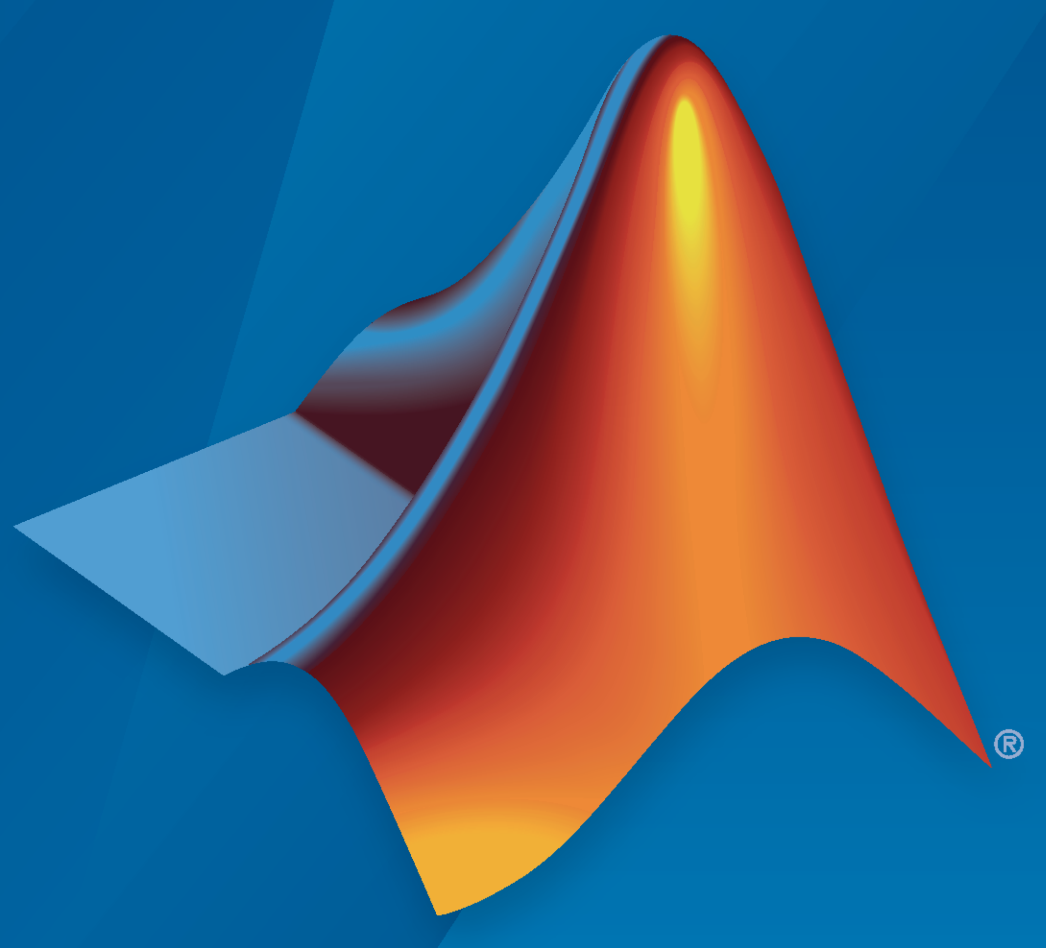
Deep Learning HDL Toolbox™

User's Guide



MATLAB®

R2023a

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Deep Learning HDL Toolbox™ User's Guide

© COPYRIGHT 2020— 2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2020	Online only	New for Version 1.0 (Release 2020b)
March 2021	Online only	Revised for Version 1.1 (Release R2021a)
September 2021	Online only	Revised for Version 1.2 (Release R2021b)
March 2022	Online only	Revised for Version 1.3 (Release R2022a)
September 2022	Online only	Revised for Version 1.4 (Release R2022b)
March 2023	Online only	Revised for Version 1.5 (Release R2023a)

Contents

1 What is Deep Learning?

Introduction to Deep Learning **1-2**

Training Process **1-3**
Training from Scratch **1-3**
Transfer Learning **1-3**
Feature Extraction **1-4**

Convolutional Neural Networks **1-5**

2 Deep Learning Processor

Deep Learning Processor IP Core Architecture **2-2**
DDR Memory **2-2**
Memory Access Arbitrator Modules **2-3**
Convolution Kernel **2-3**
Top-Level Scheduler Module **2-3**
Fully Connected Kernel **2-3**
Custom Kernel **2-3**
Profiler Utilities **2-4**

3 Applications and Examples

MATLAB Controlled Deep Learning Processor **3-2**

4 Deep Learning on FPGA Overview

Deep Learning on FPGA Workflow **4-2**

Deep Learning on FPGA Solution **4-4**
Advantages of Deep Learning on FPGA **4-4**
Deep Learning on FPGA Workflows **4-4**

5 Workflow and APIS

Prototype Deep Learning Networks on FPGA and SoC Devices **5-2**

Profile Inference Run **5-4**

Multiple Frame Support **5-7**
Input DDR Format **5-7**
Output DDR Format **5-7**
Manually Enable Multiple Frame Mode **5-8**

Initialize Deployed Deep Learning Processor Without Using a MATLAB Connection **5-9**
Prerequisites **5-9**
Generate File **5-9**
Generated File Structure **5-9**
Initiate Deployed Deep Learning Processor IP Core **5-13**

6 Fast MATLAB to FPGA Connection Using LIBIIO/Ethernet

LIBIIO/Ethernet Connection Based Deep Learning Network Deployment **6-2**
Ethernet Interface **6-2**
LIBIIO/Ethernet Performance **6-2**

7 Networks and Layers

Supported Networks, Layers, Boards, and Tools **7-2**
Supported Pretrained Networks **7-2**
Supported Layers **7-16**
Supported Boards **7-34**
Third-Party Synthesis Tools and Version Support **7-34**
Image Input Layer Normalization Hardware Implementation **7-34**

8 Custom Processor Configuration Workflow

Custom Processor Configuration Workflow **8-2**

Estimate Performance of Deep Learning Network **8-3**
Estimate Performance of Custom Deep Learning Network for Custom Processor Configuration **8-3**

Evaluate Performance of Deep Learning Network on Custom Processor Configuration . . . 8-4

Estimate Resource Utilization for Custom Processor Configuration . . . **8-10**
Estimate Resource Utilization . . . 8-10
Customize Bitstream Configuration to Meet Resource Use Requirements . . . 8-11

Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization . . . **8-17**

Generate Custom Bitstream to Meet Custom Deep Learning Network Requirements . . . **8-19**

Create Deep Learning Processor Configuration for Custom Layers . . . **8-26**
Deploy Custom Layer Networks . . . 8-26
Create a Deep learning Processor Configuration . . . 8-26
Create Custom Layer MATLAB Function . . . 8-27
Create Custom Layer Simulink Function . . . 8-28
Register Custom Layer and Model . . . 8-28
Generate Verification Model for Custom Layer . . . 8-29
Simulate and Validate Custom Layer Model . . . 8-31
Generate Custom Bitstream . . . 8-32
Deploy and Predict Custom Layer Network on Hardware . . . 8-32
Custom Layer Registration File . . . 8-32

Register, Validate, and Deploy Custom Natural Logarithm Layer Network to FPGA . . . **8-35**

9 Custom Processor Code Generation Workflow

Generate Custom Bitstream . . . **9-2**

Generate Custom Processor IP . . . **9-3**

10 Featured Examples

Get Started with Deep Learning FPGA Deployment on Intel Arria 10 SoC . . . **10-3**

Get Started with Deep Learning FPGA Deployment on Xilinx ZCU102 SoC . . . **10-6**

Get Started with Deep Learning FPGA Deployment on Xilinx ZC706 SoC . . . **10-10**

Logo Recognition Network . . . **10-13**

Deploy Transfer Learning Network for Lane Detection **10-18**

Image Category Classification by Using Deep Learning **10-23**

Defect Detection **10-32**

Profile Network to Determine Performance Bottlenecks **10-49**

Bicyclist and Pedestrian Classification by Using FPGA **10-53**

Visualize Activations of a Deep Learning Network by Using LogoNet . **10-59**

Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core **10-65**

Run a Deep Learning Network on FPGA with Live Camera Input **10-70**

Running Convolution-Only Networks by Using FPGA Deployment **10-80**

Accelerate Prototyping Workflow for Large Networks by Using Ethernet **10-86**

Create Series Network for Quantization **10-94**

Custom Deep Learning Processor Generation to Meet Performance Requirements **10-98**

Quantize Network for FPGA Deployment **10-104**

Evaluate Performance of Deep Learning Network on Custom Processor Configuration **10-110**

Customize Bitstream Configuration to Meet Resource Use Requirements **10-116**

Vehicle Detection Using DAG Network Based YOLO v2 Deployed to FPGA **10-122**

Customize Bitstream Configuration to Meet Resource Use Requirements **10-131**

Image Classification Using Neural Network on FPGA **10-137**

Classify Images on FPGA Using Quantized Neural Network **10-145**

Classify ECG Signals Using DAG Network Deployed to FPGA **10-159**

Prototype and Verify Deep Learning Networks Without Target Hardware **10-170**

Classify Images on FPGA by Using Quantized GoogLeNet Network . . **10-177**

Estimate Resource Utilization for Custom Board and Reference Design **10-191**

Speech Command Recognition by Using FPGA **10-194**

Modulation Classification by Using FPGA **10-204**

Deploy Simple Adder Network by using MATLAB Deployment Script and Deployment Instructions File **10-216**

Human Pose Estimation by Using Segmentation DAG Network Deployed to FPGA **10-224**

Semantic Segmentation of Multispectral Images by Using Quantized U-Net on FPGA **10-230**

Optimize Deep Learning Processor Configuration for Network Performance **10-239**

Run Sequence-to-Sequence Classification on FPGAs by Using Deep Learning HDL Toolbox **10-246**

Generate Word-By-Word Text on FPGAs by Using Deep Learning HDL Toolbox **10-253**

Run Sequence Forecasting on FPGA by Using Deep Learning HDL Toolbox **10-262**

Detect Objects Using YOLO v3 Network Deployed to FPGA **10-283**

Run Sequence-to-Sequence Regression on FPGAs **10-297**

Deploy and Verify YOLO v2 Vehicle Detector on FPGA **10-309**

Deploy Semantic Segmentation Network Using Dilated Convolutions on FPGA **10-324**

Run Sequence Forecasting Using a GRU Layer on an FPGA **10-333**

Deploy YAMNet Networks to FPGAs With and Without Cross-Layer Equalization **10-354**

Increase Image Resolution Using VDSR Network Running on FPGA **10-363**

Deploy Image Recognition Network on FPGA With and Without Pruning **10-379**

11 Deep Learning Quantization

Quantization Workflow Prerequisites **11-2**
Prerequisites for All Quantization Workflows **11-2**
Supported Networks and Layers **11-2**
Prerequisites for Calibration **11-2**

Prerequisites for Quantization . . . 11-3
Prerequisites for Validation . . . 11-3

Calibration . . . **11-5**
Workflow . . . 11-5

Validation . . . **11-7**
Workflow . . . 11-7

Code Generation and Deployment . . . **11-10**

12 Deep Learning Processor IP Core User Guide

Generate Custom Generic Deep Learning Processor IP Core . . . **12-2**

Deep Learning Processor IP Core . . . **12-5**

Use the Compiler Output for System Integration . . . **12-6**
External Memory Address Map . . . 12-6
Compiler Optimizations . . . 12-6
Leg Level Compilations . . . 12-7

External Memory Data Format . . . **12-9**
Key Terminology . . . 12-9
Convolution Module External Memory Data Format . . . 12-9
Fully Connected Module External Memory Data Format . . . 12-12

Deep Learning Processor IP Core Report . . . **12-14**
Summary . . . 12-14
Target Interface Configuration . . . 12-14
Register Address Mapping . . . 12-14
IP Core User Guide . . . 12-15
IP Core File List . . . 12-15

Interface with the Deep Learning Processor IP Core . . . **12-17**
Create Deep Learning Processor Configuration . . . 12-17
Select Data Processing Mode . . . 12-17
Design Processing Mode Interface Signals . . . 12-18
Design Batch Processing Mode Interface . . . 12-23
Design Streaming Mode Interface . . . 12-25
Access Data from DDR . . . 12-31

Deep Learning Processor IP Core Generation for Custom Board . . . **12-33**

13 Deep Learning HDL Toolbox Support for LSTM Networks

Support for Long Short-Term Memory Networks **13-2**
Prediction and Forecasting **13-3**

How Deep Learning HDL Toolbox Compiles the LSTM Layer **13-5**
LSTM Layer Architecture **13-5**
Compiler Interpretation **13-6**

How Deep Learning HDL Toolbox Compiles the GRU Layer **13-9**

What is Deep Learning?

- “Introduction to Deep Learning” on page 1-2
- “Training Process” on page 1-3
- “Convolutional Neural Networks” on page 1-5

Introduction to Deep Learning

Deep learning is a branch of machine learning that teaches computers to do what comes naturally to humans: learn from experience. The learning algorithms use computational methods to “learn” information directly from data without relying on a predetermined equation as model. Deep learning uses neural networks to learn useful representations of data directly from images. It is a specialized form of machine learning that can be used for applications such as classifying images, detecting objects, recognizing speech, and describing the content. The relevant features are automatically extracted from the images. The deep learning algorithms can be applied to supervised and unsupervised learning. These algorithms scale with data, that is, the performance of the network improves with size of the data.

Training Process

You can train deep learning neural networks for classification tasks by using methods such as training from scratch, or by transfer learning, or by feature extraction.

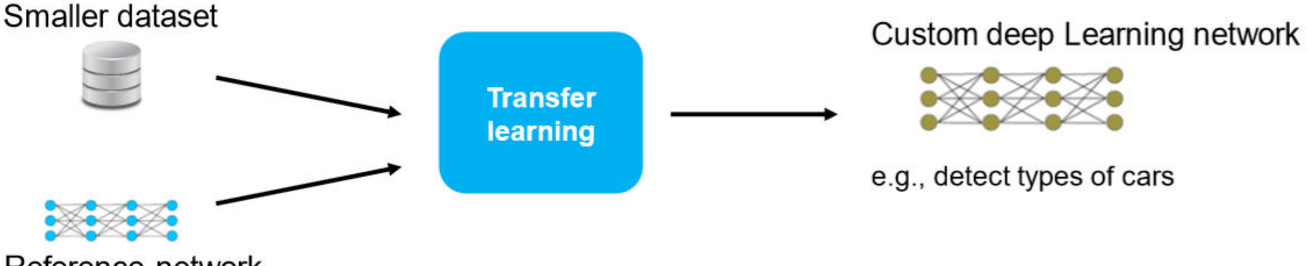
Training from Scratch

Training a deep learning neural network from scratch requires a large amount of labeled data.To create the network architecture by using Neural Network Toolbox™, you can use the built-in layers, define your own layers, or import layers from Caffe models. The neural network is then trained by using the large amounts of labeled data. Use trained network for predicting or classifying the unlabeled data. These networks can take few days or couple of weeks to train. Therefore, it is not a commonly used method for training networks.

For more information, see “Get Started with Transfer Learning”.

Transfer Learning

Transfer learning is used for cases where there is lack of labeled data. The existing network architectures, trained for scenarios with large amounts of labeled data, are used for this approach. The parameters of pretrained networks are modified to fit the unlabeled data. Therefore, transfer learning is used for transferring knowledge across various tasks. You can train or modify these networks faster so it is the most widely used training approach for deep learning applications.



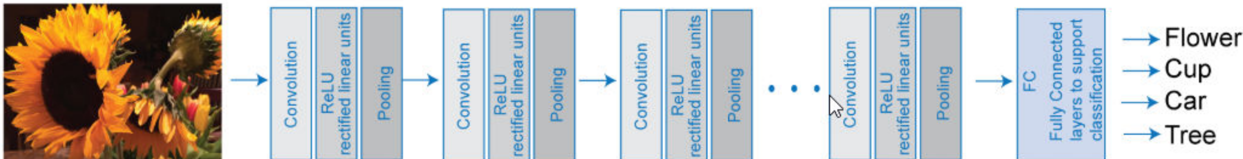
For more information, see “Get Started with Transfer Learning”

Feature Extraction

Layers in deep learning networks are trained for extracting features from the input data. This approach uses the network as a feature extractor. The features extracted after the training process can be put into various machine learning models such as Support Vector Machines (SVM).

Convolutional Neural Networks

Convolutional neural networks (CNNs) are one of the most commonly used deep learning networks.They are feedforward artificial neural networks inspired by the animal's visual cortex. These networks are designed for data with spatial and temporal information.Therefore, convolutional neural networks are widely used in image and video recognition, speech recognition, and natural language processing. The architecture of convolution neural network consists of various layers which convert the raw input pixels into a class score.



For more details, see “Learn About Convolutional Neural Networks”.

You can train CNNs from scratch, by transfer learning, or by feature extraction. You can then use the trained network for classification or regression applications.

For more details on training CNNs, see “Pretrained Deep Neural Networks” .

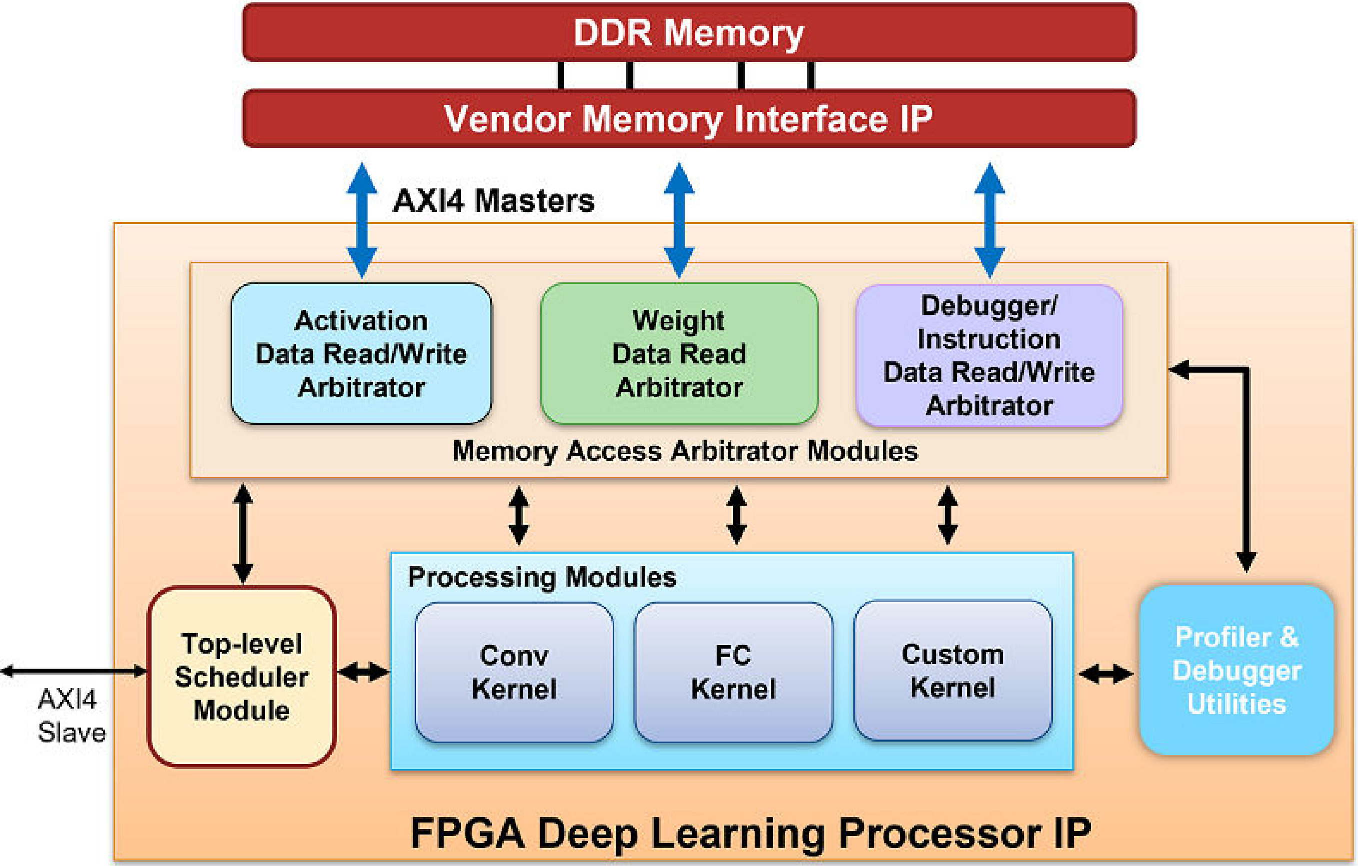
For more details on deep learning, training process, and CNNs, see Deep Learning Onramp.

Deep Learning Processor

Deep Learning Processor IP Core Architecture

Deep Learning HDL Toolbox provides a target-independent generic deep learning processor IP core that you can deploy to any custom platform. You can reuse the deep learning processor IP core and share it to accommodate deep neural networks that have various layer sizes and parameters. Use this deep learning processor IP core to rapidly prototype deep neural networks from MATLAB and deploy the network to FPGAs.

This image shows the deep learning processor IP core architecture:



To illustrate the deep learning processor IP core architecture, consider an image classification example.

DDR Memory

You can store the input images, weights, and output images in the external DDR memory. The processor consists of three AXI4 master interfaces that communicate with the external memory. You can use one of the AXI4 Master interfaces to load the input images onto the processing modules. The `compile` method generates the weight data. To retrieve the activation data from the DDR , see "External Memory Data Format" on page 12-9. You can write the weight data to a deployment file and use the deployment file to initialize the generated deep learning processor. For more information, see "Initialize Deployed Deep Learning Processor Without Using a MATLAB Connection" on page 5-9.

Memory Access Arbitrator Modules

The activation and weight memory access arbitrator modules use AXI Master interface to read and write weights and activation data to and from the processing modules. The profiler AXI Master interface reads and writes profiler timing data and instructions to the profiler module.

Convolution Kernel

The `Conv Kernel` implements layers that have a convolution layer output format. The two AXI4 master interfaces provide the weights and activations for the layer to the `Conv Kernel`. The `Conv Kernel` then performs the implemented layer operation on the input image. This kernel is generic because it can support tensors and shapes of various sizes. For a list of layers with the `conv` output format, see "Supported Layers" on page 7-16. For a list of the `conv kernel` properties, see `dlhdl.ProcessorConfig`.

Top-Level Scheduler Module

The top-level scheduler module schedules what instructions to run, what data to read from DDR, and when to read the data from DDR. The scheduler module acts as the central computer in a distributed computer architecture that distributes instructions to the processing modules. For example, if the network has a convolution layer, fully connected layer, and a multiplication layer the scheduler:

- Schedules the processing and data read instructions for the convolution layer and sends them to the `conv` kernel.
- Schedules the processing and data read instructions for the fully connected layer and sends them to the `FC` kernel.
- Schedules the processing and data read instructions for the multiplication layer and sends them to the `custom` kernel.

Fully Connected Kernel

The fully connected (FC) kernel implements layers that have a fully connected layer output format. The two AXI4 master interfaces provide the weights and activations to the `FC Kernel`. The `FC Kernel` then performs the fully-connected layer operation on the input image. This kernel is also generic because it can support tensors and shapes of various sizes. For a list of layers with FC output format, see "Supported Layers" on page 7-16. For a list of the `FC Kernel` properties, see `dlhdl.ProcessorConfig`.

Custom Kernel

The custom kernel module implements layers that are registered as a custom layer by using the `registerCustomLayer` method. To learn how to create, register, and validate your own custom layers, see "Register, Validate, and Deploy Custom Natural Logarithm Layer Network to FPGA" on page 8-35. For example, the addition layer, multiplication layer, `resize2dlayer`, and so on are implemented on the custom kernel module. For a list of layers implemented on this module, see "Supported Layers" on page 7-16. For a list of the `Custom Kernel` properties, see `dlhdl.ProcessorConfig`.

Profiler Utilities

When you set the `Profiler` argument of the `predict` or `predictAndUpdateState` methods to `on`, the profiler module collects information from the kernel, such as the `Conv Kernel` start and stop times, `FC Kernel` start and stop times, and so on. The profiler module uses this information to create a profiler table with these results. For more information, see “Profile Inference Run” on page 5-4.

See Also

`dlhdl.ProcessorConfig` | `compile`

More About

- “Custom Processor Configuration Workflow” on page 8-2
- “Deep Learning Processor IP Core” on page 12-5
- “Use the Compiler Output for System Integration” on page 12-6
- “External Memory Data Format” on page 12-9

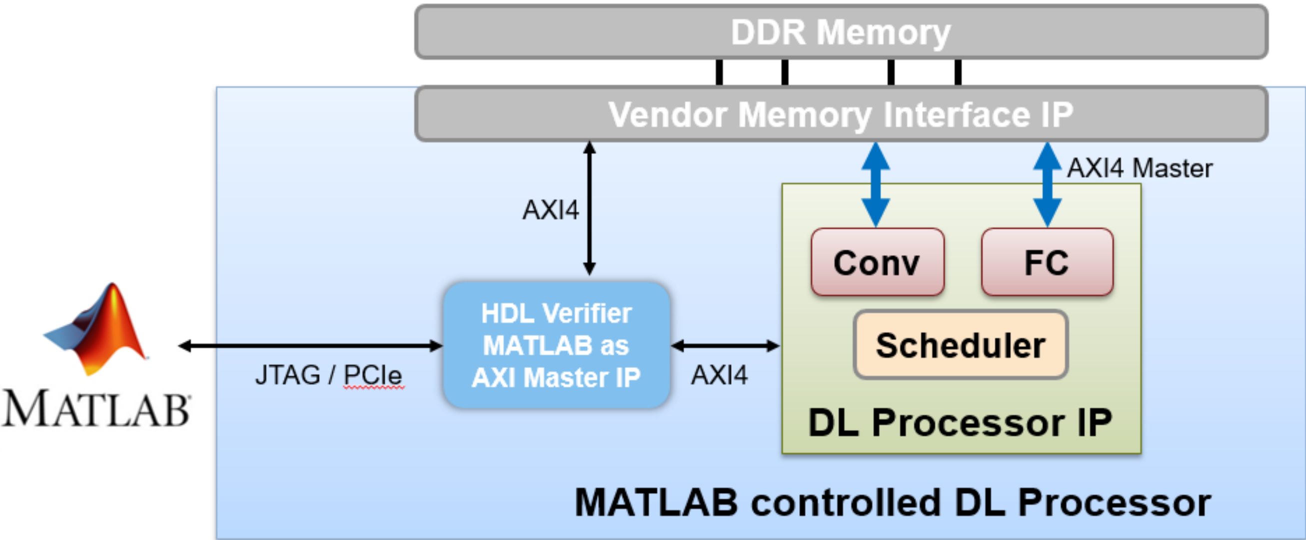
Applications and Examples

MATLAB Controlled Deep Learning Processor

To rapidly prototype the deep learning networks on FPGAs from MATLAB, use a MATLAB controlled deep learning processor. The processor integrates the generic deep learning processor with the HDL Verifier™ MATLAB as AXI Master IP. For more information on:

- Generic deep learning processor IP, see “Generate Custom Generic Deep Learning Processor IP Core” on page 12-2 .
- MATLAB as AXI Master IP, see “Set Up AXI Manager” (HDL Verifier) .

You can use this processor to run neural networks with various inputs, weights, and biases on the same FPGA platform because the deep learning processor IP core can handle tensors and shapes of any sizes. Before you use the MATLAB as AXI Master, make sure that you have installed the HDL Verifier support packages for the FPGA boards. This figure shows the MATLAB controlled deep learning processor architecture.



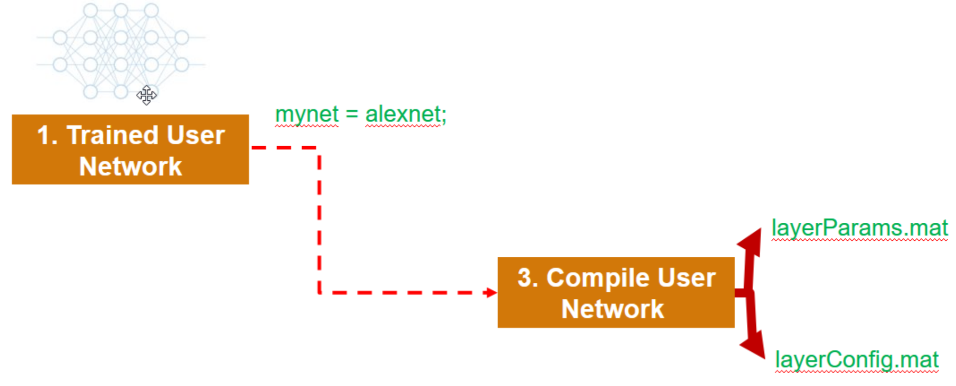
To integrate the generic deep learning processor IP with the MATLAB as AXI Master, use the AXI4 Slave interface of the deep learning processor IP core. By using a JTAG or PCI express interface, the IP responds to read or write commands from MATLAB. Therefore, you can use the MATLAB controlled deep learning processor to deploy the deep learning neural network to the FPGA boards from MATLAB, perform operations specified by the network architecture, and then return the predicted results to MATLAB. For example, see “Image Classification Using Neural Network on FPGA” on page 10-137.

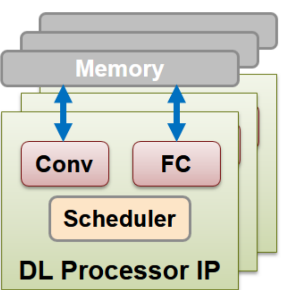
Deep Learning on FPGA Overview

- “Deep Learning on FPGA Workflow” on page 4-2
- “Deep Learning on FPGA Solution” on page 4-4

Deep Learning on FPGA Workflow

This figure illustrates deep learning on FPGA workflow.





To use the workflow:

1. **Load deep learning neural network**

   You can load the various deep learning neural networks such as Alexnet, VGG and GoogleNet onto the MATLAB framework. When you compile the network, the network parameters are saved into a structure that consists of `NetConfigs` and `layerConfigs`. `NetConfigs` consists of the weights and biases of the trained network. `layerConfig` consists of various configuration values of the trained network.
2. **Modify pretrained neural network on MATLAB using transfer learning**

   The internal network developed on the MATLAB framework is trained and modified according to the parameters of the external neural network. See also “Get Started with Transfer Learning”.
3. **Compile user network**

   Compilation of the user network usually begins with validating the architecture, types of layers present , data type of input and output parameters, and maximum number of activations. This FPGA solution supports series network architecture with data types of single and int8. For more details, see **"Product Description"**. If the user network features are different, the compiler produces an error and stops. The compiler also performs sanity check by using weight compression and weight quantization.
4. **Deploy on target FPGA board**

   By using specific APIs and the `NetConfigs` and `layerConfigs`, deploying the compiled network converts the user-trained network into a fixed bitstream and then programs the bitstream on the target FPGA.
5. **Predict outcome**

   To classify objects in the input image, use the deployed framework on the FPGA board.

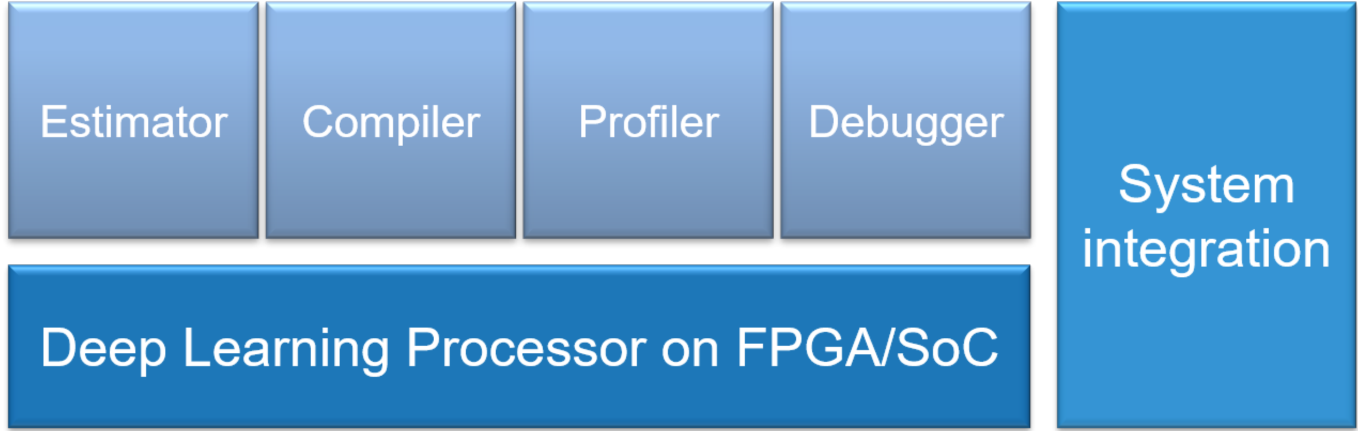
See Also

"Deep Learning on FPGA Solution" on page 4-4

Deep Learning on FPGA Solution

The deep learning on field programable gate array (FPGA) solution provides you with an end-to-end workflow to compile, deploy, profile and debug your custom pretrained deep learning networks. You can also generate a custom deep learning processor IP core that you can integrate into your custom reference design.

This figure shows the MATLAB based deep learning on FPGA solution.



The workflow is:

- Generate the external memory address map by using the `compile` function.
- Retrieve the network layer latency and overall network performance in frames per second(FPS) by using the profiler and debugger.
- Generate a custom deep learning processor IP core.
- Integrate the generated IP core into your custom reference design.

Generate the external memory address map by using the compiler. Retrieve the network layer latency and overall network performance in frames per second (FPS) by using the profiler and debugger. Generate a custom deep learning processor IP core and integrate the generated IP core into your custom reference design.

Advantages of Deep Learning on FPGA

FPGAs offer several advantages over a graphics processing unit (GPU) for deep learning applications.

- High performance by providing high throughput and low latency.
- Low power consumption by enabling you to fine-tune the hardware to your desired application.
- Cost effective because you can integrate additional capabilities on the same chip, which also saves costs and board space.

Deep Learning on FPGA Workflows

Based on your goals, use the information in this table to choose your workflow.

Goal	Workflow
Run a pretrained series network on your target FPGA board.	“Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2
Obtain the performance of your pretrained series network for a preconfigured deep learning processor.	“Estimate Performance of Deep Learning Network” on page 8-3
Customize the deep learning processor to meet your resource utilization requirements.	“Estimate Resource Utilization for Custom Processor Configuration” on page 8-10
Generate a custom deep learning processor for your FPGA.	“Generate Custom Bitstream” on page 9-2
Learn about the benefits of quantizing your pretrained series networks.	“Quantization of Deep Neural Networks”
Compare the accuracy of your quantized pretrained series networks against your single data type pretrained series network.	“Validation” on page 11-7
Run a quantized pretrained series network on your target FPGA board.	“Code Generation and Deployment” on page 11-10

Workflow and APIS

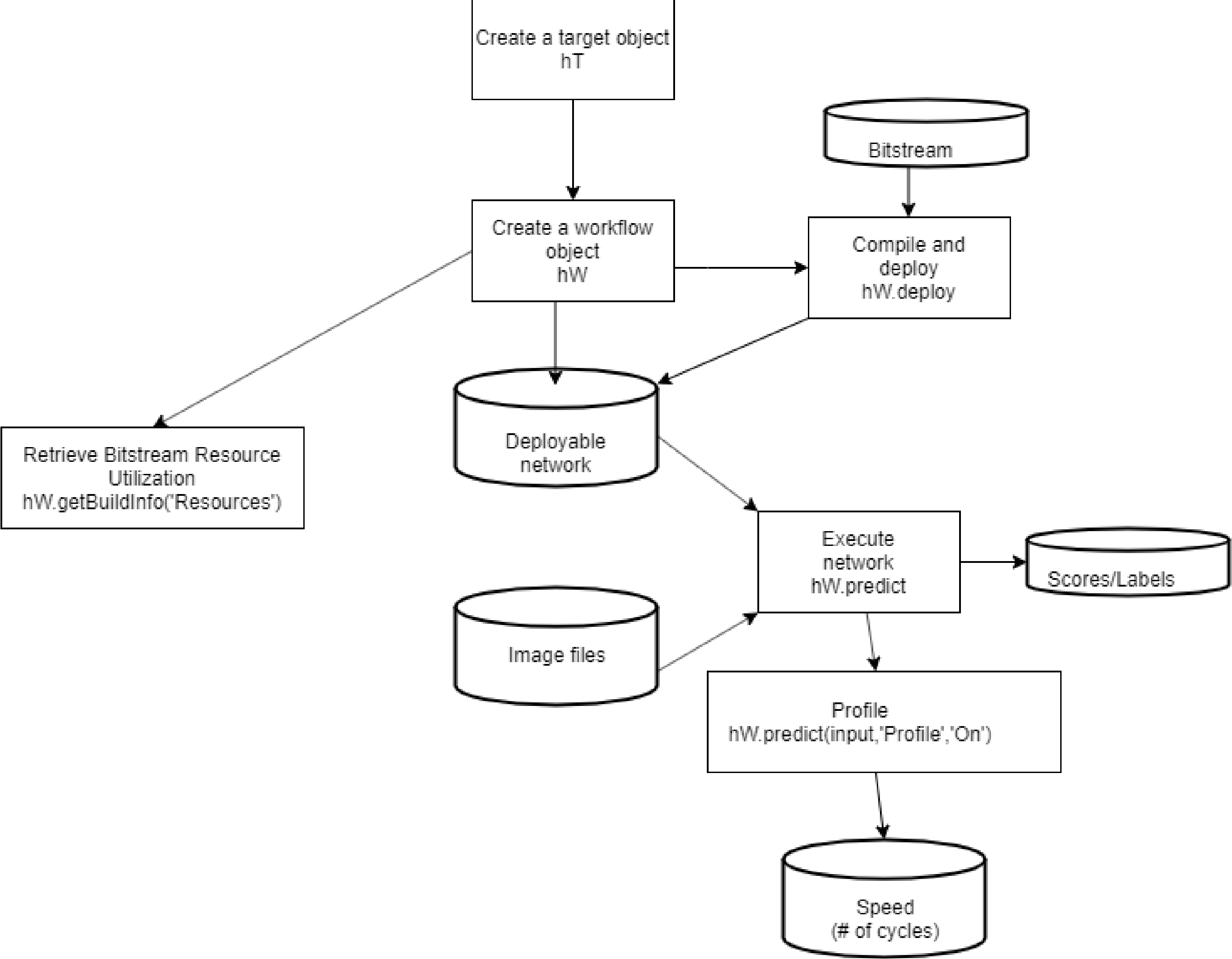
- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2
- “Profile Inference Run” on page 5-4
- “Multiple Frame Support” on page 5-7
- “Initialize Deployed Deep Learning Processor Without Using a MATLAB Connection” on page 5-9

Prototype Deep Learning Networks on FPGA and SoC Devices

To prototype and deploy your custom series deep learning network, create an object of class `dlhdl.Workflow`. Use this object to:

- Compile and deploy the deep learning network on specified target FPGA or SoC board by using the `deploy` function.
- Retrieve the bitstream resource utilization by using the `getBuildInfo` function.
- Execute the deployed deep learning network and predict the classification of input images by using the `predict` function.
- Calculate the speed and profile of the deployed deep learning network by using the `predict` function.

This workflow illustrates deploying your deep learning network to the FPGA boards.



See Also

`dlhdl.Workflow` | `dlhdl.Target`

More About

- “Get Started with Deep Learning FPGA Deployment on Xilinx ZCU102 SoC” on page 10-6

Profile Inference Run

This example shows how to retrieve the prediction and profiler results for the ResNet-18 network. View the network prediction and performance data for the layers, convolution module and fully connected modules in your pretrained deep learning network.

1. Create an object of class `Workflow` by using the `dlhdl.Workflow` class.

   See, "Create Workflow Object by using Property Name Value Pairs".
2. Set a pretrained deep learning network and bitstream for the workflow object.

   See, "Create Workflow Object by using Property Name Value Pairs".
3. Create an object of class `dlhdl.Target` and specify the target vendor and interface. See, `dlhdl.Target`.
4. To deploy the network on a specified target FPGA board, call the `deploy` method for the workflow object. See, `deploy`.
5. Call the `predict` function for the workflow object. Provide an array of images as the `InputImage` parameter. Provide arguments to turn on the profiler. See "Classify Images on FPGA Using Quantized Neural Network".

   The labels classifying the images are stored in a structure `struct` and displayed on the screen. The performance parameters of speed and latency are returned in a structure `struct`.

Use this image to run this code:

```
snet = resnet18;
hT = dlhdl.Target('Xilinx','Interface','Ethernet');
hW = dlhdl.Workflow('Net',snet,'Bitstream','zcu102_single','Target',hT);
hW.deploy;
image = imread('zebra.jpeg');
inputImg = imresize(image, [224, 224]);
imshow(inputImg);
[prediction, speed] = hW.predict(single(inputImg),'Profile','on');
[val, idx] = max(prediction);
snet.Layers(end).ClassNames{idx}
```

```
### Finished writing input activations.
### Running single input activations.
```

```
              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                    23659630                  0.10754                      1           23659630             9.3
    conv1                   2224115                  0.01011
    pool1                    572867                  0.00260
    res2a_branch2a           972699                  0.00442
    res2a_branch2b           972568                  0.00442
    res2a                    209312                  0.00095
    res2b_branch2a           972733                  0.00442
    res2b_branch2b           973022                  0.00442
    res2b                    209736                  0.00095
    res3a_branch2a           747507                  0.00340
    res3a_branch2b           904291                  0.00411
    res3a_branch1            538763                  0.00245
    res3a                    104750                  0.00048
    res3b_branch2a           904389                  0.00411
    res3b_branch2b           904367                  0.00411
    res3b                    104886                  0.00048
    res4a_branch2a           485682                  0.00221
    res4a_branch2b           880001                  0.00400
    res4a_branch1            486429                  0.00221
    res4a                     52628                  0.00024
    res4b_branch2a           880053                  0.00400
    res4b_branch2b           880035                  0.00400
    res4b                     52478                  0.00024
    res5a_branch2a          1056299                  0.00480
    res5a_branch2b          2056857                  0.00935
    res5a_branch1           1056510                  0.00480
    res5a                     26170                  0.00012
    res5b_branch2a          2057203                  0.00935
    res5b_branch2b          2057659                  0.00935
    res5b                     26381                  0.00012
    pool5                     71405                  0.00032
    fc1000                   216155                  0.00098
 * The clock frequency of the DL processor is: 220MHz
```

The profiler data returns these parameters and their values:

- `LastFrameLatency(cycles)` — Total number of clock cycles for previous frame execution.
- Clock frequency — Clock frequency information is retrieved from the bitstream that was used to deploy the network to the target board. For example, the profiler returns `* The clock frequency of the DL processor is: 220MHz`. The clock frequency of 220 MHz is retrieved from the `zcu102_single` bitstream.
- `LastFrameLatency(seconds)` — Total number of seconds for previous frame execution. The total time is calculated as `LastFrameLatency(cycles)/Clock Frequency`. For example the `conv_module` `LastFrameLatency(seconds)` is calculated as `2224115/(220*10^6)`.
- `FramesNum` — Total number of input frames to the network. This value will be used in the calculation of `Frames/s`.
- `Total Latency` — Total number of clock cycles to execute all the network layers and modules for `FramesNum`.
- `Frames/s` — Number of frames processed in one second by the network. The total `Frames/s` is calculated as `(FramesNum*Clock Frequency)/Total Latency`. For example the `Frames/s` in the example is calculated as `(1*220*10^6)/23659630`.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `predict`

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2

- “Profile Network to Determine Performance Bottlenecks” on page 10-49

Multiple Frame Support

Deep Learning HDL Toolbox supports multiple frame mode enabling you to write multiple images into the double data rate (DDR) memory and read back multiple results at the same time. To improve the performance of your deployed deep learning networks, use multiple frame mode.

Input DDR Format

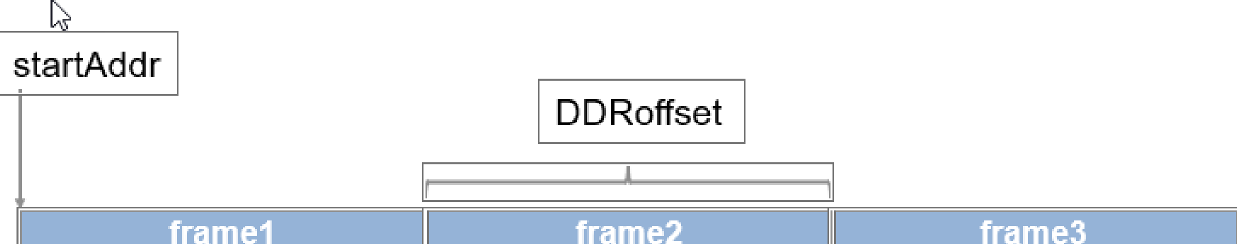
To format input images to meet the multiple frame input DDR format, you must have:

- The start address of the input data for the DDR
- The DDR offset for a single input image frame

This information is automatically generated by the `compile` method. For more information on the generated DDR address offsets, see “Use the Compiler Output for System Integration” on page 12-6.

You can also specify the maximum number of input frames as an optional argument in the `compile` method. For more information, see “Generate DDR Memory Offsets Based On Number of Input Frames”.

This graphic shows the format of the input area of the DDR for multiple input images.



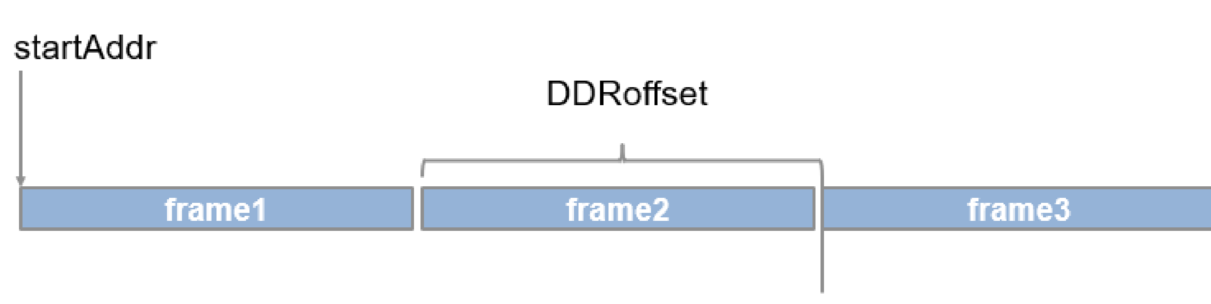
Output DDR Format

To retrieve the results for multiple image inputs from the output area of the DDR, you must have:

- The start address of the output area of the DDR
- The DDR offset of a single result

The output results must be formatted as a multiple of the FC output feature size. The information and formatting are generated by the `compile` method. For more information on the generated DDR address offsets, see “External Memory Address Map” on page 12-6.

This graphic shows the formatting of the output area of the DDR memory.



Manually Enable Multiple Frame Mode

After the deep learning network has been deployed, you can manually enable the multiple frame mode by writing the number of frames through a network configuration (NC) port. To manually enter the multiple frame mode at the MATLAB command line enter:

```
dnnfpga.hwutils.writeSignal(1, dnnfpga.hwutils.numTo8Hex(addrMap('FrameCount')),15,hT);
```

`addrMap('FrameCount')` returns the AXI register address for `FrameCount`, 15 is the number of images, and hT represents the `dlhdl.Target` class that contains the board definition and board interface definition. For more information about the AXI register addresses, see “Deep Learning Processor IP Core Report” on page 12-14.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile`

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2

Initialize Deployed Deep Learning Processor Without Using a MATLAB Connection

Generate a file that has instructions to communicate with the deployed deep learning processor IP core by using Deep Learning HDL Toolbox. Initialize the deployed deep learning processor IP core without a MATLAB connection by using a utility to parse and execute the instructions in the generated file.

Prerequisites

- Deep Learning HDL Toolbox Support Package for Intel® FPGA and SoC Devices

  or Deep Learning HDL Toolbox Support Package for Xilinx® FPGA and SoC Devices
- Set up a secure digital (SD) card by using the guided SD card setup. For Intel boards, see “Guided SD Card Set Up” (Deep Learning HDL Toolbox Support Package for Intel FPGA and SoC Devices). For Xilinx boards, see “Guided SD Card Set Up” (Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices).

Generate File

To generate a file that has the instructions to program and initialize the generated deep learning processor IP configuration, set the deployment target to `File` by using the `Target` method of the `dlhdl.Workflow` object. For example, this code generates a `dlhdl.Workflow` object that has the ResNet-18 convolutional neural network as the network, `zcu102_single` as the target bitstream, and deploys the instructions to a file that is called `zcu102socinitdata.dln`.

```
hTarget = dlhdl.Target('Xilinx',Interface = 'File', Filename = 'zcu102socinitdata.dln');
hW = dlhdl.Workflow(Network = resnet18, Bitstream = 'zcu102_single', Target = hTarget);
hW.compile;
hW.deploy;
```

When you use the `deploy` method of the `dlhdl.Workflow` object, and the interface option for `dlhdl.Target` is set to `File`, the `dlhdl.Workflow` object and associated instructions are written to the file whose name is in `Filename`.

When you do not enter the file name for the `dlhdl.Target` object, the name of the created file is the bitstream name. For example, in this code the generated file name is `zcu102_single.dln`.

```
hTarget = dlhdl.Target('Xilinx',Interface = 'File');
hW = dlhdl.Workflow(Network = resnet18, Bitstream = 'zcu102_single', Target = hTarget);
hW.compile;
hW.deploy;
```

Generated File Structure

The generated file is a binary file that consists of:

- A header section that contains information such as the date and time the file was generated, Deep Learning HDL Toolbox version, DDR address range, and so on.
- A start of data (SOD) section that indicates the start of instructions to read and write data.
- Data section that has AXI read and write transactions.
- An end of data command (EOD) that marks the end of the file.

Header Section Information

This table lists the information available in the file header section. Strings are null terminated and `uint32` data types are stored in reverse byte order. When your script reads `uint32` data types, read the data from right to left. For example, `0xa0000000` is stored in the generated files as `00 00 00 A0`.

Field	Data Type	Example Information
File version	string	'MWDLNV2'
Date and time	string	'25-Oct-2021 12:44:03'
Deep Learning HDL Toolbox name	string	Deep Learning HDL Toolbox
Deep Learning HDL Toolbox version	string	'1.2'
Deep Learning HDL Toolbox release information	string	'R2022a'
Deep Learning HDL Toolbox date	string	'30-Sep-2021'
Deep learning processor base address	uint32	0xa0000000
Deep learning processor address range	uint32	0x00010000
DDR memory base address	uint32	0x80000000
DDR memory address range	uint32	0x20000000
Target device platform	string	'Xilinx'
Device tree node name for deep learning processor IP core transmit	string	'mwipcore_dl0:mmrw0'
Device tree node name for deep learning processor IP core receive	string	'mwipcore_dl0:mmrd0'
Device tree node name for DDR memory transmit	string	'mwipcore_ddr0:mm2s0'
Device tree node name for DDR memory receive	string	'mwipcore_ddr0:s2mm0'

This image shows the header information section of the generated file.

```
00000000h: 4D 57 44 4C 4E 56 30 32 00 32 39 2D 4F 63 74 2D ; MWDLNV02.29-Oct-
00000010h: 32 30 32 31 20 30 39 3A 31 37 3A 33 33 00 44 65 ; 2021 09:17:33.De
00000020h: 65 70 20 4C 65 61 72 6E 69 6E 67 20 48 44 4C 20 ; ep Learning HDL
00000030h: 54 6F 6F 6C 62 6F 78 00 31 2E 32 00 28 52 32 30 ; Toolbox.1.2.(R20
00000040h: 32 32 61 29 00 33 30 2D 53 65 70 2D 32 30 32 31 ; 22a).30-Sep-2021
00000050h: 00 00 00 00 A0 00 00 01 00 00 00 00 80 00 00 00 ; .... .......€...
00000060h: 20 58 69 6C 69 6E 78 00 6D 77 69 70 63 6F 72 65 ;  Xilinx.mwipcore
00000070h: 5F 64 6C 30 3A 6D 6D 77 72 30 00 6D 77 69 70 63 ; _dl0:mmwr0.mwipc
00000080h: 6F 72 65 5F 64 6C 30 3A 6D 6D 72 64 30 00 6D 77 ; ore_dl0:mmrd0.mw
00000090h: 69 70 63 6F 72 65 5F 64 64 72 30 3A 6D 6D 32 73 ; ipcore_ddr0:mm2s
000000a0h: 30 00 6D 77 69 70 63 6F 72 65 5F 64 64 72 30 3A ; 0.mwipcore_ddr0:
000000b0h: 73 32 6D 6D 30 00 53 4F 44 00 57 52 44 00 57 52 ; s2mm0.SOD.WRD.WR
```

Data Section

After the header section, each block starts with a three-letter command, listed in this table:

Field	Type	Notes
SOD	string	Start of data
EOD	string	End of data
TXT	string	Text field only
WRD	string	Data to write
RDD	string	Data to read

Read Data Command

After detecting the SOD command, check for read commands. The read data command appears in the generated file when you are waiting for a done flag from the deep learning processor. When the read command is executed:

- A while loop is started.
- A read is executed from a single register.

The read command and while loop end when the value of the data read from the register is equal to the value of the data in the data-to-read section of the read command.

The read data command follows this format:

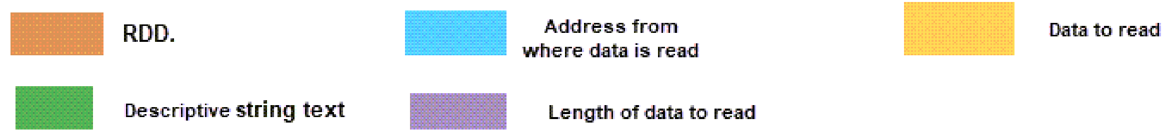
- `'RDD.'`, a null terminated string indicating start of the read command.
- `'Text description.'`, a null terminated string that indicated the address from where the data is read and length of the data to be read.
- Hexadecimal representation of the register address from where data is read. This data is specified as a `uint32` data type.
- Hexadecimal representation of the length of data to be read. This data is specified as a `uint32` data type. The length data is the number of 32-bit data packets to be read. For example, a length value of one indicates one 32-bit data packet to read.
- Hexadecimal representation of the data to be read. This data is specified as a `uint32` data type.

In the generated file, `uint32` data types are stored in reverse byte order. When your script reads `uint32` data types, read the data from right to left. For example, `0xa0000184` is stored in the generated files as `84 01 00 A0`.

This image shows a read instruction in the generated file and the structure of the read data command. `RDD.RD@ADDR:0xa0000184 Length:1. 0xa0000184 0x00000001 0x00000001`.

```
00003570h: 01 00 A0 01 00 00 00 01 00 00 00 52 44 44 00 52 ; .. ........RDD.R
00003580h: 44 40 41 44 44 52 3A 20 30 78 61 30 30 30 30 31 ; D@ADDR: 0xa00001
00003590h: 38 34 2C 20 4C 65 6E 67 74 68 3A 20 31 00 84 01 ; 84, Length: 1.„.
000035a0h: 00 A0 01 00 00 00 01 00 00 00 57 52 44 00 57 52 ; . .......WRD.WR
```

RDD.RD@ADDR: 0xa0000184, Length: 1.0xa0000184 0x000000010x00000001



Write Data Command

After detecting the SOD command, check for write commands.

The write data command follows this format:

- `'WRD.'`, a null terminated string indicating start of the write command.
- `'Text description.'`, a null terminated string that indicated the address from where the data is read and length of the data to be read
- Hexadecimal representation of the register address where data is to be written. This data is specified as a `uint32` data type.
- Hexadecimal representation of the length of data to write. This data is specified as a `uint32` data type. The length data is the number of 32-bit data packets to write. For example, a length value of 36 indicates 36 32-bit data packets to write. When there are `N` data packets to write, the write data format in the generated file after the text description field is address data, length data, data packet 1, data packet 2,..., and data packet N.
- Hexadecimal representation of the data to write. This data is specified as a `uint32` data type.

In the generated file, `uint32` data types are stored in reverse byte order. When your script reads `uint32` data types, read the data from right to left. For example, `0xa0000184` is stored in the generated files as `84 01 00 A0`.

This image shows a write instruction in the generated file and the structure of the write data command. `WRD.WR@ADDR:0x81800000 Len:36. 0x81800000 0x00000024 0x00000084.....0x00000003`. In this example, there are 36 data packets to write. The first data packet is `0x00000084` and the last data packet is `0x00000003`.

```
00003a30h: A0 01 00 00 00 00 00 00 00 57 52 44 00 57 52 40 ; .........WRD.WR@
00003a40h: 41 44 44 52 3A 20 30 78 38 31 38 30 30 30 30 30 ; ADDR: 0x81800000
00003a50h: 20 4C 65 6E 3A 20 33 36 00 00 00 80 81 24 00 00 ;  Len: 36...€ $..
00003a60h: 00 84 00 00 00 00 02 00 00 00 00 10 00 00 00 08 ; .„..............
00003a70h: 08 20 00 01 00 8C 00 00 00 F2 00 00 00 00 00 08 ; . ...Œ...ò......
00003a80h: 08 20 00 01 00 8C 00 00 00 F2 00 00 00 00 00 08 ; . ...Œ...ò......
00003a90h: 08 20 00 01 00 8C 00 00 00 F2 00 00 00 00 00 08 ; . ...Œ...ò......
00003aa0h: 08 20 00 01 00 8C 00 00 00 F2 00 00 00 84 00 00 ; . ...Œ...ò...„..
00003ab0h: 00 00 02 00 00 00 00 10 00 00 00 08 08 20 00 01 ; ............. ..
00003ac0h: 00 8C 00 00 00 F2 00 00 00 00 02 00 00 00 00 00 ; .Œ...ò..........
00003ad0h: 00 00 00 00 00 05 00 00 00 00 00 00 00 08 80 00 ; ..............€.
00003ae0h: 00 80 00 00 00 00 00 04 08 00 01 00 00 03 00 00 ; .€..............
00003af0h: 00 57 52 44 00 57 52 40 41 44 44 52 3A 20 30 78 ; .WRD.WR@ADDR: 0x
```

WRD.WR@ADDR: 0x81800000 Len: 36.0x818000000x000000240x00000084.......0x00000003

Initiate Deployed Deep Learning Processor IP Core

To initiate the deployed deep learning processor IP core, create a script to parse the generated file and extract the instructions to program the deep learning processor IP core. The script must perform these actions:

1. Take the generated file as an input and open the generated file.
2. Extract the header information.
3. Detect the start of data (SOD) command. Once the SOD command is detected:

   - Read data by extracting the address, length of data to be read, and data to read information from the read packet structure. Use the `readmemory` function with the address and length as input arguments.
   - Write data by extracting the write data address and data to write information from the write packet structure. Use the `writememory` function with the address and data to write as input arguments.
4. Detect the end of data (EOD) command and close the generated file.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy`

Related Examples

- "Deploy Simple Adder Network by using MATLAB Deployment Script and Deployment Instructions File" on page 10-216

Fast MATLAB to FPGA Connection Using LIBIIO/Ethernet

LIBIIO/Ethernet Connection Based Deep Learning Network Deployment

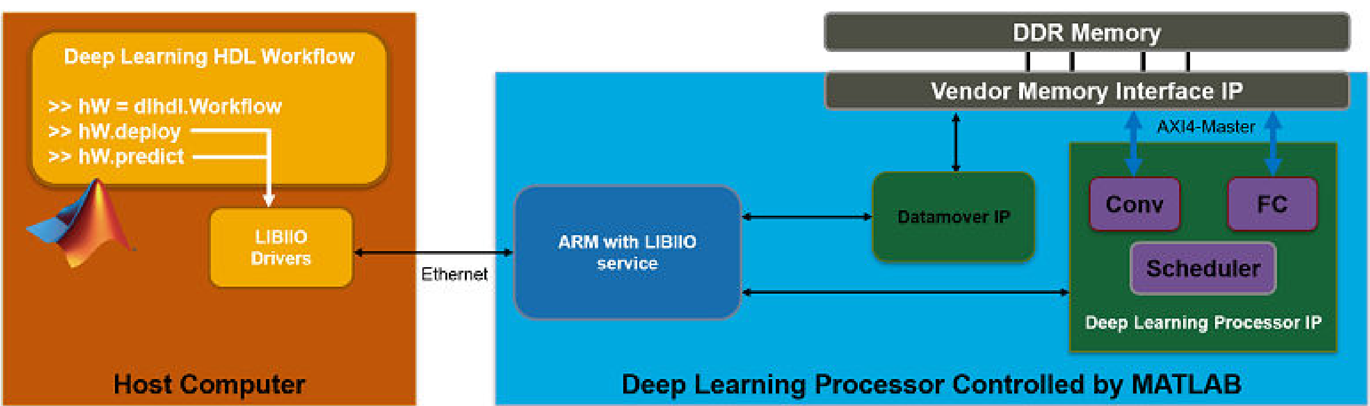
In this section...
"Ethernet Interface" on page 6-2 "LIBIIO/Ethernet Performance" on page 6-2

Ethernet Interface

The Ethernet interface leverages an ARM processor to send and receive information from the deployed deep learning network running on the FPGA. The ARM processor runs on a Linux operating system. You can use the Linux operating system services to interact with the FPGA. When you use the Ethernet interface, the bitstream is downloaded to the SD card. The bitstream is persistent through power cycles and is reprogrammed each time the FPGA is turned on. The ARM processor is configured with the correct device tree when the bitstream is programmed.

To communicate with the design running on the FPGA, MATLAB leverages the Ethernet connection between the host computer and ARM processor. The ARM processor runs a LIBIIO service, which communicates with a Datamover IP in the FPGA design. You use the Datamover IP for fast data transfers between the host computer and FPGA, which is useful when prototyping large deep learning networks that have long transfer times over JTAG. The ARM processor generates the read and write transactions to access memory locations in the onboard memory and deep learning processor.

This figure shows the high-level architecture of the Ethernet interface.



You can configure your `dlhdl.Workflow` object hardware interface to use an Ethernet connection at the time of the workflow object creation. For more information, see "Create Target Object That Has an Ethernet Interface and Set IP Address".

LIBIIO/Ethernet Performance

The improvement in performance speed of JTAG compared to LIBIIO/Ethernet is listed in this table.

Transfer Speed	JTAG	IIO	Speedup

Write Transfer Speed	225 kB/s	33 MB/s	Approximately 150x
Read Transfer Speed	162 kB/s	32 MB/s	Approximately 200x

See Also

`dlhdl.Target`

More About

- “Accelerate Prototyping Workflow for Large Networks by Using Ethernet” on page 10-86

7

Networks and Layers

Supported Networks, Layers, Boards, and Tools

In this section...

"Supported Pretrained Networks" on page 7-2
"Supported Layers" on page 7-16
"Supported Boards" on page 7-34
"Third-Party Synthesis Tools and Version Support" on page 7-34
"Image Input Layer Normalization Hardware Implementation" on page 7-34

Supported Pretrained Networks

Deep Learning HDL Toolbox supports code generation for series convolutional neural networks (CNNs or ConvNets). You can generate code for any trained CNN whose computational layers are supported for code generation. For a full list, see "Supported Layers" on page 7-16. You can use one of the pretrained networks listed in the table to generate code for your target Intel or Xilinx FPGA boards.

Network	Network Description	Type	Single Data Type (with Shipping Bitstreams)			INT8 data type (with Shipping Bitstreams)			Application Area
			ZCU102	**ZC706**	**Arria10 SoC**	**ZCU102**	**ZC706**	**Arria10 SoC**	Classification
AlexNet	AlexNet convolutional neural network.	Series Network	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	Classification

LogoNet	Logo recognition network (LogoNet) is a MATLAB developed logo identification network. For more information, see “Logo Recognition Network”.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Classification
DigitsNet	Digit classification network. See “Create Simple Deep Learning Neural Network for Classification”	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Classification

Lane detection	LaneNet convolutional neural network. For more information, see “Deploy Transfer Learning Network for Lane Detection” on page 10-18.	Series Network	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	Classification
VGG-16	VGG-16 convolutional neural network. For the pretrained VGG-16 model, see `vgg16`.	Series Network	No. Network exceeds PL DDR memory size	No. Network exceeds FC module memory size.	Yes	Yes	No. Network exceeds FC module memory size.	Yes	Classification
VGG-19	VGG-19 convolutional neural network. For the pretrained VGG-19 model, see `vgg19` .	Series Network	No. Network exceeds PL DDR memory size	No. Network exceeds FC module memory size.	Yes	Yes	No. Network exceeds FC module memory size.	Yes	Classification

Darknet-19	Darknet-19 convolutional neural network. For the pretrained darknet-19 model, see `darknet19`.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Classification
Radar Classification	Convolutional neural network that uses micro-Doppler signatures to identify and classify the object. For more information, see "Bicyclist and Pedestrian Classification by Using FPGA" on page 10-53.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Classification and Software Defined Radio (SDR)

Defect Detection `snet_defnet`	`snet_defnet` is a custom AlexNet network used to identify and classify defects. For more information, see “Defect Detection” on page 10-32.	Series Network	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	Classification
Defect Detection `snet_blemdetnet`	`snet_blemdetnet` is a custom convolutional neural network used to identify and classify defects. For more information, see “Defect Detection” on page 10-32.	Series Network	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	Classification

DarkNet-53	Darknet-53 convolutional neural network. For the pretrained DarkNet-53 model, see `darknet53`.	Directed acyclic graph (DAG) network based	Yes	Yes	Yes	Yes	Yes	No	Classification
ResNet-18	ResNet-18 convolutional neural network. For the pretrained ResNet-18 model, see `resnet18`.	Directed acyclic graph (DAG) network based	Yes	Yes	Yes	Yes	Yes	Yes	Classification
ResNet-50	ResNet-50 convolutional neural network. For the pretrained ResNet-50 model, see `resnet50`.	Directed acyclic graph (DAG) network based	No. Network exceeds PL DDR memory size.	No. Network exceeds PL DDR memory size.	Yes	Yes	Yes	Yes	Classification

ResNet-based YOLO v2	You only look once (YOLO) is an object detector that decodes the predictions from a convolutional neural network and generates bounding boxes around the objects. For more information, see “Vehicle Detection Using DAG Network Based YOLO v2 Deployed to FPGA” on page 10-122.	Directed acyclic graph (DAG) network based	Yes	Yes	Yes	Yes	Yes	Yes	Object detection

MobileNetV2	MobileNet-v2 convolutional neural network. For the pretrained MobileNet-v2 model, see `mobilenetv2`.	Directed acyclic graph (DAG) network based	Yes	Yes	Yes	Yes	Yes	Yes	Classification
GoogLeNet	GoogLeNet convolutional neural network. For the pretrained GoogLeNet model, see `googlenet`.	Directed acyclic graph (DAG) network based	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the `LRNBlockGeneration` property of the processor configuration for the bitstream and generate the bitstream again.	Classification
PoseNet	Human pose estimation network.	Directed acyclic graph (DAG) network based	Yes.	Yes	Yes	Yes	Yes	Yes	Segmentation

U-Net	U-Net convolutional neural network designed for semantic image segmentation.	Directed acyclic graph (DAG) network based	No. PL DDR memory oversize.	No. PL DDR memory oversize.	No. PL DDR memory oversize.	No. PL DDR memory oversize.	No. PL DDR memory oversize.	Yes	Segmentation
SqueezeNet-based YOLO v3	The you-only-look-once (YOLO) v3 object detector is a multi-scale object detection network that uses a feature extraction network and multiple detection heads to make predictions at multiple scales.	`dlnetwork` object	Yes	Yes	No	No	No	No	Object detection

Sequence-to-sequence classification	Classify each time step of sequence data using a long short-term memory (LSTM) network. See “Run Sequence-to-Sequence Classification on FPGAs by Using Deep Learning HDL Toolbox” on page 10-246.	Long short-term memory (LSTM) network	Yes	Yes	No	No	No	No	Sequence data classification

Time series forecasting	Forecast time series data using a long short-term memory (LSTM) network. See “Run Sequence Forecasting on FPGA by Using Deep Learning HDL Toolbox” on page 10-262	Long short-term memory (LSTM) network	Yes	Yes	No	No	No	No	Forecast time series data
Word-by-word text generation	Generate text word-by-word by using a long short-term memory (LSTM) network. See “Generate Word-By-Word Text on FPGAs by Using Deep Learning HDL Toolbox” on page 10-253.	Long short-term memory (LSTM) network	Yes	Yes	No	No	No	No	Sequence data prediction

YAMNet	Pretrained audio classification network. See `yamnet` and “Deploy YAMNet Networks to FPGAs With and Without Cross-Layer Equalization” on page 10-354.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Audio data classification

Semantic Segmentation Using Dilated Convolutions	Semantic segmentation using dilated convolution layer to increase coverage area without increasing the number of computational parameters. See “Deploy Semantic Segmentation Network Using Dilated Convolutions on FPGA” on page 10-324.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Segmentation

Time series forecasting	Forecast time series data using a long short-term memory (LSTM) network. See “Run Sequence Forecasting Using a GRU Layer on an FPGA” on page 10-333.	Gated recurrent unit (GRU) layer network	Yes	Yes	No	No	No	No	Forecast time series data
Pruned image classification network	Pruned image classification network. See “Deploy Image Recognition Network on FPGA With and Without Pruning” on page 10-379	Series network	Yes	Yes	Yes	Yes	Yes	Yes	Image classification

Very-deep super-resolution (VDSR) network	Create high resolution images from low-resolution images by using VDSR networks. See "Increase Image Resolution Using VDSR Network Running on FPGA" on page 10-363	Series network	Yes	Yes	Yes	Yes	Yes	Yes	Image processing

Supported Layers

Deep Learning HDL Toolbox supports the layers listed in these tables.

Input Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Description and Limitations	INT8 Compatible

imageInputLayer	SW	An image input layer inputs 2-D images to a network and applies data normalization. The normalization options `zero-center` and `zscore` can run on hardware if the `compile` method `HardwareNormalization` argument is enabled and the input data is of `single` data type. If the `HardwareNormalization` option is not enabled or the input data type is `int8` the normalization runs in software. Normalization specified using a function handle is not supported. See "Image Input Layer Normalization Hardware Implementation" on page 7-34. When the `Normalization` property is set to `none` the `activations` function cannot be used for the `imageInputLayer`.	Yes. Runs as single datatype in SW.
featureInputLayer	SW	A feature input layer inputs feature data to a network and applies data normalization.	No
sequenceInputLayer	SW	A sequence input layer inputs sequence data to a network.	No

Convolution and Fully Connected Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible

convolution2dLayer	HW	Convolution (Conv)	A 2-D convolutional layer applies sliding convolutional filters to the input. When generating code for a network using this layer, these limitations apply: • Filter size must be 1-66. • Stride size must be 1-15 and square. • Padding size must be in the range 0-8. • Dilation factor supported up to [16 16] and must be square. • Padding value is not supported. • When the dilation factor is a multiple of three the calculated dilated filter size must have a maximum value of the existing convolution filter size limit. In all other cases, the filter size can be as large as the maximum value of the existing convolution filter size.	Yes

<table>
<tr>
<td>groupedConvolution2dLayer</td>
<td>HW</td>
<td>Convolution (Conv)</td>
<td>A 2-D grouped convolutional layer separates the input channels into groups and applies sliding convolutional filters. Use grouped convolutional layers for channel-wise separable (also known as depth-wise separable) convolution.

Code generation is now supported for a 2-D grouped convolution layer that has the <code>NumGroups</code> property set as <code>'channel-wise'</code>.

When generating code for a network using this layer, these limitations apply:
Filter size must be 1-15 and square. For example [1 1] or [14 14]. When the <code>NumGroups</code> is set as <code>'channel-wise'</code>, filter size must be 3-14.
Stride size must be 1-15 and square.
Padding size must be in the range 0-8.</td>
<td>Yes</td>
</tr>
</table>

			• Dilation factor must be [1 1]. • When the NumGroups is not set as 'channel-wise', number of groups must be 1 or 2. • The input feature number must be greater than a single multiple of the square root of the ConvThreadNumber. • When the NumGroups is not set as 'channel-wise', the number of filters per group must be a multiple of the square root of the ConvThreadNumber.	

transposedConv2dLayer	HW	Convolution (Conv)	A transposed 2-D convolution layer upsamples feature maps. When generating code for a network using this layer, these limitations apply: • Filter size must be 1-8 and square. • Stride size must be 1-66 and square. • Padding size must be in the range 0-8. • Padding value is not supported.	Yes
fullyConnectedLayer	HW	Fully Connected (FC)	A fully connected layer multiplies the input by a weight matrix, and then adds a bias vector. When generating code for a network using this layer, these limitations apply: • The layer input and output size are limited by the values specified in “InputMemorySize” and “OutputMemorySize”.	Yes

Activation Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible

reluLayer	HW	Layer is fused.	A ReLU layer performs a threshold operation to each element of the input where any value less than zero is set to zero. A ReLU layer is supported only when it is preceded by any of these layers: • Convolution • Fully Connected • Adder	Yes
leakyReluLayer	HW	Layer is fused.	A leaky ReLU layer performs a threshold operation where any input value less than zero is multiplied by a fixed scalar. A leaky ReLU layer is supported only when it is preceded by any of these layers: • Convolution • Fully Connected • Adder	Yes

clippedReluLayer	HW	Layer is fused.	A clipped ReLU layer performs a threshold operation where any input value less than zero is set to zero and any value above the clipping ceiling is set to that clipping ceiling value. A clipped ReLU layer is supported only when it is preceded by any of these layers: • Convolution • Fully Connected • Adder	Yes
tanhLayer	HW	Inherit from input	A hyperbolic tangent (tanh) activation layer applies the tanh function on the layer inputs.	No

Normalization, Dropout, and Cropping Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible
batchNormalizationLayer	HW	Layer is fused.	A batch normalization layer normalizes each input channel across a mini-batch. A batch normalization layer is supported when preceded by an image input layer or convolution layer.	Yes

crossChannelNormalizationLayer	HW	Convolution (Conv)	A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization. The `WindowChannelSize` must be in the range of 3-9 for code generation.	Yes. Runs as single datatype in HW.
dropoutLayer	NoOP on inference	NoOP on inference	A dropout layer randomly sets input elements to zero within a given probability.	Yes

resize2dLayer	HW	Inherit from input	A 2-D resize layer resizes 2-D input by a scale factor, to a specified height and width, or to the size of a reference input feature map. When generating code for a network using this layer, these limitations apply: • The `Method` property must be set to `nearest`. • The `GeometricTransformationMode` property must be set to `half-pixel`. • The `NearestRoundingMode` property must be set to `round`. • The ratio of the output size to input size must be an integer and in the range between two and 256.	No

Pooling and Unpooling Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible

`maxPooling2dLayer`	HW	Convolution (Conv)	A max pooling layer performs downsampling by dividing the layer input into rectangular pooling regions and computing the maximum of each region. When generating code for a network using this layer, these limitations apply: • Pool size must be 1-66. • Stride size must be 1-15 and square. • Padding size must be in the range 0-6. `HasUnpoolingOutputs` is supported. When this parameter is enabled, these limitations apply for code generation for this layer: • Pool size must be 2-by-2 or 3-by-3. • The stride size must be the same as the filter size. • Padding size is not supported. • Pool size and stride size must be square. For example, [2 2].	Yes No, when `HasUnpoolingOutputs` is enabled.

maxUnpooling2dLayer	HW	Convolution (Conv)	A max unpooling layer unpools the output of a max pooling layer.	No
averagePooling2dLayer	HW	Convolution (Conv)	An average pooling layer performs downsampling by dividing the layer input into rectangular pooling regions and computing the average values of each region. When generating code for a network using this layer, these limitations apply: • Pool size must be 1-66. • Stride size must be 1-15 and square. • Padding size must be in the range 0-6.	Yes
globalAveragePooling2dLayer	HW	Convolution (Conv)	A global average pooling layer performs downsampling by computing the mean of the height and width dimensions of the input. When generating code for a network using this layer, these limitations apply: • The pool size must be 1-66 and square.	Yes

Combination Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible
`additionLayer`	HW	Inherit from input.	An addition layer adds inputs from multiple neural network layers element-wise. You can now generated code for this layer with `int8` data type when the layer is combined with a Leaky ReLU or Clipped ReLU layer. When generating code for a network using this layer, these limitations apply: • Both input layers must have the same output layer format. For example, both layers must have conv output format or fc output format.	Yes

depthConcatenationLayer	HW	Inherit from input.	A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension). When generating code for a network using this layer, these limitations apply: • The input activation feature number must be a multiple of the square root of the “ConvThreadNumber”. • Layers that have a conv output format and layers that have an FC output format cannot be concatenated together.	Yes
multiplicationLayer	HW	Inherit from input	A multiplication layer multiplies inputs from multiple neural network layers element-wise.	No

Sequence Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Description and Limitations	INT8 Compatible
lstmLayer	HW	An LSTM layer learns long-term dependencies between time steps in time series and sequence data. The layer performs additive interactions, which can help improve gradient flow over long sequences during training. When generating code for a network using this layer, these limitations apply: • The input must be of single data type. • The `OutputMode` property must be set to `sequence`.	No
gruLayer	HW	A GRU layer is an RNN layer that learns dependencies between time steps in time series and sequence data. When generating code for a network using this layer, these limitations apply: • Inputs must be of single data type. • You must set the GRU layer `OutputMode` to `sequence`.	No

Output Layer

Layer	Layer Type Hardware (HW) or Software(SW)	Description and Limitations	INT8 Compatible

softmaxLayer	SW and HW	A softmax layer applies a softmax function to the input. If the softmax layer is implemented in hardware: • The inputs must be in the range -87 to 88 . • Softmax layer followed by adder layer or depth concatenation layer is not supported. • The inputs to this layer must have the format 1-by-N, N-by-1, 1-by-1-by-N, N-by-1-by-1, and 1-by-N-by-1. • If the convolution module of the deep learning processor is enabled the square root of the convolution thread number must be an integral power of two. If not, the layer is implemented in software.	Yes. Runs as single datatype in SW.
classificationLayer	SW	A classification layer computes the cross-entropy loss for multiclass classification issues that have mutually exclusive classes.	Yes
regressionLayer	SW	A regression layer computes the half mean squared error loss for regression problems.	Yes

sigmoidLayer	SW and HW	A sigmoid layer applies a sigmoid function to the input. When the data type is `single` the sigmoid layer is implemented in the custom module of the deep learning processor configuration. When generating code for a network using this layer, with `single` data type these limitations apply: • The inputs must be in the range -87 to 88 . Runs as single datatype in SW.	Yes. When the data type is `int8` the sigmoid layer is implemented in the fully connected (FC) module of the deep learning processor configuration. When generating code for a network using this layer, with `int8` data type these limitations apply: • The inputs must be in the range -87 to 88 . • Sigmoid layer followed by adder layer or depth concatenation layer is not supported. • The inputs to this layer must have the format 1-by-N, N-by-1, 1-by-1-by-N, N-by-1-by-1, and 1-by-N-by-1. • If the convolution module of the deep learning processor is enabled the square root of the convolution thread number must be an integral power of two. If not, the layer is implemented in software.

Keras and ONNX Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible

`nnet.keras.layer.FlattenCStyleLayer`	HW	Layer will be fused	Flatten activations into 1-D layers assuming C-style (row-major) order. A `nnet.keras.layer.FlattenCStyleLayer` is supported only when it is followed by a fully connected layer.	Yes
`nnet.keras.layer.ZeroPadding2dLayer`	HW	Layer will be fused.	Zero padding layer for 2-D input. A `nnet.keras.layer.ZeroPadding2dLayer` is supported only when it is preceded by a convolution layer or a maxpool layer. Zero padding layer is supported when followed by a grouped convolution layer.	Yes

Custom Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible
Custom Layers	HW	Inherit from input	Custom layers, with or without learnable parameters, that you define for your problem. To learn how to define your custom deep learning layers, see "Create Deep Learning Processor Configuration for Custom Layers" on page 8-26 .	No

Supported Boards

These boards are supported by Deep Learning HDL Toolbox:

- Xilinx Zynq®-7000 ZC706
- Intel Arria® 10 SoC
- Xilinx Zynq UltraScale+™ MPSoC ZCU102
- Custom boards. For more information, see "Deep Learning Processor IP Core Generation for Custom Board" on page 12-33.

Third-Party Synthesis Tools and Version Support

Deep Learning HDL Toolbox has been tested with:

- Xilinx Vivado® Design Suite 2022.1
- Intel Quartus® Prime Standard 21.1

Image Input Layer Normalization Hardware Implementation

To enable hardware implementation of the normalization functions for the image input layer, set the `HardwareNormalization` argument of the `compile` method to `auto` or `on`. When `HardwareNormalization` is set to `auto`, the compile method looks for the presence of addition and multiplication layers to implement the normalization function on hardware. The normalization is implemented on hardware by:

- Creating a new constant layer, This layer holds the value which is to be subtracted.
- Using existing addition and multiplication layers. The layers to be used depends on the normalization function being implemented.

Constant Layer Buffer Content

This table describes the value stored in the constant layer buffer.

Normalization Function	Number of Constants	Constant Layer Buffer Value
`zerocenter`	1	`- Mean`
`zscore`	2	The first constant value is `-Mean`. The second constant value is `1/StandardDeviation`

See Also

More About

- "Configure FPGA Boards"

Custom Processor Configuration Workflow

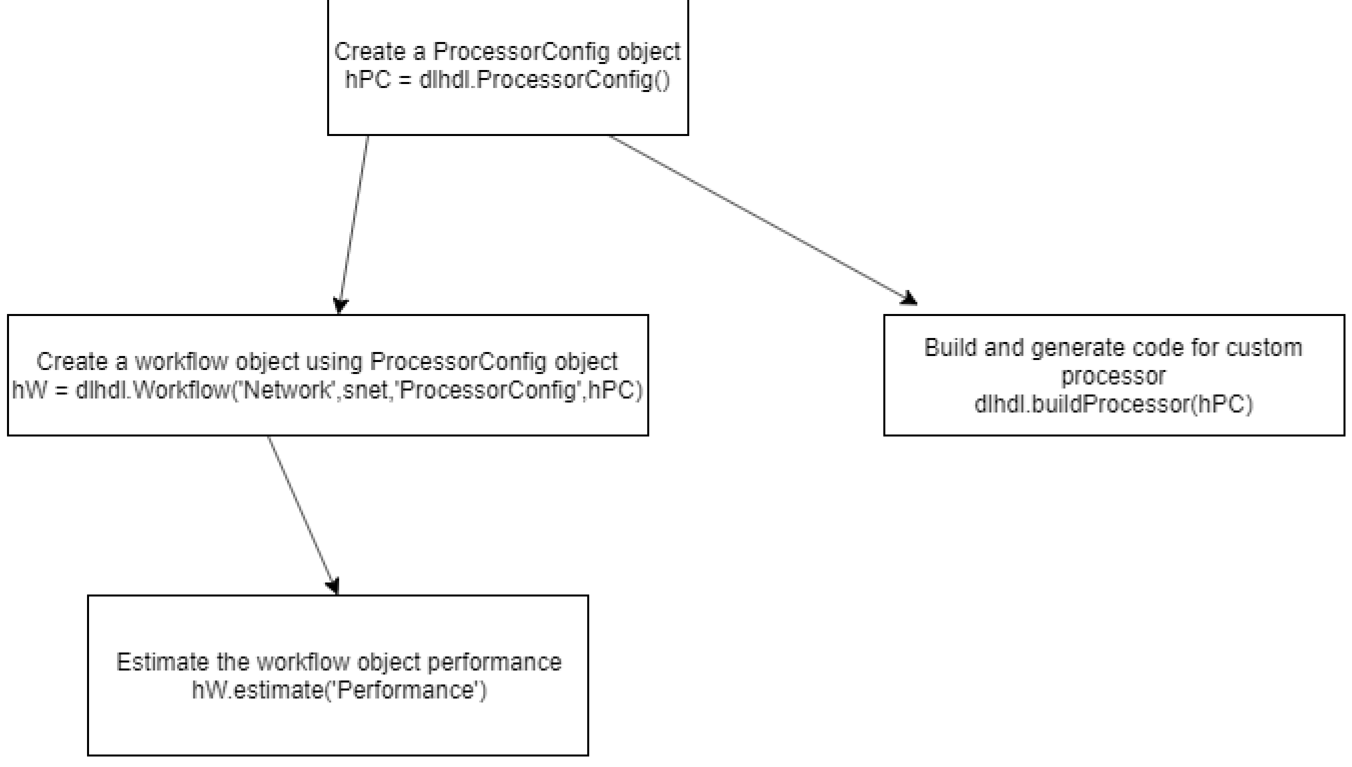
- “Custom Processor Configuration Workflow” on page 8-2
- “Estimate Performance of Deep Learning Network” on page 8-3
- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10
- “Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization” on page 8-17
- “Generate Custom Bitstream to Meet Custom Deep Learning Network Requirements” on page 8-19
- “Create Deep Learning Processor Configuration for Custom Layers” on page 8-26
- “Register, Validate, and Deploy Custom Natural Logarithm Layer Network to FPGA” on page 8-35

Custom Processor Configuration Workflow

Estimate the performance and resource utilization of your custom processor configuration by experimenting with the settings of the deep learning processor convolution and fully connected modules. For more information about the deep learning processor, see “Deep Learning Processor IP Core Architecture” on page 2-2. For information about the convolution and fully connected module parameters, see “Properties”.

After configuring your custom deep learning processor you can build and generate a custom bitstream and custom deep learning processor IP core. For more information about the custom deep learning processor IP core, see “Deep Learning Processor IP Core” on page 12-5.

The image shows the workflow to customize your deep learning processor, estimate the custom deep learning processor performance and resource utilization, and build and generate your custom deep learning processor IP core and bitstream.



See Also

`dlhdl.ProcessorConfig` | `getModuleProperty` | `setModuleProperty` | `estimatePerformance` | `estimateResources`

More About

- “Deep Learning Processor IP Core Architecture” on page 2-2
- “Estimate Performance of Deep Learning Network” on page 8-3
- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10

Estimate Performance of Deep Learning Network

To reduce the time required to design a custom deep learning network that meets performance requirements, before deploying the network, analyze layer level latencies. Compare deep learning network performances on custom bitstream processor configurations to performances on reference (shipping) bitstream processor configurations.

To learn how to use the information in the table data from the `estimatePerformance` function to calculate your network performance, see “Profile Inference Run” on page 5-4.

Estimate Performance of Custom Deep Learning Network for Custom Processor Configuration

This example shows how to calculate the performance of a deep learning network for a custom processor configuration.

1. Create a file in your current working folder called `getLogoNetwork.m`. In the file, enter:

```
function net = getLogoNetwork()
 if ~isfile('LogoNet.mat')
        url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/logo_detection/LogoNet.mat';
        websave('LogoNet.mat',url);
    end
    data = load('LogoNet.mat');
    net  = data.convnet;
end
```

   Call the function and save the result in `snet`.

```
snet = getLogoNetwork;
```

2. Create a `dlhdl.ProcessorConfig` object.

```
hPC = dlhdl.ProcessorConfig;
```

3. Call `estimatePerformance` with `snet` to retrieve the layer level latencies and performance for the LogoNet network.

```
hPC.estimatePerformance(snet)
```

```
3 Memory Regions created.


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Total Latency     Frame
                         -------------             -------------              ---------        ---------       -----
Network                    39853460                  0.19927                      1          39853460
        conv_1              6825287                  0.03413
        maxpool_1           3755088                  0.01878
        conv_2             10440701                  0.05220
        maxpool_2           1447840                  0.00724
        conv_3              9393397                  0.04697
        maxpool_3           1765856                  0.00883
        conv_4              1770484                  0.00885
        maxpool_4             28098                  0.00014
        fc_1                2644884                  0.01322
        fc_2                1692532                  0.00846
        fc_3                  89293                  0.00045
 * The clock frequency of the DL processor is: 200MHz
```

   To learn about the parameters and values returned by `estimatePerformance`, see .

Evaluate Performance of Deep Learning Network on Custom Processor Configuration

Benchmark the performance of a deep learning network on a custom bitstream configuration by comparing it to the performance on a reference (shipping) bitstream configuration. Use the comparison results to adjust your custom deep learning processor parameters to achieve optimum performance.

In this example compare the performance of the ResNet-18 network on the `zcu102_single` bitstream configuration to the performance on the default custom bitstream configuration.

Prerequisites

- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model for ResNet-18 Network

Load Pretrained Network

Load the pretrained network.

```
snet = resnet18;
```

Retrieve `zcu102_single` Bitstream Configuration

To retrieve the `zcu102_single` bitstream configuration, use the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

```
hPC_shipping = dlhdl.ProcessorConfig('Bitstream',"zcu102_single")
```

```
hPC_shipping = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                   Processing Module "adder"
                            ModuleGeneration: 'on'
                             InputMemorySize: 40
                            OutputMemorySize: 40

              Processor Top Level Properties
```

```
                     RunTimeControl: 'register'
                 InputDataInterface: 'External Memory'
                OutputDataInterface: 'External Memory'
                  ProcessorDataType: 'single'

            System Level Properties
                     TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                    TargetFrequency: 220
                      SynthesisTool: 'Xilinx Vivado'
                    ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
            SynthesisToolChipFamily: 'Zynq UltraScale+'
            SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
           SynthesisToolPackageName: ''
            SynthesisToolSpeedValue: ''
```

Estimate ResNet-18 Performance for `zcu102_single` Bitstream Configuration

To `estimate` the performance of the ResNet-18 DAG network, use the `estimatePerformance` function of the `dlhdl.ProcessorConfig` object. The function returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

```
hPC_shipping.estimatePerformance(snet)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    23634966                  0.10743                       1          236
    ____conv1               2165372                  0.00984
    ____pool1                646226                  0.00294
    ____res2a_branch2a       966221                  0.00439
    ____res2a_branch2b       966221                  0.00439
    ____res2a                210750                  0.00096
    ____res2b_branch2a       966221                  0.00439
    ____res2b_branch2b       966221                  0.00439
    ____res2b                210750                  0.00096
    ____res3a_branch1        540749                  0.00246
    ____res3a_branch2a       763860                  0.00347
    ____res3a_branch2b       919117                  0.00418
    ____res3a                105404                  0.00048
    ____res3b_branch2a       919117                  0.00418
    ____res3b_branch2b       919117                  0.00418
    ____res3b                105404                  0.00048
    ____res4a_branch1        509261                  0.00231
    ____res4a_branch2a       509261                  0.00231
    ____res4a_branch2b       905421                  0.00412
    ____res4a                 52724                  0.00024
    ____res4b_branch2a       905421                  0.00412
    ____res4b_branch2b       905421                  0.00412
    ____res4b                 52724                  0.00024
    ____res5a_branch1       1046605                  0.00476
    ____res5a_branch2a      1046605                  0.00476
```

```
    ____res5a_branch2b     2005197                  0.00911
    ____res5a                26368                  0.00012
    ____res5b_branch2a     2005197                  0.00911
    ____res5b_branch2b     2005197                  0.00911
    ____res5b                26368                  0.00012
    ____pool5                54594                  0.00025
    ____fc1000              207852                  0.00094
 * The clock frequency of the DL processor is: 220MHz
```

Create Custom Processor Configuration

To create a custom processor configuration, use the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

```
hPC_custom = dlhdl.ProcessorConfig
```

```
hPC_custom =
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                   Processing Module "adder"
                            ModuleGeneration: 'on'
                             InputMemorySize: 40
                            OutputMemorySize: 40

              Processor Top Level Properties
                              RunTimeControl: 'register'
                          InputDataInterface: 'External Memory'
                         OutputDataInterface: 'External Memory'
                           ProcessorDataType: 'single'

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                             TargetFrequency: 200
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

Estimate ResNet-18 Performance for Custom Bitstream Configuration

To `estimate` the performance of the ResNet-18 DAG network, use the `estimatePerformance` function of the `dlhdl.ProcessorConfig` object. The function returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

```
hPC_custom.estimatePerformance(snet)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   21219873                  0.10610                       1          212
    ____conv1              2165372                  0.01083
    ____pool1               646226                  0.00323
    ____res2a_branch2a      966221                  0.00483
    ____res2a_branch2b      966221                  0.00483
    ____res2a               210750                  0.00105
    ____res2b_branch2a      966221                  0.00483
    ____res2b_branch2b      966221                  0.00483
    ____res2b               210750                  0.00105
    ____res3a_branch1       540749                  0.00270
    ____res3a_branch2a      708564                  0.00354
    ____res3a_branch2b      919117                  0.00460
    ____res3a               105404                  0.00053
    ____res3b_branch2a      919117                  0.00460
    ____res3b_branch2b      919117                  0.00460
    ____res3b               105404                  0.00053
    ____res4a_branch1       509261                  0.00255
    ____res4a_branch2a      509261                  0.00255
    ____res4a_branch2b      905421                  0.00453
    ____res4a                52724                  0.00026
    ____res4b_branch2a      905421                  0.00453
    ____res4b_branch2b      905421                  0.00453
    ____res4b                52724                  0.00026
    ____res5a_branch1       751693                  0.00376
    ____res5a_branch2a      751693                  0.00376
    ____res5a_branch2b     1415373                  0.00708
    ____res5a                26368                  0.00013
    ____res5b_branch2a     1415373                  0.00708
    ____res5b_branch2b     1415373                  0.00708
    ____res5b                26368                  0.00013
    ____pool5                54594                  0.00027
    ____fc1000              207351                  0.00104
 * The clock frequency of the DL processor is: 200MHz
```

The performance of the ResNet-18 network on the custom bitstream configuration is lower than the performance on the `zcu102_single` bitstream configuration. The difference between the custom bitstream configuration and the `zcu102_single` bitstream configuration is the target frequency.

Modify Custom Processor Configuration

Modify the custom processor configuration to increase the target frequency. To learn about modifiable parameters of the processor configuration, see `dlhdl.ProcessorConfig`.

```
hPC_custom.TargetFrequency = 220;
hPC_custom
```

```
hPC_custom = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                   Processing Module "adder"
                            ModuleGeneration: 'on'
                             InputMemorySize: 40
                            OutputMemorySize: 40

              Processor Top Level Properties
                              RunTimeControl: 'register'
                          InputDataInterface: 'External Memory'
                         OutputDataInterface: 'External Memory'
                           ProcessorDataType: 'single'

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                             TargetFrequency: 220
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

Re-estimate ResNet-18 Performance for Modified Custom Bitstream Configuration

Estimate the performance of the ResNet-18 DAG network on the modified custom bitstream configuration.

```
hPC_custom.estimatePerformance(snet)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification
```

```
              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   23634966                  0.10743                       1          236
    ____conv1              2165372                  0.00984
    ____pool1               646226                  0.00294
    ____res2a_branch2a      966221                  0.00439
    ____res2a_branch2b      966221                  0.00439
    ____res2a               210750                  0.00096
    ____res2b_branch2a      966221                  0.00439
    ____res2b_branch2b      966221                  0.00439
    ____res2b               210750                  0.00096
    ____res3a_branch1       540749                  0.00246
    ____res3a_branch2a      763860                  0.00347
    ____res3a_branch2b      919117                  0.00418
    ____res3a               105404                  0.00048
    ____res3b_branch2a      919117                  0.00418
    ____res3b_branch2b      919117                  0.00418
    ____res3b               105404                  0.00048
    ____res4a_branch1       509261                  0.00231
    ____res4a_branch2a      509261                  0.00231
    ____res4a_branch2b      905421                  0.00412
    ____res4a                52724                  0.00024
    ____res4b_branch2a      905421                  0.00412
    ____res4b_branch2b      905421                  0.00412
    ____res4b                52724                  0.00024
    ____res5a_branch1      1046605                  0.00476
    ____res5a_branch2a     1046605                  0.00476
    ____res5a_branch2b     2005197                  0.00911
    ____res5a                26368                  0.00012
    ____res5b_branch2a     2005197                  0.00911
    ____res5b_branch2b     2005197                  0.00911
    ____res5b                26368                  0.00012
    ____pool5                54594                  0.00025
    ____fc1000              207852                  0.00094
 * The clock frequency of the DL processor is: 220MHz
```

See Also

dlhdl.ProcessorConfig | getModuleProperty | setModuleProperty | estimatePerformance | estimateResources

More About

- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10
- “Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization” on page 8-17

Estimate Resource Utilization for Custom Processor Configuration

To estimate the resource utilization of a custom processor configuration, compare resource utilization for a custom processor configuration to resource utilization of a reference (shipping) bitstream processor configuration. Analyze the effects of custom deep learning processor parameters on resource utilization.

Estimate Resource Utilization

Calculate resource utilization for a custom processor configuration.

1. Create a `dlhdl.ProcessorConfig` object.

```
hPC = dlhdl.ProcessorConfig
```

```
hPC =

                   Processing Module "conv"
                           ModuleGeneration: 'on'
                         LRNBlockGeneration: 'on'
                           ConvThreadNumber: 16
                            InputMemorySize: [227 227 3]
                           OutputMemorySize: [227 227 3]
                           FeatureSizeLimit: 2048

                     Processing Module "fc"
                           ModuleGeneration: 'on'
                     SoftmaxBlockGeneration: 'off'
                             FCThreadNumber: 4
                            InputMemorySize: 25088
                           OutputMemorySize: 4096

                  Processing Module "adder"
                           ModuleGeneration: 'on'
                            InputMemorySize: 40
                           OutputMemorySize: 40

             Processor Top Level Properties
                             RunTimeControl: 'register'
                         InputDataInterface: 'External Memory'
                        OutputDataInterface: 'External Memory'
                          ProcessorDataType: 'single'

                    System Level Properties
                             TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit'
                            TargetFrequency: 200
                              SynthesisTool: 'Xilinx Vivado'
                            ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                    SynthesisToolChipFamily: 'Zynq UltraScale+'
                    SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                   SynthesisToolPackageName: ''
                    SynthesisToolSpeedValue: ''
```

2. Call `estimateResources` to retrieve resource utilization.

```
hPC.estimateResources
```

```
             Deep Learning Processor Estimator Resource Results

                               DSPs          Block RAM*     LUTs(CLB/ALUT)
                          -------------    -------------    -------------
Available                      2520             912            274080
                          -------------    -------------    -------------
DL_Processor                 377( 15%)        508( 56%)     234175( 86%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

The returned table contains resource utilization for the entire processor and individual modules.

Customize Bitstream Configuration to Meet Resource Use Requirements

This example shows how to deploy a digit recognition network with a target performance of 500 frames per second (FPS) to a Xilinx™ ZCU102 ZU4CG device. The target device resource counts are:

- Digital signal processor (DSP) slice count — 240
- Block random access memory (BRAM) count — 128

The reference `zcu102_int8` bitstream configuration is for a Xilinx ZCU102 ZU9EG device. The default board resource counts are:

- Digital signal processor (DSP) slice count — 2520
- Block random access memory (BRAM) count — 912

The default board resource counts exceed the resource budget and are on the higher end of the cost spectrum. In this example, you can achieve target performance and resource use budget by quantizing the target deep learning network and customizing the bitstream configuration.

Prerequisites

- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library

Load Pretrained Network

To load the pretrained series network, that has been trained on the Modified National Institute Standards of Technology (MNIST) database, enter:

```
snet = getDigitsNetwork;
```

Quantize Network

To quantize the MNIST based digits network, enter:

```
dlquantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
Image = imageDatastore('five_28x28.pgm','Labels','five');
calibrate(dlquantObj,Image);
```

Retrieve `zcu102_int` Bitstream Configuration

To retrieve the `zcu102_int8` bitstream configuration, use the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

```
referencehPC = dlhdl.ProcessorConfig('Bitstream','zcu102_int8')
```

```
referencehPC =
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
```

```
                 ConvThreadNumber: 64
                  InputMemorySize: [227 227 3]
                 OutputMemorySize: [227 227 3]
                 FeatureSizeLimit: 2048

           Processing Module "fc"
                 ModuleGeneration: 'on'
           SoftmaxBlockGeneration: 'off'
           SigmoidBlockGeneration: 'off'
                   FCThreadNumber: 16
                  InputMemorySize: 25088
                 OutputMemorySize: 4096

       Processing Module "custom"
                 ModuleGeneration: 'on'
                         Addition: 'on'
                   Multiplication: 'on'
                         Resize2D: 'off'
                          Sigmoid: 'off'
                        TanhLayer: 'off'
                  InputMemorySize: 40
                 OutputMemorySize: 120

   Processor Top Level Properties
                   RunTimeControl: 'register'
                    RunTimeStatus: 'register'
               InputStreamControl: 'register'
              OutputStreamControl: 'register'
                     SetupControl: 'register'
                ProcessorDataType: 'int8'

          System Level Properties
                   TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                  TargetFrequency: 250
                    SynthesisTool: 'Xilinx Vivado'
                  ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
          SynthesisToolChipFamily: 'Zynq UltraScale+'
          SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
         SynthesisToolPackageName: ''
          SynthesisToolSpeedValue: ''
```

Estimate Network Performance and Resource Utilization for `zcu102_int8` Bitstream Configuration

To estimate the performance of the digits series network, use the `estimatePerformance` method of the `dlhdl.ProcessorConfig` object. The method returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

```
estimatePerformance(referencehPC,dlquantObj)
```

```
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convo
### The network includes the following layers:
     1   'imageinput'    Image Input             28×28×1 images with 'zerocenter' normalization
     2   'conv_1'        2-D Convolution         8 3×3×1 convolutions with stride [1  1] and padd
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
     5   'conv_2'        2-D Convolution         16 3×3×8 convolutions with stride [1  1] and pad
```

```
     6   'relu_2'        ReLU                     ReLU
     7   'maxpool_2'     2-D Max Pooling          2×2 max pooling with stride [2  2] and padding
     8   'conv_3'        2-D Convolution          32 3×3×16 convolutions with stride [1  1] and pa
     9   'relu_3'        ReLU                     ReLU
    10   'fc'            Fully Connected          10 fully connected layer
    11   'softmax'       Softmax                  softmax
    12   'classoutput'   Classification Output    crossentropyex with '0' and 9 other classes

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------            -------------               ---------        --
Network                     58101                  0.00023                       1
    conv_1                   4391                  0.00002
    maxpool_1                2877                  0.00001
    conv_2                   2351                  0.00001
    maxpool_2                2265                  0.00001
    conv_3                   2651                  0.00001
    fc                      43566                  0.00017
 * The clock frequency of the DL processor is: 250MHz
```

To estimate the resource use of the `zcu102_int8` bitstream, use the `estimateResources` method of the `dlhdl.ProcessorConfig` object. The method returns the estimated DSP slice and BRAM usage.

```
estimateResources(referencehPC)
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                   2520             912            274080
                        -------------    -------------    -------------
DL_Processor              805( 32%)        386( 43%)     142494( 52%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

The estimated performance is 4303 FPS and the estimated resource use counts are:

- Digital signal processor (DSP) slice count - 797
- Block random access memory (BRAM) count -386

The estimated DSP slice count and BRAM count use exceeds the target device resource budget. Customize the bitstream configuration to reduce resource use.

Create Custom Bitstream Configuration

To create a custom processor configuration, use `dlhdl.ProcessorConfig` class. To learn about the modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

To reduce the resource use for the custom bitstream, modify the `KernelDataType property` for the `conv, fc, and` adder modules. Modify the `ConvThreadNumber` property to reduce DSP slice

count. Reduce the InputMemorySize and OutputMemorySize properties for the conv module to reduce the BRAM count.

```
customhPC = dlhdl.ProcessorConfig;
customhPC.ProcessorDataType = 'int8';
customhPC.setModuleProperty('conv','ConvThreadNumber',4);
customhPC.setModuleProperty('conv','InputMemorySize',[30 30 1]);
customhPC.setModuleProperty('conv','OutputMemorySize',[30 30 1]);
customhPC
```

```
customhPC =
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                            ConvThreadNumber: 4
                             InputMemorySize: [30 30 1]
                            OutputMemorySize: [30 30 1]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                      SigmoidBlockGeneration: 'off'
                               FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                  Processing Module "custom"
                            ModuleGeneration: 'on'
                                    Addition: 'on'
                              Multiplication: 'on'
                                    Resize2D: 'off'
                                     Sigmoid: 'off'
                                   TanhLayer: 'off'
                             InputMemorySize: 40
                            OutputMemorySize: 120

              Processor Top Level Properties
                              RunTimeControl: 'register'
                               RunTimeStatus: 'register'
                          InputStreamControl: 'register'
                         OutputStreamControl: 'register'
                                SetupControl: 'register'
                           ProcessorDataType: 'int8'

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation K
                             TargetFrequency: 200
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

Estimate Network Performance and Resource Utilization for Custom Bitstream Configuration

Estimate the performance of the digits series network for the custom bitstream.

```
estimatePerformance(customhPC,dlquantObj)
```

```
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convo
### The network includes the following layers:
     1   'imageinput'    Image Input             28×28×1 images with 'zerocenter' normalization
     2   'conv_1'        2-D Convolution         8 3×3×1 convolutions with stride [1  1] and padd
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
     5   'conv_2'        2-D Convolution         16 3×3×8 convolutions with stride [1  1] and pad
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
     8   'conv_3'        2-D Convolution         32 3×3×16 convolutions with stride [1  1] and pa
     9   'relu_3'        ReLU                    ReLU
    10   'fc'            Fully Connected         10 fully connected layer
    11   'softmax'       Softmax                 softmax
    12   'classoutput'   Classification Output   crossentropyex with '0' and 9 other classes

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------            -------------              ---------       --
Network                     433577                  0.00217                       1           4
    conv_1                   26160                  0.00013
    maxpool_1                31888                  0.00016
    conv_2                   44736                  0.00022
    maxpool_2                22337                  0.00011
    conv_3                  265045                  0.00133
    fc                       43411                  0.00022
 * The clock frequency of the DL processor is: 200MHz
```

Estimate the resource use of the custom bitstream.

```
estimateResources(customhPC)
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912            274080
                        -------------    -------------    -------------
DL_Processor                139(  6%)        108( 12%)      56270( 21%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

The estimated performance is 461.3 FPS and the estimated resource use counts are:

- Digital signal processor (DSP) slice count - 131
- Block random access memory (BRAM) count -108

The estimated resources of the customized bitstream match the user target device resource budget and the estimated performance matches the target network performance.

See Also

`dlhdl.ProcessorConfig` | `getModuleProperty` | `setModuleProperty` | `estimatePerformance` | `estimateResources`

More About

- "Estimate Performance of Deep Learning Network" on page 8-3
- "Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization" on page 8-17

Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization

Analyze how deep learning processor parameters affect deep learning network performance and bitstream resource utilization. Identify parameters that help improve performance and reduce resource utilization.

This table lists the deep learning processor parameters and their effects on performance and resource utilization.

Deep Learning Processor Parameter	Deep Learning Processor Module	Parameter Action	Effect on Performance	Effect on Resource Utilization
"TargetFrequency"	Base module	Increase target frequency.	Improves performance.	Marginal increase in lookup table (LUT) utilization.
"ConvThreadNumber"	`conv`	Increase thread number.	Improves performance.	Increases resource utilization.
"InputMemorySize"	`conv`	Increase input memory size.	Improves performance.	Increases block RAM (BRAM) resource utilization.
"OutputMemorySize"	`conv`	Increase output memory size.	Improves performance.	Increases block RAM (BRAM) resource utilization.
"FeatureSizeLimit"	`conv`	Increase feature size limit.	Improves performance on networks with layers that have a large number of features.	Increases block RAM (BRAM) resource utilization.
"FCThreadNumber"	`fc`	Increase thread number.	Improves performance.	Increases resource utilization.
"InputMemorySize"	`fc`	Increase input memory size.	Improves performance.	Increases Block RAM (BRAM) resource utilization.
"OutputMemorySize"	`fc`	Increase output memory size.	Improves performance.	Increases Block RAM (BRAM) resource utilization.
"InputMemorySize"	`custom`	Increase input memory size	Improves performance for DAG networks only	Increases resource utilization for DAG networks only.

“OutputMemorySize”	`custom`	Increase output memory size	Improves performance for DAG networks only	Increases resource utilization for DAG networks only.
“ProcessorDataType”	`Top Level`	Change data type to int8.	Improves performance. There could be a drop in accuracy.	Reduces resource utilization.

See Also

`dlhdl.ProcessorConfig` | `getModuleProperty` | `setModuleProperty` | `estimatePerformance` | `estimateResources`

More About

- “Estimate Performance of Deep Learning Network” on page 8-3
- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10

Generate Custom Bitstream to Meet Custom Deep Learning Network Requirements

Deploy your custom network that only has layers with the convolution module output format or only layers with the fully connected module output format by generating a resource optimized custom bitstream that satisfies your performance and resource requirements. Bitstream generated using the default deep learning processor configuration consists of the convolution (conv), fully connected (fc), and adder modules. The generated default bitstreams could exceed your resource utilization requirements which could drive up costs. To generate a bitstream that consists of only the layers in your custom deep learning network, modify the deep learning processor configuration by using the `setModuleProperty` function of the `dlhdl.ProcessorConfig` object.

In this example, you have a network that has only layers that have the fully connected module output format. Generate a custom bitstream that consists of the fully connected module only by removing the convolution and adder modules from the deep learning processor configuration. To remove the convolution and adder modules:

- Turn off the `ModuleGeneration` property for the individual modules in the deep learning processor configuration.
- Use the `optimizeConfigurationForNetwork` function. The function takes the deep learning network object as the input and returns an optimized custom deep learning processor configuration.
- Rapidly verify the resource utilization of the optimized deep learning processor configuration by using the `estimateResources` function.

Prerequisites

- Deep Learning HDL Toolbox™ Support Package for Xilinx™ FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Create Custom Processor Configuration

Create a custom processor configuration. Save the configuration to `hPC`.

```
hPC = dlhdl.ProcessorConfig
```

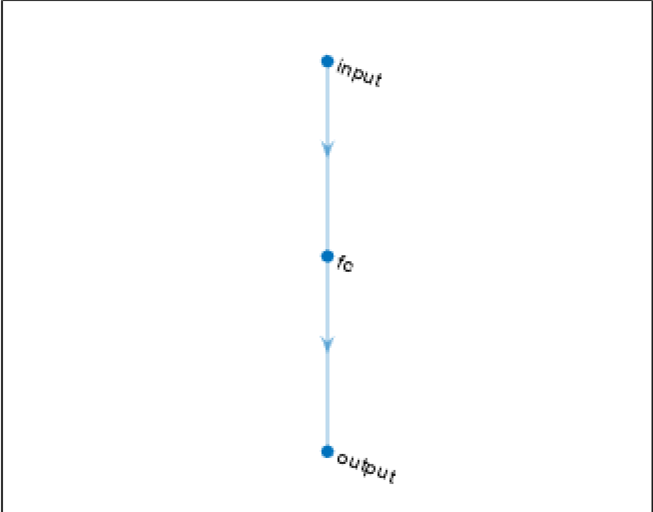
```
hPC = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                      SigmoidBlockGeneration: 'off'
                              FCThreadNumber: 4
```

```
                     InputMemorySize: 25088
                    OutputMemorySize: 4096

          Processing Module "custom"
                    ModuleGeneration: 'on'
                            Addition: 'on'
                      Multiplication: 'on'
                            Resize2D: 'off'
                             Sigmoid: 'off'
                           TanhLayer: 'off'
                     InputMemorySize: 40
                    OutputMemorySize: 120

      Processor Top Level Properties
                      RunTimeControl: 'register'
                       RunTimeStatus: 'register'
                  InputStreamControl: 'register'
                 OutputStreamControl: 'register'
                        SetupControl: 'register'
                   ProcessorDataType: 'single'

             System Level Properties
                      TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                     TargetFrequency: 200
                       SynthesisTool: 'Xilinx Vivado'
                     ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
             SynthesisToolChipFamily: 'Zynq UltraScale+'
             SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
            SynthesisToolPackageName: ''
             SynthesisToolSpeedValue: ''
```

Optimize Processor Configuration for a Custom Fully Connected (FC) Layer only Network

To optimize your processor configuration, create a custom fully connected layer only network. Call the custom network `fcnet`.

```
layers = [ ...
    imageInputLayer([28 28 3],'Normalization','none','Name','input')
    fullyConnectedLayer(10,'Name','fc')
    regressionLayer('Name','output')];
layers(2).Weights = rand(10,28*28*3);
layers(2).Bias = rand(10,1);
fcnet = assembleNetwork(layers);
plot(fcnet);
```



Retrieve the resource utilization for the default custom processor configuration by using `estimateResources`. Retrieve the performance for the custom network `fcnet` by using `estimatePerformance`.

```
hPC.estimateResources
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520              912            274080
                        -------------    -------------    -------------
DL_Processor                381( 16%)        508( 56%)     216119( 79%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

```
hPC.estimatePerformance(fcnet)
```

```
### The network includes the following layers:
     1   'input'    Image Input         28×28×3 images             (SW Layer)
     2   'fc'       Fully Connected     10 fully connected layer  (HW Layer)
     3   'output'   Regression Output   mean-squared-error        (SW Layer)

### Notice: The layer 'input' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------           -------------              ---------        --
Network                     137574                  0.00069                     1            1
    ____fc                  137574                  0.00069
 * The clock frequency of the DL processor is: 200MHz
```

The target device resource counts are:

- Digital signal processor (DSP) slice count — 240
- Block random access memory (BRAM) count — 128

The estimated performance is 1454 frames per second (FPS). The estimated resource use counts are:

- Digital signal processor (DSP) slice count — 381
- Block random access memory (BRAM) count — 508

The estimated DSP slice count and BRAM count use exceeds the target device resource budget. Customize the bitstream configuration to reduce resource use by customizing the processor configuration.

Customize Processor Configuration by Using ModuleGeneration Property

Create a deep learning network processor configuration object. Save it to `hPC_moduleoff`. Turn off the convolution and adder modules in the custom deep learning processor configuration.

```
hPC_moduleoff = dlhdl.ProcessorConfig;
hPC_moduleoff.setModuleProperty('conv','ModuleGeneration','off');
hPC_moduleoff.setModuleProperty('adder','ModuleGeneration','off');
```

Retrieve the resource utilization for the default custom processor configuration by using `estimateResources`. Retrieve the performance for the custom network `fcnet` by using `estimatePerformance`.

```
hPC_moduleoff.estimateResources
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912           274080
                        -------------    -------------    -------------
DL_Processor                   17(  1%)         44(  5%)      25760( 10%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

```
hPC_moduleoff.estimatePerformance(fcnet)
```

```
### The network includes the following layers:
     1   'input'    Image Input         28×28×3 images            (SW Layer)
     2   'fc'       Fully Connected     10 fully connected layer  (HW Layer)
     3   'output'   Regression Output   mean-squared-error        (SW Layer)

### Notice: The layer 'input' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    137574                  0.00069                       1            1
    ____fc                 137574                  0.00069
 * The clock frequency of the DL processor is: 200MHz
```

The target device resource counts are:

- Digital signal processor (DSP) slice count — 240
- Block random access memory (BRAM) count — 128

The estimated performance is 1454 frames per second (FPS). The estimated resource use counts are:

- Digital signal processor (DSP) slice count — 17
- Block random access memory (BRAM) count — 44

The estimated resources of the customized bitstream match the user target device resource budget. The estimated performance matches the target network performance.

Customize Processor Configuration by Using `optimizeConfigurationForNetwork`

Create a deep learning network processor configuration object. Save it to `hPC_optimized`. Generate an optimized deep learning processor configuration by using the `optimizeConfigurationForNetwork` function.

```
hPC_optimized = dlhdl.ProcessorConfig;
hPC_optimized.optimizeConfigurationForNetwork(fcnet);
```

```
### Optimizing processor configuration for deep learning network begin.
### Note: Processing module "conv" property "ModuleGeneration" changed from "true" to "false".
### Note: Processing module "fc" property "InputMemorySize" changed from "25088" to "2352".
### Note: Processing module "fc" property "OutputMemorySize" changed from "4096" to "128".
### Note: Processing module "custom" property "ModuleGeneration" changed from "true" to "false".

                    Processing Module "conv"
                            ModuleGeneration: 'off'

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                      SigmoidBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 2352
                            OutputMemorySize: 128

                  Processing Module "custom"
                            ModuleGeneration: 'off'

              Processor Top Level Properties
                              RunTimeControl: 'register'
                               RunTimeStatus: 'register'
                          InputStreamControl: 'register'
                         OutputStreamControl: 'register'
                                SetupControl: 'register'
                           ProcessorDataType: 'single'

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation K
                             TargetFrequency: 200
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

```
### Optimizing processor configuration for deep learning network complete.
```

Retrieve the resource utilization for the default custom processor configuration by using `estimateResources.` Retrieve the performance for the custom network `fcnet` by using `estimatePerformance`.

```
hPC_optimized.estimateResources
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912           274080
                        -------------    -------------    -------------
DL_Processor                 17(  1%)         20(  3%)      25760( 10%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

```
hPC_optimized.estimatePerformance(fcnet)
```

```
### The network includes the following layers:
     1   'input'    Image Input         28×28×3 images            (SW Layer)
     2   'fc'       Fully Connected     10 fully connected layer  (HW Layer)
     3   'output'   Regression Output   mean-squared-error        (SW Layer)

### Notice: The layer 'input' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------             ---------        --
Network                    137574                  0.00069                        1           1
    ____fc                 137574                  0.00069
 * The clock frequency of the DL processor is: 200MHz
```

The target device resource counts are:

- Digital signal processor (DSP) slice count — 240
- Block random access memory (BRAM) count — 128

The estimated performance is 1454 frames per second (FPS). The estimated resource use counts are:

- Digital signal processor (DSP) slice count — 17
- Block random access memory (BRAM) count — 20

The estimated resources of the customized bitstream match the user target device resource budget. The estimated performance matches the target network performance.

Generate Custom Bitstream

Generate a custom bitstream using the processor configuration that matches your performance and resource requirements.

To deploy `fcnet` using the bitstream generated by using the `ModuleOff` property, uncomment this line of code:

```
%   dlhdl.buildProcessor(hPC_moduleoff)
```

To deploy `fcnet` using the bitstream generated by using the `optimizeConfigurationForNetwork` function, uncomment this line of code:

```
%   dlhdl.buildProcessor(hPC_optimized)
```

See Also

`dlhdl.ProcessorConfig` | `getModuleProperty` | `setModuleProperty` | `estimatePerformance` | `estimateResources`

More About

- “Estimate Performance of Deep Learning Network” on page 8-3
- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10
- “Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization” on page 8-17

Create Deep Learning Processor Configuration for Custom Layers

Deep learning networks use custom layers to perform actions such as resizing 2-D inputs by a scale factor, performing element-wise multiplications, and so on. If your network requires layers to perform certain actions and the layers are not provided by Deep Learning Toolbox™, create a custom layer. Rapidly prototype, validate and deploy your networks that have custom layers by:

- Creating and registering your custom layer function and Simulink® model.
- Validating your custom layer
- Generating a custom bitstream

Deploy the network that has custom layers to a target board by using the custom bitstream

Deploy Custom Layer Networks

1. Create a custom processor configuration object by using the `dlhdl.ProcessorConfig` object.
2. For layers that use a custom function, create a MATLAB function and Simulink model that replicates your custom layer function.
3. Register your custom layer function and Simulink model by using the `registerCustomLayer` method.
4. Enable the registered custom layers in your custom deep learning processor configuration.
5. Simulate and verify your custom layer by using a generated verification model. Generate a verification model by using the `openCustomLayerModel` method. Verify the custom layer by using the `verifyCustomLayerModel` method. This step is optional.
6. Generate a custom bitstream by using the `dlhdl.buildProcessor` function.
7. Create a workflow object that has your custom layer network and custom bitstream by using the `dlhdl.Workflow` object.
8. Compile and deploy the workflow object by using the `compile` and `deploy` methods.
9. Retrieve the prediction results from the deployed network by using the `predict` method.

Tip If you are creating a layer with multiple inputs, then you must set the `NumInputs` properties in the layer constructor.

Retrieve the prediction results from the deployed network by using MATLAB.

Create a Deep learning Processor Configuration

To generate a custom processor configuration, use the `dlhdl.ProcessorConfig` object. The generated deep learning processor configuration object has a `custom` module that contains the preconfigured custom layers. Save the deep learning processor configuration to a variable named `hPC`.

```
hPC = dlhdl.ProcessorConfig
```

```
hPC =
```

```
              Processing Module "conv"
                    ModuleGeneration: 'on'
                  LRNBlockGeneration: 'off'
         SegmentationBlockGeneration: 'on'
                    ConvThreadNumber: 16
                     InputMemorySize: [227 227 3]
                    OutputMemorySize: [227 227 3]
                    FeatureSizeLimit: 2048

                Processing Module "fc"
                    ModuleGeneration: 'on'
              SoftmaxBlockGeneration: 'off'
              SigmoidBlockGeneration: 'off'
                      FCThreadNumber: 4
                     InputMemorySize: 25088
                    OutputMemorySize: 4096

            Processing Module "custom"
                    ModuleGeneration: 'on'
                            Addition: 'on'
                      Multiplication: 'on'
                     InputMemorySize: 40
                    OutputMemorySize: 40

        Processor Top Level Properties
                      RunTimeControl: 'register'
                       RunTimeStatus: 'register'
                  InputStreamControl: 'register'
                 OutputStreamControl: 'register'
                   ProcessorDataType: 'single'

               System Level Properties
                      TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit'
                     TargetFrequency: 200
                       SynthesisTool: 'Xilinx Vivado'
                     ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
             SynthesisToolChipFamily: 'Zynq UltraScale+'
             SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
            SynthesisToolPackageName: ''
             SynthesisToolSpeedValue: ''
```

Create Custom Layer MATLAB Function

Design the custom layer function by using a MATLAB function. The custom layer function must:

- Have a maximum of two inputs and one output.
- Use only element-wise operations. These operations are not element-wise operations:
  - Matrix multiplication
  - Flatten
  - Reshape
  - Concatenation
  - Batch normalization

This example code shows the MATLAB function for a custom signum layer.

```
classdef SignumLayer < nnet.layer.Layer
    % Example custom Signum layer.

    properties
        testPropertyValue1 single = 3;
        testPropertyValue2 single = 4;
    end

    methods
        function layer = SignumLayer(name)
            % Set layer name.
            layer.Name = name;
            % Set layer description.
            layer.Description = "custom signum layer";
        end
```

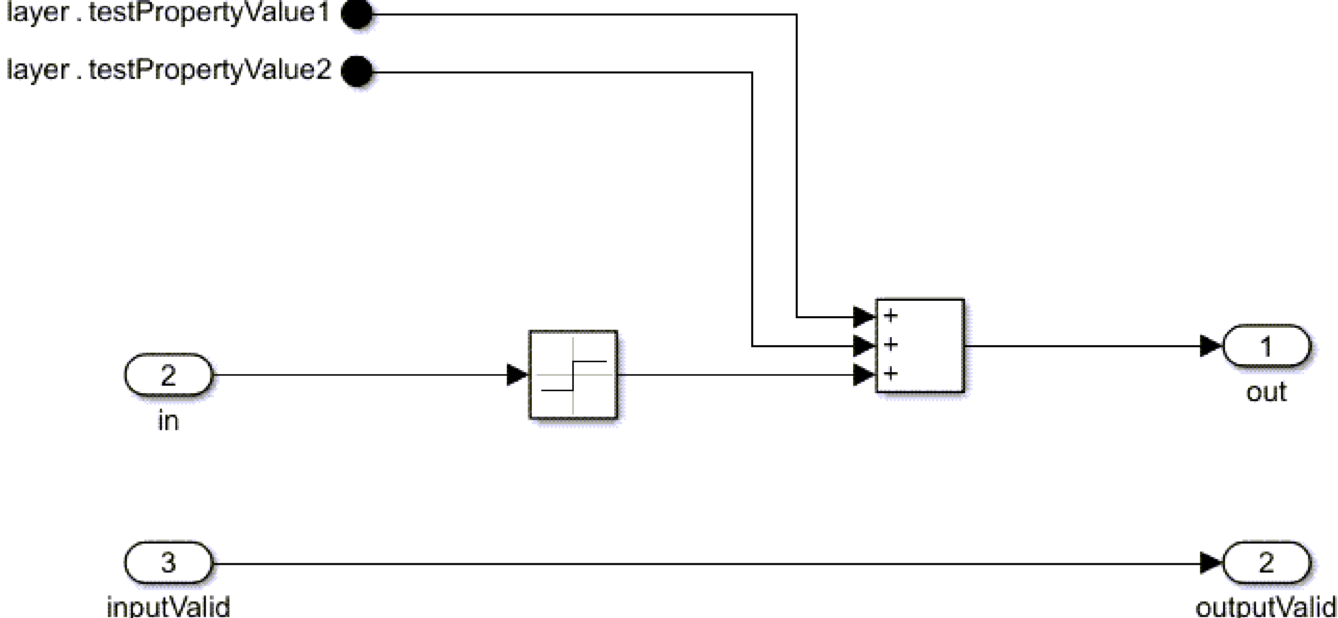
```
        function Z = predict(layer, X)
            % Z = predict(layer, X) forwards the input data X through the
            % layer and outputs the result Z.

            Z = sign(X) + layer.testPropertyValue1 + layer.testPropertyValue2;

        end
    end
end
```

Create Custom Layer Simulink Function

Design the custom layer model in Simulink. Your model design must:

- Use subsystem reference blocks only. Model reference blocks are not supported.
- Model the `inputValid` and `outputValid` signals.
- Have the same inputs and outputs as the custom layer MATLAB function.

This image shows the Simulink model for the custom signum layer.



Register Custom Layer and Model

Register an instance of the custom layer and custom layer Simulink model by using the `registerCustomLayer` method.

```
hSignum = SignumLayer('Signum1');
registerCustomLayer(hPC, Layer = hSignum, Model = 'mySignumModel.slx');
hPC
```

The custom deep learning processor configuration has a `Signum` layer under the `custom` processing module. The custom signum layer is enabled for bitstream generation.

```
hPC
```

```
hPC = 

                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                             ConvThreadNumber: 16
                              InputMemorySize: [227 227 3]
                             OutputMemorySize: [227 227 3]
                             FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                      SigmoidBlockGeneration: 'off'
                               FCThreadNumber: 4
                              InputMemorySize: 25088
                             OutputMemorySize: 4096

                  Processing Module "custom"
                            ModuleGeneration: 'on'
                                    Addition: 'on'
                              Multiplication: 'on'
                                 SignumLayer: 'on'
                              InputMemorySize: 40
                             OutputMemorySize: 40

              Processor Top Level Properties
                               RunTimeControl: 'register'
                                RunTimeStatus: 'register'
                           InputStreamControl: 'register'
                          OutputStreamControl: 'register'
                            ProcessorDataType: 'single'

                     System Level Properties
                               TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit'
                              TargetFrequency: 200
                                SynthesisTool: 'Xilinx Vivado'
                              ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

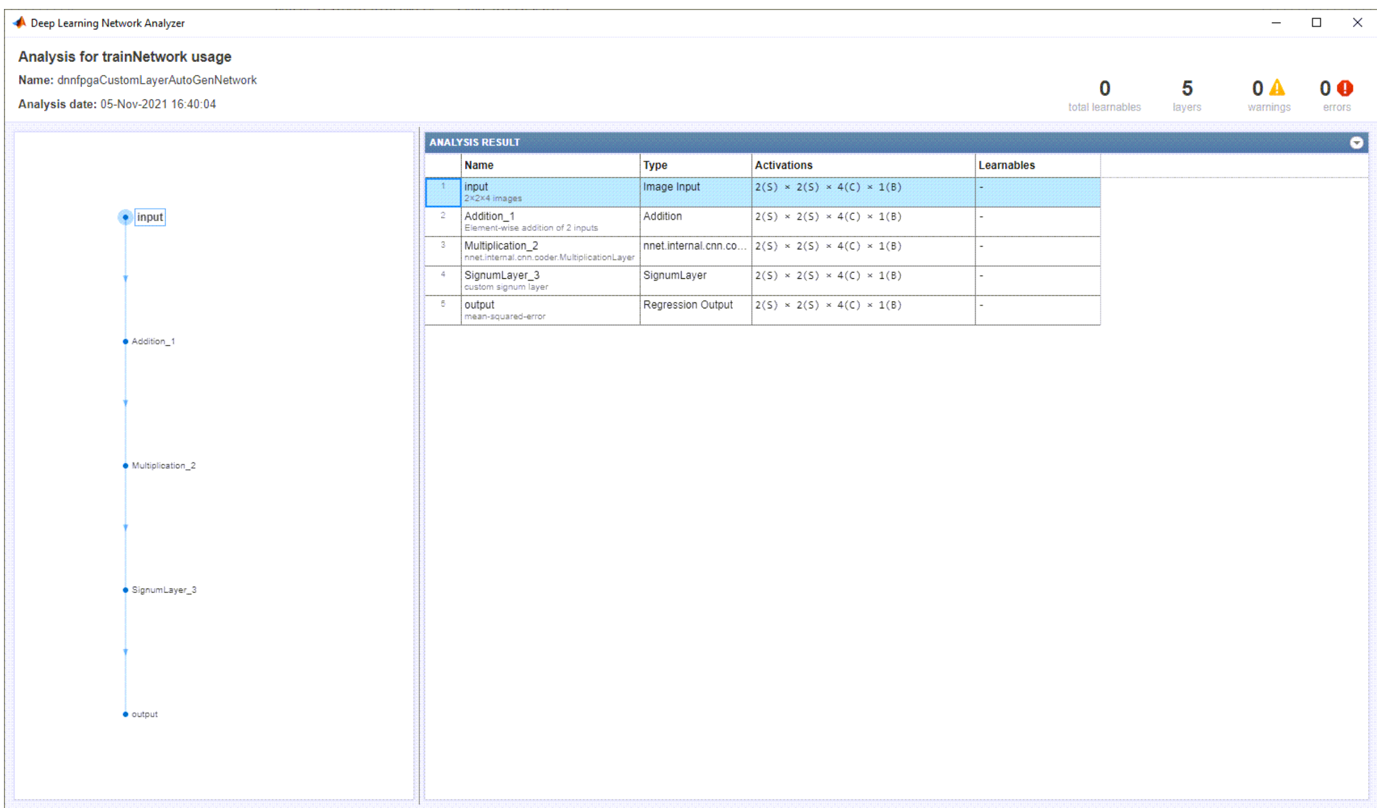
Generate Verification Model for Custom Layer

Generate a verification model for your custom layer by using the `openCustomLayerModel` method. Generate a test network and test image for your custom layer network by specifying blank arguments for the `Network` and `InputImages` arguments of the `openCustomLayerModel` method. The size of the test image matches the input layer size of the created test network.

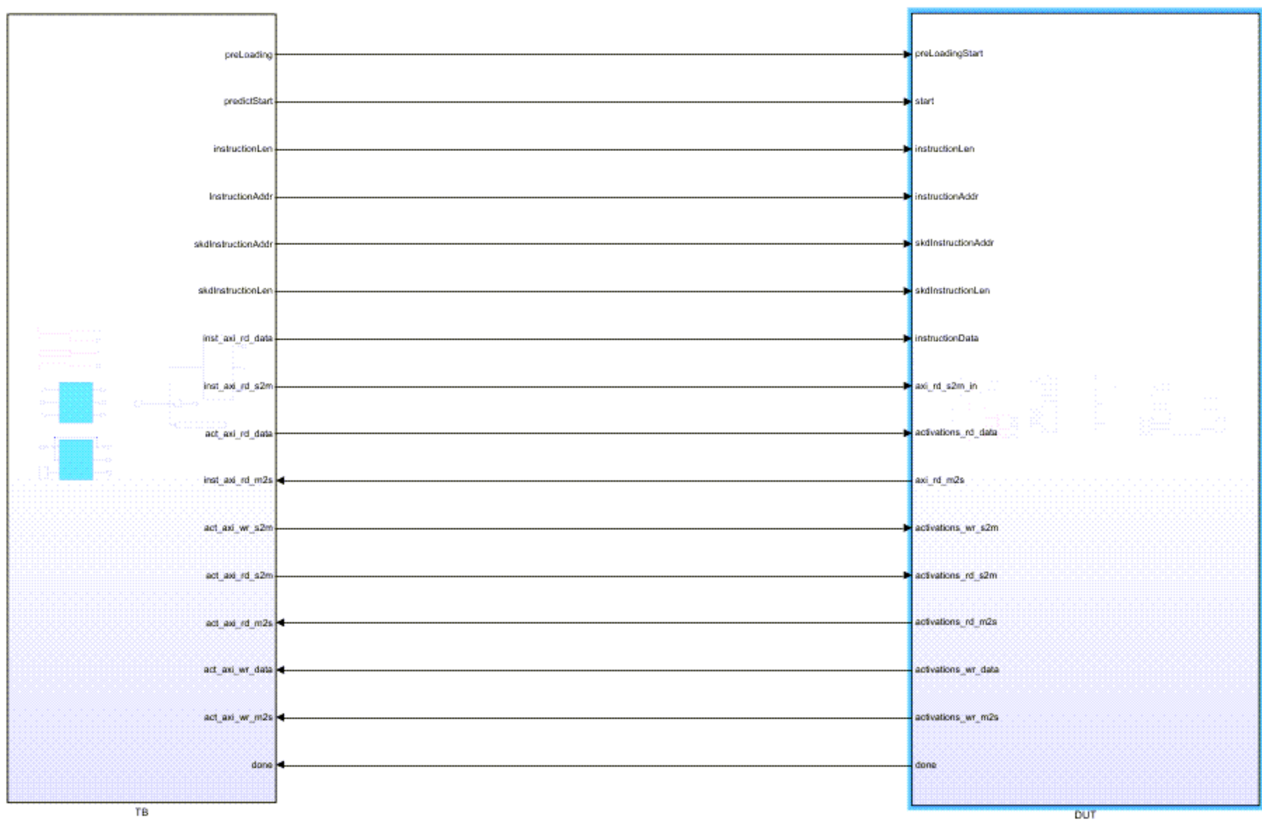
```
openCustomLayerModel(hPC)
```

```
### The 'Network' property is empty for the given object. An auto-generated network is provided.
### Custom layer verification model generation begin.
### Compiling network for Deep Learning FPGA prototyping ...
### Custom layer verification model generation complete.
```

An input image of size two-by-two-by four is created for the generated verification model. This image shows the auto-generated network for the custom layer model.



The `openCustomLayerModel` method generates a verification model file called `dnnfpgaCustomLayerVerificationModel.slx` for your custom layer. The generated verification model consists of a test bench block `TB` and a design under test block `DUT`. The testbench block contains tests signals that are applied to your custom layer model which is a part of the design under test block to verify the functionality of the custom layer and prediction accuracy of the network that has the custom layer. This image shows the generated verification model blocks.



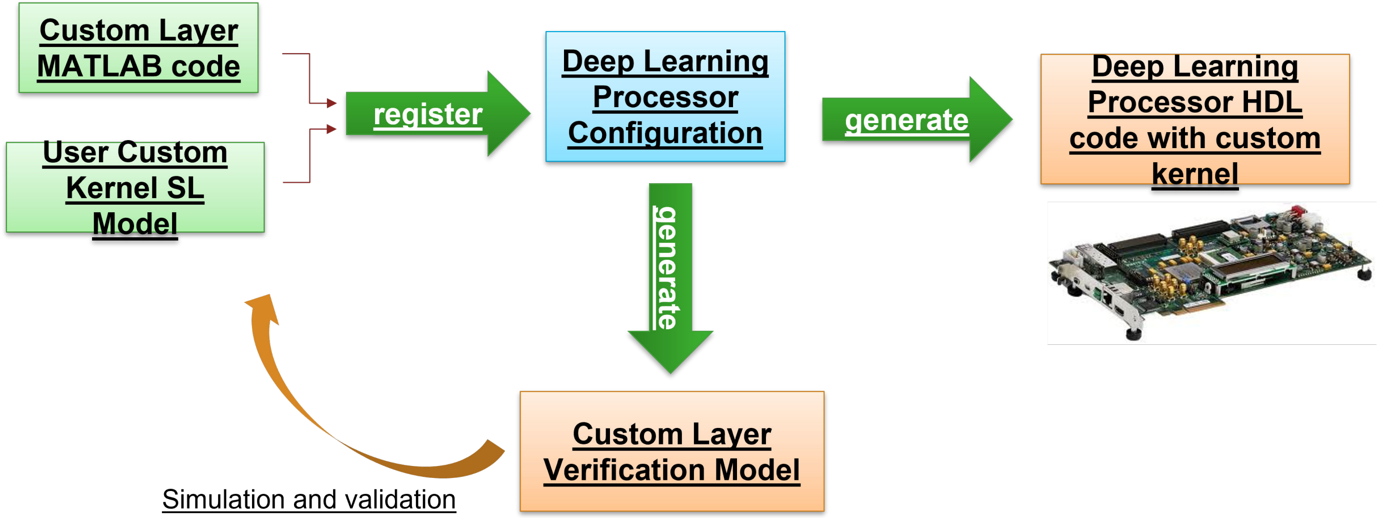
Simulate and Validate Custom Layer Model

Before you verify your custom layer model by using the `verifyCustomLayerModel` method, open the `dnnfpgaCustomLayerVerificationModel.slx` verification model. The `verifyCustomLayerModel` method verifies the functionality of the custom layer and prediction accuracy of the network that has the custom layer.

```
verifyCustomLayerModel(hPC)
```

```
### Custom layer verification model simulation and validation begin.
### Compiling Simulink model 'dnnfpgaCustomLayerVerificationModel' ...
### Complete Simulink model 'dnnfpgaCustomLayerVerificationModel' compilation.
Verification passed.
### Custom layer verification model simulation and validation complete.
```

Use the generated verification model to simulates, test, iterate and develop your custom kernel Simulink model. This image shows the custom kernel development process.



Generate Custom Bitstream

Generate a custom bitstream that has the name `myCustomLayer.bit` by using the `dlhdl.buildProcessor` function. Save the generated bitstream to the `myCustomLayer_prj` folder.

```
dlhdl.buildProcessor(hPC, ProjectFolder = 'myCustomLayer_prj', ProcessorName = 'myCustomLayer');
```

Deploy and Predict Custom Layer Network on Hardware

Deploy the custom layer network by creating a `dlhdl.Workflow` object with the custom layer network as the `Network` argument and the custom bitstream `myCustomLayer.bit` as the `Bitstream` argument. To retrieve the prediction results from the deployed network use MATLAB and the `predict` method.

```
hTarget = dlhdl.Target('Xilinx','Interface','JTAG');
hW = dlhdl.Workflow(Network = myCustomNet, Bitstream =
'myCustomLayer.bit'
,
...
Target = hTarget);
hW.compile;
hW.deploy;
image = randi(255, [2,2,4]);
hW.predict(single(image),Profile =
'on'
);
```

Custom Layer Registration File

To reuse your verified custom layers, register them by using a custom layer registration file. Custom registration layer files must be named `dlhdl_customLayer_registration.m`. The custom layer registration file contains a list of `dlhdl.CustomLayer` objects. A specific board can have multiple custom layer registration files on the MATLAB path. Do not list `dlhdl.CustomLayer` objects in more than one custom layer registration file.

When the processor configuration object is created, Deep Learning HDL Toolbox searches the MATLAB path for files named `dlhdl_customLayer_registration.m`, and uses the information in the files to populate the registered custom layer information. List only custom layers in the custom layer registration file after they have been verified by using the `verifyCustomLayerModel` method.

This script is an example of a `dlhdl_customLayer_registration.m` file.

```
function customLayerList = dlhdl_customLayer_registration
% Custom layer registration file
% 1. Any registration file with this name on MATLAB path will be picked up.
% 2. Registration file returns a cell array of dlhdl.CustomLayer
% object which are used to register custom layer information for Deep
% Learning HDL Toolbox workflow
% 3. Use dlhdl.CustomLayer object to register a layer class, and a
% model file path relative to the location of this registration file

%   Copyright 2021 The MathWorks, Inc.

customLayerList = { ...
    dlhdl.CustomLayer('Name', 'Addition',       'Layer', additionLayer(2),       'Model', 'model/customLayers/dnnfpgaAdditionLayerModel.
    dlhdl.CustomLayer('Name', 'Multiplication', 'Layer', multiplicationLayer(2), 'Model', 'model/customLayers/dnnfpgaMultiplicationLayer
    };

end
```

To register the custom signum layer for reuse, create this `dlhdl_customLayer_registration.m` file and place it on the MATLAB path.

```
function customLayerList = dlhdl_customLayer_registration
% Custom layer registration file
% 1. Any registration file with this name on MATLAB path will be picked up.
% 2. Registration file returns a cell array of dlhdl.CustomLayer
% object which are used to register custom layer information for Deep
% Learning HDL Toolbox workflow
% 3. Use dlhdl.CustomLayer object to register a layer class, and a
% model file path relative to the location of this registration file

%   Copyright 2021 The MathWorks, Inc.

customLayerList = { ...
    dlhdl.CustomLayer('Name','Signum','Layer',SignumLayer('Signum'),'Model','C:\Users\skapali\dnnfpgaSignumLayerModel.slx'), ...
   };

end
```

Create a `dlhdl.ProcessorConfig` object. The custom signum layer now appears in the default processor configuration object under the `custom` processing module.

```
hPC = dlhdl.ProcessorConfig
```

```
hPC = 

                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                      SigmoidBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                  Processing Module "custom"
                            ModuleGeneration: 'on'
                                    Addition: 'on'
                              Multiplication: 'on'
                                      Signum: 'on'
                             InputMemorySize: 40
```

```
                  OutputMemorySize: 40

    Processor Top Level Properties
                    RunTimeControl: 'register'
                     RunTimeStatus: 'register'
                InputStreamControl: 'register'
               OutputStreamControl: 'register'
                 ProcessorDataType: 'single'

           System Level Properties
                    TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit'
                   TargetFrequency: 200
                     SynthesisTool: 'Xilinx Vivado'
                   ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
           SynthesisToolChipFamily: 'Zynq UltraScale+'
           SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
          SynthesisToolPackageName: ''
           SynthesisToolSpeedValue: '
```

For an example of how to create, register, validate, and deploy a network with a custom `log` layer, see “Register, Validate, and Deploy Custom Natural Logarithm Layer Network to FPGA” on page 8-35 .

See Also

`dlhdl.ProcessorConfig` | `registerCustomLayer` | `openCustomLayerModel` | `verifyCustomLayerModel`

Register, Validate, and Deploy Custom Natural Logarithm Layer Network to FPGA

This example shows how to register, validate, and deploy a custom natural logarithm (log) layer network by using Deep Learning HDL Toolbox™. To deploy the network with the custom natural logarithm (log) layer:

1. Create a custom processor configuration by using the `dlhdl.ProcessorConfig` object.
2. Create a MATLAB® Function and Simulink® model to represent your custom layer.
3. Register the custom natural logarithm log) layer by using the `registerCustomLayer` method.
4. Simulate and verify your custom layer by using a generated verification model. Generate a verification model by using the `openCustomLayerModel` method. Verify the custom layer by using the `verifyCustomLayerModel` method.
5. Generate a custom bitstream by using the `dlhdl.buildProcessor` function.
6. Create a workflow object that has your custom layer network and custom bitstream by using the `dlhdl.Workflow` object.
7. Compile and deploy the workflow object by using the `compile` and `deploy` methods.

To retrieve the prediction results from the deployed custom layer network, use MATLAB®.

Create a Deep Learning Processor Configuration

To generate a custom processor configuration, use the `dlhdl.ProcessorConfig` object. The generated deep learning processor configuration object has a custom module that contains the preconfigured custom layers. Save the deep learning processor configuration to a variable `hPC`.

```
hPC = dlhdl.ProcessorConfig
```

```
hPC = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                             ConvThreadNumber: 16
                              InputMemorySize: [227 227 3]
                             OutputMemorySize: [227 227 3]
                             FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                      SigmoidBlockGeneration: 'off'
                               FCThreadNumber: 4
                              InputMemorySize: 25088
                             OutputMemorySize: 4096

                  Processing Module "custom"
                            ModuleGeneration: 'on'
                                    Addition: 'on'
                              Multiplication: 'on'
                                    Resize2D: 'off'
                                     Sigmoid: 'off'
```

```
                   TanhLayer: 'off'
             InputMemorySize: 40
            OutputMemorySize: 120

   Processor Top Level Properties
              RunTimeControl: 'register'
               RunTimeStatus: 'register'
          InputStreamControl: 'register'
         OutputStreamControl: 'register'
                SetupControl: 'register'
           ProcessorDataType: 'single'

          System Level Properties
              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
             TargetFrequency: 200
               SynthesisTool: 'Xilinx Vivado'
             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
     SynthesisToolChipFamily: 'Zynq UltraScale+'
     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
    SynthesisToolPackageName: ''
     SynthesisToolSpeedValue: ''
```

Create Custom Layer MATLAB Function

Design the custom layer function by using a MATLAB function. The custom layer function must:

- Have a maximum of two inputs and one output.
- Use only element-wise operations. These operations are not element-wise operations, Matrix multiplication, flatten, reshape, concatenation, batch normalization

This example code shows the MATLAB function for a custom logarithm layer.

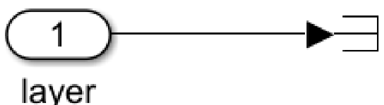
```
classdef LogLayer < nnet.layer.Layer
    % To create a custom network with exponential layer for verification
    % and for using it in yolov2transform layer, this class can be used.

    methods
        % function layer = LogLayer(varargin)
        %     p = inputParser;
        %     addParameter(p,'Name', []);
        %     parse(p,varargin{:});
        %     layer.Name = p.Results.Name;
        %     layer.Type = 'Log';
        % end

        function layer = LogLayer(name)
            layer.Name = name;
            layer.Description = 'Custom Log Layer';
        end

        function Z = predict(~, X)
            % Forward input data through the layer at prediction time and
            % output the result
            Z = log(X);
        end
    end
```

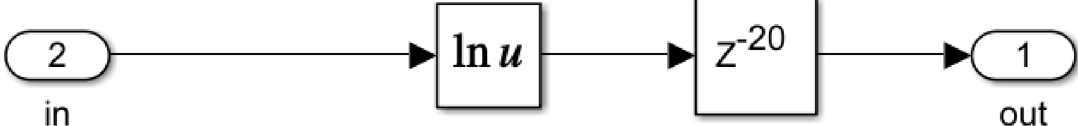
```
end
```

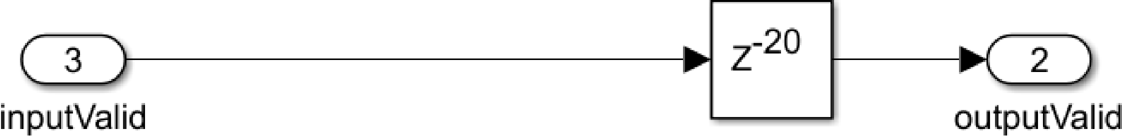
Create Custom Layer Simulink Function

Design the custom layer model in Simulink. Your model design must:

- Use subsystem reference blocks only. Model reference blocks are not supported.
- Model the `inputvalid` and `outputValid` signals.
- Have the same inputs and outputs as the custom layer MATLAB function.







Register Custom Layer and Model

To register an instance of the custom layer and custom layer Simulink® model use the `registerCustomLayer` method. Deep Learning HDL Toolbox™ uses the Simulink® model to generate a verification model for the custom layer.

```
hLogLayer = LogLayer('customLog');
registerCustomLayer(hPC,Layer = hLogLayer, Model = 'dnnfpgaLogLayerModel.slx');
hPC
```

```
hPC = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                             ConvThreadNumber: 16
```

```
                  InputMemorySize: [227 227 3]
                 OutputMemorySize: [227 227 3]
                 FeatureSizeLimit: 2048

           Processing Module "fc"
                 ModuleGeneration: 'on'
           SoftmaxBlockGeneration: 'off'
           SigmoidBlockGeneration: 'off'
                   FCThreadNumber: 4
                  InputMemorySize: 25088
                 OutputMemorySize: 4096

       Processing Module "custom"
                 ModuleGeneration: 'on'
                         Addition: 'on'
                   Multiplication: 'on'
                         Resize2D: 'off'
                          Sigmoid: 'off'
                        TanhLayer: 'off'
                         LogLayer: 'on'
                  InputMemorySize: 40
                 OutputMemorySize: 120

   Processor Top Level Properties
                   RunTimeControl: 'register'
                    RunTimeStatus: 'register'
               InputStreamControl: 'register'
              OutputStreamControl: 'register'
                     SetupControl: 'register'
                ProcessorDataType: 'single'

          System Level Properties
                   TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                  TargetFrequency: 200
                    SynthesisTool: 'Xilinx Vivado'
                  ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
          SynthesisToolChipFamily: 'Zynq UltraScale+'
          SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
         SynthesisToolPackageName: ''
          SynthesisToolSpeedValue: ''
```

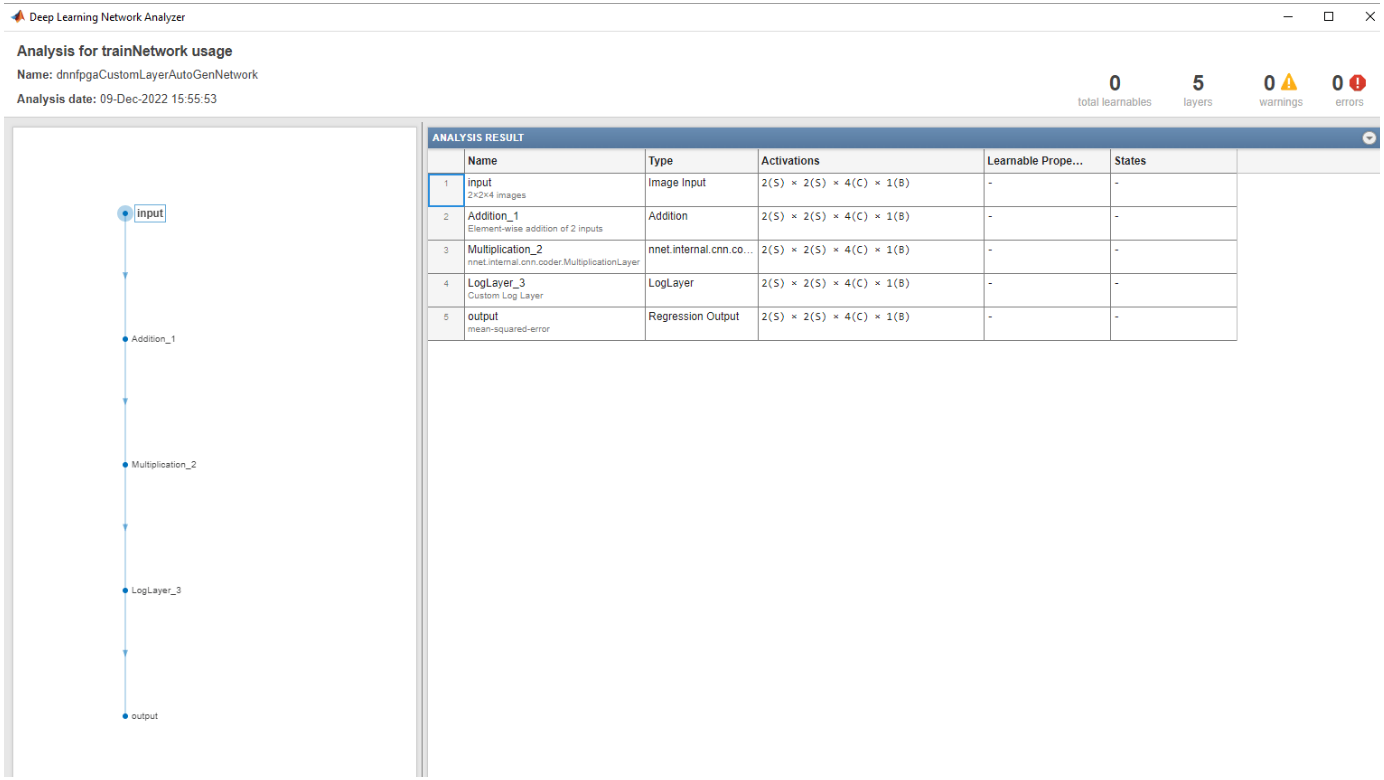
The custom deep learning processor configuration has a `Log` layer under the custom processing module. The custom natural logarithm (log) layer is enabled by default for the bitstream generation.

Generate Verification Model for Custom Layer

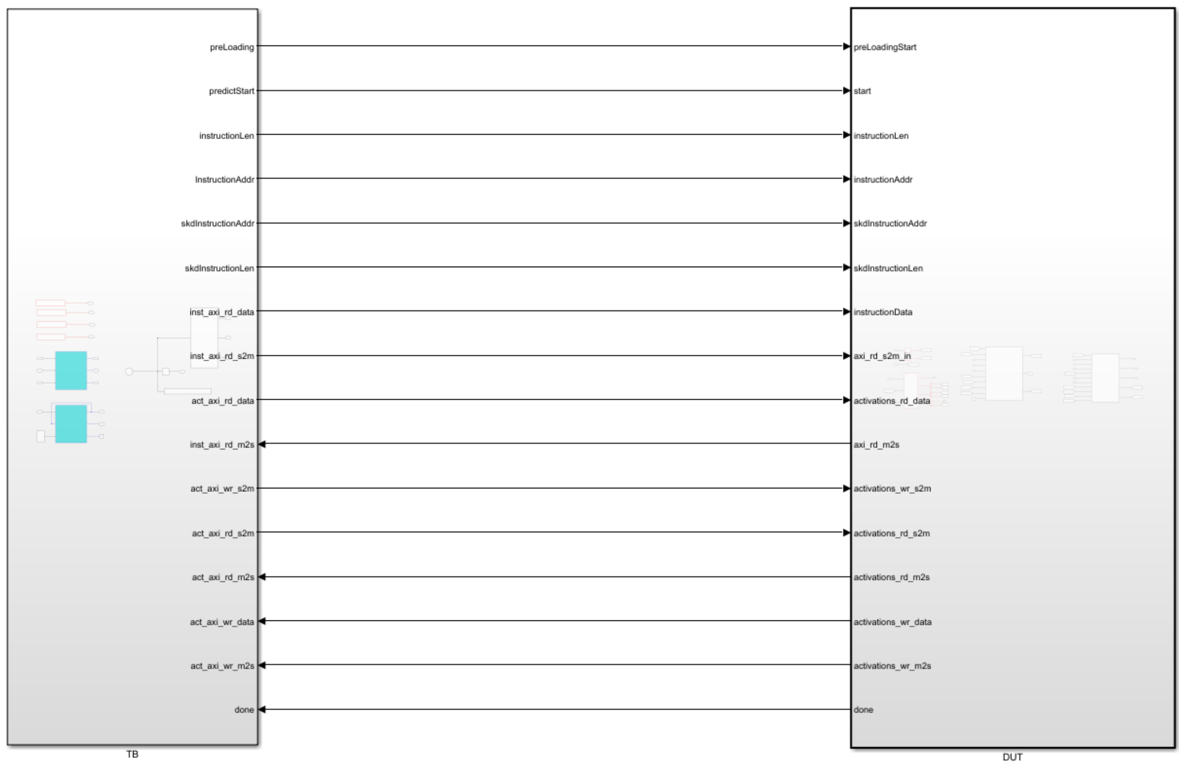
Generate a verification model for your custom layer by using the `openCustomLayerModel` method. Generate a test network and a test image for your custom layer network by specifying blank arguments for the `Network` and `InputImages` arguments of the `openCustomLayerModel` method. The size of the test image matches the input layer size of the created test network.

```
openCustomLayerModel(hPC);
```

An input image of size two-by-two-by-four is created for the generated verification model. This image shows the auto-generated network for the custom layer model.



The `openCustomLayerModel` method generates a verification model file called `dnnfpgaCustomLayerVerificationModel.slx` for your custom layer. The generated verification model consists a test bench block `TB` and a design under test block `DUT`. The test bench block contains test signals that are applied to your custom layer model which is a part of the design under test block to verify the functionality of the custom layer and prediction accuracy of the network that has the custom layer. This image shows the generated verification model blocks.

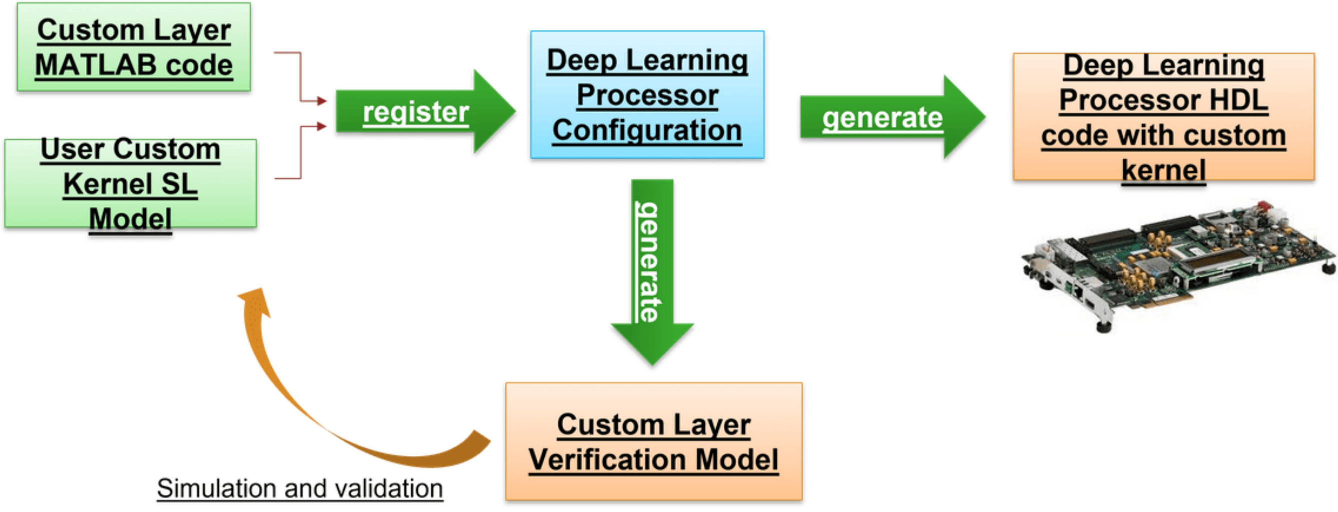


Simulate and Validate Custom Layer Model

Before you verify your custom layer model by using the `verifyCustomLayerModel` method, open the `dnnfpgaCustomLayerVerificationModel.slx` verification model. The `verifyCustomLayerModel` verifies the functionality of the custom layer and prediction accuracy of the network which has the custom layer.

```
open_system('dnnfpgaCustomLayerVerificationModel.slx');
verifyCustomLayerModel(hPC);
```

Use the generated verification model to simulate, test, iterate, and develop your custom kernel Simulink model. This image shows the custom kernel development process.



Generate Custom Bitstream

Generate a custom bitstream that has the name `myCustomLayer.bit` by using the `dlhdl.buildProcessor` function. Save the generated bitstream to the `myCustomLayer_prj` folder.

```
dlhdl.buildProcessor(hPC,ProjectFolder = 'myCustomLayer_prj',ProcessorName ='myCustomLayer');
```

Deploy and Predict Custom Layer Network on Hardware

Deploy the custom layer network by creating a `dlhdl.Workflow` object with the custom layer network as the `Network` argument and the custom bitstream `myCustomLayer.bit` as the `Bitstream` argument. To retrieve the prediction results from the deployed network use MATLAB and the `predict` method.

```
hTarget = dlhdl.Target('Xilinx','Interface','JTAG');
hW = dlhdl.Workflow(Network = myCustomNet, Bitstream='myCustomLayer.bit',Target=hTarget);
compile(hW);
deploy(hW);
image = randi(255, [2,2,4]);
predict(hW,single(image),Profile='on');
```

See Also

`dlhdl.Workflow` | `dlhdl.ProcessorConfig` | `registerCustomLayer` | `openCustomLayerModel` | `verifyCustomLayerModel` | `dlhdl.buildProcessor`

More About

- “Create Deep Learning Processor Configuration for Custom Layers” on page 8-26

Custom Processor Code Generation Workflow

- “Generate Custom Bitstream” on page 9-2
- “Generate Custom Processor IP” on page 9-3

Generate Custom Bitstream

To deploy a deep learning network to your custom target device, generate a custom bitstream by using the `dlhdl.ProcessorConfig` object.

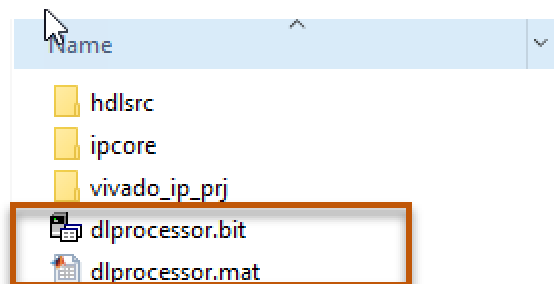
1. Create a `dlhdl.ProcessorConfig` object.

   ```
   hPC = dlhdl.ProcessorConfig;
   ```
2. Set up the tool path to your design tool. For example, to set up the path to the Vivado design tool, enter:

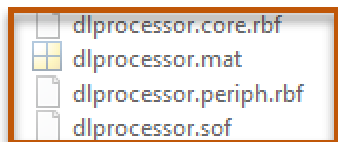
   ```
   hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
   ```
3. Generate the custom bitstream.

   ```
   dlhdl.buildProcessor(hPC);
   ```
4. Locate the bitstream file and associated MAT file at `cwd\dlhdl_prj\`, where `cwd` is your current working folder. The name of the bitstream file is `dlprocessor.bit` . The name of the MAT file is `dlprocessor.mat`.

   To use the generated bitstream for the supported Xilinx boards, copy the `dlprocessor.bit` and `dlprocessor.mat` files to the present working folder.

   To use the generated bitstream for the supported Intel boards, copy the `dlprocessor.core.rbf`, `dlprocessor.mat`, `dlprocessor.periph.rbf`, and `dlprocessor.sof` files to the same present working folder.

5. Deploy the custom bitstream and deep learning network to your target device.

   ```
   hTarget = dlhdl.Target('Xilinx');
   net = resnet18;
   hW = dlhdl.Workflow('Network',net,'Bitstream','dlprocessor.bit','Target',hTarget);
   % If your custom bitstream files are in a different folder, use:
   % hW = dlhdl.Workflow('Network',snet,'Bitstream',...
   % 'C:\yourfolder\dlprocessor.bit','Target',hTarget);
   hW.compile;
   hW.deploy;
   ```

See Also

`dlhdl.ProcessorConfig` | `dlhdl.buildProcessor` | `dlhdl.Workflow`

Generate Custom Processor IP

Generate a custom deep learning processor IP core from a custom deep learning processor configuration. The generated deep learning processor IP core is shared and reusable. Integrate the generated deep learning processor IP core into your custom reference design. The `dlhdl.buildProcessor` API builds the `dlhdl.ProcessorConfig` object to generate a custom processor IP and related code that you can use in your custom reference designs.

1. Create a `dlhdl.ProcessorConfig` object.

   ```
   hPC = dlhdl.ProcessorConfig;
   ```
2. Set up the tool path to your design tool. For example, to set up the path to the Vivado design tool, enter:

   ```
   hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
   ```
3. Generate the custom processor IP.

   ```
   dlhdl.buildProcessor(hPC);
   ```

See Also

`dlhdl.ProcessorConfig` | `dlhdl.buildProcessor`

More About

- “Deep Learning Processor IP Core” on page 12-5

Featured Examples

- “Get Started with Deep Learning FPGA Deployment on Intel Arria 10 SoC” on page 10-3
- “Get Started with Deep Learning FPGA Deployment on Xilinx ZCU102 SoC” on page 10-6
- “Get Started with Deep Learning FPGA Deployment on Xilinx ZC706 SoC” on page 10-10
- “Logo Recognition Network” on page 10-13
- “Deploy Transfer Learning Network for Lane Detection” on page 10-18
- “Image Category Classification by Using Deep Learning” on page 10-23
- “Defect Detection” on page 10-32
- “Profile Network to Determine Performance Bottlenecks” on page 10-49
- “Bicyclist and Pedestrian Classification by Using FPGA ” on page 10-53
- “Visualize Activations of a Deep Learning Network by Using LogoNet” on page 10-59
- “Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core” on page 10-65
- “Run a Deep Learning Network on FPGA with Live Camera Input” on page 10-70
- “Running Convolution-Only Networks by Using FPGA Deployment” on page 10-80
- “Accelerate Prototyping Workflow for Large Networks by Using Ethernet” on page 10-86
- “Create Series Network for Quantization” on page 10-94
- “Custom Deep Learning Processor Generation to Meet Performance Requirements” on page 10-98
- “Quantize Network for FPGA Deployment” on page 10-104
- “Evaluate Performance of Deep Learning Network on Custom Processor Configuration” on page 10-110
- “Customize Bitstream Configuration to Meet Resource Use Requirements” on page 10-116
- “Vehicle Detection Using DAG Network Based YOLO v2 Deployed to FPGA” on page 10-122
- “Customize Bitstream Configuration to Meet Resource Use Requirements” on page 10-131
- “Image Classification Using Neural Network on FPGA” on page 10-137
- “Classify Images on FPGA Using Quantized Neural Network” on page 10-145
- “Classify ECG Signals Using DAG Network Deployed to FPGA” on page 10-159
- “Prototype and Verify Deep Learning Networks Without Target Hardware” on page 10-170
- “Classify Images on FPGA by Using Quantized GoogLeNet Network” on page 10-177
- “Estimate Resource Utilization for Custom Board and Reference Design” on page 10-191
- “Speech Command Recognition by Using FPGA” on page 10-194
- “Modulation Classification by Using FPGA” on page 10-204
- “Deploy Simple Adder Network by using MATLAB Deployment Script and Deployment Instructions File” on page 10-216
- “Human Pose Estimation by Using Segmentation DAG Network Deployed to FPGA” on page 10-224
- “Semantic Segmentation of Multispectral Images by Using Quantized U-Net on FPGA” on page 10-230

- “Optimize Deep Learning Processor Configuration for Network Performance” on page 10-239
- “Run Sequence-to-Sequence Classification on FPGAs by Using Deep Learning HDL Toolbox” on page 10-246
- “Generate Word-By-Word Text on FPGAs by Using Deep Learning HDL Toolbox” on page 10-253
- “Run Sequence Forecasting on FPGA by Using Deep Learning HDL Toolbox” on page 10-262
- “Detect Objects Using YOLO v3 Network Deployed to FPGA” on page 10-283
- “Run Sequence-to-Sequence Regression on FPGAs” on page 10-297
- “Deploy and Verify YOLO v2 Vehicle Detector on FPGA” on page 10-309
- “Deploy Semantic Segmentation Network Using Dilated Convolutions on FPGA” on page 10-324
- “Run Sequence Forecasting Using a GRU Layer on an FPGA ” on page 10-333
- “Deploy YAMNet Networks to FPGAs With and Without Cross-Layer Equalization” on page 10-354
- “Increase Image Resolution Using VDSR Network Running on FPGA” on page 10-363
- “Deploy Image Recognition Network on FPGA With and Without Pruning” on page 10-379

Get Started with Deep Learning FPGA Deployment on Intel Arria 10 SoC

This example shows how to create, compile, and deploy a `dlhdl.Workflow` object that has a handwritten character detection series network object by using the Deep Learning HDL Toolbox™ Support Package for Intel FPGA and SoC. Use MATLAB® to retrieve the prediction results from the target device.

Prerequisites

- Intel Arria™ 10 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Intel FPGA and SoC
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox™

Load the Pretrained SeriesNetwork

To load the pretrained series network, that has been trained on the Modified National Institute Standards of Technology (MNIST) database[1], enter:

```
snet = getDigitsNetwork;
```

To view the layers of the pretrained series network, enter:

```
analyzeNetwork(snet)
```

Create Target Object

Create a target object that has a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use JTAG, install Intel™ Quartus™ Prime Standard Edition 20.1. Set up the path to your installed Intel Quartus Prime executable if it is not already set up. For example, to set the toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Altera Quartus II','ToolPath', 'C:\altera\20.1\quartus\bin64');
```

```
hTarget = dlhdl.Target('Intel')
```

```
hTarget = 
  Target with properties:

       Vendor: 'Intel'
    Interface: JTAG
```

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pretrained MNIST neural network, snet, as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Intel Arria 10 SOC board and the bitstream uses a single data type.

```
hW = dlhdl.Workflow('network', snet, 'Bitstream', 'arria10soc_single','Target',hTarget);
```

Compile the MNIST Series Network

To compile the MNIST series network, run the compile function of the `dlhdl.Workflow` object.

```
dn = hW.compile;
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SystemBufferOffset"        "0x00800000"     "28.0 MB"
    "InstructionDataOffset"     "0x02400000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02800000"     "4.0 MB"
    "FCWeightDataOffset"        "0x02c00000"     "4.0 MB"
    "EndOffset"                 "0x03000000"     "Total: 48.0 MB"
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Intel Arria 10 SoC hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

```
hW.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 28-Jun-2020 13:45:47
```

Run Prediction for Example Image

To load the example image, execute the predict function of the `dlhdl.Workflow` object, and then display the FPGA result, enter:

```
inputImg = imread('five_28x28.pgm');
```

Run prediction with the profile 'on' to see the latency and throughput results.

```
[prediction, speed] = hW.predict(single(inputImg),'Profile','on');
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    49243                  0.00033                       1
    conv_module            25983                  0.00017
        conv_1              6813                  0.00005
        maxpool_1           4705                  0.00003
        conv_2              5205                  0.00003
        maxpool_2           3839                  0.00003
        conv_3              5481                  0.00004
```

```
    fc_module                23260                  0.00016
        fc                   23260                  0.00016
 * The clock frequency of the DL processor is: 150MHz
```

```
[val, idx] = max(prediction);
fprintf('The prediction result is %d\n', idx-1);
```

```
The prediction result is 5
```

Bibliography

1. LeCun, Y., C. Cortes, and C. J. C. Burges. "The MNIST Database of Handwritten Digits." http://yann.lecun.com/exdb/mnist/.

See Also

More About

- "Create Simple Deep Learning Neural Network for Classification"

Get Started with Deep Learning FPGA Deployment on Xilinx ZCU102 SoC

This example shows how to create, compile, and deploy a `dlhdl.Workflow` object that has a handwritten character detection series network as the network object by using the Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC. Use MATLAB® to retrieve the prediction results from the target device.

Prerequisites

- Xilinx ZCU102 SoC development kit.
- Deep Learning HDL Toolbox™
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™

Load the Pretrained Series Network

To load the pretrained series network, that has been trained on the Modified National Institute Standards of Technology (MNIST) database[1], enter:

```
snet = getDigitsNetwork;
```

To view the layers of the pretrained series network, enter:

```
analyzeNetwork(snet)
```

Create Target Object

Create a target object that has a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet.

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet')
```

```
hTarget = 
  Target with properties:

       Vendor: 'Xilinx'
    Interface: Ethernet
    IPAddress: '192.168.1.101'
     Username: 'root'
         Port: 22
```

Create WorkFlow Object

Create an object of the `dlhdl.Workflow` class. Specify the network and the bitstream name during the object creation. Specify saved pretrained MNIST neural network, snet, as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx ZCU102 SOC board and the bitstream uses a single data type.

```
hW = dlhdl.Workflow('network', snet, 'Bitstream', 'zcu102_single','Target',hTarget)
```

Compile the MNIST Series Network

To compile the MNIST series network, run the compile function of the `dlhdl.Workflow` object.

```
dn = hW.compile;
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single ...
### The network includes the following layers:

     1   'imageinput'    Image Input             28×28×1 images with 'zerocenter' normalization
     2   'conv_1'        Convolution             8 3×3×1 convolutions with stride [1  1] and padd
     3   'batchnorm_1'   Batch Normalization     Batch normalization with 8 channels
     4   'relu_1'        ReLU                    ReLU
     5   'maxpool_1'     Max Pooling             2×2 max pooling with stride [2  2] and padding
     6   'conv_2'        Convolution             16 3×3×8 convolutions with stride [1  1] and pad
     7   'batchnorm_2'   Batch Normalization     Batch normalization with 16 channels
     8   'relu_2'        ReLU                    ReLU
     9   'maxpool_2'     Max Pooling             2×2 max pooling with stride [2  2] and padding
    10   'conv_3'        Convolution             32 3×3×16 convolutions with stride [1  1] and pa
    11   'batchnorm_3'   Batch Normalization     Batch normalization with 32 channels
    12   'relu_3'        ReLU                    ReLU
    13   'fc'            Fully Connected         10 fully connected layer
    14   'softmax'       Softmax                 softmax
    15   'classoutput'   Classification Output   crossentropyex with '0' and 9 other classes

3 Memory Regions created.

Skipping: imageinput
Compiling leg: conv_1>>relu_3 ...
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
### Notice: (Layer  1) The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented
### Notice: (Layer 10) The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is imp
Compiling leg: conv_1>>relu_3 ... complete.
Compiling leg: fc ...
### Notice: (Layer  1) The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented
### Notice: (Layer  3) The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is imp
Compiling leg: fc ... complete.
Skipping: softmax
Skipping: classoutput
Creating Schedule...
.......
Creating Schedule...complete.
Creating Status Table...
......
Creating Status Table...complete.
Emitting Schedule...
......
Emitting Schedule...complete.
Emitting Status Table...
........
Emitting Status Table...complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________
```

```
    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "28.0 MB"
    "InstructionDataOffset"     "0x02800000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02c00000"     "4.0 MB"
    "FCWeightDataOffset"        "0x03000000"     "4.0 MB"
    "EndOffset"                 "0x03400000"     "Total: 52.0 MB"

### Network compilation complete.
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

```
hW.deploy
```

```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'

Downloading target FPGA device configuration over Ethernet to SD card done. The system will now r


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 30-Dec-2020 15:13:03
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 30-Dec-2020 15:13:03
```

Run Prediction for Example Image

To load the example image, execute the predict function of the `dlhdl.Workflow` object, and then display the FPGA result, enter:

```
inputImg = imread('five_28x28.pgm');
```

Run prediction with the profile 'on' to see the latency and throughput results.

```
[prediction, speed] = hW.predict(single(inputImg),'Profile','on');
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
```

```
                         -------------          -------------          ---------        --
Network                      98117                  0.00045                    1
        conv_1                6607                  0.00003
        maxpool_1             4716                  0.00002
        conv_2                4637                  0.00002
        maxpool_2             2977                  0.00001
        conv_3                6752                  0.00003
        fc                   72428                  0.00033
 * The clock frequency of the DL processor is: 220MHz
```

```
[val, idx] = max(prediction);
fprintf('The prediction result is %d\n', idx-1);
```

```
The prediction result is 5
```

Bibliography

1. LeCun, Y., C. Cortes, and C. J. C. Burges. "The MNIST Database of Handwritten Digits." http://yann.lecun.com/exdb/mnist/.

See Also

More About

- "Create Simple Deep Learning Neural Network for Classification"

Get Started with Deep Learning FPGA Deployment on Xilinx ZC706 SoC

This example shows how to create, compile, and deploy a handwritten character detection series network to an FPGA and use MATLAB® to retrieve the prediction results.

Prerequisites

- Xilinx® Zynq® ZC706 Evaluation Kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx® FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Load Pretrained Network

Load the pretrained series network trained on the Modified National Institute of Standards and Technology (MNIST) database[1].

```
snet = getDigitsNetwork;
```

View the layers of the pretrained series network by using the Deep network Designer app.

```
deepNetworkDesigner(snet)
```

Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Create a programming interface with custom name for your target device and a JTAG interface to connect the target device to the host computer. To use JTAG, install Xilinx™ Vivado™ Design Suite 2020.2. Set the toolpath by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
```

```
hTarget = dlhdl.Target('Xilinx');
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx® Zynq® ZC706 Evaluation Kit and the bitstream uses the single data type. .

```
hW = dlhdl.Workflow(Network=snet,Bitstream='zc706_single',Target=hTarget);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment.

```
dn = compile(hW)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________
```

```
    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SystemBufferOffset"        "0x00800000"     "28.0 MB"
    "InstructionDataOffset"     "0x02400000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02800000"     "4.0 MB"
    "FCWeightDataOffset"        "0x02c00000"     "4.0 MB"
    "EndOffset"                 "0x03000000"     "Total: 48.0 MB"

dn = struct with fields:
       Operators: [1×1 struct]
    LayerConfigs: [1×1 struct]
      NetConfigs: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx® Zynq® UltraScale+ MPSoC ZCU102 hardware, run the `deploy` method of the `dlhdl.Workflow` object. This method programs the FPGA board using the output of the compile method and the programming file, downloads the network weights and biases, displays progress messages, and the time it takes to deploy the network.

```
deploy(hW)
```

```
### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 12-Jun-2020 14:54:22
```

Test Network

Load the example image.

```
inputImg = imread('five_28x28.pgm');
```

Classify the image on the FPGA by using the `predict` method of the `dlhdl.Workflow` object and display the results.

```
[prediction,speed] = hW.predict(single(inputImg),'Profile','on');
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------              -------------              ---------        --
Network                       80141                  0.00160                       1
    conv_module               47601                  0.00095
        conv_1                10047                  0.00020
        maxpool_1              6999                  0.00014
        conv_2                11367                  0.00023
        maxpool_2              5465                  0.00011
        conv_3                13783                  0.00028
    fc_module                 32540                  0.00065
        fc                    32540                  0.00065
 * The clock frequency of the DL processor is: 50MHz
```

```
[val,idx] = max(prediction);
fprintf('The prediction result is %d\n', idx-1);
```

```
The prediction result is 5
```

Bibliography

1 LeCun, Y., C. Cortes, and C. J. C. Burges. "The MNIST Database of Handwritten Digits." http://yann.lecun.com/exdb/mnist/.

See Also

dlhdl.Workflow | dlhdl.Target | compile | deploy | predict

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2

Logo Recognition Network

This example shows how to create, compile, and deploy a `dlhdl.Workflow` object that has Logo Recognition Network as the network object using the Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC. Use MATLAB® to retrieve the prediction results from the target device.

The Logo Recognition Network

Logos assist users in brand identification and recognition. Many companies incorporate their logos in advertising, documentation materials, and promotions. The logo recognition network (logonet) was developed in MATLAB® and can recognize 32 logos under various lighting conditions and camera motions. Because this network focuses only on recognition, you can use it in applications where localization is not required.

Prerequisites

- Xilinx ZCU102 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

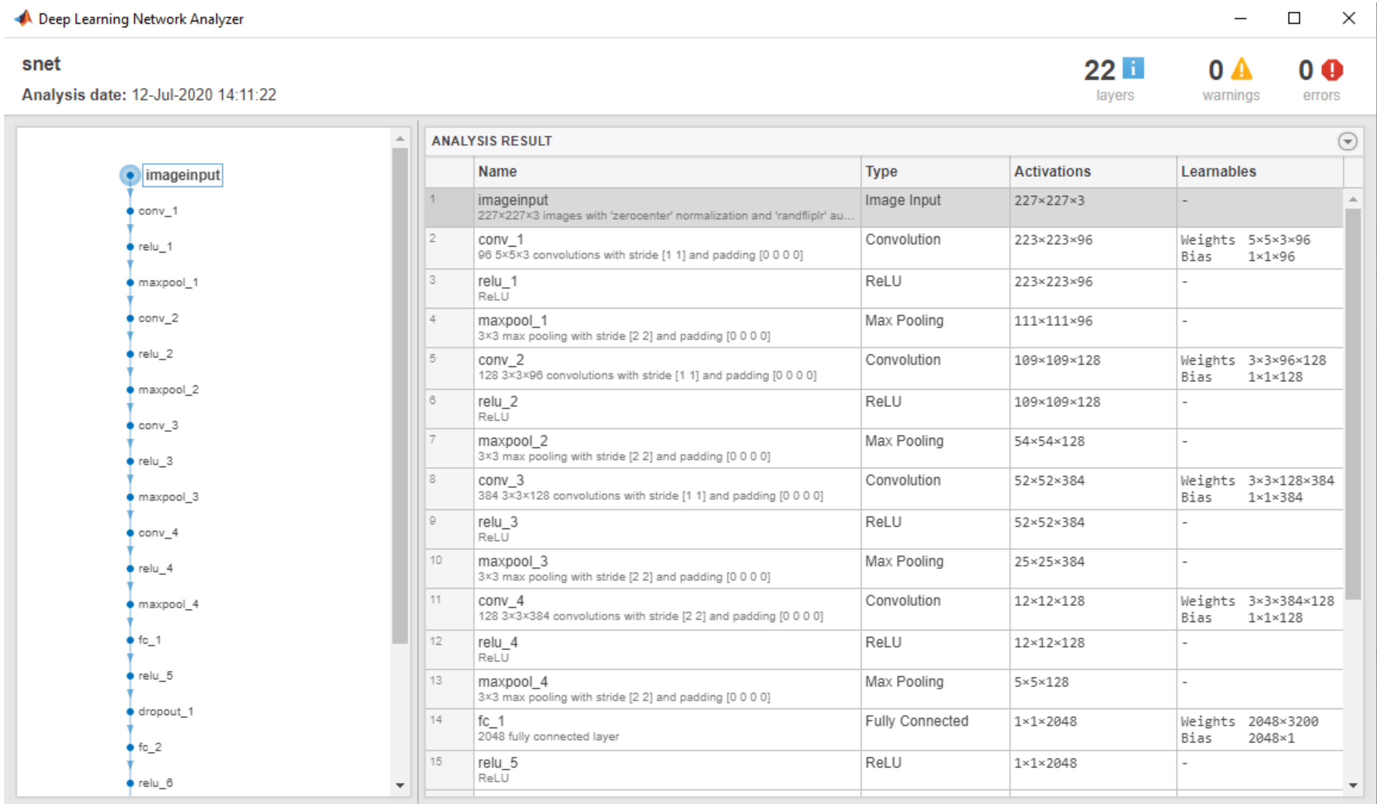
Load the Pretrained Series Network

To load the pretrained series network logonet, enter:

```
snet = getLogoNetwork;
```

To view the layers of the pretrained series network, enter:

```
analyzeNetwork(snet)
```



	Name	Type	Activations	Learnables
1	imageinput 227×227×3 images with 'zerocenter' normalization and 'randfliplr' au...	Image Input	227×227×3	-
2	conv_1 96 5×5×3 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	223×223×96	Weights 5×5×3×96 Bias 1×1×96
3	relu_1 ReLU	ReLU	223×223×96	-
4	maxpool_1 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	111×111×96	-
5	conv_2 128 3×3×96 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	109×109×128	Weights 3×3×96×128 Bias 1×1×128
6	relu_2 ReLU	ReLU	109×109×128	-
7	maxpool_2 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	54×54×128	-
8	conv_3 384 3×3×128 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	52×52×384	Weights 3×3×128×384 Bias 1×1×384
9	relu_3 ReLU	ReLU	52×52×384	-
10	maxpool_3 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	25×25×384	-
11	conv_4 128 3×3×384 convolutions with stride [2 2] and padding [0 0 0 0]	Convolution	12×12×128	Weights 3×3×384×128 Bias 1×1×128
12	relu_4 ReLU	ReLU	12×12×128	-
13	maxpool_4 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	5×5×128	-
14	fc_1 2048 fully connected layer	Fully Connected	1×1×2048	Weights 2048×3200 Bias 2048×1
15	relu_5 ReLU	ReLU	1×1×2048	-

Create Target Object

Create a target object that has a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use JTAG, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.b
```

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Create WorkFlow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pretrained logonet neural network, `snet`, as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example the target FPGA board is the Xilinx ZCU102 SOC board. The bitstream uses a single data type.

```
hW = dlhdl.Workflow('network', snet, 'Bitstream', 'zcu102_single','Target',hTarget);
% If running on Xilinx ZC706 board, instead of the above command,
% uncomment the command below.
%
% hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'zc706_single','Target',hTarget);
```

Compile the Logo Recognition Network

To compile the logo recognition network, run the compile function of the `dlhdl.Workflow` object.

```
dn = hW.compile
```

```
          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SystemBufferOffset"        "0x01c00000"     "60.0 MB"
    "InstructionDataOffset"     "0x05800000"     "12.0 MB"
    "ConvWeightDataOffset"      "0x06400000"     "32.0 MB"
    "FCWeightDataOffset"        "0x08400000"     "44.0 MB"
    "EndOffset"                 "0x0b000000"     "Total: 176.0 MB"

dn = struct with fields:
       Operators: [1×1 struct]
    LayerConfigs: [1×1 struct]
      NetConfigs: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

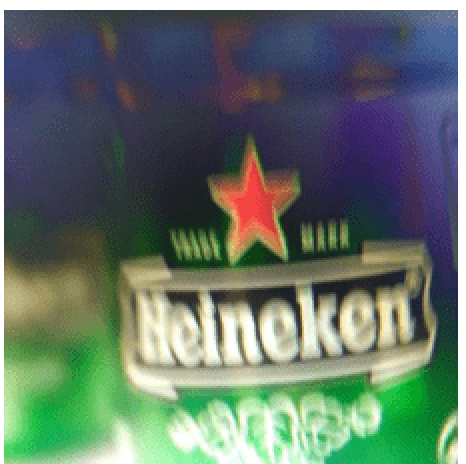
```
 hW.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to FC Processor.
### 33% finished, current time is 28-Jun-2020 12:40:14.
### 67% finished, current time is 28-Jun-2020 12:40:14.
### FC Weights loaded. Current time is 28-Jun-2020 12:40:14
```

Load the Example Image

Load the example image.

```
image = imread('heineken.png');
inputImg = imresize(image, [227, 227]);
imshow(inputImg);
```



Run the Prediction

Execute the predict function on the `dlhdl.Workflow` object and display the result:

```
[prediction, speed] = hW.predict(single(inputImg),'Profile','on');
```

```
### Finished writing input activations.
### Running single input activations.

              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------              -------------             ---------        --
Network                    38865102                  0.17666                      1          388
    conv_module            34299592                  0.15591
        conv_1              6955899                  0.03162
        maxpool_1           3306384                  0.01503
        conv_2             10396300                  0.04726
        maxpool_2           1207215                  0.00549
        conv_3              9269094                  0.04213
        maxpool_3           1367650                  0.00622
        conv_4              1774679                  0.00807
        maxpool_4             22464                  0.00010
    fc_module               4565510                  0.02075
        fc_1                2748478                  0.01249
        fc_2                1758315                  0.00799
        fc_3                  58715                  0.00027
 * The clock frequency of the DL processor is: 220MHz
```

```
[val, idx] = max(prediction);
snet.Layers(end).ClassNames{idx}
```

```
ans =
'heineken'
```

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `compile` | `deploy` | `predict`

More About

- "Prototype Deep Learning Networks on FPGA and SoC Devices" on page 5-2

Deploy Transfer Learning Network for Lane Detection

This example shows how to create, compile, and deploy a lane detection convolutional neural network (CNN) to an FPGA, and use MATLAB® to retrieve the prediction results.

Prerequisites

- Xilinx ZCU102 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Load the Pretrained SeriesNetwork

Load the pretrained lanenet network.

```
snet = getLaneDetectionNetwork;
```

Normalize Input Layer

Normalize the input layer by modifying its type.

```
inputlayer = imageInputLayer(snet.Layers(1).InputSize,Normalization='none');
snet = SeriesNetwork([inputlayer;snet.Layers(2:end)]);
```

View the layers of the network by using the Deep Network Designer app.

```
deepNetworkDesigner(snet)
```

Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Create a programming interface with custom name for your target device and an Ethernet interface to connect the target device to the host computer.

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Generate Custom Bitstream to Deploy Network

The lane detection network consists of multiple cross-channel normalization layers. To support this layer on hardware, enable the `LRNBlockGeneration` property of the conv module in the bitstream that you need to use for FPGA inference. The zcu102_single bitstream does not have this property turned on. A new bitstream can be generated using the following lines of code. The generated bitstream can be used along with a dlhdl.Workflow object for inference.

When you create a dlhdl.ProcessorConfig object for a reference bitstream, make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example the target FPGA board is the Xilinx ZCU102 SoC board and the date type is single. Update the processor configuration with the `LRNBlockGeneration` property enabled and the `SegmentationBlockGeneration` property disabled. Disabling the `SegmentationBlockGeneration` property ensures that the Deep Learning IP fits on the FPGA and avoids overuse of resources. If targeting the Xilinx ZC706 board, replace `'zcu102_single'` with '`zc706_single`' in the first command.

```
hPC = dlhdl.ProcessorConfig('Bitstream', 'zcu102_single');
setModuleProperty(hPC,'conv', 'LRNBlockGeneration', 'on');
setModuleProperty(hPC,'conv', 'SegmentationBlockGeneration', 'off');
```

Generate a custom bitstream by using the `dlhdl.buildProcessor` function. To learn how to use the generated bitstream file, see “Generate Custom Bitstream” on page 9-2.

```
dlhdl.buildProcessor(hPC)
```

If targeting the Xilinx ZC706 board, replace 'zcu102_single' with 'zc706_single' in the first command above.

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx® Zynq® UltraScale+™ MPSoC ZCU102 board and the bitstream uses the single data type.

```
hW = dlhdl.Workflow(Network=snet,Bitstream='dlprocessor.bit',Target=hTarget);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment.

```
dn = compile(hW);
```

```
          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"        
    "OutputResultOffset"        "0x01800000"     "4.0 MB"         
    "SystemBufferOffset"        "0x01c00000"     "28.0 MB"        
    "InstructionDataOffset"     "0x03800000"     "4.0 MB"         
    "ConvWeightDataOffset"      "0x03c00000"     "16.0 MB"        
    "FCWeightDataOffset"        "0x04c00000"     "148.0 MB"       
    "EndOffset"                 "0x0e000000"     "Total: 224.0 MB"
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx® Zynq® UltraScale+ MPSoC ZCU102 hardware, run the `deploy` method of the `dlhdl.Workflow` object. This method programs the FPGA board using the output of the compile method and the programming file, downloads the network weights and biases, displays progress messages, and the time it takes to deploy the network.

```
deploy(hW)
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to FC Processor.
### 13% finished, current time is 28-Jun-2020 12:36:09.
### 25% finished, current time is 28-Jun-2020 12:36:10.
### 38% finished, current time is 28-Jun-2020 12:36:11.
### 50% finished, current time is 28-Jun-2020 12:36:12.
### 63% finished, current time is 28-Jun-2020 12:36:13.
### 75% finished, current time is 28-Jun-2020 12:36:14.
### 88% finished, current time is 28-Jun-2020 12:36:14.
### FC Weights loaded. Current time is 28-Jun-2020 12:36:15
```

Test Network

Run the `demoOnVideo` helper function. This function loads the example video, executes the `predict` method of the `dlhdl.Workflow` object, and then plots the result. See Helper Functions on page 10-20

```
demoOnVideo(hW,1);
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   24904175                  0.11320                       1          249
    conv_module            8967009                  0.04076
        conv1              1396633                  0.00635
        norm1               623003                  0.00283
        pool1               226855                  0.00103
        conv2              3410044                  0.01550
        norm2               378531                  0.00172
        pool2               233635                  0.00106
        conv3              1139419                  0.00518
        conv4               892918                  0.00406
        conv5               615897                  0.00280
        pool5                50189                  0.00023
    fc_module             15937166                  0.07244
        fc6               15819257                  0.07191
        fcLane1             117125                  0.00053
        fcLane2                782                  0.00000
 * The clock frequency of the DL processor is: 220MHz
```

Helper Functions

```
function demoOnVideo (hW, frameLimit)

if nargin < 2
    frameLimit = 1000000;
end

writeToFile = false;

videoFile = 'caltech_cordova1.avi';

if ~isfile(videoFile)
      url = append('https://www.mathworks.com/supportfiles/gpucoder/media/', videoFile);
       websave('caltech_cordova1.avi', url);
end

ss = getLaneDetectionData();

sensor = caltechMonoCamera();

%Initialize video readers and writers
vR = VideoReader(videoFile);
vPlayer = vision.DeployableVideoPlayer();
```

```
if writeToFile
    [~, name, ext] = fileparts(videoFile);
    outFileName = [name '_out' ext];
    vW = VideoWriter(outFileName);
    vW.FrameRate = vR.FrameRate;
    open(vW);
end

isOpen = true;

frameCount = 0;
while frameCount < frameLimit && isOpen && hasFrame(vR)
    testImg = readFrame(vR);
    inputImg = imresize(testImg, [227 227]);

%     profile off
    outputs = hW.predict(inputImg, 'Profile', 'on');

    laneim = showNetworkOutputs(testImg, outputs, ss.laneCoeffMeans, ss.laneCoeffsStds, sensor);
    step(vPlayer, laneim);
    frameCount = frameCount + 1;
    if writeToFile
        writeVideo(vW, laneim);
    end
    isOpen = vPlayer.isOpen();
end

if writeToFile
    close(vW);
end

release(vPlayer);
delete(vR);

end

function laneim = showNetworkOutputs(img, lanecoeffsNetworkOutput, laneCoeffMeans, laneCoeffStds
%

params = lanecoeffsNetworkOutput .* laneCoeffStds + laneCoeffMeans;

isRightLaneFound = abs(params(6)) > 0.5;
isLeftLaneFound =  abs(params(3)) > 0.5;

if isRightLaneFound
    rtBoundary = parabolicLaneBoundary(params(4:6));
else
    rtBoundary = parabolicLaneBoundary.empty(1, 0);
end

if isLeftLaneFound
    ltBoundary = parabolicLaneBoundary(params(1:3));
else
    ltBoundary = parabolicLaneBoundary.empty(1, 0);
end

laneboundaries = [ltBoundary, rtBoundary];
```

```
vehicleXPoints = 3:30;
laneim = insertLaneBoundary(img,laneboundaries,sensor, vehicleXPoints, 'Color', 'green');

end
```

See Also

dlhdl.Workflow | dlhdl.Target | compile | deploy | predict

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2
- “Start Deep Learning Faster Using Transfer Learning”

Image Category Classification by Using Deep Learning

This example shows you how to create, compile, and deploy a `dlhdl.Workflow` object with ResNet-18 as the network object by using the Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC. Use MATLAB® to retrieve the prediction results from the target device. ResNet-18 is a pretrained convolutional neural network that has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee, mug, pencil,and many animals). You can also use VGG-19 and DarkNet-19 as the network objects.

Prerequisites

- Xilinx ZCU102 SoC Development Kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™ Model for ResNet-18 Network
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Load the Pretrained Network

To load the pretrained Directed Acyclic Graph (DAG) network resnet18, enter:

```
net = resnet18;
```

To load the pretrained series network vgg19, enter:

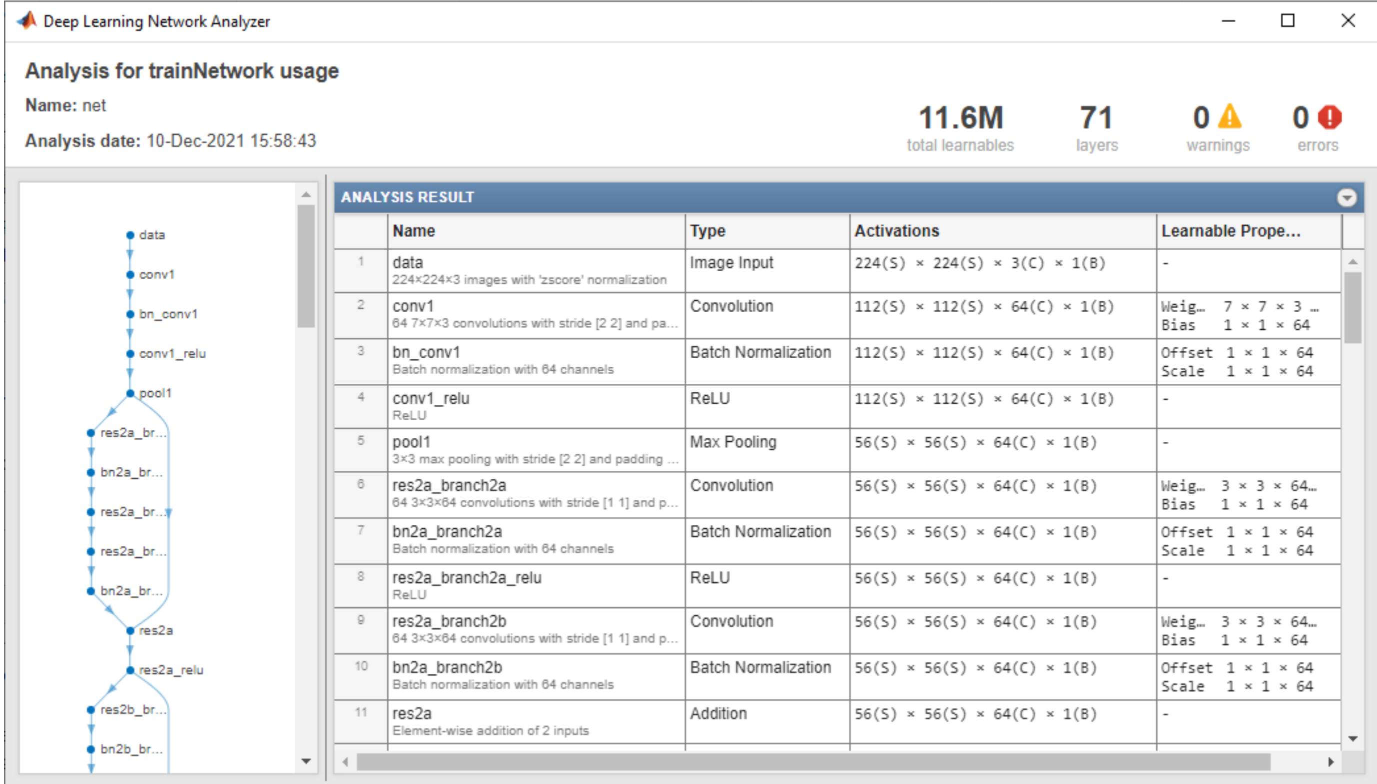
```
% net = vgg19;
```

To load the pretrained series network darknet19, enter:

```
% net = darknet19;
```

The pretrained ResNet-18 network contains 71 layers including the input, convolution, batch normalization, ReLU, max pooling, addition, global average pooling, fully connected, and the softmax layers. To view the layers of the pretrained ResNet-18 network, enter:

```
analyzeNetwork(net)
```



Create Target Object

Use the `dlhdl.Target` class to create a target object with a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use JTAG,Install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
```

```
hTarget = dlhdl.Target('Xilinx', 'Interface', 'Ethernet');
```

Create WorkFlow Object

Use the `dlhdl.Workflow` class to create an object. When you create the object, specify the network and the bitstream name. Specify the saved pretrained ResNet-18 neural network as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx ZCU102 SoC board. The bitstream uses a single data type.

```
hW = dlhdl.Workflow('Network', net, 'Bitstream', 'zcu102_single', 'Target', hTarget);
```

Compile the ResNet-18 DAG network

To compile the ResNet-18 DAG network, run the compile method of the `dlhdl.Workflow` object. You can optionally specify the maximum number of input frames. You can also optionally specify the input image normalization to happen in software.

```
dn = compile(hW, 'InputFrameNumberLimit', 15, 'HardwareNormalization', 'off')
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### The network includes the following layers:
     1   'data'                              Image Input                   224×224×3 images with
     2   'conv1'                             Convolution                   64 7×7×3 convolutions w
     3   'bn_conv1'                          Batch Normalization           Batch normalization wit
     4   'conv1_relu'                        ReLU                          ReLU
     5   'pool1'                             Max Pooling                   3×3 max pooling with st
     6   'res2a_branch2a'                    Convolution                   64 3×3×64 convolutions
     7   'bn2a_branch2a'                     Batch Normalization           Batch normalization wit
     8   'res2a_branch2a_relu'               ReLU                          ReLU
     9   'res2a_branch2b'                    Convolution                   64 3×3×64 convolutions
    10   'bn2a_branch2b'                     Batch Normalization           Batch normalization wit
    11   'res2a'                             Addition                      Element-wise addition o
    12   'res2a_relu'                        ReLU                          ReLU
    13   'res2b_branch2a'                    Convolution                   64 3×3×64 convolutions
    14   'bn2b_branch2a'                     Batch Normalization           Batch normalization wit
    15   'res2b_branch2a_relu'               ReLU                          ReLU
    16   'res2b_branch2b'                    Convolution                   64 3×3×64 convolutions
    17   'bn2b_branch2b'                     Batch Normalization           Batch normalization wit
    18   'res2b'                             Addition                      Element-wise addition o
    19   'res2b_relu'                        ReLU                          ReLU
    20   'res3a_branch2a'                    Convolution                   128 3×3×64 convolutions
    21   'bn3a_branch2a'                     Batch Normalization           Batch normalization wit
    22   'res3a_branch2a_relu'               ReLU                          ReLU
    23   'res3a_branch2b'                    Convolution                   128 3×3×128 convolution
    24   'bn3a_branch2b'                     Batch Normalization           Batch normalization wit
    25   'res3a'                             Addition                      Element-wise addition o
    26   'res3a_relu'                        ReLU                          ReLU
    27   'res3a_branch1'                     Convolution                   128 1×1×64 convolutions
    28   'bn3a_branch1'                      Batch Normalization           Batch normalization wit
    29   'res3b_branch2a'                    Convolution                   128 3×3×128 convolution
    30   'bn3b_branch2a'                     Batch Normalization           Batch normalization wit
    31   'res3b_branch2a_relu'               ReLU                          ReLU
    32   'res3b_branch2b'                    Convolution                   128 3×3×128 convolution
    33   'bn3b_branch2b'                     Batch Normalization           Batch normalization wit
    34   'res3b'                             Addition                      Element-wise addition o
    35   'res3b_relu'                        ReLU                          ReLU
    36   'res4a_branch2a'                    Convolution                   256 3×3×128 convolution
    37   'bn4a_branch2a'                     Batch Normalization           Batch normalization wit
    38   'res4a_branch2a_relu'               ReLU                          ReLU
    39   'res4a_branch2b'                    Convolution                   256 3×3×256 convolution
    40   'bn4a_branch2b'                     Batch Normalization           Batch normalization wit
    41   'res4a'                             Addition                      Element-wise addition o
    42   'res4a_relu'                        ReLU                          ReLU
    43   'res4a_branch1'                     Convolution                   256 1×1×128 convolution
    44   'bn4a_branch1'                      Batch Normalization           Batch normalization wit
    45   'res4b_branch2a'                    Convolution                   256 3×3×256 convolution
    46   'bn4b_branch2a'                     Batch Normalization           Batch normalization wit
    47   'res4b_branch2a_relu'               ReLU                          ReLU
    48   'res4b_branch2b'                    Convolution                   256 3×3×256 convolution
    49   'bn4b_branch2b'                     Batch Normalization           Batch normalization wit
    50   'res4b'                             Addition                      Element-wise addition o
    51   'res4b_relu'                        ReLU                          ReLU
    52   'res5a_branch2a'                    Convolution                   512 3×3×256 convolution
    53   'bn5a_branch2a'                     Batch Normalization           Batch normalization wit
    54   'res5a_branch2a_relu'               ReLU                          ReLU
    55   'res5a_branch2b'                    Convolution                   512 3×3×512 convolution
```

```
    56   'bn5a_branch2b'                     Batch Normalization         Batch normalization wi
    57   'res5a'                             Addition                    Element-wise addition 
    58   'res5a_relu'                        ReLU                        ReLU
    59   'res5a_branch1'                     Convolution                 512 1×1×256 convolutio
    60   'bn5a_branch1'                      Batch Normalization         Batch normalization wi
    61   'res5b_branch2a'                    Convolution                 512 3×3×512 convolutio
    62   'bn5b_branch2a'                     Batch Normalization         Batch normalization wi
    63   'res5b_branch2a_relu'               ReLU                        ReLU
    64   'res5b_branch2b'                    Convolution                 512 3×3×512 convolutio
    65   'bn5b_branch2b'                     Batch Normalization         Batch normalization wi
    66   'res5b'                             Addition                    Element-wise addition 
    67   'res5b_relu'                        ReLU                        ReLU
    68   'pool5'                             2-D Global Average Pooling   2-D global average poo
    69   'fc1000'                            Fully Connected             1000 fully connected l
    70   'prob'                              Softmax                     softmax
    71   'ClassificationLayer_predictions'   Classification Output       crossentropyex with 't

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification
### Compiling layer group: conv1>>pool1 ...
### Compiling layer group: conv1>>pool1 ... complete.
### Compiling layer group: res2a_branch2a>>res2a_branch2b ...
### Compiling layer group: res2a_branch2a>>res2a_branch2b ... complete.
### Compiling layer group: res2b_branch2a>>res2b_branch2b ...
### Compiling layer group: res2b_branch2a>>res2b_branch2b ... complete.
### Compiling layer group: res3a_branch1 ...
### Compiling layer group: res3a_branch1 ... complete.
### Compiling layer group: res3a_branch2a>>res3a_branch2b ...
### Compiling layer group: res3a_branch2a>>res3a_branch2b ... complete.
### Compiling layer group: res3b_branch2a>>res3b_branch2b ...
### Compiling layer group: res3b_branch2a>>res3b_branch2b ... complete.
### Compiling layer group: res4a_branch1 ...
### Compiling layer group: res4a_branch1 ... complete.
### Compiling layer group: res4a_branch2a>>res4a_branch2b ...
### Compiling layer group: res4a_branch2a>>res4a_branch2b ... complete.
### Compiling layer group: res4b_branch2a>>res4b_branch2b ...
### Compiling layer group: res4b_branch2a>>res4b_branch2b ... complete.
### Compiling layer group: res5a_branch1 ...
### Compiling layer group: res5a_branch1 ... complete.
### Compiling layer group: res5a_branch2a>>res5a_branch2b ...
### Compiling layer group: res5a_branch2a>>res5a_branch2b ... complete.
### Compiling layer group: res5b_branch2a>>res5b_branch2b ...
### Compiling layer group: res5b_branch2a>>res5b_branch2b ... complete.
### Compiling layer group: pool5 ...
### Compiling layer group: pool5 ... complete.
### Compiling layer group: fc1000 ...
### Compiling layer group: fc1000 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "12.0 MB"        
    "OutputResultOffset"        "0x00c00000"     "4.0 MB"         
    "SchedulerDataOffset"       "0x01000000"     "4.0 MB"         
```

```
    "SystemBufferOffset"        "0x01400000"     "28.0 MB"
    "InstructionDataOffset"     "0x03000000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03400000"     "52.0 MB"
    "FCWeightDataOffset"        "0x06800000"     "4.0 MB"
    "EndOffset"                 "0x06c00000"     "Total: 108.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

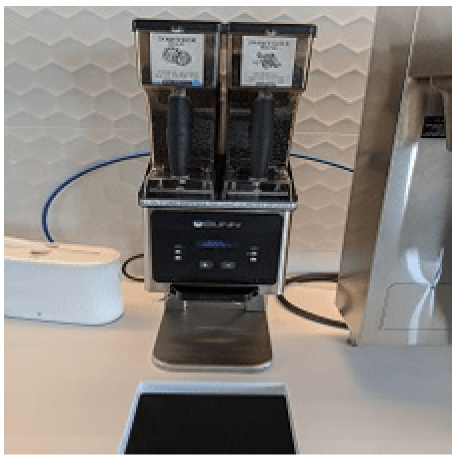
```
deploy(hW)
```

```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'

Downloading target FPGA device configuration over Ethernet to SD card done. The system will now r


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 10-Dec-2021 16:01:37
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 10-Dec-2021 16:01:37
```

Load Image for Prediction

Load the example image.

```
imgFile = 'espressomaker.jpg';
inputImg = imresize(imread(imgFile), [224,224]);
imshow(inputImg)
```



Run Prediction for One Image

Execute the predict method on the `dlhdl.Workflow` object and then show the label in the MATLAB command window.

```
[prediction, speed] = predict(hW,single(inputImg),'Profile','on');
```

```
### Finished writing input activations.
### Running single input activation.
```

```
Deep Learning Processor Profiler Performance Results
```

```
                     LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                           -------------             -------------              ---------        --
Network                     24100982                  0.10955                       1          241
    conv1                    2225590                  0.01012
    pool1                     577207                  0.00262
    res2a_branch2a            973263                  0.00442
    res2a_branch2b            973083                  0.00442
    res2a                     307582                  0.00140
    res2b_branch2a            973221                  0.00442
    res2b_branch2b            973548                  0.00443
    res2b                     307602                  0.00140
    res3a_branch1             541072                  0.00246
    res3a_branch2a            749668                  0.00341
    res3a_branch2b            908194                  0.00413
    res3a                     153885                  0.00070
    res3b_branch2a            908013                  0.00413
    res3b_branch2b            907705                  0.00413
    res3b                     153935                  0.00070
    res4a_branch1             491540                  0.00223
    res4a_branch2a            491680                  0.00223
    res4a_branch2b            889776                  0.00404
    res4a                      77044                  0.00035
    res4b_branch2a            889897                  0.00404
    res4b_branch2b            889873                  0.00404
    res4b                      77053                  0.00035
    res5a_branch1            1057762                  0.00481
    res5a_branch2a           1057907                  0.00481
    res5a_branch2b           2058997                  0.00936
    res5a                      38602                  0.00018
    res5b_branch2a           2058860                  0.00936
    res5b_branch2b           2059549                  0.00936
    res5b                      38704                  0.00018
    pool5                      73721                  0.00034
    fc1000                    216262                  0.00098
 * The clock frequency of the DL processor is: 220MHz
```

```
[val, idx] = max(prediction);
net.Layers(end).ClassNames{idx}
```

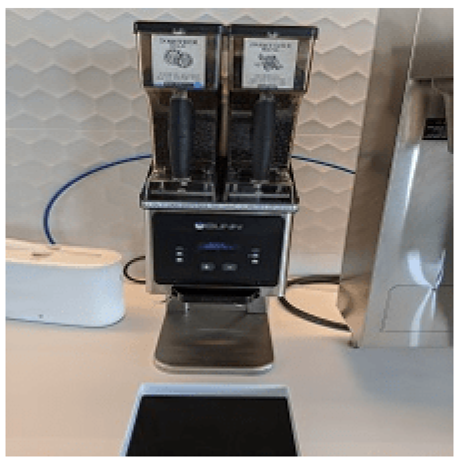
```
ans =
'Polaroid camera'
```

Run Prediction for Multiple Images

Load multiple images and retrieve their prediction reults by using the mulltiple frame support feature. For more information, see “Multiple Frame Support” on page 5-7.

The `demoOnImage` function loads multiple images and retrieves their prediction results. The `annotateresults` function displays the image prediction result on top of the images which are assembled into a 3-by-5 array.

```
imshow(inputImg)
```



```
demoOnImage;
```

```
### Finished writing input activations.
### Running in multi-frame mode with 15 inputs.
FPGA PREDICTION: binder
FPGA PREDICTION: file
FPGA PREDICTION: barber chair
FPGA PREDICTION: mixing bowl
FPGA PREDICTION: washbasin
FPGA PREDICTION: desk
FPGA PREDICTION: envelope
FPGA PREDICTION: Polaroid camera
FPGA PREDICTION: typewriter keyboard
FPGA PREDICTION: monitor
FPGA PREDICTION: sunglass
FPGA PREDICTION: ballpoint
FPGA PREDICTION: can opener
FPGA PREDICTION: analog clock
FPGA PREDICTION: ashcan
```

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `compile` | `deploy` | `predict` | **Deep Network Designer**

More About

- "Prototype Deep Learning Networks on FPGA and SoC Devices" on page 5-2

Defect Detection

This example shows how to deploy a custom trained series network to detect defects in objects such as hexagon nuts. The custom networks were trained by using transfer learning. Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training signals. This example uses two trained series networks, `trainedDefNet.mat` and `trainedBlemDetNet.mat`.

Prerequisites

- Xilinx ZCU102 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Load Pretrained Networks

Load the custom pretrained series network `trainedDefNet`.

```
if ~isfile('trainedDefNet.mat')
        url = 'https://www.mathworks.com/supportfiles/dlhdl/trainedDefNet.mat';
        websave('trainedDefNet.mat',url);
    end
    net1 = load('trainedDefNet.mat');
   snet_defnet = net1.custom_alexnet
```

```
snet_defnet = 
  SeriesNetwork with properties:

         Layers: [25×1 nnet.cnn.layer.Layer]
     InputNames: {'data'}
    OutputNames: {'output'}
```

Analyze the network. `analyzeNetwork` displays an interactive plot of the network architecture and a table containing information about the network layers.

```
    analyzeNetwork(snet_defnet)
```

Load the network `snet_blemdetnet`.

```
if ~isfile('trainedBlemDetNet.mat')
        url = 'https://www.mathworks.com/supportfiles/dlhdl/trainedBlemDetNet.mat';
        websave('trainedBlemDetNet.mat',url);
    end
    net2 = load('trainedBlemDetNet.mat');
    snet_blemdetnet = net2.convnet
```

```
snet_blemdetnet = 
  SeriesNetwork with properties:
```

```
        Layers: [12×1 nnet.cnn.layer.Layer]
    InputNames: {'imageinput'}
   OutputNames: {'classoutput'}
```

Analyze the network. `analyzeNetwork` displays an interactive plot of the network architecture and a table containing information about the network layers.

```
analyzeNetwork(snet_blemdetnet)
```

Create Target Object

Create a target object that has a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use the JTAG connection, install the Xilinx™ Vivado™ Design Suite 2020.2.

Set the Xilinx Vivado toolpath.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
```

```
hT = dlhdl.Target('Xilinx','Interface','Ethernet')
```

```
hT =
  Target with properties:

       Vendor: 'Xilinx'
    Interface: Ethernet
    IPAddress: '192.168.1.101'
     Username: 'root'
         Port: 22
```

Generate Bitstream to Run Network

The defect detection network consists of multiple Cross Channel Normalization layers. To support this layer on hardware, the 'LRNBlockGeneration' property of the conv module needs to be turned on in the bitstream used for FPGA inference. The shipping zcu102_single bitstream does not have this property turned on. A new bitstream can be generated using the following lines of code. The generated bitstream can be used along with a dlhdl.Workflow object for inference.

When creating a dlhdl.ProcessorConfig object for an existing shipping bitstream, make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example the target FPGA board is the Xilinx ZCU102 SoC board and the date type is single. Update the processor configuration with 'LRNBlockGeneration' turned on and 'SegmentationBlockGeneration' turned off. Turn the latter off to fit the Deep Learning IP on the FPGA and avoid overutilization of resources.

```
hPC = dlhdl.ProcessorConfig('Bitstream', 'zcu102_single');
hPC.setModuleProperty('conv', 'LRNBlockGeneration', 'on');
hPC.setModuleProperty('conv', 'SegmentationBlockGeneration', 'off');
dlhdl.buildProcessor(hPC)
```

To learn how to use the generated bitstream file, see “Generate Custom Bitstream” on page 9-2.

Create Workflow Object for trainedDefNet Network

Create an object of the `dlhdl.Workflow` class. When you create the class, specify the network and the bitstream name. Make sure to use the generated bitstream which enables processing of Cross

Channel Normalization layers on the FPGA. Specify the saved pretrained neural network, snet_defnet, as the network.

```
hW = dlhdl.Workflow('Network',snet_defnet,'Bitstream','dlprocessor.bit','Target',hT);
```

Compile trainedDefNet Series Network

Run the compile function of the `dlhdl.Workflow` object.

```
hW.compile
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single ...
### The network includes the following layers:

     1   'data'     Image Input                   128×128×1 images with 'zerocenter' normalizatio
     2   'conv1'    Convolution                   96 11×11×1 convolutions with stride [4  4] and
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 channels per
     5   'pool1'    Max Pooling                   3×3 max pooling with stride [2  2] and padding
     6   'conv2'    Grouped Convolution           2 groups of 128 5×5×48 convolutions with stride
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 channels per
     9   'pool2'    Max Pooling                   3×3 max pooling with stride [2  2] and padding
    10   'conv3'    Convolution                   384 3×3×256 convolutions with stride [1  1] and
    11   'relu3'    ReLU                          ReLU
    12   'conv4'    Grouped Convolution           2 groups of 192 3×3×192 convolutions with strid
    13   'relu4'    ReLU                          ReLU
    14   'conv5'    Grouped Convolution           2 groups of 128 3×3×192 convolutions with strid
    15   'relu5'    ReLU                          ReLU
    16   'pool5'    Max Pooling                   3×3 max pooling with stride [2  2] and padding
    17   'fc6'      Fully Connected               4096 fully connected layer
    18   'relu6'    ReLU                          ReLU
    19   'drop6'    Dropout                       50% dropout
    20   'fc7'      Fully Connected               4096 fully connected layer
    21   'relu7'    ReLU                          ReLU
    22   'drop7'    Dropout                       50% dropout
    23   'fc8'      Fully Connected               2 fully connected layer
    24   'prob'     Softmax                       softmax
    25   'output'   Classification Output         crossentropyex with classes 'ng' and 'ok'

3 Memory Regions created.

Skipping: data
Compiling leg: conv1>>pool5 ...
Compiling leg: conv1>>pool5 ... complete.
Compiling leg: fc6>>fc8 ...
Compiling leg: fc6>>fc8 ... complete.
Skipping: prob
Skipping: output
Creating Schedule...
.......
Creating Schedule...complete.
Creating Status Table...
......
Creating Status Table...complete.
Emitting Schedule...
......
```

```
Emitting Schedule...complete.
Emitting Status Table...
........
Emitting Status Table...complete.

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "8.0 MB"         
    "OutputResultOffset"        "0x00800000"     "4.0 MB"         
    "SchedulerDataOffset"       "0x00c00000"     "4.0 MB"         
    "SystemBufferOffset"        "0x01000000"     "28.0 MB"        
    "InstructionDataOffset"     "0x02c00000"     "4.0 MB"         
    "ConvWeightDataOffset"      "0x03000000"     "12.0 MB"        
    "FCWeightDataOffset"        "0x03c00000"     "84.0 MB"        
    "EndOffset"                 "0x09000000"     "Total: 144.0 MB"

### Network compilation complete.

ans = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming the FPGA device and displays progress messages and the time it takes to deploy the network.

```
hW.deploy
```

```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'

Downloading target FPGA device configuration over Ethernet to SD card done. The system will now 


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Dec-2020 16:16:31
### Loading weights to FC Processor.
### 20% finished, current time is 16-Dec-2020 16:16:32.
### 40% finished, current time is 16-Dec-2020 16:16:32.
### 60% finished, current time is 16-Dec-2020 16:16:33.
```

```
### 80% finished, current time is 16-Dec-2020 16:16:34.
### FC Weights loaded. Current time is 16-Dec-2020 16:16:34
```

Run Prediction for One Image

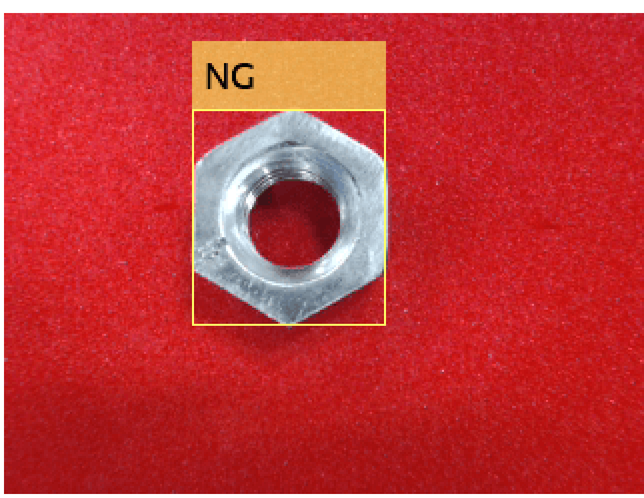
Load an image from the attached `testImages` folder and resize the image to match the network image input layer dimensions. Run the predict function of the `dlhdl.Workflow` object to retrieve and display the defect prediction from the FPGA.

```
wi = uint32(320);
he = uint32(240);
ch = uint32(3);
filename = fullfile(pwd,'ng1.png');
img=imread(filename);
img = imresize(img, [he, wi]);
img = mat2ocv(img);

    % Extract ROI for preprocessing
    [Iori, imgPacked, num, bbox] = myNDNet_Preprocess(img);

    % Row-major to column-major conversion
    imgPacked2 = zeros([128,128,4],'uint8');
    for c = 1:4
        for i = 1:128
            for j = 1:128
                imgPacked2(i,j,c) = imgPacked((i-1)*128 + (j-1) + (c-1)*128*128 + 1);
            end
        end
    end

    % Classify detected nuts by using CNN
    scores = zeros(2,4);
    for i = 1:num
         [scores(:,i), speed] = hW.predict(single(imgPacked2(:,:,i)),'Profile','on');
    end
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   12231156                  0.05560                       1          122
    conv1                   414021                  0.00188
    norm1                   172325                  0.00078
    pool1                    56747                  0.00026
    conv2                   654112                  0.00297
    norm2                   119403                  0.00054
    pool2                    43611                  0.00020
    conv3                   777446                  0.00353
    conv4                   595551                  0.00271
    conv5                   404425                  0.00184
    pool5                    17831                  0.00008
    fc6                    1759699                  0.00800
    fc7                    7030188                  0.03196
```

```
    fc8                        185672                    0.00084
 * The clock frequency of the DL processor is: 220MHz
```

```
Iori = reshape(Iori, [1, he*wi*ch]);
bbox = reshape(bbox, [1,16]);
scores = reshape(scores, [1, 8]);

% Insert an annotation for postprocessing
out = myNDNet_Postprocess(Iori, num, bbox, scores, wi, he, ch);

sz = [he wi ch];
out = ocv2mat(out,sz);
imshow(out)
```



Create Workflow Object for `trainedBlemDetNet` Network

Create an object of the `dlhdl.Workflow` class. When you create the class, specify the network and the bitstream name. Make sure to use the generated bitstream which enables processing of Cross Channel Normalization layers on the FPGA. Specify the saved pretrained neural network, `trainedblemDetNet`, as the network.

```
hW = dlhdl.Workflow('Network',snet_blemdetnet,'Bitstream','dlprocessor.bit','Target',hT)
```

Compile trainedBlemDetNet Series Network

Run the compile function of the `dlhdl.Workflow` object.

```
hW.compile
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single ...
### The network includes the following layers:
```

```
     1   'imageinput'    Image Input                   128×128×1 images with 'zerocenter' normali
     2   'conv_1'        Convolution                   20 5×5×1 convolutions with stride [1  1] a
     3   'relu_1'        ReLU                          ReLU
     4   'maxpool_1'     Max Pooling                   2×2 max pooling with stride [2  2] and pad
     5   'crossnorm'     Cross Channel Normalization   cross channel normalization with 5 channel
     6   'conv_2'        Convolution                   20 5×5×20 convolutions with stride [1  1]
     7   'relu_2'        ReLU                          ReLU
     8   'maxpool_2'     Max Pooling                   2×2 max pooling with stride [2  2] and pad
     9   'fc_1'          Fully Connected               512 fully connected layer
    10   'fc_2'          Fully Connected               2 fully connected layer
    11   'softmax'       Softmax                       softmax
    12   'classoutput'   Classification Output         crossentropyex with classes 'ng' and 'ok'

3 Memory Regions created.

Skipping: imageinput
Compiling leg: conv_1>>maxpool_2 ...
Compiling leg: conv_1>>maxpool_2 ... complete.
Compiling leg: fc_1>>fc_2 ...
Compiling leg: fc_1>>fc_2 ... complete.
Skipping: softmax
Skipping: classoutput
Creating Schedule...
.......
Creating Schedule...complete.
Creating Status Table...
......
Creating Status Table...complete.
Emitting Schedule...
......
Emitting Schedule...complete.
Emitting Status Table...
........
Emitting Status Table...complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "8.0 MB"
    "OutputResultOffset"        "0x00800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x01000000"     "28.0 MB"
    "InstructionDataOffset"     "0x02c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03000000"     "4.0 MB"
    "FCWeightDataOffset"        "0x03400000"     "36.0 MB"
    "EndOffset"                 "0x05800000"     "Total: 88.0 MB"

### Network compilation complete.

ans = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming the FPGA device and displays progress messages and the time it takes to deploy the network.

```
hW.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Dec-2020 16:16:47
### Loading weights to FC Processor.
### 50% finished, current time is 16-Dec-2020 16:16:48.
### FC Weights loaded. Current time is 16-Dec-2020 16:16:48
```

Run Prediction for One Image

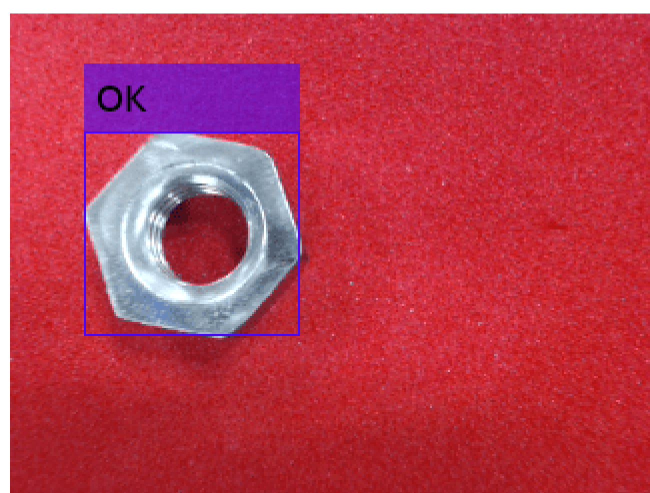
Load an image from the attached `testImages` folder and resize the image to match the network image input layer dimensions. Run the predict function of the `dlhdl.Workflow` object to retrieve and display the defect prediction from the FPGA.

```
wi = uint32(320);
he = uint32(240);
ch = uint32(3);

filename = fullfile(pwd,'ok1.png');
img=imread(filename);
img = imresize(img, [he, wi]);
img = mat2ocv(img);

    % Extract ROI for preprocessing
    [Iori, imgPacked, num, bbox] = myNDNet_Preprocess(img);

    % Row-major to column-major conversion
    imgPacked2 = zeros([128,128,4],'uint8');
    for c = 1:4
        for i = 1:128
            for j = 1:128
                imgPacked2(i,j,c) = imgPacked((i-1)*128 + (j-1) + (c-1)*128*128 + 1);
            end
        end
    end

    % classify detected nuts by using CNN
    scores = zeros(2,4);
    for i = 1:num
         [scores(:,i), speed] = hW.predict(single(imgPacked2(:,:,i)),'Profile','on');
    end
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
```

```
                             -------------          -------------          ---------       --
Network                       4892622               0.02224                    1          48
    conv_1                     467921               0.00213
    maxpool_1                  188086               0.00085
    crossnorm                  159500               0.00072
    conv_2                     397561               0.00181
    maxpool_2                   41455               0.00019
    fc_1                      3614625               0.01643
    fc_2                        23355               0.00011
 * The clock frequency of the DL processor is: 220MHz
```

```
    Iori = reshape(Iori, [1, he*wi*ch]);
    bbox = reshape(bbox, [1,16]);
    scores = reshape(scores, [1, 8]);

    % Insert annotation for postprocessing
    out = myNDNet_Postprocess(Iori, num, bbox, scores, wi, he, ch);

    sz = [he wi ch];
    out = ocv2mat(out,sz);
    imshow(out)
```



Quantize and Deploy `trainedBlemDetNet` Network

The `trainedBlemDetNet` network improves performance to 45 frames per second. The target performance of the deployed network is 100 frames per second while staying within the target resource utilization budget. The resource utilization budget takes into consideration parameters such as memory size and onboard IO. While you can increase the resource utilization budget by choosing a larger board, doing so increases the cost. Instead, improve the deployed network performance and stay within the resource utilization budget by quantizing the network. Quantize and deploy the `trainedBlemDetNet` network.

Load the data set as an image datastore. The imageDatastore labels the images based on folder names and stores the data. Divide the data into calibration and validation data sets. Use 50% of the images for calibration and 50% of the images for validation. Expedite the calibration and validation process by using a subset of the calibration and validation image sets.

```
if ~isfile('dataSet.zip')
        url = 'https://www.mathworks.com/supportfiles/dlhdl/dataSet.zip';
        websave('dataSet.zip',url);
end
unzip('dataSet.zip')
unzip('dataset.zip')
imageData = imageDatastore(fullfile('dataset'),...
'IncludeSubfolders',true,'FileExtensions','.PNG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
calibrationData_reduced = calibrationData.subset(1:20);
validationData_reduced = validationData.subset(1:1);
```

Create a quantized network by using the `dlquantizer` object. Set the target execution environment to FPGA.

```
dlQuantObj = dlquantizer(snet_blemdetnet,'ExecutionEnvironment','FPGA')
```

```
dlQuantObj = 
  dlquantizer with properties:

           NetworkObject: [1×1 SeriesNetwork]
    ExecutionEnvironment: 'FPGA'
```

Use the `calibrate` function to exercise the network by using sample inputs and collect the range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The `calibrate` function returns a table. Each row of the table contains range information for a learnable parameter of the quantized network.

```
dlQuantObj.calibrate(calibrationData_reduced)
```

```
ans=21×5 table
        Optimized Layer Name          Network Layer Name    Learnables / Activations     MinValue
    ____________________________    __________________    ________________________    __________

    {'conv_1_Weights'          }      {'conv_1'    }           "Weights"                -0.29022
    {'conv_1_Bias'             }      {'conv_1'    }           "Bias"                  -0.021907
    {'conv_2_Weights'          }      {'conv_2'    }           "Weights"                -0.10499
    {'conv_2_Bias'             }      {'conv_2'    }           "Bias"                  -0.010084
    {'fc_1_Weights'            }      {'fc_1'      }           "Weights"               -0.051599
    {'fc_1_Bias'               }      {'fc_1'      }           "Bias"                 -0.0048897
    {'fc_2_Weights'            }      {'fc_2'      }           "Weights"               -0.071356
    {'fc_2_Bias'               }      {'fc_2'      }           "Bias"                  -0.062086
    {'imageinput'              }      {'imageinput'}           "Activations"                   0
    {'imageinput_normalization'}      {'imageinput'}           "Activations"             -184.37
    {'conv_1'                  }      {'conv_1'    }           "Activations"             -112.18
    {'relu_1'                  }      {'relu_1'    }           "Activations"                   0
    {'maxpool_1'               }      {'maxpool_1' }           "Activations"                   0
    {'crossnorm'               }      {'crossnorm' }           "Activations"                   0
    {'conv_2'                  }      {'conv_2'    }           "Activations"             -117.79
    {'relu_2'                  }      {'relu_2'    }           "Activations"                   0
```

⋮

The `trainedBlemDetNet` network consists of a Cross Channel Normalization layer. To support this layer on hardware, the 'LRNBlockGeneration' property of the conv module needs to be turned on in the bitstream used for FPGA inference. The shipping zcu102_int8 bitstream does not have this property turned on. A new bitstream can be generated using the following lines of code. The generated bitstream can be used along with a dlhdl.Workflow object for inference.

When creating a dlhdl.ProcessorConfig object for an existing shipping bitstream, make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example the target FPGA board is the Xilinx ZCU102 SoC board and the date type is int8. Update the processor configuration with 'LRNBlockGeneration' turned on and 'SegmentationBlockGeneration' turned off. Turn the latter off to fit the Deep Learning IP on the FPGA and avoid overutilization of resources.

```
% hPC = dlhdl.ProcessorConfig('Bitstream', 'zcu102_int8');
% hPC.setModuleProperty('conv', 'LRNBlockGeneration', 'on');
% hPC.setModuleProperty('conv', 'SegmentationBlockGeneration', 'off');
% dlhdl.buildProcessor(hPC)
```

To learn how to use the generated bitstream file, see “Generate Custom Bitstream” on page 9-2.

Create an object of the `dlhdl.Workflow` class. When you create the class, specify the network and the bitstream name. Make sure to use this newly generated bitstream which enables processing of Cross Channel Normalization layers on the FPGA. Specify the saved pretrained quantized `trainedblemDetNet` object `dlQuantObj` as the network.

```
hW = dlhdl.Workflow('Network', dlQuantObj, 'Bitstream', 'dlprocessor.bit','Target',hT);
```

To compile the quantized network, run the compile function of the `dlhdl.Workflow` object.

```
hW.compile('InputFrameNumberLimit',30)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8 ...
### The network includes the following layers:

     1   'imageinput'    Image Input                    128×128×1 images with 'zerocenter' normali
     2   'conv_1'        Convolution                    20 5×5×1 convolutions with stride [1  1] a
     3   'relu_1'        ReLU                           ReLU
     4   'maxpool_1'     Max Pooling                    2×2 max pooling with stride [2  2] and pad
     5   'crossnorm'     Cross Channel Normalization    cross channel normalization with 5 channe
     6   'conv_2'        Convolution                    20 5×5×20 convolutions with stride [1  1]
     7   'relu_2'        ReLU                           ReLU
     8   'maxpool_2'     Max Pooling                    2×2 max pooling with stride [2  2] and pad
     9   'fc_1'          Fully Connected                512 fully connected layer
    10   'fc_2'          Fully Connected                2 fully connected layer
    11   'softmax'       Softmax                        softmax
    12   'classoutput'   Classification Output          crossentropyex with classes 'ng' and 'ok'

3 Memory Regions created.

Skipping: imageinput
Compiling leg: conv_1>>maxpool_2 ...
Compiling leg: conv_1>>maxpool_2 ... complete.
Compiling leg: fc_1>>fc_2 ...
```

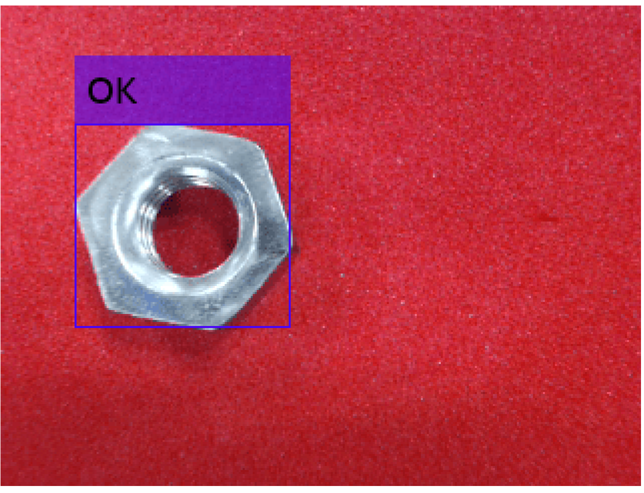
```
Compiling leg: fc_1>>fc_2 ... complete.
Skipping: softmax
Skipping: classoutput
Creating Schedule...
.........
Creating Schedule...complete.
Creating Status Table...
........
Creating Status Table...complete.
Emitting Schedule...
......
Emitting Schedule...complete.
Emitting Status Table...
..........
Emitting Status Table...complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    ________________

    "InputDataOffset"           "0x00000000"     "16.0 MB"
    "OutputResultOffset"        "0x01000000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01400000"     "4.0 MB"
    "SystemBufferOffset"        "0x01800000"     "28.0 MB"
    "InstructionDataOffset"     "0x03400000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03800000"     "4.0 MB"
    "FCWeightDataOffset"        "0x03c00000"     "12.0 MB"
    "EndOffset"                 "0x04800000"     "Total: 72.0 MB"

### Network compilation complete.

ans = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
```

To deploy the network on the Xilinx ZCU102 SoC hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming the FPGA device and displays progress messages and the time it takes to deploy the network.

```
hW.deploy
```

```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'

Downloading target FPGA device configuration over Ethernet to SD card done. The system will now
```

```
System is rebooting .
```




```
 . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Dec-2020 16:18:03
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 16-Dec-2020 16:18:03
```

Load an image from the attached `testImages` folder and resize the image to match the network image input layer dimensions. Run the predict function of the `dlhdl.Workflow` object to retrieve and display the defect prediction from the FPGA.

```
wi = uint32(320);
he = uint32(240);
ch = uint32(3);

filename = fullfile(pwd,'ok1.png');
img=imread(filename);
img = imresize(img, [he, wi]);
img = mat2ocv(img);

    % Extract ROI for preprocessing
    [Iori, imgPacked, num, bbox] = myNDNet_Preprocess(img);

    % row-major > column-major conversion
    imgPacked2 = zeros([128,128,4],'uint8');
    for c = 1:4
        for i = 1:128
            for j = 1:128
                imgPacked2(i,j,c) = imgPacked((i-1)*128 + (j-1) + (c-1)*128*128 + 1);
            end
        end
```

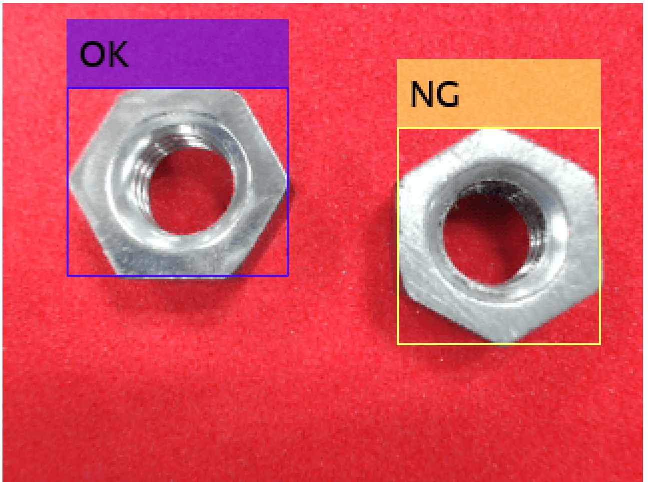
```
    end

    % classify detected nuts by using CNN
    scores = zeros(2,4);
    for i = 1:num
         [scores(:,i), speed] = hW.predict(single(imgPacked2(:,:,i)),'Profile','on');
    end

### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------            -------------              ---------        --
Network                    1754969                  0.00798                       1           17
    conv_1                  271340                  0.00123
    maxpool_1                87533                  0.00040
    crossnorm               125737                  0.00057
    conv_2                  149972                  0.00068
    maxpool_2                19657                  0.00009
    fc_1                   1085683                  0.00493
    fc_2                     14928                  0.00007
 * The clock frequency of the DL processor is: 220MHz


    Iori = reshape(Iori, [1, he*wi*ch]);
    bbox = reshape(bbox, [1,16]);
    scores = reshape(scores, [1, 8]);

    % Insert an annotation for postprocessing
    out = myNDNet_Postprocess(Iori, num, bbox, scores, wi, he, ch);

    sz = [he wi ch];
    out = ocv2mat(out,sz);
    imshow(out)
```

To test that the quantized network can identify all test cases deploy an additional image, resize the image to match the network image input layer dimensions, and run the predict function of the `dlhdl.Workflow` object to retrieve and display the defect prediction from the FPGA.

```
wi = uint32(320);
he = uint32(240);
ch = uint32(3);

filename = fullfile(pwd,'okng.png');
img=imread(filename);
img = imresize(img, [he, wi]);
img = mat2ocv(img);

    % Extract ROI for preprocessing
    [Iori, imgPacked, num, bbox] = myNDNet_Preprocess(img);

    % row-major > column-major conversion
    imgPacked2 = zeros([128,128,4],'uint8');
    for c = 1:4
        for i = 1:128
            for j = 1:128
                imgPacked2(i,j,c) = imgPacked((i-1)*128 + (j-1) + (c-1)*128*128 + 1);
            end
        end
    end

    % classify detected nuts by using CNN
    scores = zeros(2,4);
    for i = 1:num
         [scores(:,i), speed] = hW.predict(single(imgPacked2(:,:,i)),'Profile','on');
    end

### Finished writing input activations.
### Running single input activations.
```

```
              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    1754614                  0.00798                       1          17
    conv_1                  271184                  0.00123
    maxpool_1                87557                  0.00040
    crossnorm               125768                  0.00057
    conv_2                  149819                  0.00068
    maxpool_2                19602                  0.00009
    fc_1                   1085664                  0.00493
    fc_2                     14930                  0.00007
 * The clock frequency of the DL processor is: 220MHz


### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    1754486                  0.00797                       1          17
    conv_1                  271014                  0.00123
    maxpool_1                87662                  0.00040
    crossnorm               125835                  0.00057
    conv_2                  149789                  0.00068
    maxpool_2                19661                  0.00009
    fc_1                   1085505                  0.00493
    fc_2                     14930                  0.00007
 * The clock frequency of the DL processor is: 220MHz
```

```
    Iori = reshape(Iori, [1, he*wi*ch]);
    bbox = reshape(bbox, [1,16]);
    scores = reshape(scores, [1, 8]);

    % Insert an annotation for postprocessing
    out = myNDNet_Postprocess(Iori, num, bbox, scores, wi, he, ch);

    sz = [he wi ch];
    out = ocv2mat(out,sz);
    imshow(out)
```



Quantizing the network improves the performance from 45 frames per second to 125 frames per second and reduces the deployed network size from 88 MB to 72 MB.

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `compile` | `deploy` | `predict` | **Deep Network Designer**

More About

- "Prototype Deep Learning Networks on FPGA and SoC Devices" on page 5-2

Profile Network to Determine Performance Bottlenecks

This example shows how to identify performance bottlenecks in a deep learning network on an FPGA by using the `Profile` option of the `predict` method.

Prerequisites

- Xilinx® ZCU102 SoC development kit.
- Deep Learning HDL Toolbox™ Support Package for Xilinx® FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Load the Pretrained SeriesNetwork

Load the pretrained digits network:

```
snet = getDigitsNetwork;
snet.Layers
```

```
ans = 
  15×1 Layer array with layers:

     1   'imageinput'    Image Input             28×28×1 images with 'zerocenter' normalization
     2   'conv_1'        Convolution             8 3×3×1 convolutions with stride [1  1] and padd
     3   'batchnorm_1'   Batch Normalization     Batch normalization with 8 channels
     4   'relu_1'        ReLU                    ReLU
     5   'maxpool_1'     Max Pooling             2×2 max pooling with stride [2  2] and padding
     6   'conv_2'        Convolution             16 3×3×8 convolutions with stride [1  1] and pad
     7   'batchnorm_2'   Batch Normalization     Batch normalization with 16 channels
     8   'relu_2'        ReLU                    ReLU
     9   'maxpool_2'     Max Pooling             2×2 max pooling with stride [2  2] and padding
    10   'conv_3'        Convolution             32 3×3×16 convolutions with stride [1  1] and pa
    11   'batchnorm_3'   Batch Normalization     Batch normalization with 32 channels
    12   'relu_3'        ReLU                    ReLU
    13   'fc'            Fully Connected         10 fully connected layer
    14   'softmax'       Softmax                 softmax
    15   'classoutput'   Classification Output   crossentropyex with '0' and 9 other classes
```

Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Specify that the interface is for a Xilinx board with an Ethernet interface. To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx',Interface="Ethernet");
```

To use the JTAG interface, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado tool path and use the JTAG interface, enter:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
hTarget = dlhdl.Target('Xilinx',Interface='JTAG');
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and FPGA board. In this

example the target FPGA board is the Xilinx ZCU102 SOC board. The bitstream uses a single data type.

```
hW = dlhdl.Workflow(Network=snet,Bitstream="zcu102_single",Target=hTarget);
```

To run the example in a Xilinx ZC706 board, enter:

```
hW = dlhdl.Workflow(Network=snet,Bitstream='zc706_single',Target=hTarget);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment.

```
dn = compile(hW);
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SystemBufferOffset"        "0x00800000"     "28.0 MB"
    "InstructionDataOffset"     "0x02400000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02800000"     "4.0 MB"
    "FCWeightDataOffset"        "0x02c00000"     "4.0 MB"
    "EndOffset"                 "0x03000000"     "Total: 48.0 MB"
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` method of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board and download the network weights and biases. The `deploy` function starts programming the FPGA device and displays progress messages, and the required time to deploy the network.

```
deploy(hW);
```

```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'

Downloading target FPGA device configuration over Ethernet to SD card done. The system will now r


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 28-Jun-2020 12:24:21
```

Test Network

Load the example image.

```
inputImg = imread('five_28x28.pgm');
```

Classify the image on the FPGA by using the `predict` method of the `dlhdl.Workflow` object and display the results.

```
[~,speed] = predict(hW,single(inputImg),'Profile','on');
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    73231                  0.00033                       1
    conv_module            26847                  0.00012
        conv_1              6618                  0.00003
        maxpool_1           4823                  0.00002
        conv_2              4876                  0.00002
        maxpool_2           3551                  0.00002
        conv_3              7039                  0.00003
    fc_module              46384                  0.00021
        fc                 46384                  0.00021
 * The clock frequency of the DL processor is: 220MHz
```

Identify and Display the Bottleneck Layer

Remove the module- and network-level results contained in the `NumFrames`, `Total latency`, and `Frames/s` columns from the results table. Retain only the network layer profiler results. After you identify the bottleneck layer, display the bottleneck layer index, running time, and information.

```
speed('Network',:) = [];
speed('____conv_module',:) = [];
speed('____fc_module',:)  = [];
speed = removevars(speed, {'NumFrames','Total Latency(cycles)','Frame/s'});
```

Sort the performance results in descending order.

```
speed = sortrows(speed,'Latency(cycles)','descend');
```

The last layer in this sorted table is the bottleneck layer. In this network the bottleneck layer is the `fc` layer.

```
layerSpeed = speed(1,:);
layerName = strip(layerSpeed.Properties.RowNames{1},'_');
for idx = 1:length(snet.Layers)
    currLayer = snet.Layers(idx);
    if strcmp(currLayer.Name, layerName)
        bottleNeckLayer = currLayer;
        break;
    end
end
```

Display this information for the bottleneck layer:

- Layer index
- Percentage of time the layer runs
- Layer information

```
dnnfpga.disp(['Bottleneck layer index is ', num2str(idx), '.']);
```

```
### Bottleneck layer index is 13.
```

```
percent = layerSpeed.("Latency(cycles)")/sum(speed.("Latency(cycles)")) * 100;
dispStr = sprintf('It accounts for about %0.2f percent of the total running time.', percent);
dnnfpga.disp(dispStr);
```

```
### It accounts for about 63.29 percent of the total running time.
```

```
dnnfpga.disp('Bottleneck layer information: ');
```

```
### Bottleneck layer information:
```

```
disp(currLayer);
```

```
  FullyConnectedLayer with properties:

          Name: 'fc'

   Hyperparameters
     InputSize: 1568
    OutputSize: 10

   Learnable Parameters
       Weights: [10×1568 single]
          Bias: [10×1 single]

  Show all properties
```

See Also

dlhdl.Workflow | dlhdl.Target | compile | deploy | predict

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2
- “Profile Inference Run” on page 5-4

Bicyclist and Pedestrian Classification by Using FPGA

This example shows how to deploy a custom trained series network to detect pedestrians and bicyclists based on their micro-Doppler signatures. This network is taken from the Pedestrian and Bicyclist Classification Using Deep Learning example from the Phased Array Toolbox. For more details on network training and input data, see “Pedestrian and Bicyclist Classification Using Deep Learning” (Radar Toolbox).

Prerequisites

- Xilinx™ Vivado™ Design Suite 2020.2
- Zynq® UltraScale+™ MPSoC ZCU102 Evaluation Kit
- HDL Verifier™ Support Package for XIlinx FPGA Boards
- MATLAB™ Coder ™ Interface for Deep Learning Libraries
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

The data files used in this example are:

- The MAT File `trainedNetBicPed.mat` contains a model trained on training data set `trainDataNoCar` and its label set `trainLabelNoCar`.
- The MAT File `testDataBicPed.mat` contains the test data set `testDataNoCar` and its label set `testLabelNoCar`.

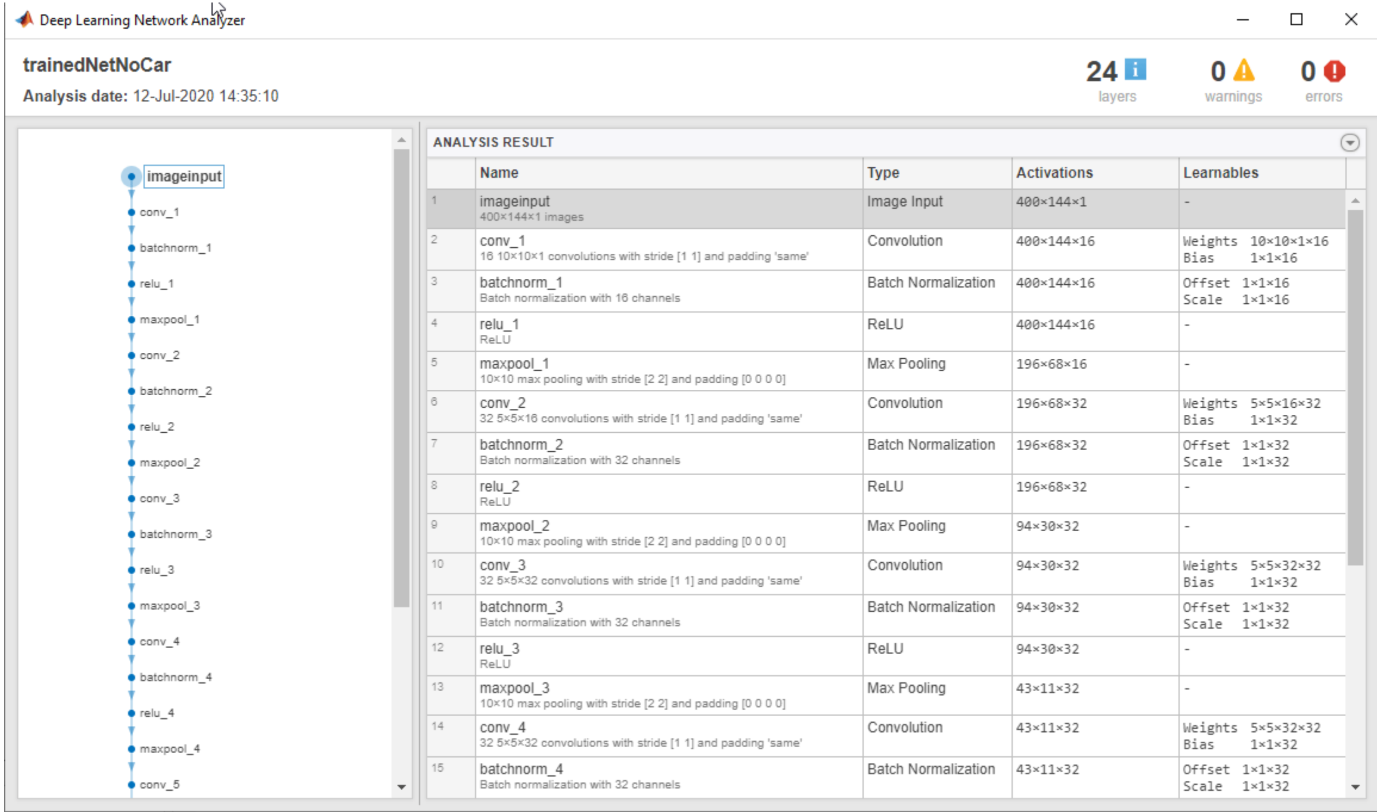
Load Data and Network

Load a pretrained network. Load test data and its labels.

```
load('trainedNetBicPed.mat','trainedNetNoCar')
load('testDataBicPed.mat')
```

View the layers of the pre-trained series network

```
analyzeNetwork(trainedNetNoCar);
```



	Name	Type	Activations	Learnables
1	imageinput 400×144×1 images	Image Input	400×144×1	-
2	conv_1 16 10×10×1 convolutions with stride [1 1] and padding 'same'	Convolution	400×144×16	Weights 10×10×1×16 Bias 1×1×16
3	batchnorm_1 Batch normalization with 16 channels	Batch Normalization	400×144×16	Offset 1×1×16 Scale 1×1×16
4	relu_1 ReLU	ReLU	400×144×16	-
5	maxpool_1 10×10 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	196×68×16	-
6	conv_2 32 5×5×16 convolutions with stride [1 1] and padding 'same'	Convolution	196×68×32	Weights 5×5×16×32 Bias 1×1×32
7	batchnorm_2 Batch normalization with 32 channels	Batch Normalization	196×68×32	Offset 1×1×32 Scale 1×1×32
8	relu_2 ReLU	ReLU	196×68×32	-
9	maxpool_2 10×10 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	94×30×32	-
10	conv_3 32 5×5×32 convolutions with stride [1 1] and padding 'same'	Convolution	94×30×32	Weights 5×5×32×32 Bias 1×1×32
11	batchnorm_3 Batch normalization with 32 channels	Batch Normalization	94×30×32	Offset 1×1×32 Scale 1×1×32
12	relu_3 ReLU	ReLU	94×30×32	-
13	maxpool_3 10×10 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	43×11×32	-
14	conv_4 32 5×5×32 convolutions with stride [1 1] and padding 'same'	Convolution	43×11×32	Weights 5×5×32×32 Bias 1×1×32
15	batchnorm_4 Batch normalization with 32 channels	Batch Normalization	43×11×32	Offset 1×1×32 Scale 1×1×32

Set up HDL Toolpath

Set up the path to your installed Xilinx™ Vivado™ Design Suite 2020.2 executable if it is not already set up. For example, to set the toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado','ToolPath', 'C:\Vivado\2020.2\bin');
```

Create Target Object

Create a target object for your target device with a vendor name and an interface to connect your target device to the host computer. Interface options are JTAG (default) and Ethernet. Vendor options are Intel or Xilinx. Use the installed Xilinx Vivado Design Suite over an Ethernet connection to program the device.

```
hT = dlhdl.Target('Xilinx', 'Interface', 'Ethernet');
```

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pre-trained series network, `trainedNetNoCar`, as the network. Make sure the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Zynq UltraScale+ MPSoC ZCU102 board. The bitstream uses a single data type. .

```
hW = dlhdl.Workflow('Network', trainedNetNoCar, 'Bitstream', 'zcu102_single', 'Target', hT);
```

Compile `trainedNetNoCar` Series Network

To compile the `trainedNetNoCar` series network, run the compile function of the `dlhdl.Workflow` object .

```
dn = hW.compile;
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "28.0 MB"
    "OutputResultOffset"        "0x01c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x02000000"     "28.0 MB"
    "InstructionDataOffset"     "0x03c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04000000"     "4.0 MB"
    "FCWeightDataOffset"        "0x04400000"     "4.0 MB"
    "EndOffset"                 "0x04800000"     "Total: 72.0 MB"
```

Program the Bitstream onto FPGA and Download Network Weights

To deploy the network on the Zynq® UltraScale+™ MPSoC ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object . This function uses the output of the compile function to program the FPGA board by using the programming file.The function also downloads the network weights and biases. The deploy function checks for the Xilinx Vivado tool and the supported tool version. It then starts programming the FPGA device by using the bitstream, displays progress messages and the time it takes to deploy the network.

```
hW.deploy;
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Deep learning network programming has been skipped as the same network is already loaded on t
```

Run Predictions on Micro-Doppler Signatures

Classify one input from the sample test data set by using the predict function of the `dlhdl.Workflow` object and display the label. The inputs to the network correspond to the sonograms of the micro-Doppler signatures for a pedestrian or a bicyclist or a combination of both.

```
testImg = single(testDataNoCar(:, :, :, 1));
testLabel = testLabelNoCar(1);
classnames = trainedNetNoCar.Layers(end).Classes;

% Get predictions from network on single test input
score = hW.predict(testImg, 'Profile', 'On')
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    9430692                  0.04287                       1           94
    conv_module            9411355                  0.04278
        conv_1             4178753                  0.01899
        maxpool_1          1394883                  0.00634
```

```
        conv_2              1975197                  0.00898
        maxpool_2            706156                  0.00321
        conv_3               813598                  0.00370
        maxpool_3            121790                  0.00055
        conv_4               148165                  0.00067
        maxpool_4             22255                  0.00010
        conv_5                41999                  0.00019
        avgpool2d              8674                  0.00004
    fc_module                 19337                  0.00009
        fc                    19337                  0.00009
 * The clock frequency of the DL processor is: 220MHz

score = 1×5 single row vector

    0.9956    0.0000    0.0000    0.0044    0.0000
```

```
[~, idx1] = max(score);
predTestLabel = classnames(idx1)
```

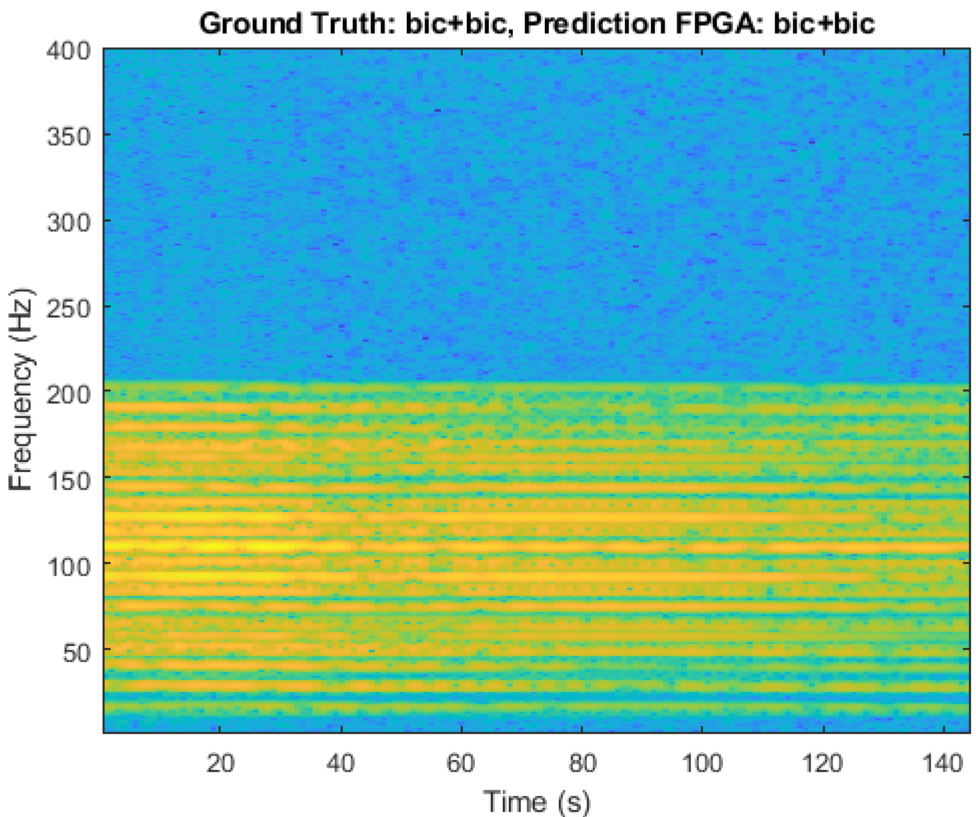
```
predTestLabel = categorical
     ped
```

Load five random images from the sample test data set and execute the predict function of the `dlhdl.Workflow` object to display the labels alongside the signatures. The predictions will happen at once since the input is concatenated along the fourth dimension.

```
numTestFrames = size(testDataNoCar, 4);
numView = 5;
listIndex = randperm(numTestFrames, numView);
testImgBatch = single(testDataNoCar(:, :, :, listIndex));
testLabelBatch = testLabelNoCar(listIndex);

% Get predictions from network using DL HDL Toolbox on FPGA
[scores, speed] = hW.predict(testImgBatch, 'Profile', 'On');
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                 9446929                  0.04294                       5           471
    conv_module         9427488                  0.04285
        conv_1          4195175                  0.01907
        maxpool_1       1394705                  0.00634
        conv_2          1975204                  0.00898
        maxpool_2        706332                  0.00321
        conv_3           813499                  0.00370
        maxpool_3        121869                  0.00055
        conv_4           148063                  0.00067
        maxpool_4         22019                  0.00010
        conv_5            42053                  0.00019
        avgpool2d          8684                  0.00004
    fc_module             19441                  0.00009
```

```
        fc                     19441                   0.00009
 * The clock frequency of the DL processor is: 220MHz

[~, idx2] = max(scores, [], 2);
predTestLabelBatch = classnames(idx2);

% Display the micro-doppler signatures along with the ground truth and
% predictions.
for k = 1:numView
    index = listIndex(k);
    imagesc(testDataNoCar(:, :, :, index));
    axis xy
    xlabel('Time (s)')
    ylabel('Frequency (Hz)')
    title('Ground Truth: '+string(testLabelNoCar(index))+', Prediction FPGA: '+string(predTestLab
    drawnow;
    pause(3);
end
```



The image shows the micro-Doppler signatures of two bicyclists (bic+bic) which is the ground truth. The ground truth is the classification of the image against which the network prediction is compared. The network prediction retrieved from the FPGA correctly predicts that the image has two bicyclists.

See Also

dlhdl.Workflow | dlhdl.Target | compile | deploy | predict

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2
- “Profile Inference Run” on page 5-4

Visualize Activations of a Deep Learning Network by Using LogoNet

This example shows how to feed an image to a convolutional neural network and display the activations of the different layers of the network. Examine the activations and discover which features the network learns by comparing areas of activation to the original image. Channels in earlier layers learn simple features like color and edges, while channels in the deeper layers learn complex features. Identifying features in this way can help you understand what the network has learned.

Logo Recognition Network

Logos assist in brand identification and recognition. Many companies incorporate their logos in advertising, documentation materials, and promotions. The logo recognition network (LogoNet) was developed in MATLAB® and can recognize 32 logos under various lighting conditions and camera motions. Because this network focuses only on recognition, you can use it in applications where localization is not required.

Prerequisites

- Intel® Arria10 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Intel FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Computer Vision Toolbox™

Load Pretrained Series Network

To load the pretrained series network LogoNet, enter:

```
snet = getLogoNetwork;
```

Create Target Object

Create a target object that has a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use JTAG, install Intel™ Quartus™ Prime Standard Edition 20.1. Set up the path to your installed Intel Quartus Prime executable if it is not already set up. For example, to set the toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Altera Quartus II','ToolPath', 'C:\altera\20.1\quartus\bin64');
```

To create the target object, enter:

```
hTarget = dlhdl.Target('Intel','Interface','JTAG');
```

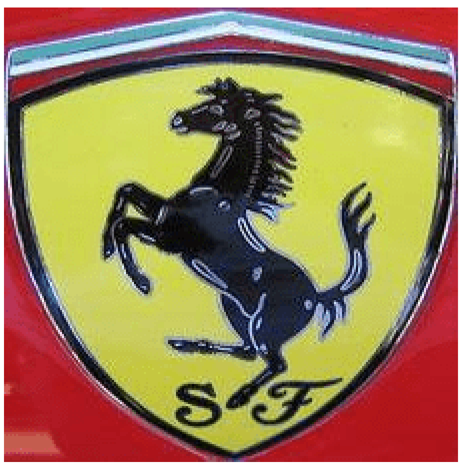
Create Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pretrained LogoNet neural network, `snet`, as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Intel Arria10 SOC board. The bitstream uses a single data type.

```
hW = dlhdl.Workflow('network', snet, 'Bitstream', 'arria10soc_single','Target',hTarget);
```

Read and show an image. Save its size for future use.

```
im = imread('ferrari.jpg');
imshow(im)
```



```
imgSize = size(im);
imgSize = imgSize(1:2);
```

View Network Architecture

Analyze the network to see which layers you can view. The convolutional layers perform convolutions by using learnable parameters. The network learns to identify useful features, often including one feature per channel. The first convolutional layer has 64 channels.

```
analyzeNetwork(snet)
```

The Image Input layer specifies the input size. Before passing the image through the network, you can resize it. The network can also process larger images.. If you feed the network larger images, the activations also become larger. Because the network is trained on images of size 227-by-227, it is not trained to recognize larger objects or features.

Show Activations of First Maxpool Layer

Investigate features by observing which areas in the maxpool layers activate on an image and comparing that image to the corresponding areas in the original images. Each layer of a convolutional neural network consists of many 2-D arrays called *channels*. Pass the image through the network and examine the output activations of the `maxpool_1` layer.

```
act1 = hW.activations(single(im),'maxpool_1','Profiler','on');
```

```
          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________
```

```
    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "136.0 MB"
    "SystemBufferOffset"        "0x0a000000"     "64.0 MB"
    "InstructionDataOffset"     "0x0e000000"     "8.0 MB"
    "ConvWeightDataOffset"      "0x0e800000"     "4.0 MB"
    "EndOffset"                 "0x0ec00000"     "Total: 236.0 MB"

### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.

### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   10182024                  0.06788                       1          101
    conv_module           10182024                  0.06788
        conv_1             7088885                  0.04726
        maxpool_1          3093166                  0.02062
 * The clock frequency of the DL processor is: 150MHz
```

The activations are returned as a 3-D array, with the third dimension indexing the channel on the maxpool_1 layer. To show these activations using the imtile function, reshape the array to 4-D. The third dimension in the input to imtile represents the image color. Set the third dimension to have size 1 because the activations do not have color. The fourth dimension indexes the channel.

```
sz = size(act1);
act1 = reshape(act1,[sz(1) sz(2) 1 sz(3)]);
```

Display the activations. Each activation can take any value, so normalize the output using the mat2gray. All activations are scaled so that the minimum activation is 0 and the maximum activation is 1. Display the 96 images on an 12-by-8 grid, one for each channel in the layer.

```
I = imtile(mat2gray(act1),'GridSize',[12 8]);
imshow(I)
```

Investigate Activations in Specific Channels

Each tile in the activations grid is the output of a channel in the `maxpool_1` layer. White pixels represent strong positive activations and black pixels represent strong negative activations. A channel that is mostly gray does not activate as strongly on the input image. The position of a pixel in the activation of a channel corresponds to the same position in the original image. A white pixel at a location in a channel indicates that the channel is strongly activated at that position.

Resize the activations in channel 33 to be the same size as the original image and display the activations.

```
act1ch33 = act1(:,:,:,22);
act1ch33 = mat2gray(act1ch33);
act1ch33 = imresize(act1ch33,imgSize);

I = imtile({im,act1ch33});
imshow(I)
```

Find Strongest Activation Channel

Find interesting channels by programmatically investigating channels with large activations. Find the channel that has the largest activation by using the `max` function, resize the channel output, and display the activations.

```
[maxValue,maxValueIndex] = max(max(max(act1)));
act1chMax = act1(:,:,:,maxValueIndex);
act1chMax = mat2gray(act1chMax);
act1chMax = imresize(act1chMax,imgSize);

I = imtile({im,act1chMax});
imshow(I)
```

Compare the strongest activation channel image to the original image. This channel activates on edges. It activates positively on light left/dark right edges and negatively on dark left/light right edges.

See Also

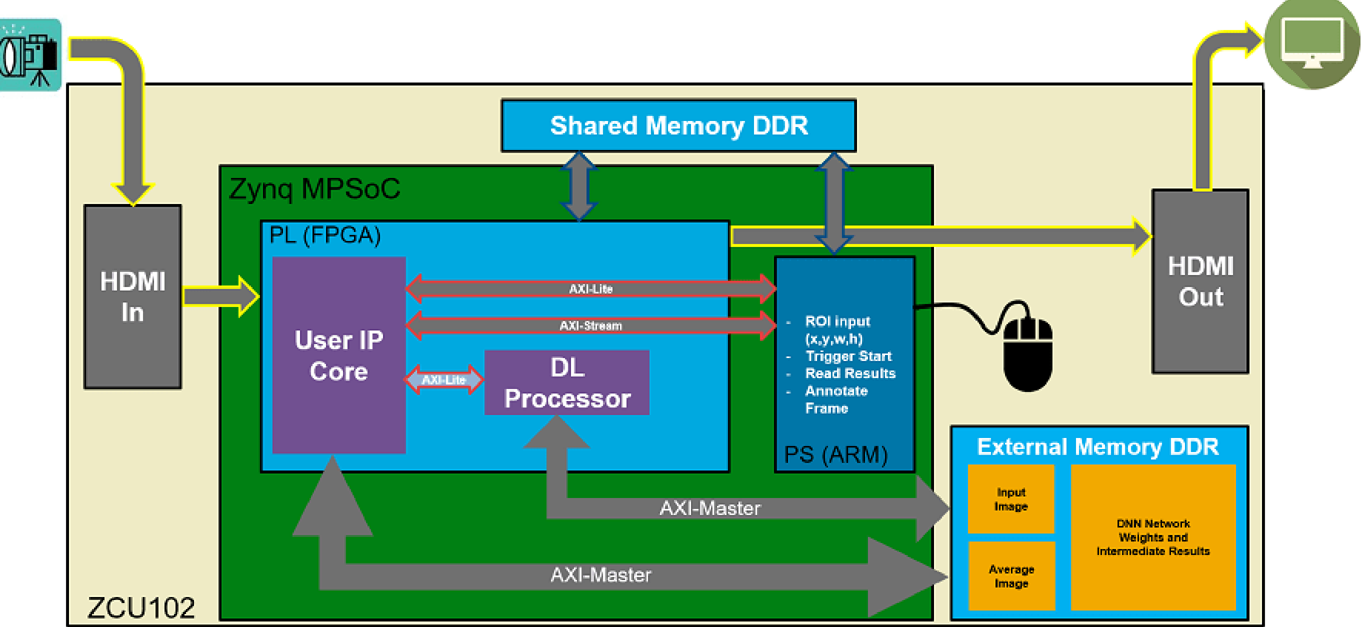
`dlhdl.Workflow` | `dlhdl.Target` | `activations` | `compile` | `deploy` | `predict`

More About

- "Prototype Deep Learning Networks on FPGA and SoC Devices" on page 5-2
- "Profile Inference Run" on page 5-4

Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core

This example shows how to create an HDL Coder™ reference design that contains a generated deep learning processor IP core. The reference design receives a live camera input and uses a deployed series network to classify the objects in the camera input. This figure is a high-level architectural diagram that shows the reference design that will be implemented on the Xilinx™ Zynq™ Ultrascale+ (TM) MPsoC ZCU102 Evaluation Kit.



The user IP core block:

- Extracts the region of interest (ROI) based on ROI dimensions from the processing system (PS) (ARM).
- Performs downsampling on the input image.
- Zero-centers the input image.
- Transfers the preprocessed image to the external DDR memory.
- Triggers the deep learning processor IP core.
- Notifies the PS(ARM) processor.

The deep learning processor IP core accesses the preprocessed inputs, performs the object classification and loads the output results back into the external DDR memory.

The PS (ARM):

- Takes the ROI dimensions and passes them to the user IP core.
- Performs post-processing on the image data.
- Annotates the object classification results from the deep learning processor IP core on the output video frame.

You can also use MATLAB® to retrieve the classification results and verify the generated deep learning processor IP core. The user DUT for this reference design is the preprocessing algorithm (User IP Core). You can design the preprocessing DUT algorithm in Simulink®, generate the DUT IP core, and integrate the generated DUT IP core into the larger system that contains the deep learning processor IP core. To learn how to generate the DUT IP core, see “Run a Deep Learning Network on FPGA with Live Camera Input” on page 10-70.

Generate Deep Learning Processor IP Core

Follow these steps to configure and generate the deep learning processor IP core into the reference design.

1. Create a custom deep learning processor configuration.

```
hPC = dlhdl.ProcessorConfig
```

To learn more about the deep learning processor architecture, see “Deep Learning Processor IP Core Architecture” on page 2-2. To get information about the custom processor configuration parameters and modifying the parameters, see `getModuleProperty` and `setModuleProperty`.

2. Generate the Deep Learning Processor IP core.

To learn how to generate the custom deep learning processor IP, see “Generate Custom Processor IP” on page 9-3. The deep learning processor IP core is generated by using the HDL Coder™ IP core generation workflow. For more information, see “Custom IP Core Generation” (HDL Coder).

```
dlhdl.buildProcessor(hPC)
```

The generated IP core files are located at `cwd\dlhdl_prj\ipcore`. `cwd` is the current working directory. The `ipcore` folder contains an HTML report located at `cwd\dlhdl_prj\ipcore\DUT_ip_v1_0\doc`.

IP Core Generation Report for testbench

Summary

IP core name	DUT_ip
IP core version	1.0
IP core folder	dlhdl_prj\ipcore\DUT_ip_v1_0
IP core zip file name	DUT_ip_v1_0.zip
Target platform	Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
Target tool	Xilinx Vivado
Target language	VHDL
Reference Design	AXI-Stream DDR Memory Access : 3-AXIM
Model	testbench
Model version	1.1208
HDL Coder version	3.17
IP core generated on	16-Jul-2020 08:51:10
IP core generated for	DUT

Target Interface Configuration

You chose the following target interface configuration for testbench :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
dut_rd_data	Inport	single (4)	AXI4 Master Activation Data Read	Data	
inputStart	Inport	boolean	AXI4	x"224"	
debugEnable	Inport	boolean	AXI4	x"140"	
dut_rd_s2m	Inport	bus	AXI4 Master Activation Data Read	Read Slave to Master Bus	
dut_wr_s2m	Inport	bus	AXI4 Master Activation Data Write	Write Slave to Master Bus	
start	Inport	boolean	AXI4	x"138"	
debugSelect	Inport	uint32	AXI4	x"14C"	
image_valid	Inport	boolean	AXI4	x"160"	
image_data	Inport	single	AXI4	x"168"	
image_addr	Inport	ufix18	AXI4	x"164"	
debugDMAEnable	Inport	boolean	AXI4	x"144"	
read_addr	Inport	ufix18	AXI4	x"16C"	
debugDMALength	Inport	uint32	AXI4	x"148"	
debugDMAWidth	Inport	uint32	AXI4	x"150"	
debugDMAOffset	Inport	uint32	AXI4	x"154"	
debugDMADirection	Inport	boolean	AXI4	x"158"	
debugDMAStart	Inport	boolean	AXI4	x"15C"	
debug_wr_s2m	Inport	bus	AXI4 Master Debug Write	Write Slave to Master Bus	
preLoadingStart	Inport	boolean	AXI4	x"228"	
nc_LCtotalLength_IP0	Inport	uint32	AXI4	x"22C"	
nc_LCoffset_IP0	Inport	uint32	AXI4	x"230"	
nc_LCtotalLength_Conv	Inport	uint32	AXI4	x"234"	
nc_LCoffset_Conv	Inport	uint32	AXI4	x"238"	

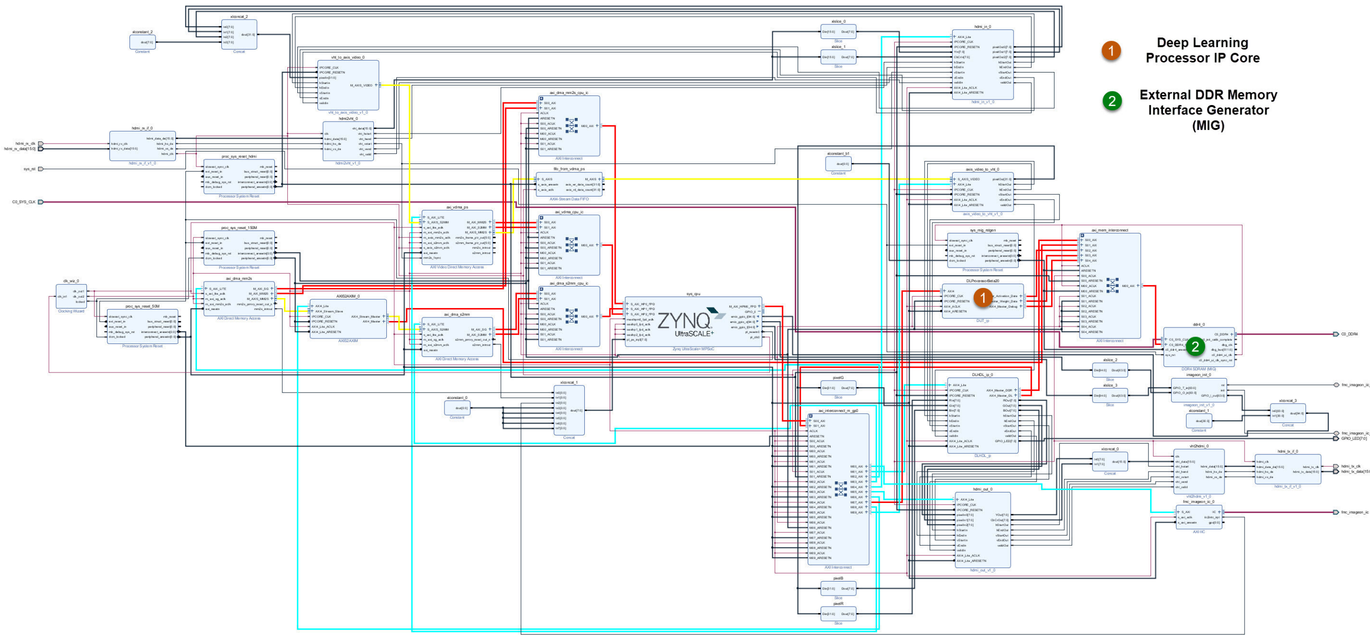
The HTML report contains a description of the deep learning processor IP core, instructions for using the core and integrating the core into your Vivado™ reference design, and a list of AXI4 registers. You will need the AXI4 register list to enter addresses into the Vivado™ Address Mapping tool. For more information about the AXI4 registers, see “Deep Learning Processor IP Core Report” on page 12-14.

Integrate the Generated Deep Learning Processor IP Core into the Reference Design

Insert the generated deep learning processor IP core into your reference design. After inserting the generated deep learning processor IP core into the reference design, you must:

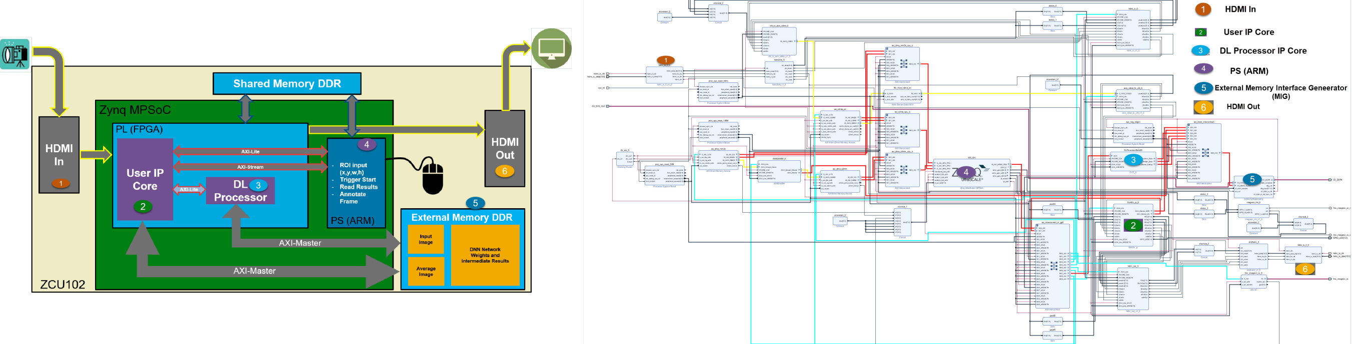
- Connect the generated deep learning processor IP core AXI4 slave interface to an AXI4 master device such as a JTAG AXI master IP core or a Zynq™ processing system (PS). Use the AXI4 master device to communicate with the deep learning processor IP core.
- Connect the vendor provided external memory interface IP core to the three AXI4 master interfaces of the generated deep learning processor IP core.

The deep learning processor IP core uses the external memory interface to access the external DDR memory. The image shows the deep learning processor IP core integrated into the Vivado™ reference design and connected to the DDR memory interface generator (MIG) IP.



Connect the External Memory Interface Generator

In your Vivado™ reference design add an external memory interface generator (MIG) block and connect the generated deep learning processor IP core to the MIG module. The MIG module is connected to the processor IP core through an AXI interconnect module. The image shows the high level architectural design and the Vivado™ reference design implementation.



Create the Reference Design Definition File

The following code describes the contents of the ZCU102 reference design definition file **plugin_rd.m** for the above Vivado™ reference design. For more details on how to define and register the custom board, refer to the “Define Custom Board and Reference Design for Zynq Workflow” (HDL Coder).

```
function hRD = plugin_rd(varargin)

% Parse config
config = ZynqVideoPSP.common.parse_config(...
    'ToolVersion', '2019.1', ...
    'Board', 'zcu102', ...
```

```
    'Design', 'visionzynq_base', ...
   'ColorSpace', 'RGB' ...
);
% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.BoardName = ZynqVideoPSP.ZCU102Hdmicam.BoardName();
hRD.ReferenceDesignName = 'HDMI RGB with DL Processor';
% Tool information
hRD.SupportedToolVersion = {'2019.1'}
...
```

Verify the Reference Design

After creating the reference design, use the HDL Coder™ IP core generation workflow to generate the bitstream and program the ZCU102 board. You can then use MATLAB® and a `dlhdl.Workflow` object to verify the deep learning processor IP core or you can use the HDL Coder™ workflow to prototype the entire system. To verify the reference design, see “Run a Deep Learning Network on FPGA with Live Camera Input” on page 10-70.

See Also

`dlhdl.ProcessorConfig` | `dlhdl.buildProcessor`

More About

- “Use the Compiler Output for System Integration” on page 12-6

Run a Deep Learning Network on FPGA with Live Camera Input

This example shows how to model preprocessing logic that receives a live camera input. You implement it on a Zynq® Ultrascale+™ MPSoC ZCU102 board by using a custom video reference design that has an integrated deep learning processor IP core for object classification. This example uses the HDL Coder™ HW/SW co-design workflow. For this example, you need:

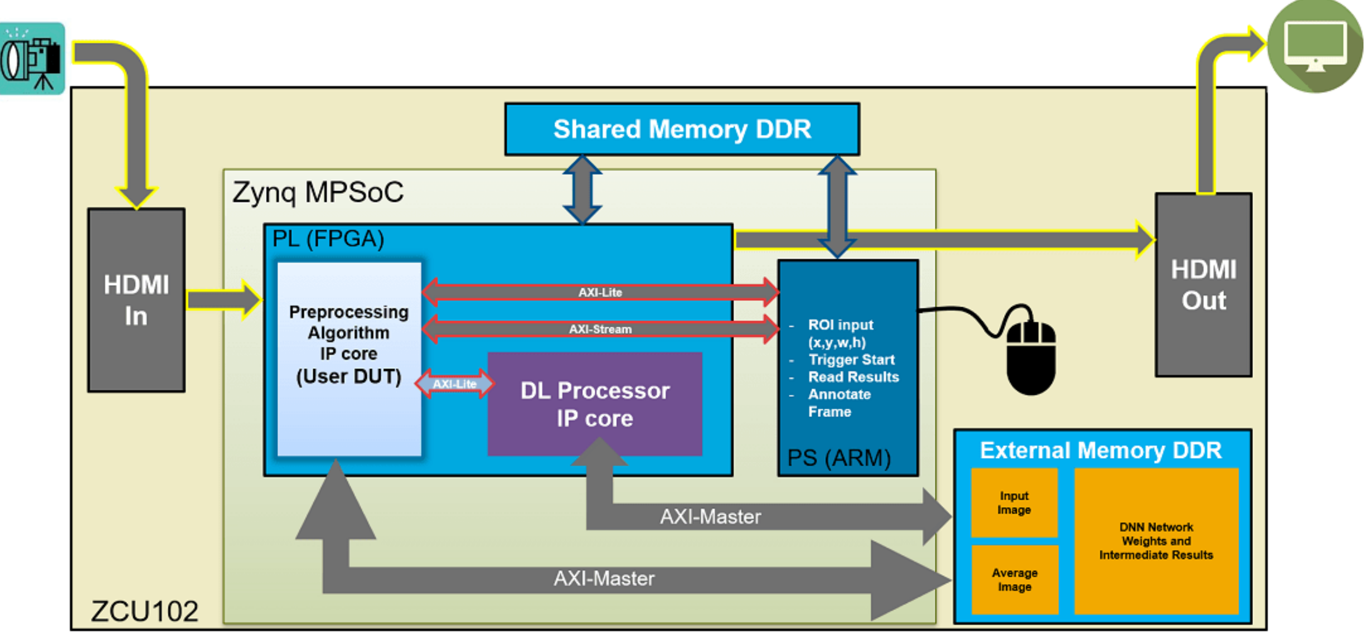
- Deep Learning HDL Toolbox™
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- Deep Learning Toolbox™
- HDL Coder™
- Simulink™

Introduction

In this example, you:

1. Model the preprocessing logic that processes the live camera input for the deep learning processor IP core. The processed video frame is sent to the external DDR memory on the FPGA board.
2. Simulate the model in Simulink® to verify the algorithm functionality.
3. Implement the preprocessing logic on a ZCU102 board by using a custom video reference design which includes the generated deep learning processor IP core.
4. Individually validate the preprocessing logic on the FPGA board.
5. Individually validate the deep learning processor IP core functionality by using the Deep Learning HDL Toolbox™ prototyping workflow.
6. Deploy and validate the entire system on a ZCU102 board.

This figure is a high-level architectural diagram of the system. The result of the deep learning network prediction is sent to the ARM processor. The ARM processor annotates the deep learning network prediction onto the output video frame.



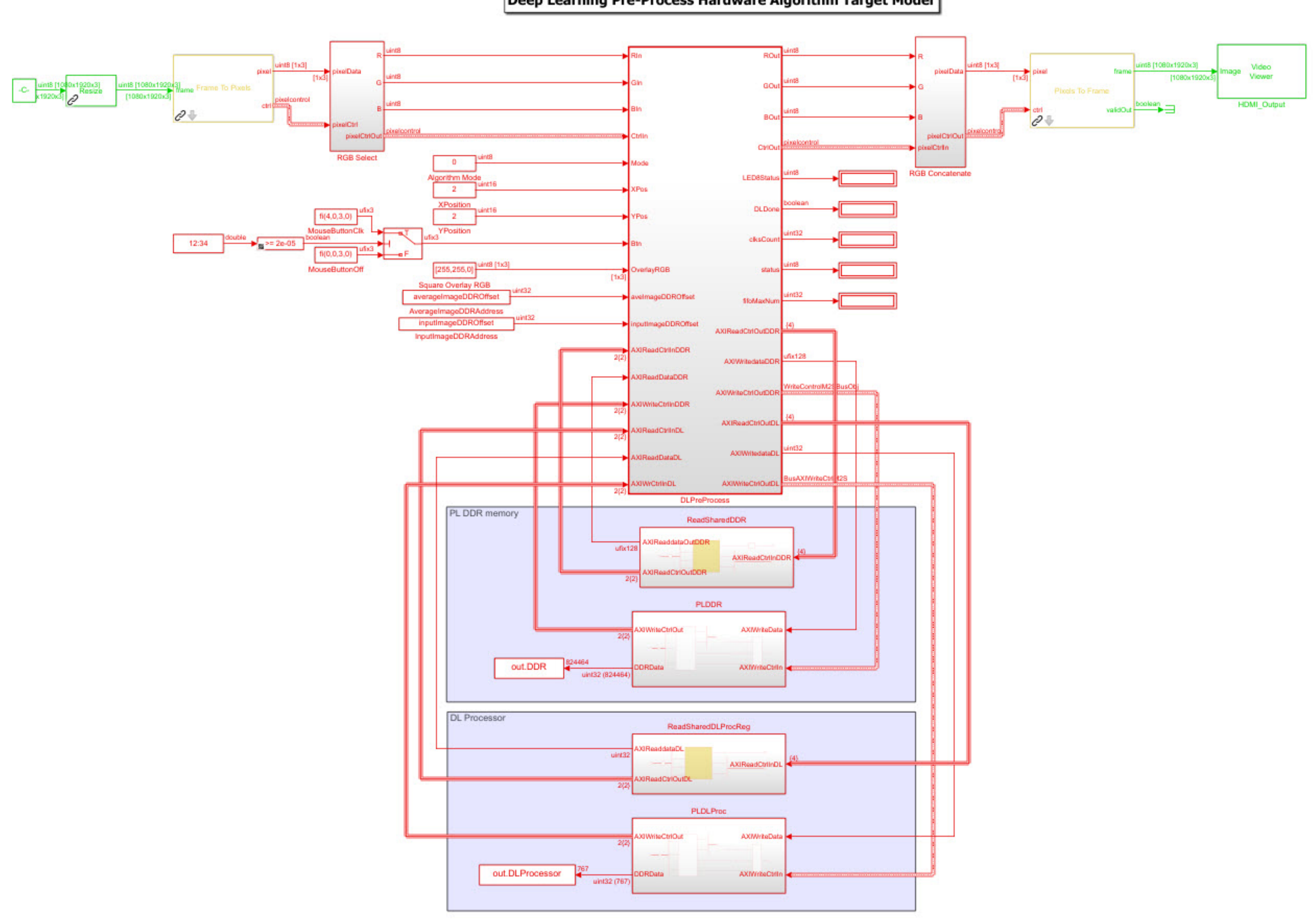
The objective of this system is to receive the live camera input through the HDMI input of the FMC daughter card on the ZCU102 board. You design the preprocessing logic in Simulink® to select and resize the region of interest (ROI). You then transmit the processed image frame to the deep learning processor IP core to run image classification by using a deep learning network.

Select and Resize the Region of Interest

Model the preprocessing logic to process the live camera input for the deep learning network and send the video frame to external DDR memory on the FPGA board. This logic is modeled in the DUT subsystem:

- Image frame selection logic that allows you to use your cursor to choose an ROI from the incoming camera frame. The selected ROI is the input to the deep learning network.
- Image resizing logic that resizes the ROI image to match the input image size of the deep learning network.
- AXI4 Master interface logic that sends the resized image frame into the external DDR memory, where the deep learning processor IP core reads the input. To model the AXI4 Master interface, see “Model Design for AXI4 Master Interface Generation” (HDL Coder).

This figure shows the Simulink® model for the preprocessing logic DUT.



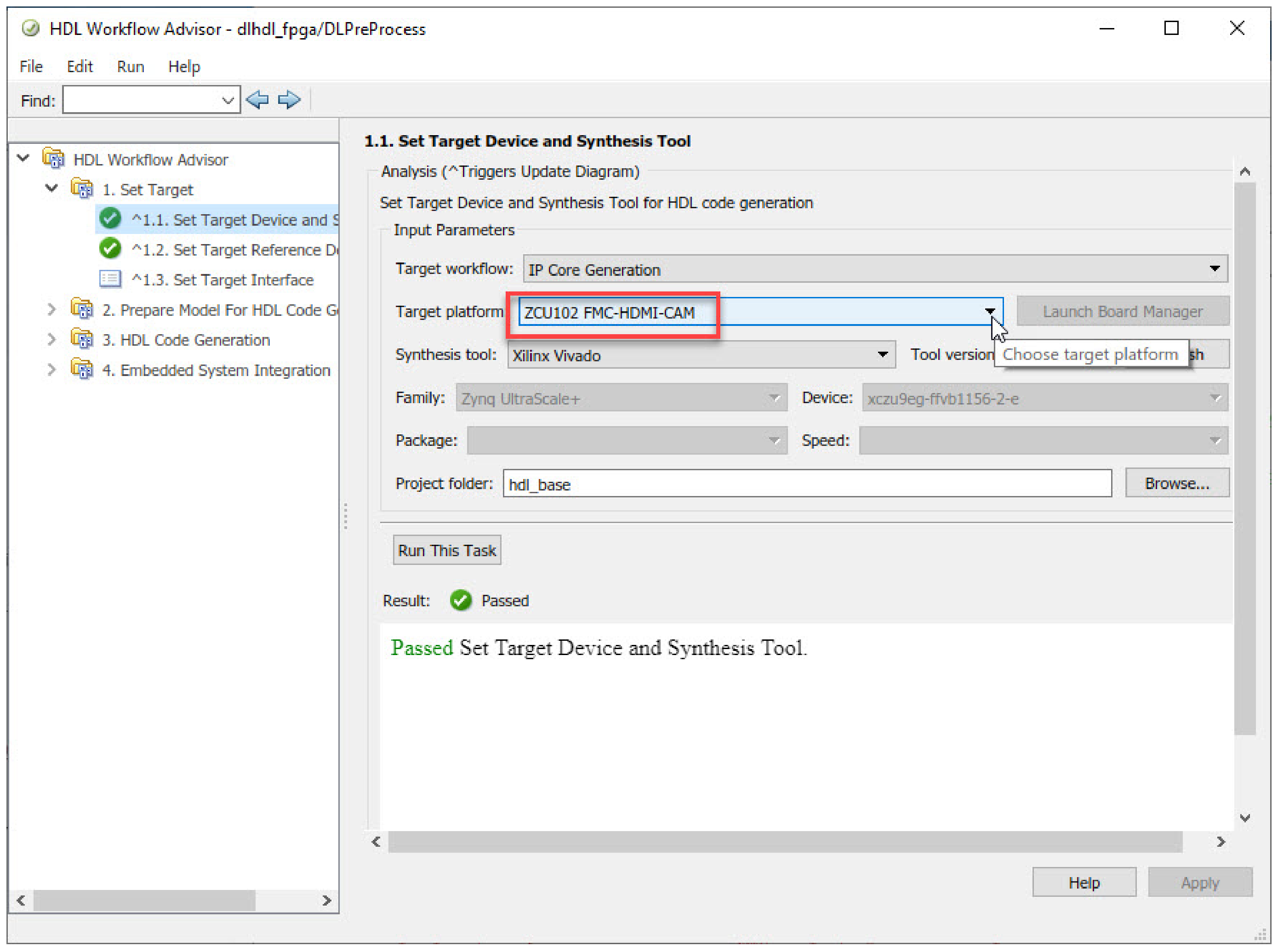
Generate Preprocessing Logic HDL IP Core

To implement the preprocessing logic model on a ZCU102 SoC board, create an HDL Coder™ reference design in Vivado™ which receives the live camera input and transmits the processed video data to the deep learning processor IP core. To create a custom video reference design that integrates the deep learning processor IP core, see “Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core” on page 10-65.

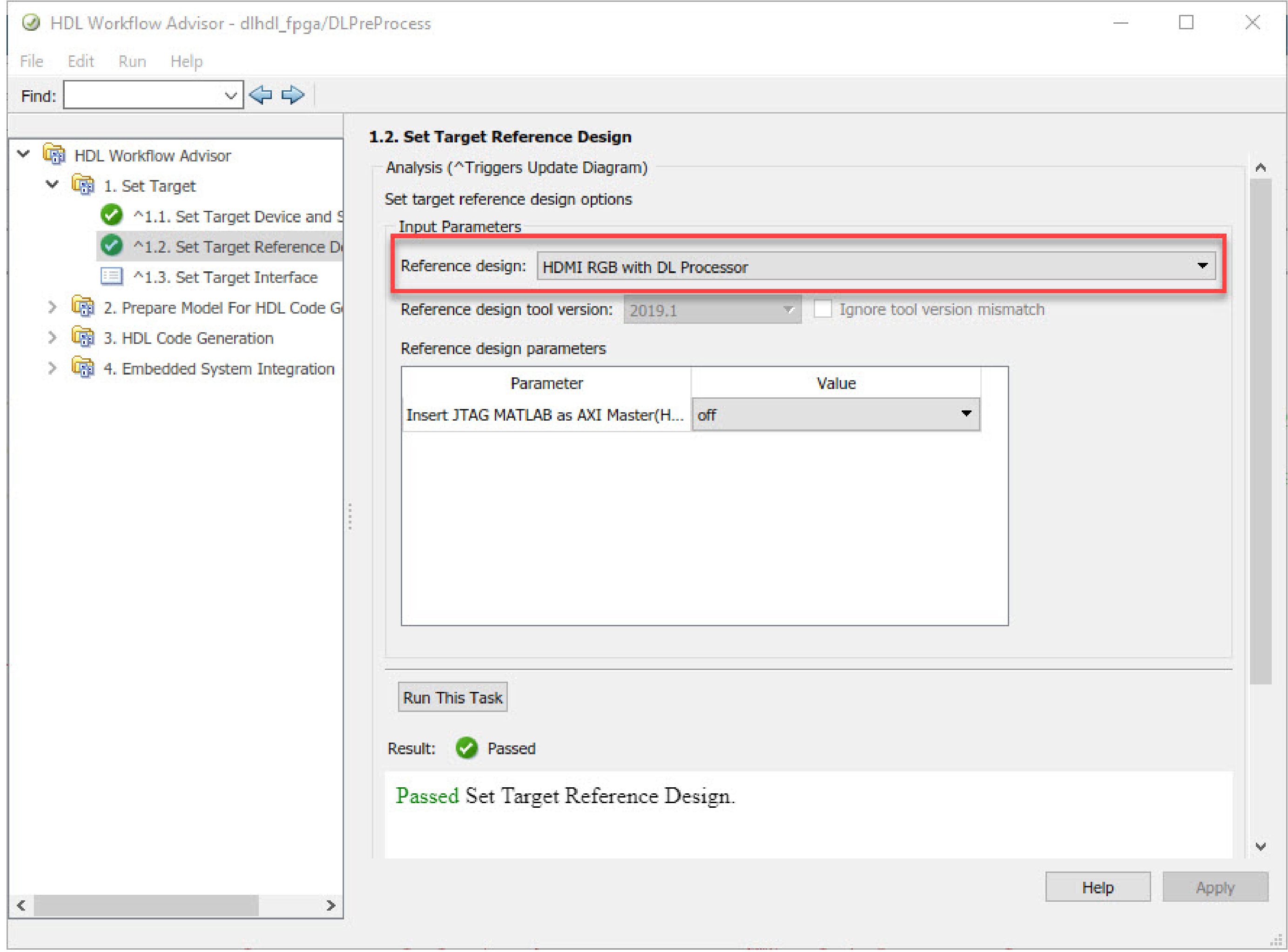
Start the HDL Coder HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy the preprocessing logic model on Zynq hardware. This workflow is the standard HDL Coder workflow. In this example the only difference is that this reference design contains the generated deep learning processor IP core. For more details refer to the “Getting Started with Targeting Xilinx Zynq Platform” (HDL Coder) example.

1. Start the HDL Workflow Advisor from the model by right-clicking the `DLPreProcess` DUT subsystem and selecting **HDL Advisor Workflow**.

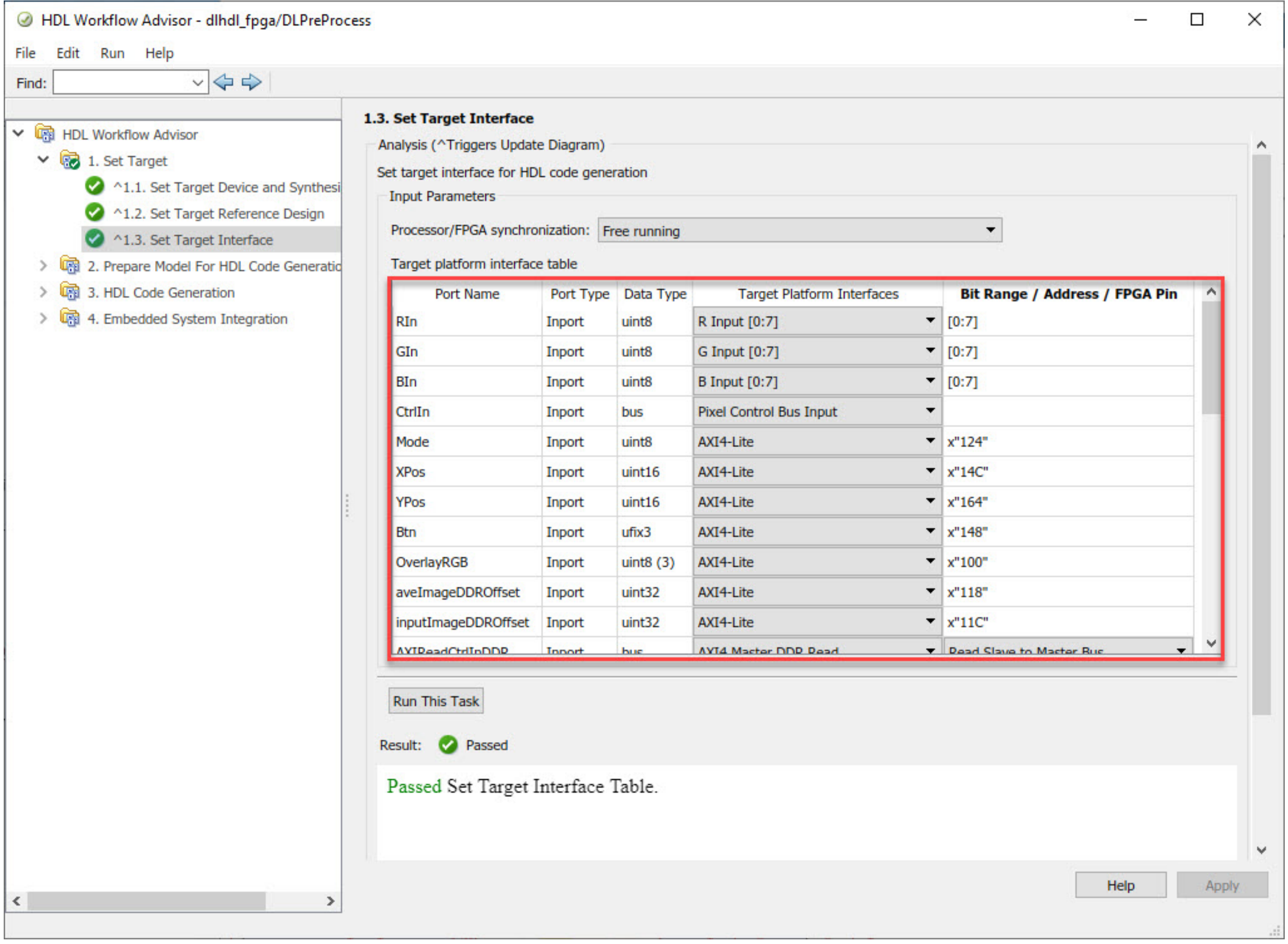
In Task 1.1, **IP Core Generation** is selected for **Target workflow** and **ZCU102-FMC-HDMI-CAM** is selected for **Target platform**.



In Task 1.2, **HDMI RGB with DL Processor** is selected for **Reference Design**.



In Task 1.3, the **Target platform interface table** is loaded as shown in the following screenshot. Here you can map the ports of the DUT subsystem to the interfaces in the reference design.



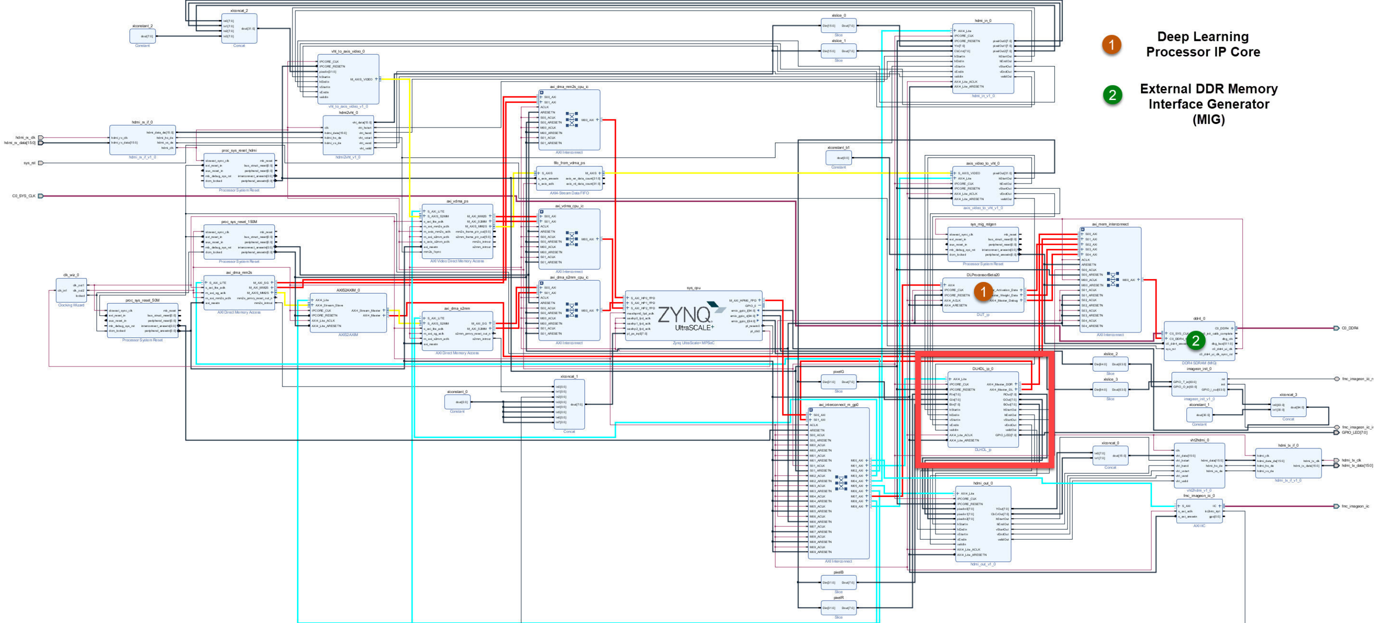
2. Right-click Task 3.2, **Generate RTL Code and IP Core**, and then select **Run to Selected Task**. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

Integrate IP into the Custom Video Reference Design

In the HDL Workflow Advisor, run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Zynq hardware.

1. Run Task 4.1, **Create Project**. This task inserts the generated IP core into the **HDMI RGB with DL Processor** reference design. To create a reference design that integrates the deep learning processor IP core, see “Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core” on page 10-65.

2. Click the link in the **Result** pane to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram, which includes the generated preprocessing HDL IP core, the deep learning processor IP core and the Zynq processor.



3. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model and build and download the FPGA bitstream.

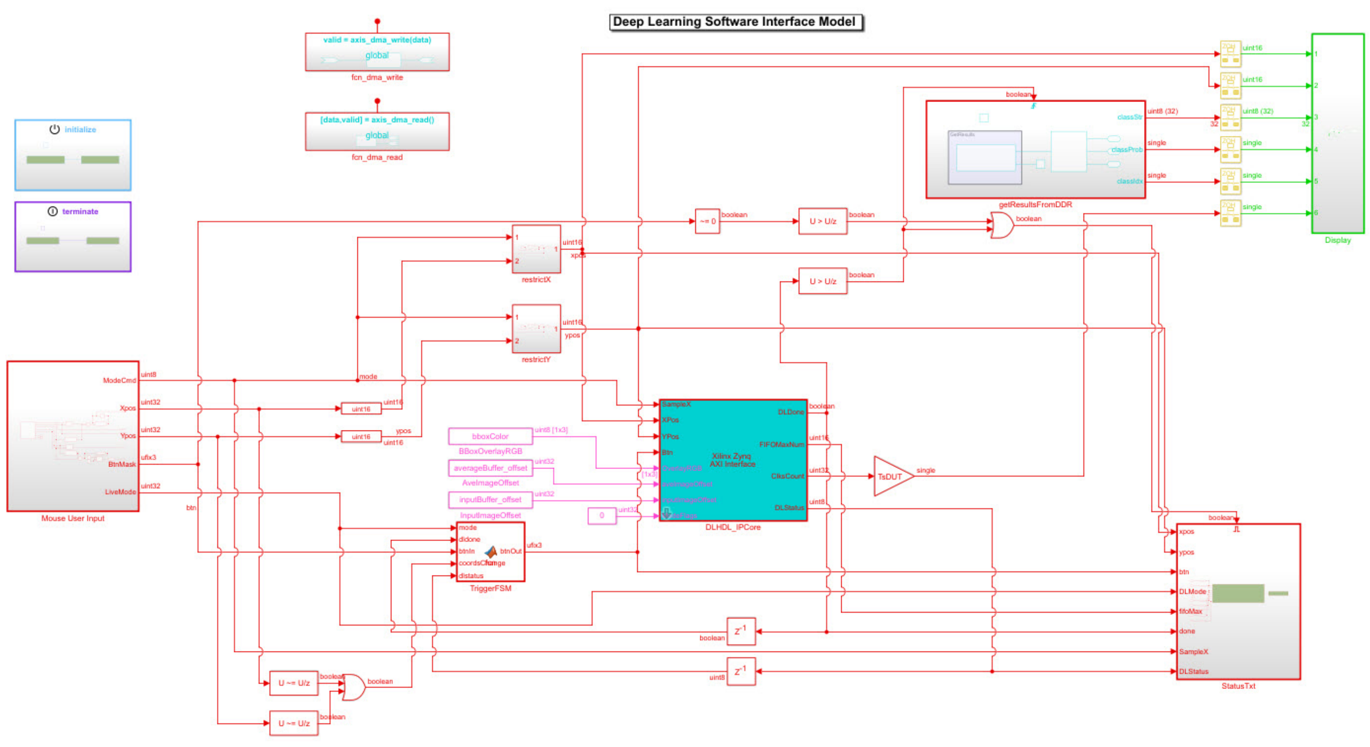
Deploy and Validate the Integrated Reference Design

To validate the integrated reference design that includes the generated preprocessing logic IP core, deep learning processor IP core, and the Zynq processor:

1. Individually validate the preprocessing logic on the FPGA board.
2. Individually validate the deep learning processor IP core functionality by using the Deep Learning HDL Toolbox™ prototyping workflow.
3. Deploy and validate the entire system on a ZCU102 board.
4. Deploy the entire system as an executable file on the SD card on the ZCU102 board.

1. Using the standard HDL Coder hardware/software co-design workflow, you can validate that the preprocessing logic works as expected on the FPGA. The HDL Workflow Advisor generates a software interface subsystem during Task 4.2 **Generate Software Interface Model**, which you can use in your software model for interfacing with the FPGA logic. From the software model, you can tune and probe the FPGA design on the hardware by using Simulink External Mode. Instruct the FPGA preprocessing logic to capture an input frame and send it to the external DDR memory.

You can then use `fpga` object to create a connection from MATLAB to the ZCU102 board and read the contents of the external DDR memory into MATLAB for validation. to use the `fpga` object, see “Create Host Interface Script to Control and Rapidly Prototype HDL IP Core” (HDL Coder).



2. The generated deep learning processor IP core has Ethernet and JTAG interfaces for communications in the generated bitstream. You can individually validate the deep learning processor IP core by using the `dlhdl.Workflow` object.

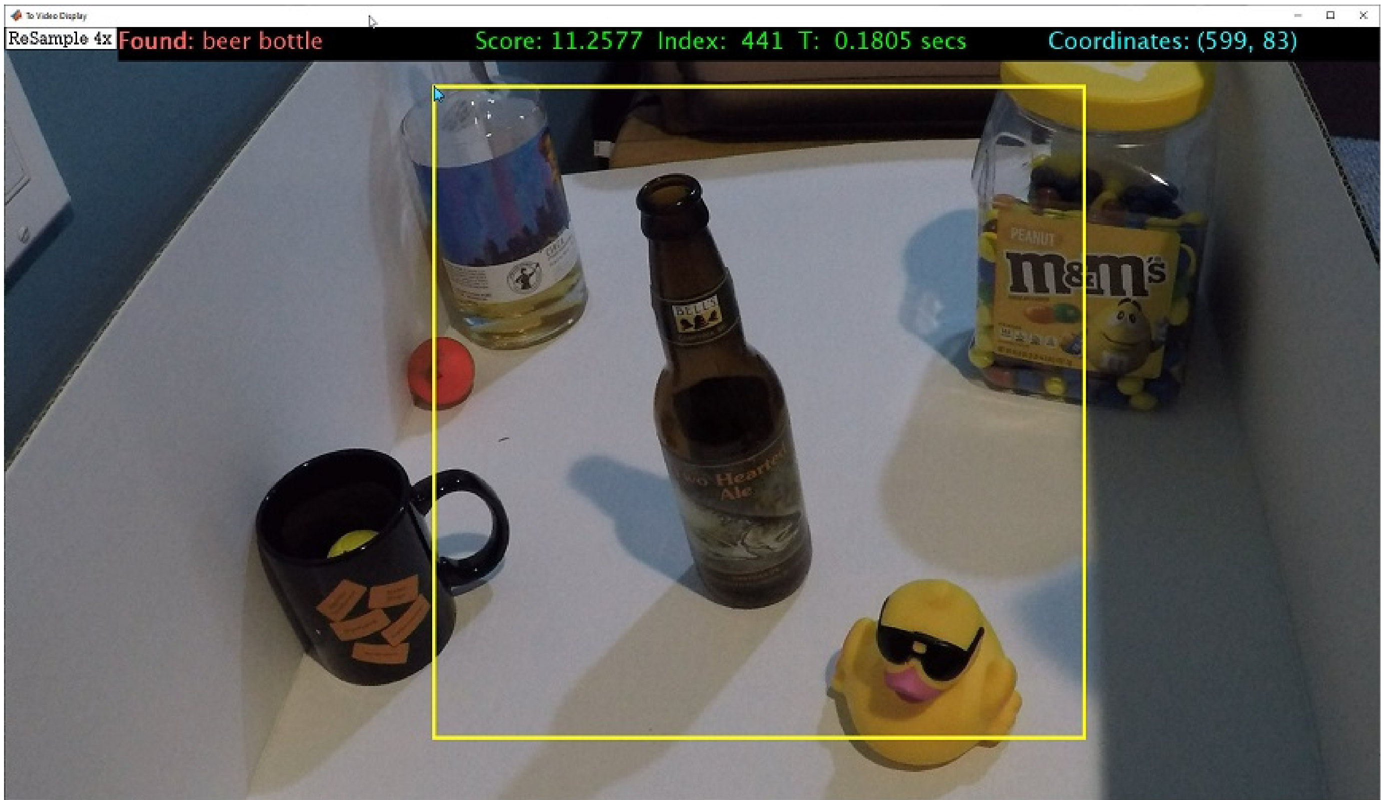
3. After you individually validate the preprocessing logic IP core and the deep learning processor IP core, you can prototype the entire integrated system on the FPGA board. Using Simulink External mode, instruct the FPGA preprocessing logic to send a processed input image frame to the DDR buffer, instruct the deep learning processor IP core to read from the same DDR buffer, and execute the prediction.

The deep learning processor IP core sends the result back to the external DDR memory. The software model running on the ARM processor retrieves the prediction result and annotates the prediction on the output video stream. This screenshot shows that you can read the ARM processor prediction result by using a serial connection.

COM5 - PuTTY

```
 1) envelope  8.8149 550
 2) laptop  7.6177 621
 3) binder  7.4577 447
 4) notebook  7.4564 682
 5) rule  7.4436 770
Class: envelope  Prob: 8.814860  Idx: 550.000000
SampleX: 2 XY:[117, 22]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7D
Top 5  Run 280
----------------------------------
 1) velvet 10.0684 886
 2) envelope  9.0011 550
 3) rule  8.8459 770
 4) wool  8.8402 912
 5) jean  8.4882 609
Class: velvet  Prob: 10.068416  Idx: 886.000000
SampleX: 2 XY:[970, 175]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7F
Top 5  Run 281
----------------------------------
 1) velvet 10.6247 886
 2) envelope  9.8796 550
 3) wool  9.2945 912
 4) rule  9.0598 770
 5) bath towel  8.8611 435
Class: velvet  Prob: 10.624667  Idx: 886.000000
SampleX: 2 XY:[993, 154]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7C
Top 5  Run 282
----------------------------------
 1) lipstick 10.4688 630
 2) pill bottle  8.7858 721
 3) beer bottle  8.5406 441
 4) thimble  8.4648 856
 5) saltshaker  8.3658 774
Class: lipstick  Prob: 10.468786  Idx: 630.000000
SampleX: 2 XY:[1084, 230]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7F
Top 5  Run 283
----------------------------------
 1) lipstick 10.1775 630
 2) pill bottle  9.0086 721
 3) loupe  8.8113 634
 4) hair spray  8.7907 586
 5) beer bottle  8.4889 441
Class: lipstick  Prob: 10.177537  Idx: 630.000000
SampleX: 2 XY:[1151, 233]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7D
Top 5  Run 284
----------------------------------
 1) beer bottle 15.1420 441
 2) whiskey jug 12.0200 902
 3) wine bottle 11.8346 908
 4) vase 11.3211 884
 5) pop bottle 11.2343 738
Class: beer bottle  Prob: 15.141971  Idx: 441.000000
SampleX: 2 XY:[1155, 233]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7E
Top 5  Run 285
----------------------------------
 1) beer bottle 15.5207 441
 2) pop bottle 11.7170 738
 3) wine bottle 11.7112 908
 4) whiskey jug 10.4509 902
 5) vase  9.9780 884
Class: beer bottle  Prob: 15.520669  Idx: 441.000000
```

This screenshot shows the frame captured from the output video stream which includes the ROI selection and the annotated prediction result.



4. After completing all your verification steps, manually deploy the entire reference design as an executable on the SD card on the ZCU102 board by using the ARM processor. Once the manual deployment is completed a MATLAB connection to the FPGA board is not required to operate the reference design.

See Also

`dlhdl.ProcessorConfig` | `dlhdl.buildProcessor`

More About

- "Use the Compiler Output for System Integration" on page 12-6

Running Convolution-Only Networks by Using FPGA Deployment

Typical series classification networks include a sequence of convolution layers followed by one or more fully connected layers. Recent research results indicate that better performance is achieved for feature extraction and recognition by using the convolution layer activations directly, instead of those from the subsequent fully connected layers.

To understand and debug convolutional networks, running and visualizing data is a useful tool. This example shows how to deploy, run, and debug a convolution-only network by using FPGA deployment..

Prerequisites

- Xilinx™ Zynq™ ZCU102 Evaluation Kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox™ Model for Resnet-50 Network

Resnet-50 Network

ResNet-50 is a convolutional neural network that is 50 layers deep. This pretrained network can classify images into 1000 object categories (such as keyboard, mouse, pencil, and more).The network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224. This example uses ResNet50 as a starting point.

Load Resnet-50 Network

Load the ResNet-50 network.

```
rnet = resnet50;
```

To visualize the structure of the Resnet-50 network, at the MATLAB® command prompt, enter:

```
analyzeNetwork(rnet)
```

Create A Convolution Only Network

A convolution only network is created by selecting a subset of the ResNet-50 network. The subset includes only the first five layers of the ResNet50 network which are convolutional in nature.

To create the convolution only network, enter:

```
layers = rnet.Layers(1:5);
outLayer = regressionLayer('Name','output');
layers(end+1) = outLayer;

snet = assembleNetwork(layers);
```

Create Target Object

To deploy the network on an FPGA, create a target object with a custom name and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use JTAG, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
%hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'D:/share/apps/HDLTools/Vivado/2020.2-
```

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pretrained convolutional only network, `snet`, as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example the target FPGA board is the Xilinx ZCU102 SOC board. The bitstream uses a single data type. Use the `dlhdl.Workflow` object to deploy networks which include both convolution and fully connected layers or only convolution layers.

```
hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'zcu102_single','Target',hTarget);
```

Compile Convolution Only Network

To compile the convolution only network, run the compile function of the `dlhdl.Workflow` object.

```
hW.compile
```

```
dn = hW.compile
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "24.0 MB"
    "SystemBufferOffset"        "0x03000000"     "28.0 MB"
    "InstructionDataOffset"     "0x04c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x05000000"     "4.0 MB"
    "EndOffset"                 "0x05400000"     "Total: 84.0 MB"

dn = struct with fields:
       Operators: [1×1 struct]
    LayerConfigs: [1×1 struct]
      NetConfigs: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. The function also downloads the network weights and biases. The deploy function programs the FPGA device, displays progress messages, and the time it takes to deploy the network.

```
hW.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Deep learning network programming has been skipped as the same network is already loaded on t
```

Load Example Image

Load and display an image to use as an input image to the series network.

```
I = imread('daisy.jpg');
imshow(I)
```

Run the Prediction

Execute the predict function of the `dlhdl.Workflow` object.

```
[P, speed] = hW.predict(single(I),'Profile','on');
```

```
### Finished writing input activations.
### Running single input activations.

              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------              -------------             ---------        --
Network                     2813005                  0.01279                      1           28
    conv_module             2813005                  0.01279
        conv1               2224168                  0.01011
        max_pooling2d_1      588864                  0.00268
 * The clock frequency of the DL processor is: 220MHz
```

The result data is returned as a 3-D array, with the third dimension indexing across the 64 feature images.

```
sz = size(P)
```

```
sz = 1×3

    56    56    64
```

To visualize all 64 features in a single image, the data is reshaped into four dimensions, which is appropriate input to the `imtile` function

```
R = reshape(P, [sz(1) sz(2) 1 sz(3)]);
sz = size(R)
```

```
sz = 1×4

    56    56     1    64
```

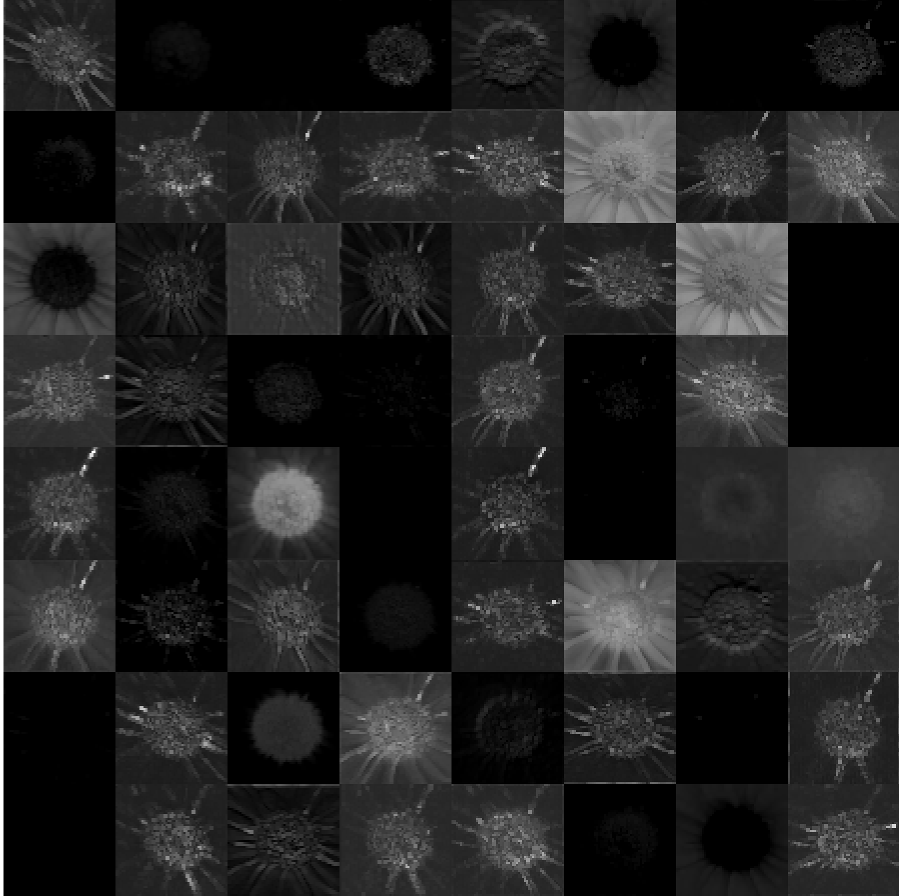
The third dimension in the input to `imtile` function represents the image color. Set the third dimension to size 1 because the activation signals in this example are scalars and do not include color. The fourth dimension indexes the channel.

The input to `imtile` is normalized using `mat2gray`. All values are scaled so that the minimum activation is 0 and the maximum activation is 1.

```
J = imtile(mat2gray(R), 'GridSize', [8 8]);
```

A grid size of 8-by-8 is selected because there are 64 features to display.

```
imshow(J)
```



The image shows activation data for each of the 64 features. Bright features indicate a strong activation.

The output from the convolutional layers only network differs from that of a network with convolution and fully connected layers. Convolution layers are used to reduce the input image size while maintaining features which are needed to get a good prediction. Convolution only layer networks are used to study feature extraction. Earlier convolution layers are used to extract low level features such as edges, colors, gradients and so on. Later convolution layers are used to extract high level features such as patterns, curves, lines and so on. These high level features can then be used to identify objects.

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `activations` | `compile` | `deploy` | `predict`

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2
- “Profile Inference Run” on page 5-4

Accelerate Prototyping Workflow for Large Networks by Using Ethernet

This example shows how to deploy a deep learning network and obtain prediction results using the Ethernet connection to your target device. You can significantly speed up the deployment and prediction times for large deep learning networks by using Ethernet versus JTAG. This example shows the workflow on a ZCU102 SoC board. The example also works on the other boards supported by Deep Learning HDL Toolbox. See “Supported Networks, Layers, Boards, and Tools” on page 7-2.

Prerequisites

- Xilinx ZCU102 SoC Development Kit. For help with board setup, see “Guided SD Card Set Up” (Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices).
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox™ Model for ResNet-18 Network

Introduction

Deep Learning HDL Toolbox establishes a connection between the host computer and FPGA board to prototype deep learning networks on hardware. This connection is used to deploy deep learning networks and run predictions. The connection provides two services:

- Programming the bitstream onto the FPGA
- Communicating with the design running on FPGA from MATLAB

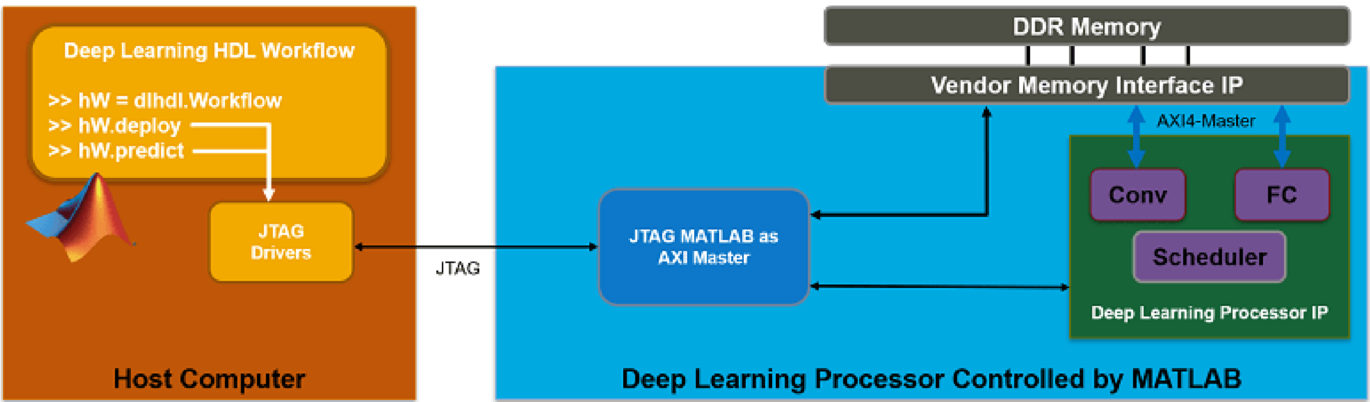
There are two hardware interfaces for establishing a connection between the host computer and FPGA board: JTAG and Ethernet.

JTAG Interface

The JTAG interface, programs the bitstream onto the FPGA over JTAG. The bitstream is not persistent through power cycles. You must reprogram the bitstream each time the FPGA is turned on.

MATLAB uses JTAG to control an AXI Master IP in the FPGA design, to communicate with the design running on the FPGA. You can use the AXI Master IP to read and write memory locations in the onboard memory and deep learning processor.

This figure shows the high-level architecture of the JTAG interface.

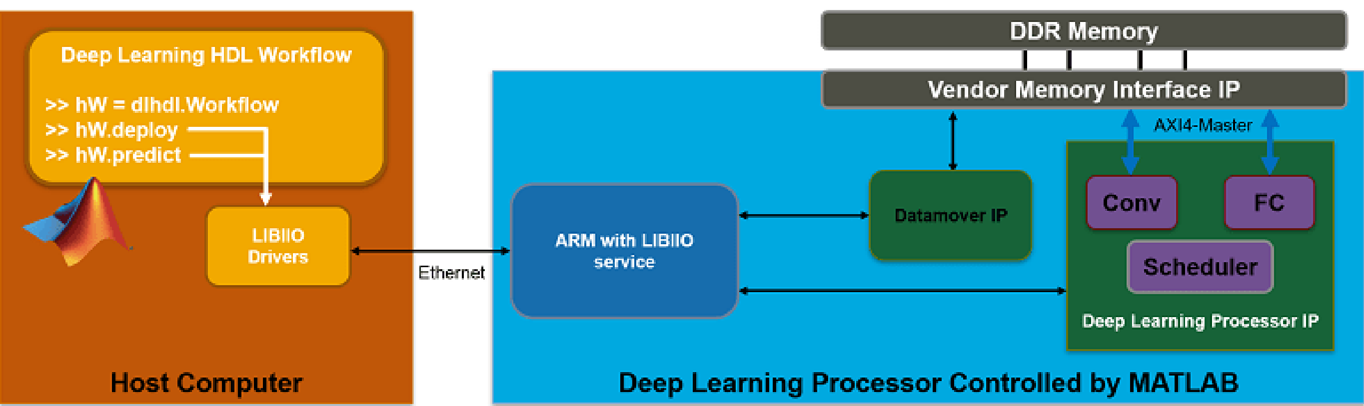


Ethernet Interface

The Ethernet interface leverages the ARM processor to send and receive information from the design running on the FPGA. The ARM processor runs on a Linux operating system. You can use the Linux operating system services to interact with the FPGA. When using the Ethernet interface, the bitstream is downloaded to the SD card. The bitstream is persistent through power cycles and is reprogrammed each time the FPGA is turned on. The ARM processor is configured with the correct device tree when the bitstream is programmed.

To communicate with the design running on the FPGA, MATLAB leverages the Ethernet connection between the host computer and ARM processor. The ARM processor runs a LIBIIO service, which communicates with a datamover IP in the FPGA design. The datamover IP is used for fast data transfers between the host computer and FPGA, which is useful when prototyping large deep learning networks that would have long transfer times over JTAG. The ARM processor generates the read and write transactions to access memory locations in both the onboard memory and deep learning processor.

The figure below shows the high-level architecture of the Ethernet interface.



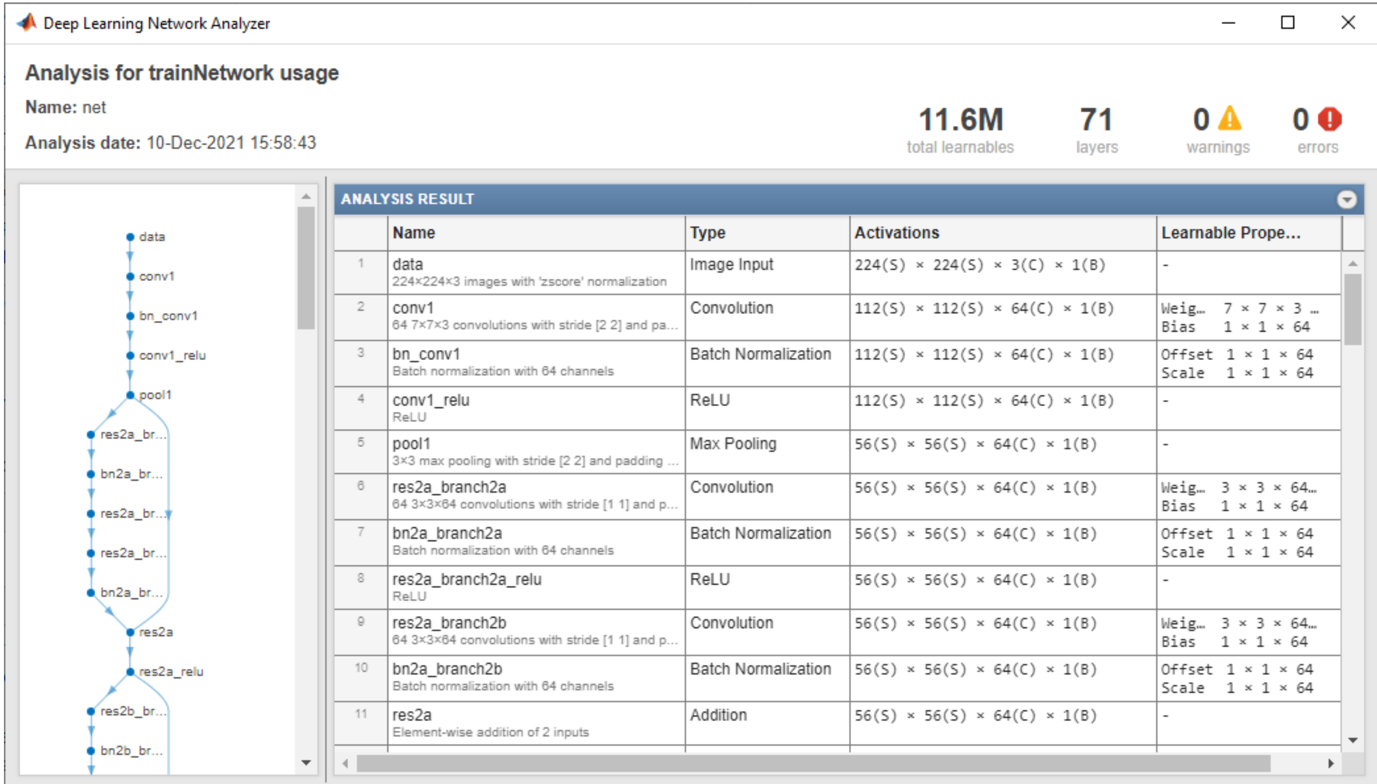
Load and Compile Deep Learning Network

This example uses the pretrained DAG network `resnet18`. This network is a larger network that has significant improvement in transfer time when deploying it to the FPGA by using Ethernet. To load `resnet18`, run the command:

```
net = resnet18;
```

The pretrained ResNet-18 network contains 71 layers including the input, convolution, batch normalization, ReLU, max pooling, addition, global average pooling, fully connected, and the softmax layers. To view the layers of the network enter:

```
analyzeNetwork(net);
```



To deploy the deep learning network on the target FPGA board, create a `dlhdl.Workflow` object that has the pretrained network `net` as the network and the bitstream for your target FPGA board. This example uses the bitstream `'zcu102_single'`, which has single data type and is configured for the ZCU102 board. To run this example on a different board, use the bitstream for your board.

```
hW = dlhdl.Workflow('Network', net, 'Bitstream', 'zcu102_single');
```

`Compile` the resnet18 network for deployment to the FPGA.

```
compile(hW);
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### The network includes the following layers:
     1   'data'                      Image Input             224×224×3 images with
     2   'conv1'                     Convolution             64 7×7×3 convolutions w
     3   'bn_conv1'                  Batch Normalization     Batch normalization wi
     4   'conv1_relu'                ReLU                    ReLU
     5   'pool1'                     Max Pooling             3×3 max pooling with s
     6   'res2a_branch2a'            Convolution             64 3×3×64 convolutions
     7   'bn2a_branch2a'             Batch Normalization     Batch normalization wi
     8   'res2a_branch2a_relu'       ReLU                    ReLU
     9   'res2a_branch2b'            Convolution             64 3×3×64 convolutions
```

```
    10   'bn2a_branch2b'              Batch Normalization     Batch normalization wit
    11   'res2a'                      Addition                Element-wise addition o
    12   'res2a_relu'                 ReLU                    ReLU
    13   'res2b_branch2a'             Convolution             64 3×3×64 convolutions
    14   'bn2b_branch2a'              Batch Normalization     Batch normalization wit
    15   'res2b_branch2a_relu'        ReLU                    ReLU
    16   'res2b_branch2b'             Convolution             64 3×3×64 convolutions
    17   'bn2b_branch2b'              Batch Normalization     Batch normalization wit
    18   'res2b'                      Addition                Element-wise addition o
    19   'res2b_relu'                 ReLU                    ReLU
    20   'res3a_branch2a'             Convolution             128 3×3×64 convolutions
    21   'bn3a_branch2a'              Batch Normalization     Batch normalization wit
    22   'res3a_branch2a_relu'        ReLU                    ReLU
    23   'res3a_branch2b'             Convolution             128 3×3×128 convolution
    24   'bn3a_branch2b'              Batch Normalization     Batch normalization wit
    25   'res3a'                      Addition                Element-wise addition o
    26   'res3a_relu'                 ReLU                    ReLU
    27   'res3a_branch1'              Convolution             128 1×1×64 convolutions
    28   'bn3a_branch1'               Batch Normalization     Batch normalization wit
    29   'res3b_branch2a'             Convolution             128 3×3×128 convolution
    30   'bn3b_branch2a'              Batch Normalization     Batch normalization wit
    31   'res3b_branch2a_relu'        ReLU                    ReLU
    32   'res3b_branch2b'             Convolution             128 3×3×128 convolution
    33   'bn3b_branch2b'              Batch Normalization     Batch normalization wit
    34   'res3b'                      Addition                Element-wise addition o
    35   'res3b_relu'                 ReLU                    ReLU
    36   'res4a_branch2a'             Convolution             256 3×3×128 convolution
    37   'bn4a_branch2a'              Batch Normalization     Batch normalization wit
    38   'res4a_branch2a_relu'        ReLU                    ReLU
    39   'res4a_branch2b'             Convolution             256 3×3×256 convolution
    40   'bn4a_branch2b'              Batch Normalization     Batch normalization wit
    41   'res4a'                      Addition                Element-wise addition o
    42   'res4a_relu'                 ReLU                    ReLU
    43   'res4a_branch1'              Convolution             256 1×1×128 convolution
    44   'bn4a_branch1'               Batch Normalization     Batch normalization wit
    45   'res4b_branch2a'             Convolution             256 3×3×256 convolution
    46   'bn4b_branch2a'              Batch Normalization     Batch normalization wit
    47   'res4b_branch2a_relu'        ReLU                    ReLU
    48   'res4b_branch2b'             Convolution             256 3×3×256 convolution
    49   'bn4b_branch2b'              Batch Normalization     Batch normalization wit
    50   'res4b'                      Addition                Element-wise addition o
    51   'res4b_relu'                 ReLU                    ReLU
    52   'res5a_branch2a'             Convolution             512 3×3×256 convolution
    53   'bn5a_branch2a'              Batch Normalization     Batch normalization wit
    54   'res5a_branch2a_relu'        ReLU                    ReLU
    55   'res5a_branch2b'             Convolution             512 3×3×512 convolution
    56   'bn5a_branch2b'              Batch Normalization     Batch normalization wit
    57   'res5a'                      Addition                Element-wise addition o
    58   'res5a_relu'                 ReLU                    ReLU
    59   'res5a_branch1'              Convolution             512 1×1×256 convolution
    60   'bn5a_branch1'               Batch Normalization     Batch normalization wit
    61   'res5b_branch2a'             Convolution             512 3×3×512 convolution
    62   'bn5b_branch2a'              Batch Normalization     Batch normalization wit
    63   'res5b_branch2a_relu'        ReLU                    ReLU
    64   'res5b_branch2b'             Convolution             512 3×3×512 convolution
    65   'bn5b_branch2b'              Batch Normalization     Batch normalization wit
    66   'res5b'                      Addition                Element-wise addition o
    67   'res5b_relu'                 ReLU                    ReLU
```

```
    68   'pool5'                             2-D Global Average Pooling   2-D global average poo
    69   'fc1000'                            Fully Connected              1000 fully connected la
    70   'prob'                              Softmax                      softmax
    71   'ClassificationLayer_predictions'   Classification Output        crossentropyex with 'te

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'data' of type 'ImageInputLayer' is split into 'data', 'data_norm_add', an
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification
### Compiling layer group: conv1>>pool1 ...
### Compiling layer group: conv1>>pool1 ... complete.
### Compiling layer group: res2a_branch2a>>res2a_branch2b ...
### Compiling layer group: res2a_branch2a>>res2a_branch2b ... complete.
### Compiling layer group: res2b_branch2a>>res2b_branch2b ...
### Compiling layer group: res2b_branch2a>>res2b_branch2b ... complete.
### Compiling layer group: res3a_branch1 ...
### Compiling layer group: res3a_branch1 ... complete.
### Compiling layer group: res3a_branch2a>>res3a_branch2b ...
### Compiling layer group: res3a_branch2a>>res3a_branch2b ... complete.
### Compiling layer group: res3b_branch2a>>res3b_branch2b ...
### Compiling layer group: res3b_branch2a>>res3b_branch2b ... complete.
### Compiling layer group: res4a_branch1 ...
### Compiling layer group: res4a_branch1 ... complete.
### Compiling layer group: res4a_branch2a>>res4a_branch2b ...
### Compiling layer group: res4a_branch2a>>res4a_branch2b ... complete.
### Compiling layer group: res4b_branch2a>>res4b_branch2b ...
### Compiling layer group: res4b_branch2a>>res4b_branch2b ... complete.
### Compiling layer group: res5a_branch1 ...
### Compiling layer group: res5a_branch1 ... complete.
### Compiling layer group: res5a_branch2a>>res5a_branch2b ...
### Compiling layer group: res5a_branch2a>>res5a_branch2b ... complete.
### Compiling layer group: res5b_branch2a>>res5b_branch2b ...
### Compiling layer group: res5b_branch2a>>res5b_branch2b ... complete.
### Compiling layer group: pool5 ...
### Compiling layer group: pool5 ... complete.
### Compiling layer group: fc1000 ...
### Compiling layer group: fc1000 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"        
    "OutputResultOffset"        "0x01800000"     "4.0 MB"         
    "SchedulerDataOffset"       "0x01c00000"     "8.0 MB"         
    "SystemBufferOffset"        "0x02400000"     "28.0 MB"        
    "InstructionDataOffset"     "0x04000000"     "4.0 MB"         
    "ConvWeightDataOffset"      "0x04400000"     "52.0 MB"        
    "FCWeightDataOffset"        "0x07800000"     "4.0 MB"         
    "EndOffset"                 "0x07c00000"     "Total: 124.0 MB"

### Network compilation complete.
```

The output displays the size of the compiled network which is 124 MB. The entire 124 MB is transferred to the FPGA by using the `deploy` method. Due to the large size of the network, the

transfer can take a significant amount of time if using JTAG. When using Ethernet, the transfer happens quickly.

Deploy Deep Learning Network to FPGA

Before deploying a network, you must first establish a connection to the FPGA board. The `dlhdl.Target` object represents this connection between the host computer and the FPGA. Create two target objects, one for connection through the JTAG interface and one for connection through the Ethernet interface. To use the JTAG connection, install Xilinx™ Vivado™ Design Suite 2020.2 and set the path to your installed Xilinx Vivado executable if it is not already set up.

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.b
hTargetJTAG = dlhdl.Target('Xilinx', 'Interface', 'JTAG')
```

```
hTargetJTAG = 
  TargetJTAG with properties:

    Interface: JTAG
       Vendor: 'Xilinx'
```

```
hTargetEthernet = dlhdl.Target('Xilinx', 'Interface', 'Ethernet')
```

```
hTargetEthernet = 
  TargetEthernet with properties:

    Interface: Ethernet
    IPAddress: '192.168.1.101'
     Username: 'root'
         Port: 22
       Vendor: 'Xilinx'
```

To deploy the network, assign the target object to the `dlhdl.Workflow` object and execute the `deploy` method. The deployment happens in two stages. First, the bitstream is programmed onto the FPGA. Then, the network is transferred to the onboard memory.

Select the JTAG interface and time the operation. This operation might take several minutes.

```
hW.Target = hTargetJTAG;
tic;
deploy(hW);
```

```
### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 13-Dec-2021 13:55:43
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 13-Dec-2021 13:55:51
```

```
elapsedTimeJTAG = toc
```

```
elapsedTimeJTAG = 419.3838
```

Use the Ethernet interface by setting the `dlhdl.Workflow` target object to `hTargetEthernet` and running the `deploy` function. There is a significant acceleration in the network deployment when you use Ethernet to deploy the bitstream and network to the FPGA.

```
hW.Target = hTargetEthernet;
tic;
deploy(hW);
```

```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'

Downloading target FPGA device configuration over Ethernet to SD card done. The system will now


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 13-Dec-2021 13:56:31
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 13-Dec-2021 13:56:31
```

```
elapsedTimeEthernet = toc
```

```
elapsedTimeEthernet = 39.4850
```

Changing from JTAG to Ethernet, the `deploy` function reprograms the bitstream, which accounts for most of the elapsed time. Reprogramming is due to different methods that are used to program the bitstream for the different hardware interfaces. The Ethernet interface configures the ARM processor and uses a persistent programming method so that the bitstream is reprogrammed each time the board is turned on. When deploying different deep learning networks by using the same bitstream and hardware interface, you can skip the bitstream programming, which further speeds up network deployment.

Run Prediction for Example Image

Run a prediction for an example image by using the `predict` method.

```
imgFile = 'monitor.jpg';
inputImg = imresize(imread(imgFile), [224,224]);
imshow(inputImg)
```

```
prediction = predict(hW,single(inputImg));

### Finished writing input activations.
### Running single input activation.

[val, idx] = max(prediction);
result = net.Layers(end).ClassNames{idx}

result =
'monitor'
```

Release any hardware resources associated with the `dlhdl.Target` objects.

```
release(hTargetJTAG)
release(hTargetEthernet)
```

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `activations` | `compile` | `deploy` | `predict`

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2
- “Profile Inference Run” on page 5-4

Create Series Network for Quantization

This example shows how to fine-tune a pretrained AlexNet convolutional neural network to perform classification on a new collection of images.

AlexNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input and outputs a label for the object in the image together with the probabilities for each of the object categories.

Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training images.

Load Training Data

Unzip and load the new images as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

```
unzip('logos_dataset.zip');

imds = imageDatastore('logos_dataset', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. `splitEachLabel` splits the `images` datastore into two new datastores.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

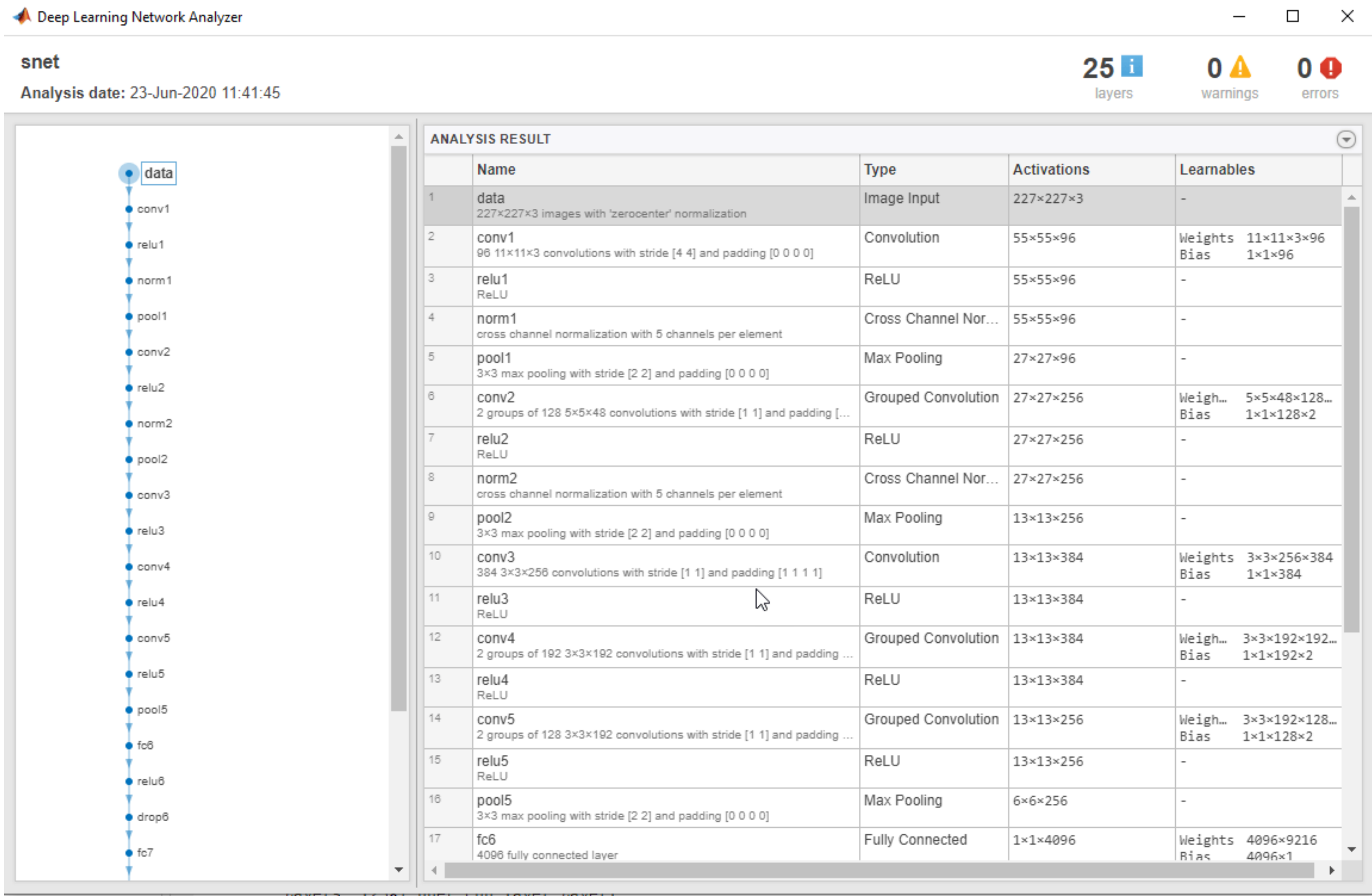
Load Pretrained Network

Load the pretrained AlexNet neural network. If Deep Learning Toolbox™ Model *for AlexNet Network* is not installed, then the software provides a download link. AlexNet is trained on more than one million images and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the model has learned rich feature representations for a wide range of images.

```
snet = alexnet;
```

Use `analyzeNetwork` to display an interactive visualization of the network architecture and detailed information about the network layers.

```
analyzeNetwork(snet)
```



	Name	Type	Activations	Learnables
1	data 227×227×3 images with 'zerocenter' normalization	Image Input	227×227×3	-
2	conv1 96 11×11×3 convolutions with stride [4 4] and padding [0 0 0 0]	Convolution	55×55×96	Weights 11×11×3×96 Bias 1×1×96
3	relu1 ReLU	ReLU	55×55×96	-
4	norm1 cross channel normalization with 5 channels per element	Cross Channel Nor...	55×55×96	-
5	pool1 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	27×27×96	-
6	conv2 2 groups of 128 5×5×48 convolutions with stride [1 1] and padding [...	Grouped Convolution	27×27×256	Weigh... 5×5×48×128... Bias 1×1×128×2
7	relu2 ReLU	ReLU	27×27×256	-
8	norm2 cross channel normalization with 5 channels per element	Cross Channel Nor...	27×27×256	-
9	pool2 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	13×13×256	-
10	conv3 384 3×3×256 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	13×13×384	Weights 3×3×256×384 Bias 1×1×384
11	relu3 ReLU	ReLU	13×13×384	-
12	conv4 2 groups of 192 3×3×192 convolutions with stride [1 1] and padding ...	Grouped Convolution	13×13×384	Weigh... 3×3×192×192... Bias 1×1×192×2
13	relu4 ReLU	ReLU	13×13×384	-
14	conv5 2 groups of 128 3×3×192 convolutions with stride [1 1] and padding ...	Grouped Convolution	13×13×256	Weigh... 3×3×192×128... Bias 1×1×128×2
15	relu5 ReLU	ReLU	13×13×256	-
16	pool5 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	6×6×256	-
17	fc6 4096 fully connected layer	Fully Connected	1×1×4096	Weights 4096×9216 Bias 4096×1

The first layer, the image input layer, requires input images of size 227-by-227-by-3, where 3 is the number of color channels.

```
inputSize = snet.Layers(1).InputSize
```

```
inputSize = 1×3

   227   227     3
```

Replace Final Layers

The last three layers of the pretrained network `net` are configured for 1000 classes. These three layers must be fine-tuned for the new classification problem. Extract all layers, except the last three, from the pretrained network.

```
layersTransfer = snet.Layers(1:end-3);
```

Transfer the layers to the new classification task by replacing the last three layers with a fully connected layer, a softmax layer, and a classification output layer. Specify the options of the new fully connected layer according to the new data. Set the fully connected layer to have the same size as the number of classes in the new data. To learn faster in the new layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the fully connected layer.

```
numClasses = numel(categories(imdsTrain.Labels))
```

```
numClasses = 32
```

```
layers = [
    layersTransfer
    fullyConnectedLayer(numClasses,'WeightLearnRateFactor',20,'BiasLearnRateFactor',20)
    softmaxLayer
    classificationLayer];
```

Train Network

The network requires input images of size 227-by-227-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

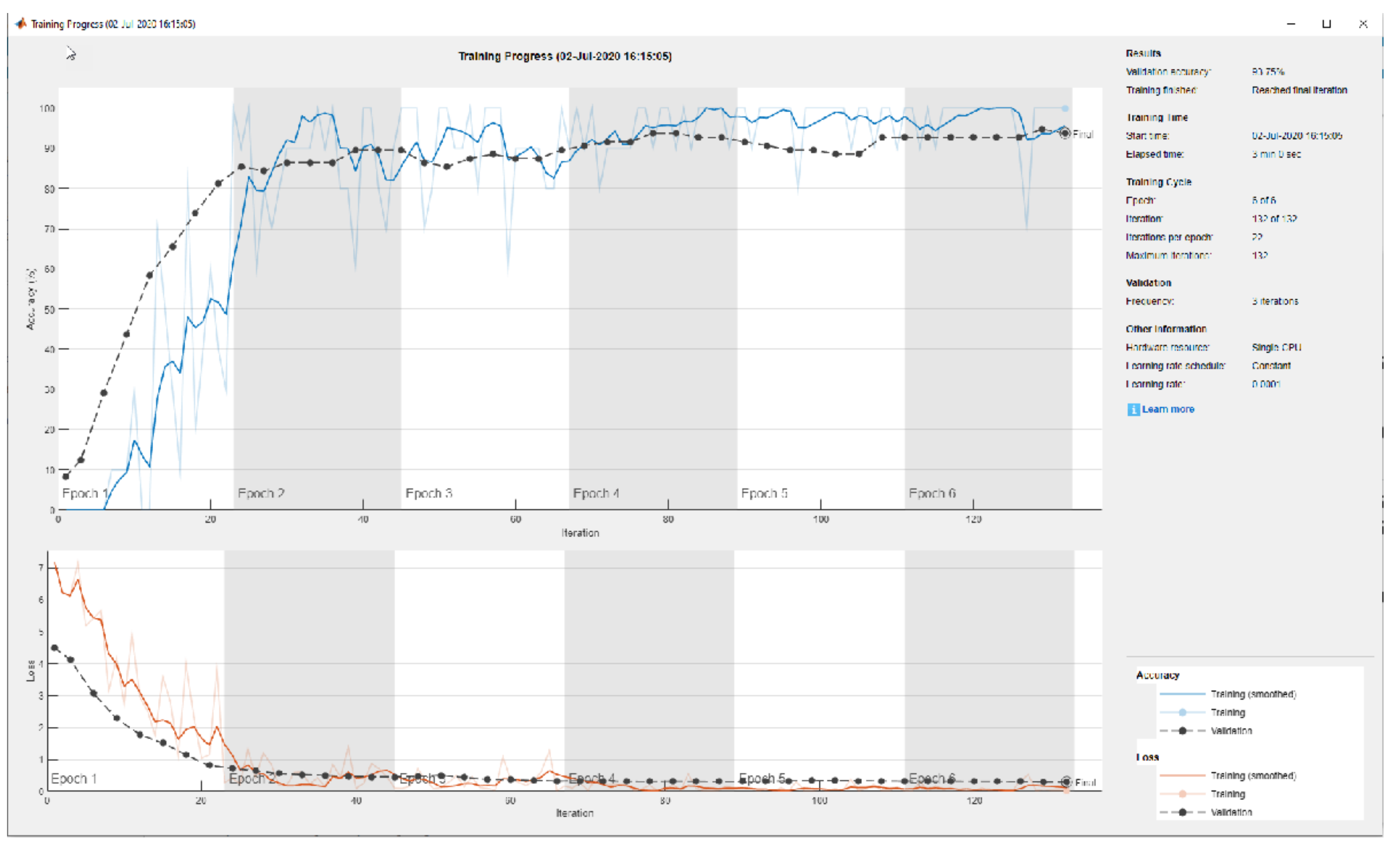
```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate factors for the fully connected layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size and validation data. The software validates the network every `ValidationFrequency` iterations during training.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',10, ...
    'MaxEpochs',6, ...
    'InitialLearnRate',1e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',3, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network that consists of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 6.1, 6.3, or higher). Otherwise, it uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

```
netTransfer = trainNetwork(augimdsTrain,layers,options);
```


Training Progress (02-Jul-2020 16:15:05)
Results
Validation accuracy:
Training finished:
Reached final iteration
Training Time
Start time:
02-Jul-2020 16:15:05
Elapsed time:
3 min 0 sec
Training Cycle
Epoch:
6 of 6
Iteration:
132 of 132
Iterations per epoch:
22
Maximum iterations:
132
Validation
Frequency:
3 iterations
Other Information
Hardware resource:
Single CPU
Learning rate schedule:
Constant
Learning rate:
0.0001
Learn more
Accuracy
Training (smoothed)
Training
Validation
Loss
Training (smoothed)
Training
Validation
Accuracy (%)
Loss
Iteration
Epoch 1
Epoch 2
Epoch 3
Epoch 4
Epoch 5
Epoch 6
Final

Custom Deep Learning Processor Generation to Meet Performance Requirements

This example shows how to create a custom processor configuration and estimate the performance of a pretrained series network. You can then modify parameters of the custom processor configuration and re-estimate the performance. Once you have achieved your performance requirements you can generate a custom bitstream by using the custom processor configuration.

Prerequisites

- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library
- MATLAB Coder Interface for Deep Learning

Load Pretrained Series Network

To load the pretrained series network LogoNet, enter:

```
snet = getLogoNetwork;
```

Define Training and Validation Data Sets

This example uses the `logos_dataset` data set. The data set consists of 320 images. Create an `augmentedImageDatastore` object to use for training and validation.

```
curDir = pwd;
unzip('logos_dataset.zip');

imds = imageDatastore('logos_dataset', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

Create Custom Processor Configuration

To create a custom processor configuration, use the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

```
hPC = dlhdl.ProcessorConfig;
hPC.TargetFrequency = 220;
hPC
```

```
hPC = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                             ConvThreadNumber: 16
                              InputMemorySize: [227 227 3]
                             OutputMemorySize: [227 227 3]
```

```
                    FeatureSizeLimit: 2048

              Processing Module "fc"
                    ModuleGeneration: 'on'
              SoftmaxBlockGeneration: 'off'
              SigmoidBlockGeneration: 'off'
                      FCThreadNumber: 4
                     InputMemorySize: 25088
                    OutputMemorySize: 4096

          Processing Module "custom"
                    ModuleGeneration: 'on'
                            Addition: 'on'
                      Multiplication: 'on'
                            Resize2D: 'off'
                             Sigmoid: 'off'
                           TanhLayer: 'off'
                     InputMemorySize: 40
                    OutputMemorySize: 120

      Processor Top Level Properties
                      RunTimeControl: 'register'
                       RunTimeStatus: 'register'
                  InputStreamControl: 'register'
                 OutputStreamControl: 'register'
                        SetupControl: 'register'
                   ProcessorDataType: 'single'

             System Level Properties
                      TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                     TargetFrequency: 220
                       SynthesisTool: 'Xilinx Vivado'
                     ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
             SynthesisToolChipFamily: 'Zynq UltraScale+'
             SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
            SynthesisToolPackageName: ''
             SynthesisToolSpeedValue: ''
```

Estimate LogoNet Performance

To `estimate` the performance of the LogoNet series network, use the `estimatePerformance` function of the `dlhdl.ProcessorConfig` object. The function returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

```
hPC.estimatePerformance(snet)
```

```
### Notice: The layer 'imageinput' of type 'ImageInputLayer' is split into an image input layer
### The network includes the following layers:
     1   'imageinput'    Image Input             227×227×3 images with 'zerocenter' normalizatio
     2   'conv_1'        2-D Convolution         96 5×5×3 convolutions with stride [1  1] and pa
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     2-D Max Pooling         3×3 max pooling with stride [2  2] and padding
     5   'conv_2'        2-D Convolution         128 3×3×96 convolutions with stride [1  1] and
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     2-D Max Pooling         3×3 max pooling with stride [2  2] and padding
     8   'conv_3'        2-D Convolution         384 3×3×128 convolutions with stride [1  1] and
     9   'relu_3'        ReLU                    ReLU
```

```
    10   'maxpool_3'     2-D Max Pooling        3×3 max pooling with stride [2  2] and padding
    11   'conv_4'        2-D Convolution        128 3×3×384 convolutions with stride [2  2] and
    12   'relu_4'        ReLU                   ReLU
    13   'maxpool_4'     2-D Max Pooling        3×3 max pooling with stride [2  2] and padding
    14   'fc_1'          Fully Connected        2048 fully connected layer
    15   'relu_5'        ReLU                   ReLU
    16   'fc_2'          Fully Connected        2048 fully connected layer
    17   'relu_6'        ReLU                   ReLU
    18   'fc_3'          Fully Connected        32 fully connected layer
    19   'softmax'       Softmax                softmax
    20   'classoutput'   Classification Output  crossentropyex with 'adidas' and 31 other classe

### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------             ---------        --
Network                    39199107                  0.17818                       1           391
    ____imageinput_norm      216472                  0.00098
    ____conv_1              6832680                  0.03106
    ____maxpool_1           3705912                  0.01685
    ____conv_2             10454501                  0.04752
    ____maxpool_2           1173810                  0.00534
    ____conv_3              9364533                  0.04257
    ____maxpool_3           1229970                  0.00559
    ____conv_4              1759348                  0.00800
    ____maxpool_4             24450                  0.00011
    ____fc_1                2651288                  0.01205
    ____fc_2                1696632                  0.00771
    ____fc_3                  89511                  0.00041
 * The clock frequency of the DL processor is: 220MHz
```

The estimated frames per second is 5.5 Frames/s. To improve the network performance, modify the custom processor convolution module kernel data type, convolution processor thread number, fully connected module kernel data type, and fully connected module thread number. For more information about these processor parameters, see `getModuleProperty` and `setModuleProperty`.

Create Modified Custom Processor Configuration

To create a custom processor configuration, use the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

```
hPCNew = dlhdl.ProcessorConfig;
hPCNew.TargetFrequency = 300;
hPCNew.ProcessorDataType = 'int8';
hPCNew.setModuleProperty('conv', 'ConvThreadNumber', 64);
hPCNew.setModuleProperty('fc', 'FCThreadNumber',   16);
hPCNew
```

```
hPCNew = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
```

```
                    ConvThreadNumber: 64
                     InputMemorySize: [227 227 3]
                    OutputMemorySize: [227 227 3]
                    FeatureSizeLimit: 2048

              Processing Module "fc"
                    ModuleGeneration: 'on'
              SoftmaxBlockGeneration: 'off'
              SigmoidBlockGeneration: 'off'
                      FCThreadNumber: 16
                     InputMemorySize: 25088
                    OutputMemorySize: 4096

          Processing Module "custom"
                    ModuleGeneration: 'on'
                            Addition: 'on'
                      Multiplication: 'on'
                            Resize2D: 'off'
                             Sigmoid: 'off'
                           TanhLayer: 'off'
                     InputMemorySize: 40
                    OutputMemorySize: 120

      Processor Top Level Properties
                      RunTimeControl: 'register'
                       RunTimeStatus: 'register'
                  InputStreamControl: 'register'
                 OutputStreamControl: 'register'
                        SetupControl: 'register'
                   ProcessorDataType: 'int8'

             System Level Properties
                      TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                     TargetFrequency: 300
                       SynthesisTool: 'Xilinx Vivado'
                     ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
             SynthesisToolChipFamily: 'Zynq UltraScale+'
             SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
            SynthesisToolPackageName: ''
             SynthesisToolSpeedValue: ''
```

Quantize LogoNet Series Network

To quantize the LogoNet network, enter:

```
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
imageData_reduced = imageData.subset(1:20);
dlquantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
dlquantObj.calibrate(imageData_reduced)
```

Estimate LogoNet Performance

To estimate the performance of the LogoNet series network, use the `estimatePerformance` function of the `dlhdl.ProcessorConfig` object. The function returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

```
hPCNew.estimatePerformance(dlquantObj)
```

```
### The network includes the following layers:
     1   'imageinput'    Image Input             227×227×3 images with 'zerocenter' normalization
     2   'conv_1'        2-D Convolution         96 5×5×3 convolutions with stride [1  1] and pad
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     2-D Max Pooling         3×3 max pooling with stride [2  2] and padding
     5   'conv_2'        2-D Convolution         128 3×3×96 convolutions with stride [1  1] and p
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     2-D Max Pooling         3×3 max pooling with stride [2  2] and padding
     8   'conv_3'        2-D Convolution         384 3×3×128 convolutions with stride [1  1] and
     9   'relu_3'        ReLU                    ReLU
    10   'maxpool_3'     2-D Max Pooling         3×3 max pooling with stride [2  2] and padding
    11   'conv_4'        2-D Convolution         128 3×3×384 convolutions with stride [2  2] and
    12   'relu_4'        ReLU                    ReLU
    13   'maxpool_4'     2-D Max Pooling         3×3 max pooling with stride [2  2] and padding
    14   'fc_1'          Fully Connected         2048 fully connected layer
    15   'relu_5'        ReLU                    ReLU
    16   'fc_2'          Fully Connected         2048 fully connected layer
    17   'relu_6'        ReLU                    ReLU
    18   'fc_3'          Fully Connected         32 fully connected layer
    19   'softmax'       Softmax                 softmax
    20   'classoutput'   Classification Output   crossentropyex with 'adidas' and 31 other classe

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    13829465                  0.04610                       1           138
    ____conv_1              3487680                  0.01163
    ____maxpool_1           1852092                  0.00617
    ____conv_2              2939191                  0.00980
    ____maxpool_2            586689                  0.00196
    ____conv_3              2577951                  0.00859
    ____maxpool_3            614769                  0.00205
    ____conv_4               611644                  0.00204
    ____maxpool_4             12201                  0.00004
    ____fc_1                 665265                  0.00222
    ____fc_2                 425425                  0.00142
    ____fc_3                  56558                  0.00019
 * The clock frequency of the DL processor is: 300MHz
```

The estimated frames per second is 21.7 Frames/s.

Generate Custom Processor and Bitstream

Use the new custom processor configuration to build and generate a custom processor and bitstream. Use the custom bitstream to deploy the LogoNet network to your target FPGA board.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
dlhdl.buildProcessor(hPCNew);
```

To learn how to use the generated bitstream file, see "Generate Custom Bitstream" on page 9-2.

The generated bitstream in this example is similar to the `zcu102_int8` bitstream. To deploy the quantized LogoNet network using the `zcu102_int8` bitstream, see “Classify Images on FPGA Using Quantized Neural Network” on page 10-145.

See Also

`dlhdl.ProcessorConfig` | `estimatePerformance` | `estimateResources`

More About

- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10

Quantize Network for FPGA Deployment

Reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. This example shows how to use Deep Learning Toolbox Model Quantization Library and Deep Learning HDL Toolbox to deploy the `int8` network to a target FPGA board.

For this example, you need:

- Deep Learning Toolbox ™
- Deep Learning HDL Toolbox ™
- Deep Learning Toolbox Model Quantization Library
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- MATLAB Coder Interface for Deep Learning.

Load Pretrained Network

Load the pretrained LogoNet network and analyze the network architecture.

```
snet = getLogoNetwork;
deepNetworkDesigner(snet);
```

Load Data

This example uses the logos_dataset data set. The data set consists of 320 images. Each image is 227-by-227 in size and has three color channels (RGB). Create an `augmentedImageDatastore` object for calibration and validation. Expedite calibration and validation by reducing the calibration data set to 20 images. The MATLAB simulation workflow has a maximum limit of five images when validating the quantized network. Reduce the validation data set sizes to five images. The FPGA validation workflow has a maximum limit of one image when validating the quantized network. Reduce the FPGA validation data set to a single image.

```
curDir = pwd;
unzip("logos_dataset.zip");
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
calibrationData_reduced = calibrationData.subset(1:20);
validationData_simulation = validationData.subset(1:5);
validationData_FPGA = validationData.subset(1:1);
```

Generate Calibration Result File for the Network

Create a `dlquantizer` object and specify the network to quantize. Specify the execution environment as FPGA.

```
dlQuantObj_simulation = dlquantizer(snet,'ExecutionEnvironment',"FPGA",'Simulation','on');
dlQuantObj_FPGA = dlquantizer(snet,'ExecutionEnvironment',"FPGA");
```

Use the `calibrate` function to exercise the network with sample inputs and collect the range information. The `calibrate` function collects the dynamic ranges of the weights and biases. The calibrate function returns a table. Each row of the table contains range information for a learnable parameter of the quantized network.

```
calibrate(dlQuantObj_simulation,calibrationData_reduced)
```

```
ans=35×5 table
         Optimized Layer Name         Network Layer Name    Learnables / Activations     MinValue
    ____________________________    __________________    ________________________    ___________

    {'conv_1_Weights'          }      {'conv_1'    }           "Weights"               -0.048978
    {'conv_1_Bias'             }      {'conv_1'    }           "Bias"                    0.99996
    {'conv_2_Weights'          }      {'conv_2'    }           "Weights"               -0.055518
    {'conv_2_Bias'             }      {'conv_2'    }           "Bias"                -0.00061171
    {'conv_3_Weights'          }      {'conv_3'    }           "Weights"               -0.045942
    {'conv_3_Bias'             }      {'conv_3'    }           "Bias"                 -0.0013998
    {'conv_4_Weights'          }      {'conv_4'    }           "Weights"               -0.045967
    {'conv_4_Bias'             }      {'conv_4'    }           "Bias"                   -0.00164
    {'fc_1_Weights'            }      {'fc_1'      }           "Weights"               -0.051394
    {'fc_1_Bias'               }      {'fc_1'      }           "Bias"                -0.00052319
    {'fc_2_Weights'            }      {'fc_2'      }           "Weights"                -0.05016
    {'fc_2_Bias'               }      {'fc_2'      }           "Bias"                 -0.0017564
    {'fc_3_Weights'            }      {'fc_3'      }           "Weights"               -0.050706
    {'fc_3_Bias'               }      {'fc_3'      }           "Bias"                   -0.02951
    {'imageinput'              }      {'imageinput'}           "Activations"                   0
    {'imageinput_normalization'}      {'imageinput'}           "Activations"             -139.34
      ⋮
```

```
calibrate(dlQuantObj_FPGA,calibrationData_reduced)
```

```
ans=35×5 table
         Optimized Layer Name         Network Layer Name    Learnables / Activations     MinValue
    ____________________________    __________________    ________________________    ___________

    {'conv_1_Weights'          }      {'conv_1'    }           "Weights"               -0.048978
    {'conv_1_Bias'             }      {'conv_1'    }           "Bias"                    0.99996
    {'conv_2_Weights'          }      {'conv_2'    }           "Weights"               -0.055518
    {'conv_2_Bias'             }      {'conv_2'    }           "Bias"                -0.00061171
    {'conv_3_Weights'          }      {'conv_3'    }           "Weights"               -0.045942
    {'conv_3_Bias'             }      {'conv_3'    }           "Bias"                 -0.0013998
    {'conv_4_Weights'          }      {'conv_4'    }           "Weights"               -0.045967
    {'conv_4_Bias'             }      {'conv_4'    }           "Bias"                   -0.00164
    {'fc_1_Weights'            }      {'fc_1'      }           "Weights"               -0.051394
    {'fc_1_Bias'               }      {'fc_1'      }           "Bias"                -0.00052319
    {'fc_2_Weights'            }      {'fc_2'      }           "Weights"                -0.05016
    {'fc_2_Bias'               }      {'fc_2'      }           "Bias"                 -0.0017564
    {'fc_3_Weights'            }      {'fc_3'      }           "Weights"               -0.050706
    {'fc_3_Bias'               }      {'fc_3'      }           "Bias"                   -0.02951
    {'imageinput'              }      {'imageinput'}           "Activations"                   0
    {'imageinput_normalization'}      {'imageinput'}           "Activations"             -139.34
      ⋮
```

Create Target Object

Create a target object with a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use JTAG, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado
\2020.2\bin\vivado.bat');
```

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Alternatively, you can also use the JTAG interface.

```
% hTarget = dlhdl.Target('Xilinx', 'Interface', 'JTAG');
```

Create dlQuantizationOptions Object

Create a `dlquantizationOptions` object. Specify the target bitstream and target board interface. The default metric function is a Top-1 accuracy metric function.

```
options_FPGA = dlquantizationOptions('Bitstream','zcu102_int8','Target',hTarget);
options_simulation = dlquantizationOptions;
```

To use a custom metric function, specify the metric function in the `dlquantizationOptions` object.

```
options_FPGA = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationData_FPGA
options_simulation = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x,snet,validationDat
```

Validate Quantized Network

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network. The `validate` function simulates the quantized network in MATLAB. The `validate` function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the single-data-type network object to the results of the quantized network object.

```
prediction_simulation = dlQuantObj_simulation.validate(validationData_simulation,options_simulati
```

```
Compiling leg: conv_1>>relu_4 ...
Compiling leg: conv_1>>relu_4 ... complete.
Compiling leg: maxpool_4 ...
Compiling leg: maxpool_4 ... complete.
Compiling leg: fc_1>>fc_3 ...
Compiling leg: fc_1>>fc_3 ... complete.

prediction_simulation = struct with fields:
       NumSamples: 5
    MetricResults: [1×1 struct]
       Statistics: []
```

For validation on an FPGA, the validate function:

- Programs the FPGA board by using the output of the `compile` method and the programming file
- Downloads the network weights and biases
- Compares the performance of the network before and after quantization

```
prediction_FPGA = dlQuantObj_FPGA.validate(validationData_FPGA,options_FPGA)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### The network includes the following layers:
     1   'imageinput'    Image Input             227×227×3 images with 'zerocenter' normalizatio
     2   'conv_1'        Convolution             96 5×5×3 convolutions with stride [1  1] and pad
     3   'relu_1'        ReLU                    ReLU
```

```
     4   'maxpool_1'     Max Pooling             3×3 max pooling with stride [2  2] and padding
     5   'conv_2'        Convolution             128 3×3×96 convolutions with stride [1  1] and p
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     Max Pooling             3×3 max pooling with stride [2  2] and padding
     8   'conv_3'        Convolution             384 3×3×128 convolutions with stride [1  1] and
     9   'relu_3'        ReLU                    ReLU
    10   'maxpool_3'     Max Pooling             3×3 max pooling with stride [2  2] and padding
    11   'conv_4'        Convolution             128 3×3×384 convolutions with stride [2  2] and
    12   'relu_4'        ReLU                    ReLU
    13   'maxpool_4'     Max Pooling             3×3 max pooling with stride [2  2] and padding
    14   'fc_1'          Fully Connected         2048 fully connected layer
    15   'relu_5'        ReLU                    ReLU
    16   'dropout_1'     Dropout                 50% dropout
    17   'fc_2'          Fully Connected         2048 fully connected layer
    18   'relu_6'        ReLU                    ReLU
    19   'dropout_2'     Dropout                 50% dropout
    20   'fc_3'          Fully Connected         32 fully connected layer
    21   'softmax'       Softmax                 softmax
    22   'classoutput'   Classification Output   crossentropyex with 'adidas' and 31 other classe

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple
### Compiling layer group: conv_1>>relu_4 ...
### Compiling layer group: conv_1>>relu_4 ... complete.
### Compiling layer group: maxpool_4 ...
### Compiling layer group: maxpool_4 ... complete.
### Compiling layer group: fc_1>>fc_3 ...
### Compiling layer group: fc_1>>fc_3 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "12.0 MB"
    "OutputResultOffset"        "0x00c00000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01000000"     "4.0 MB"
    "SystemBufferOffset"        "0x01400000"     "36.0 MB"
    "InstructionDataOffset"     "0x03800000"     "8.0 MB"
    "ConvWeightDataOffset"      "0x04000000"     "12.0 MB"
    "FCWeightDataOffset"        "0x04c00000"     "12.0 MB"
    "EndOffset"                 "0x05800000"     "Total: 88.0 MB"

### Network compilation complete.

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Deep learning network programming has been skipped as the same network is already loaded on t
### Finished writing input activations.
### Running single input activation.


              Deep Learning Processor Bitstream Build Info

Resource                   Utilized           Total        Percentage
------------------        ----------      ------------    ------------
LUTs (CLB/ALM)*               248358            274080           90.62
DSPs                             384              2520           15.24
```

```
Block RAM                        581                912           63.71
* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive L

### Notice: The layer 'imageinput' of type 'ImageInputLayer' is split into an image input layer 
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------              ---------        --
Network                   40142478                  0.18247                       1          401
    ____imageinput_norm     216472                  0.00098
    ____conv_1             6825671                  0.03103
    ____maxpool_1          3755088                  0.01707
    ____conv_2            10440701                  0.04746
    ____maxpool_2          1447840                  0.00658
    ____conv_3             9405685                  0.04275
    ____maxpool_3          1765856                  0.00803
    ____conv_4             1819636                  0.00827
    ____maxpool_4            28098                  0.00013
    ____fc_1               2651288                  0.01205
    ____fc_2               1696632                  0.00771
    ____fc_3                 89511                  0.00041
 * The clock frequency of the DL processor is: 220MHz



              Deep Learning Processor Bitstream Build Info

Resource                   Utilized           Total        Percentage
------------------        ----------      ------------    ------------
LUTs (CLB/ALM)*              168645           274080           61.53
DSPs                            800             2520           31.75
Block RAM                       453              912           49.67
* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive L

### Finished writing input activations.
### Running single input activation.

prediction_FPGA = struct with fields:
       NumSamples: 1
    MetricResults: [1×1 struct]
       Statistics: [2×7 table]
```

View Performance of Quantized Neural Network

Display the accuracy of the quantized network.

```
prediction_simulation.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
```

```
     {'Quantized'     }          1
```

```
prediction_FPGA.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
     {'Quantized'     }           1
```

Display the performance of the quantized network in frames per second.

```
prediction_FPGA.Statistics.FramesPerSecond(2)
```

```
ans = 19.0828
```

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `compile` | `deploy` | `predict` | `dlquantizer` | `dlquantizationOptions` | `calibrate` | `validate` | **Deep Network Designer**

More About

- "Quantization of Deep Neural Networks"

Evaluate Performance of Deep Learning Network on Custom Processor Configuration

Benchmark the performance of a deep learning network on a custom bitstream configuration by comparing it to the performance on a reference (shipping) bitstream configuration. Use the comparison results to adjust your custom deep learning processor parameters to achieve optimum performance.

In this example compare the performance of the ResNet-18 network on the `zcu102_single` bitstream configuration to the performance on the default custom bitstream configuration.

Prerequisites

- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model for ResNet-18 Network

Load Pretrained Network

Load the pretrained network.

```
snet = resnet18;
```

Retrieve `zcu102_single` Bitstream Configuration

To retrieve the `zcu102_single` bitstream configuration, use the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

```
hPC_shipping = dlhdl.ProcessorConfig('Bitstream',"zcu102_single")
```

```
hPC_shipping = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                   Processing Module "adder"
                            ModuleGeneration: 'on'
                             InputMemorySize: 40
                            OutputMemorySize: 40

              Processor Top Level Properties
```

```
                     RunTimeControl: 'register'
                 InputDataInterface: 'External Memory'
                OutputDataInterface: 'External Memory'
                  ProcessorDataType: 'single'

            System Level Properties
                     TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                    TargetFrequency: 220
                      SynthesisTool: 'Xilinx Vivado'
                    ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
            SynthesisToolChipFamily: 'Zynq UltraScale+'
            SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
           SynthesisToolPackageName: ''
            SynthesisToolSpeedValue: ''
```

Estimate ResNet-18 Performance for `zcu102_single` Bitstream Configuration

To `estimate` the performance of the ResNet-18 DAG network, use the `estimatePerformance` function of the `dlhdl.ProcessorConfig` object. The function returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

```
hPC_shipping.estimatePerformance(snet)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification(


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   23634966                  0.10743                       1          236
    ____conv1              2165372                  0.00984
    ____pool1               646226                  0.00294
    ____res2a_branch2a      966221                  0.00439
    ____res2a_branch2b      966221                  0.00439
    ____res2a               210750                  0.00096
    ____res2b_branch2a      966221                  0.00439
    ____res2b_branch2b      966221                  0.00439
    ____res2b               210750                  0.00096
    ____res3a_branch1       540749                  0.00246
    ____res3a_branch2a      763860                  0.00347
    ____res3a_branch2b      919117                  0.00418
    ____res3a               105404                  0.00048
    ____res3b_branch2a      919117                  0.00418
    ____res3b_branch2b      919117                  0.00418
    ____res3b               105404                  0.00048
    ____res4a_branch1       509261                  0.00231
    ____res4a_branch2a      509261                  0.00231
    ____res4a_branch2b      905421                  0.00412
    ____res4a                52724                  0.00024
    ____res4b_branch2a      905421                  0.00412
    ____res4b_branch2b      905421                  0.00412
    ____res4b                52724                  0.00024
    ____res5a_branch1      1046605                  0.00476
    ____res5a_branch2a     1046605                  0.00476
```

```
    ____res5a_branch2b     2005197                  0.00911
    ____res5a                26368                  0.00012
    ____res5b_branch2a     2005197                  0.00911
    ____res5b_branch2b     2005197                  0.00911
    ____res5b                26368                  0.00012
    ____pool5                54594                  0.00025
    ____fc1000              207852                  0.00094
 * The clock frequency of the DL processor is: 220MHz
```

Create Custom Processor Configuration

To create a custom processor configuration, use the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

```
hPC_custom = dlhdl.ProcessorConfig
```

```
hPC_custom = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                   Processing Module "adder"
                            ModuleGeneration: 'on'
                             InputMemorySize: 40
                            OutputMemorySize: 40

              Processor Top Level Properties
                              RunTimeControl: 'register'
                          InputDataInterface: 'External Memory'
                         OutputDataInterface: 'External Memory'
                           ProcessorDataType: 'single'

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                             TargetFrequency: 200
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

Estimate ResNet-18 Performance for Custom Bitstream Configuration

To `estimate` the performance of the ResNet-18 DAG network, use the `estimatePerformance` function of the `dlhdl.ProcessorConfig` object. The function returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

```
hPC_custom.estimatePerformance(snet)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification(


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    21219873                  0.10610                       1           212
    ____conv1               2165372                  0.01083
    ____pool1                646226                  0.00323
    ____res2a_branch2a       966221                  0.00483
    ____res2a_branch2b       966221                  0.00483
    ____res2a                210750                  0.00105
    ____res2b_branch2a       966221                  0.00483
    ____res2b_branch2b       966221                  0.00483
    ____res2b                210750                  0.00105
    ____res3a_branch1        540749                  0.00270
    ____res3a_branch2a       708564                  0.00354
    ____res3a_branch2b       919117                  0.00460
    ____res3a                105404                  0.00053
    ____res3b_branch2a       919117                  0.00460
    ____res3b_branch2b       919117                  0.00460
    ____res3b                105404                  0.00053
    ____res4a_branch1        509261                  0.00255
    ____res4a_branch2a       509261                  0.00255
    ____res4a_branch2b       905421                  0.00453
    ____res4a                 52724                  0.00026
    ____res4b_branch2a       905421                  0.00453
    ____res4b_branch2b       905421                  0.00453
    ____res4b                 52724                  0.00026
    ____res5a_branch1        751693                  0.00376
    ____res5a_branch2a       751693                  0.00376
    ____res5a_branch2b      1415373                  0.00708
    ____res5a                 26368                  0.00013
    ____res5b_branch2a      1415373                  0.00708
    ____res5b_branch2b      1415373                  0.00708
    ____res5b                 26368                  0.00013
    ____pool5                 54594                  0.00027
    ____fc1000               207351                  0.00104
 * The clock frequency of the DL processor is: 200MHz
```

The performance of the ResNet-18 network on the custom bitstream configuration is lower than the performance on the `zcu102_single` bitstream configuration. The difference between the custom bitstream configuration and the `zcu102_single` bitstream configuration is the target frequency.

Modify Custom Processor Configuration

Modify the custom processor configuration to increase the target frequency. To learn about modifiable parameters of the processor configuration, see `dlhdl.ProcessorConfig`.

```
hPC_custom.TargetFrequency = 220;
hPC_custom
```

```
hPC_custom =
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                   Processing Module "adder"
                            ModuleGeneration: 'on'
                             InputMemorySize: 40
                            OutputMemorySize: 40

              Processor Top Level Properties
                              RunTimeControl: 'register'
                          InputDataInterface: 'External Memory'
                         OutputDataInterface: 'External Memory'
                           ProcessorDataType: 'single'

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                             TargetFrequency: 220
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

Re-estimate ResNet-18 Performance for Modified Custom Bitstream Configuration

Estimate the performance of the ResNet-18 DAG network on the modified custom bitstream configuration.

```
hPC_custom.estimatePerformance(snet)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification(
```

```
              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   23634966                  0.10743                       1          236
    ____conv1              2165372                  0.00984
    ____pool1               646226                  0.00294
    ____res2a_branch2a      966221                  0.00439
    ____res2a_branch2b      966221                  0.00439
    ____res2a               210750                  0.00096
    ____res2b_branch2a      966221                  0.00439
    ____res2b_branch2b      966221                  0.00439
    ____res2b               210750                  0.00096
    ____res3a_branch1       540749                  0.00246
    ____res3a_branch2a      763860                  0.00347
    ____res3a_branch2b      919117                  0.00418
    ____res3a               105404                  0.00048
    ____res3b_branch2a      919117                  0.00418
    ____res3b_branch2b      919117                  0.00418
    ____res3b               105404                  0.00048
    ____res4a_branch1       509261                  0.00231
    ____res4a_branch2a      509261                  0.00231
    ____res4a_branch2b      905421                  0.00412
    ____res4a                52724                  0.00024
    ____res4b_branch2a      905421                  0.00412
    ____res4b_branch2b      905421                  0.00412
    ____res4b                52724                  0.00024
    ____res5a_branch1      1046605                  0.00476
    ____res5a_branch2a     1046605                  0.00476
    ____res5a_branch2b     2005197                  0.00911
    ____res5a                26368                  0.00012
    ____res5b_branch2a     2005197                  0.00911
    ____res5b_branch2b     2005197                  0.00911
    ____res5b                26368                  0.00012
    ____pool5                54594                  0.00025
    ____fc1000              207852                  0.00094
 * The clock frequency of the DL processor is: 220MHz
```

See Also

dlhdl.ProcessorConfig | estimatePerformance | estimateResources

More About

- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10

Customize Bitstream Configuration to Meet Resource Use Requirements

This example shows how to deploy a digit recognition network with a target performance of 500 frames per second (FPS) to a Xilinx™ ZCU102 ZU4CG device. The target device resource counts are:

- Digital signal processor (DSP) slice count — 240
- Block random access memory (BRAM) count — 128

The reference `zcu102_int8` bitstream configuration is for a Xilinx ZCU102 ZU9EG device. The default board resource counts are:

- Digital signal processor (DSP) slice count — 2520
- Block random access memory (BRAM) count — 912

The default board resource counts exceed the resource budget and are on the higher end of the cost spectrum. In this example, you can achieve target performance and resource use budget by quantizing the target deep learning network and customizing the bitstream configuration.

Prerequisites

- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library

Load Pretrained Network

To load the pretrained series network, that has been trained on the Modified National Institute Standards of Technology (MNIST) database, enter:

```
snet = getDigitsNetwork;
```

Quantize Network

To quantize the MNIST based digits network, enter:

```
dlquantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
Image = imageDatastore('five_28x28.pgm','Labels','five');
calibrate(dlquantObj,Image);
```

Retrieve `zcu102_int` Bitstream Configuration

To retrieve the `zcu102_int8` bitstream configuration, use the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

```
referencehPC = dlhdl.ProcessorConfig('Bitstream','zcu102_int8')
```

```
referencehPC =
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
```

```
        SegmentationBlockGeneration: 'on'
                    ConvThreadNumber: 64
                     InputMemorySize: [227 227 3]
                    OutputMemorySize: [227 227 3]
                    FeatureSizeLimit: 2048

              Processing Module "fc"
                    ModuleGeneration: 'on'
              SoftmaxBlockGeneration: 'off'
              SigmoidBlockGeneration: 'off'
                      FCThreadNumber: 16
                     InputMemorySize: 25088
                    OutputMemorySize: 4096

          Processing Module "custom"
                    ModuleGeneration: 'on'
                            Addition: 'on'
                      Multiplication: 'on'
                            Resize2D: 'off'
                             Sigmoid: 'off'
                           TanhLayer: 'off'
                     InputMemorySize: 40
                    OutputMemorySize: 120

      Processor Top Level Properties
                      RunTimeControl: 'register'
                       RunTimeStatus: 'register'
                  InputStreamControl: 'register'
                 OutputStreamControl: 'register'
                        SetupControl: 'register'
                   ProcessorDataType: 'int8'

             System Level Properties
                      TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                     TargetFrequency: 250
                       SynthesisTool: 'Xilinx Vivado'
                     ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
             SynthesisToolChipFamily: 'Zynq UltraScale+'
             SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
            SynthesisToolPackageName: ''
             SynthesisToolSpeedValue: ''
```

Estimate Network Performance and Resource Utilization for `zcu102_int8` Bitstream Configuration

To estimate the performance of the digits series network, use the `estimatePerformance` method of the `dlhdl.ProcessorConfig` object. The method returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

```
estimatePerformance(referencehPC,dlquantObj)
```

```
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### The network includes the following layers:
     1   'imageinput'    Image Input             28×28×1 images with 'zerocenter' normalization
     2   'conv_1'        2-D Convolution         8 3×3×1 convolutions with stride [1  1] and padd
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
```

```
     5   'conv_2'        2-D Convolution         16 3×3×8 convolutions with stride [1  1] and pad
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
     8   'conv_3'        2-D Convolution         32 3×3×16 convolutions with stride [1  1] and pa
     9   'relu_3'        ReLU                    ReLU
    10   'fc'            Fully Connected         10 fully connected layer
    11   'softmax'       Softmax                 softmax
    12   'classoutput'   Classification Output   crossentropyex with '0' and 9 other classes

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------            -------------               ---------        --
Network                    58101                  0.00023                       1
    conv_1                  4391                  0.00002
    maxpool_1               2877                  0.00001
    conv_2                  2351                  0.00001
    maxpool_2               2265                  0.00001
    conv_3                  2651                  0.00001
    fc                     43566                  0.00017
 * The clock frequency of the DL processor is: 250MHz
```

To estimate the resource use of the `zcu102_int8` bitstream, use the `estimateResources` method of the `dlhdl.ProcessorConfig` object. The method returns the estimated DSP slice and BRAM usage.

```
estimateResources(referencehPC)
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912            274080
                        -------------    -------------    -------------
DL_Processor                805( 32%)        386( 43%)     142494( 52%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

The estimated performance is 4303 FPS and the estimated resource use counts are:

- Digital signal processor (DSP) slice count - 797
- Block random access memory (BRAM) count -386

The estimated DSP slice count and BRAM count use exceeds the target device resource budget. Customize the bitstream configuration to reduce resource use.

Create Custom Bitstream Configuration

To create a custom processor configuration, use `dlhdl.ProcessorConfig` class. To learn about the modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

To reduce the resource use for the custom bitstream, modify the `KernelDataType property` for the `conv, fc, and` adder modules. Modify the `ConvThreadNumber` property to reduce DSP slice

count. Reduce the InputMemorySize and OutputMemorySize properties for the conv module to reduce the BRAM count.

```
customhPC = dlhdl.ProcessorConfig;
customhPC.ProcessorDataType = 'int8';
customhPC.setModuleProperty('conv','ConvThreadNumber',4);
customhPC.setModuleProperty('conv','InputMemorySize',[30 30 1]);
customhPC.setModuleProperty('conv','OutputMemorySize',[30 30 1]);
customhPC
```

```
customhPC =
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                            ConvThreadNumber: 4
                             InputMemorySize: [30 30 1]
                            OutputMemorySize: [30 30 1]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                      SigmoidBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                  Processing Module "custom"
                            ModuleGeneration: 'on'
                                    Addition: 'on'
                              Multiplication: 'on'
                                    Resize2D: 'off'
                                     Sigmoid: 'off'
                                   TanhLayer: 'off'
                             InputMemorySize: 40
                            OutputMemorySize: 120

              Processor Top Level Properties
                              RunTimeControl: 'register'
                               RunTimeStatus: 'register'
                          InputStreamControl: 'register'
                         OutputStreamControl: 'register'
                                SetupControl: 'register'
                           ProcessorDataType: 'int8'

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation K
                             TargetFrequency: 200
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

Estimate Network Performance and Resource Utilization for Custom Bitstream Configuration

Estimate the performance of the digits series network for the custom bitstream.

```
estimatePerformance(customhPC,dlquantObj)
```

```
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convo
### The network includes the following layers:
     1   'imageinput'    Image Input             28×28×1 images with 'zerocenter' normalization
     2   'conv_1'        2-D Convolution         8 3×3×1 convolutions with stride [1  1] and padd
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
     5   'conv_2'        2-D Convolution         16 3×3×8 convolutions with stride [1  1] and pad
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
     8   'conv_3'        2-D Convolution         32 3×3×16 convolutions with stride [1  1] and pa
     9   'relu_3'        ReLU                    ReLU
    10   'fc'            Fully Connected         10 fully connected layer
    11   'softmax'       Softmax                 softmax
    12   'classoutput'   Classification Output   crossentropyex with '0' and 9 other classes

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------            -------------               ---------        --
Network                     433577                  0.00217                       1           4
    conv_1                   26160                  0.00013
    maxpool_1                31888                  0.00016
    conv_2                   44736                  0.00022
    maxpool_2                22337                  0.00011
    conv_3                  265045                  0.00133
    fc                       43411                  0.00022
 * The clock frequency of the DL processor is: 200MHz
```

Estimate the resource use of the custom bitstream.

```
estimateResources(customhPC)
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912             274080
                        -------------    -------------    -------------
DL_Processor                139(  6%)        108( 12%)      56270( 21%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

The estimated performance is 461.3 FPS and the estimated resource use counts are:

- Digital signal processor (DSP) slice count - 131
- Block random access memory (BRAM) count -108

The estimated resources of the customized bitstream match the user target device resource budget and the estimated performance matches the target network performance.

See Also

`dlhdl.ProcessorConfig` | `estimatePerformance` | `estimateResources`

More About

- "Estimate Resource Utilization for Custom Processor Configuration" on page 8-10
- "Estimate Performance of Deep Learning Network" on page 8-3

Vehicle Detection Using DAG Network Based YOLO v2 Deployed to FPGA

This example shows how to train and deploy a you only look once (YOLO) v2 object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several techniques for object detection exist, including Faster R-CNN and you only look once (YOLO) v2. This example trains a YOLO v2 vehicle detector using the `trainYOLOv2ObjectDetector` function.

Load Dataset

This example uses a small vehicle dataset that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small dataset is useful for exploring the YOLO v2 training procedure, but in practice, more labeled images are needed to train a robust detector. The data set is attached to the example. Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

```
% Add the fullpath to the local vehicle data folder.
vehicleDataset.imageFilename = fullfile(pwd,vehicleDataset.imageFilename);
```

Split the dataset into training and test sets. Select 60% of the data for training and the rest for testing the trained detector.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * length(shuffledIndices) );
trainingDataTbl = vehicleDataset(shuffledIndices(1:idx),:);
testDataTbl = vehicleDataset(shuffledIndices(idx+1:end),:);
```

Use `imageDatastore` and `boxLabelDataStore` to create datastores for loading the image and label data during training and evaluation.

```
imdsTrain = imageDatastore(trainingDataTbl{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

imdsTest = imageDatastore(testDataTbl{:,'imageFilename'});
bldsTest = boxLabelDatastore(testDataTbl(:,'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
testData = combine(imdsTest,bldsTest);
```

Create a YOLO v2 Object Detection Network

A YOLO v2 object detection network is composed of two subnetworks. A feature extraction network followed by a detection network. The feature extraction network is typically a pretrained CNN (for

details, see “Pretrained Deep Neural Networks”). This example uses ResNet-18 for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-50 depending on application requirements. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific for YOLO v2.

Use the `yolov2Layers` (Computer Vision Toolbox) function to create a YOLO v2 object detection network automatically given a pretrained ResNet-18 feature extraction network. `yolov2Layers` requires you to specify several inputs that parameterize a YOLO v2 network:

- Network input size
- Anchor boxes
- Feature extraction network

First, specify the network input size and the number of classes. When choosing the network input size, consider the minimum size required by the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [224 224 3], which is the minimum size required to run the network.

```
inputSize = [224 224 3];
```

Define the number of object classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Note that the training images used in this example are bigger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use `estimateAnchorBoxes` (Computer Vision Toolbox) to estimate anchor boxes based on the size of objects in the training data. To account for the resizing of the images prior to training, resize the training data for estimating anchor boxes. Use `transform` to preprocess the training data, then define the number of anchor boxes and estimate the anchor boxes. Resize the training data to the input image size of the network using the supporting function `yolo_preprocessData`.

```
trainingDataForEstimation = transform(trainingData,@(data)yolo_preprocessData(data,inputSize));
numAnchors = 7;
[anchorBoxes, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
```

```
anchorBoxes = 7×2

   145   126
    91    86
   161   132
    41    34
    67    64
   136   111
    33    23


meanIoU = 0.8651
```

For more information on choosing anchor boxes, see “Estimate Anchor Boxes From Training Data” (Computer Vision Toolbox) (Computer Vision Toolbox) (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” (Computer Vision Toolbox) (Computer Vision Toolbox).

Now, use resnet18 to load a pretrained ResNet-18 model.

```
featureExtractionNetwork = resnet18;
```

Select `'res4b_relu'` as the feature extraction layer to replace the layers after `'res4b_relu'` with the detection subnetwork. This feature extraction layer outputs feature maps that are downsampled by a factor of 16. This amount of downsampling is a good trade-off between spatial resolution and the strength of the extracted features, as features extracted further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis.

```
featureLayer = 'res4b_relu';
```

Create the YOLO v2 object detection network. .

```
lgraph = yolov2Layers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer);
```

You can visualize the network using `analyzeNetwork` or Deep Network Designer from Deep Learning Toolbox™.

If more control is required over the YOLO v2 network architecture, use Deep Network Designer to design the YOLO v2 detection network manually. For more information, see “Design a YOLO v2 Detection Network” (Computer Vision Toolbox) (Computer Vision Toolbox).

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` to augment the training data by randomly flipping the image and associated box labels horizontally. Note that data augmentation is not applied to the test and validation data. Ideally, test and validation data should be representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@yolo_augmentData);
```

Preprocess Training Data and Train YOLO v2 Object Detector

Preprocess the augmented training data, and the validation data to prepare for training.

```
preprocessedTrainingData = transform(augmentedTrainingData,@(data)yolo_preprocessData(data,inputS
```

Use `trainingOptions` to specify network training options. Set `'ValidationData'` to the preprocessed validation data. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...
        'MiniBatchSize', 16, ....
        'InitialLearnRate',1e-3, ...
        'MaxEpochs',20,...
        'CheckpointPath', tempdir, ...
        'Shuffle','never');
```

Use `trainYOLOv2ObjectDetector` (Computer Vision Toolbox) function to train YOLO v2 object detector.

```
[detector,info] = trainYOLOv2ObjectDetector(preprocessedTrainingData,lgraph,options);
```

```
*************************************************************************
Training a YOLO v2 Object Detector for the following object classes:

* vehicle

Training on single CPU.
Initializing input data normalization.
|========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |     RMSE     |     Loss     |      Rate       |
|========================================================================================|
|       1 |           1 |       00:00:02 |         8.43 |         71.1 |          0.0010 |
|       5 |          50 |       00:01:26 |         0.71 |          0.5 |          0.0010 |
|      10 |         100 |       00:02:46 |         0.75 |          0.6 |          0.0010 |
|      14 |         150 |       00:04:04 |         0.53 |          0.3 |          0.0010 |
|      19 |         200 |       00:05:23 |         0.48 |          0.2 |          0.0010 |
|      20 |         220 |       00:05:53 |         0.57 |          0.3 |          0.0010 |
|========================================================================================|
Detector training complete.
*************************************************************************
```

As a quick test, run the detector on one test image. Make sure you resize the image to the same size as the training images.

```
I = imread(testDataTbl.imageFilename{2});
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
```

Display the results.

```
I_new = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I_new)
```

Load Pretrained Network

Load the pretrained network.

```
snet=detector.Network;
I_pre=yolo_pre_proc(I);
```

Use `analyzeNetwork` to obtain information about the network layers:

```
analyzeNetwork(snet)
```

Create Target Object

Create a target object for your target device with a vendor name and an interface to connect your target device to the host computer. Interface options are JTAG (default) and Ethernet. Vendor options are Intel or Xilinx. Use the installed Xilinx Vivado Design Suite over an Ethernet connection to program the device.

```
hTarget = dlhdl.Target('Xilinx', 'Interface', 'Ethernet');
```

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pre-trained series network, `trainedNetNoCar,` as the network. Make sure the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Zynq UltraScale+ MPSoC ZCU102 board. The bitstream uses a single data type.

```
hW=dlhdl.Workflow('Network', snet, 'Bitstream', 'zcu102_single','Target',hTarget)
```

```
hW = 
  Workflow with properties:

            Network: [1×1 DAGNetwork]
          Bitstream: 'zcu102_single'
    ProcessorConfig: []
             Target: [1×1 dlhdl.Target]
```

Compile YOLO v2 Object Detector

To compile the `snet` series network, run the compile function of the `dlhdl.Workflow` object .

```
dn = hW.compile
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single ...
### The network includes the following layers:

     1   'data'                   Image Input             224×224×3 images with 'zscore' norma
     2   'conv1'                  Convolution             64 7×7×3 convolutions with stride [2
     3   'bn_conv1'               Batch Normalization     Batch normalization with 64 channels
     4   'conv1_relu'             ReLU                    ReLU
     5   'pool1'                  Max Pooling             3×3 max pooling with stride [2  2] ar
     6   'res2a_branch2a'         Convolution             64 3×3×64 convolutions with stride [1
     7   'bn2a_branch2a'          Batch Normalization     Batch normalization with 64 channels
     8   'res2a_branch2a_relu'    ReLU                    ReLU
     9   'res2a_branch2b'         Convolution             64 3×3×64 convolutions with stride [1
```

```
    10   'bn2a_branch2b'         Batch Normalization       Batch normalization with 64 channels
    11   'res2a'                 Addition                  Element-wise addition of 2 inputs
    12   'res2a_relu'            ReLU                      ReLU
    13   'res2b_branch2a'        Convolution               64 3×3×64 convolutions with stride [1
    14   'bn2b_branch2a'         Batch Normalization       Batch normalization with 64 channels
    15   'res2b_branch2a_relu'   ReLU                      ReLU
    16   'res2b_branch2b'        Convolution               64 3×3×64 convolutions with stride [1
    17   'bn2b_branch2b'         Batch Normalization       Batch normalization with 64 channels
    18   'res2b'                 Addition                  Element-wise addition of 2 inputs
    19   'res2b_relu'            ReLU                      ReLU
    20   'res3a_branch2a'        Convolution               128 3×3×64 convolutions with stride
    21   'bn3a_branch2a'         Batch Normalization       Batch normalization with 128 channels
    22   'res3a_branch2a_relu'   ReLU                      ReLU
    23   'res3a_branch2b'        Convolution               128 3×3×128 convolutions with stride
    24   'bn3a_branch2b'         Batch Normalization       Batch normalization with 128 channels
    25   'res3a'                 Addition                  Element-wise addition of 2 inputs
    26   'res3a_relu'            ReLU                      ReLU
    27   'res3a_branch1'         Convolution               128 1×1×64 convolutions with stride
    28   'bn3a_branch1'          Batch Normalization       Batch normalization with 128 channels
    29   'res3b_branch2a'        Convolution               128 3×3×128 convolutions with stride
    30   'bn3b_branch2a'         Batch Normalization       Batch normalization with 128 channels
    31   'res3b_branch2a_relu'   ReLU                      ReLU
    32   'res3b_branch2b'        Convolution               128 3×3×128 convolutions with stride
    33   'bn3b_branch2b'         Batch Normalization       Batch normalization with 128 channels
    34   'res3b'                 Addition                  Element-wise addition of 2 inputs
    35   'res3b_relu'            ReLU                      ReLU
    36   'res4a_branch2a'        Convolution               256 3×3×128 convolutions with stride
    37   'bn4a_branch2a'         Batch Normalization       Batch normalization with 256 channels
    38   'res4a_branch2a_relu'   ReLU                      ReLU
    39   'res4a_branch2b'        Convolution               256 3×3×256 convolutions with stride
    40   'bn4a_branch2b'         Batch Normalization       Batch normalization with 256 channels
    41   'res4a'                 Addition                  Element-wise addition of 2 inputs
    42   'res4a_relu'            ReLU                      ReLU
    43   'res4a_branch1'         Convolution               256 1×1×128 convolutions with stride
    44   'bn4a_branch1'          Batch Normalization       Batch normalization with 256 channels
    45   'res4b_branch2a'        Convolution               256 3×3×256 convolutions with stride
    46   'bn4b_branch2a'         Batch Normalization       Batch normalization with 256 channels
    47   'res4b_branch2a_relu'   ReLU                      ReLU
    48   'res4b_branch2b'        Convolution               256 3×3×256 convolutions with stride
    49   'bn4b_branch2b'         Batch Normalization       Batch normalization with 256 channels
    50   'res4b'                 Addition                  Element-wise addition of 2 inputs
    51   'res4b_relu'            ReLU                      ReLU
    52   'yolov2Conv1'           Convolution               256 3×3×256 convolutions with stride
    53   'yolov2Batch1'          Batch Normalization       Batch normalization with 256 channels
    54   'yolov2Relu1'           ReLU                      ReLU
    55   'yolov2Conv2'           Convolution               256 3×3×256 convolutions with stride
    56   'yolov2Batch2'          Batch Normalization       Batch normalization with 256 channels
    57   'yolov2Relu2'           ReLU                      ReLU
    58   'yolov2ClassConv'       Convolution               42 1×1×256 convolutions with stride
    59   'yolov2Transform'       YOLO v2 Transform Layer.  YOLO v2 Transform Layer with 7 anchor
    60   'yolov2OutputLayer'     YOLO v2 Output            YOLO v2 Output with 7 anchors.

### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
5 Memory Regions created.

Skipping: data
Compiling leg: conv1>>pool1 ...
Compiling leg: conv1>>pool1 ... complete.
```

```
Compiling leg: res2a_branch2a>>res2a_branch2b ...
Compiling leg: res2a_branch2a>>res2a_branch2b ... complete.
Compiling leg: res2b_branch2a>>res2b_branch2b ...
Compiling leg: res2b_branch2a>>res2b_branch2b ... complete.
Compiling leg: res3a_branch1 ...
Compiling leg: res3a_branch1 ... complete.
Compiling leg: res3a_branch2a>>res3a_branch2b ...
Compiling leg: res3a_branch2a>>res3a_branch2b ... complete.
Compiling leg: res3b_branch2a>>res3b_branch2b ...
Compiling leg: res3b_branch2a>>res3b_branch2b ... complete.
Compiling leg: res4a_branch1 ...
Compiling leg: res4a_branch1 ... complete.
Compiling leg: res4a_branch2a>>res4a_branch2b ...
Compiling leg: res4a_branch2a>>res4a_branch2b ... complete.
Compiling leg: res4b_branch2a>>res4b_branch2b ...
Compiling leg: res4b_branch2a>>res4b_branch2b ... complete.
Compiling leg: yolov2Conv1>>yolov2ClassConv ...
Compiling leg: yolov2Conv1>>yolov2ClassConv ... complete.
Skipping: yolov2Transform
Skipping: yolov2OutputLayer
Creating Schedule...
.....................
Creating Schedule...complete.
Creating Status Table...
....................
Creating Status Table...complete.
Emitting Schedule...
....................
Emitting Schedule...complete.
Emitting Status Table...
......................
Emitting Status Table...complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    ________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x02000000"     "28.0 MB"
    "InstructionDataOffset"     "0x03c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04000000"     "20.0 MB"
    "EndOffset"                 "0x05400000"     "Total: 84.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
```

Program the Bitstream onto FPGA and Download Network Weights

To deploy the network on the Zynq® UltraScale+™ MPSoC ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object . This function uses the output of the compile function to program the FPGA board by using the programming file.The function also downloads the network weights and biases. The deploy function checks for the Xilinx Vivado tool and the supported tool version. It then starts programming the FPGA device by using the bitstream, displays progress messages and the time it takes to deploy the network.

```
hW.deploy
```

```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'

Downloading target FPGA device configuration over Ethernet to SD card done. The system will now r


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 04-Jan-2021 13:59:03
```

Load the Example Image and Run The Prediction

Execute the predict function on the `dlhdl.Workflow` object and display the result:

```
[prediction, speed] = hW.predict(I_pre,'Profile','on');
```

```
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   16974672                  0.07716                       1           169
    conv1                  2224187                  0.01011
    pool1                   573166                  0.00261
    res2a_branch2a          972763                  0.00442
    res2a_branch2b          972632                  0.00442
    res2a                   209363                  0.00095
    res2b_branch2a          972674                  0.00442
    res2b_branch2b          973107                  0.00442
    res2b                   209914                  0.00095
    res3a_branch1           538478                  0.00245
    res3a_branch2a          747078                  0.00340
    res3a_branch2b          904530                  0.00411
    res3a                   104830                  0.00048
    res3b_branch2a          904540                  0.00411
    res3b_branch2b          904278                  0.00411
    res3b                   104900                  0.00048
```

```
    res4a_branch1           485804                  0.00221
    res4a_branch2a          485923                  0.00221
    res4a_branch2b          880309                  0.00400
    res4a                    52446                  0.00024
    res4b_branch2a          880071                  0.00400
    res4b_branch2b          880065                  0.00400
    res4b                    52456                  0.00024
    yolov2Conv1             880210                  0.00400
    yolov2Conv2             880375                  0.00400
    yolov2ClassConv         179300                  0.00081
 * The clock frequency of the DL processor is: 220MHz
```

Display the prediction results.

```
[bboxesn, scoresn, labelsn] = yolo_post_proc(prediction,I_pre,anchorBoxes,{'Vehicle'});
I_new3 = insertObjectAnnotation(I,'rectangle',bboxesn,scoresn);
figure
imshow(I_new3)
```

See Also

dlhdl.Workflow | dlhdl.Target | compile | deploy | predict | dlquantizer | dlquantizationOptions | calibrate | validate

More About

- “Quantization of Deep Neural Networks”

Customize Bitstream Configuration to Meet Resource Use Requirements

This example shows how to deploy a digit recognition network with a target performance of 500 frames per second (FPS) to a Xilinx™ ZCU102 ZU4CG device. The target device resource counts are:

- Digital signal processor (DSP) slice count — 240
- Block random access memory (BRAM) count — 128

The reference `zcu102_int8` bitstream configuration is for a Xilinx ZCU102 ZU9EG device. The default board resource counts are:

- Digital signal processor (DSP) slice count — 2520
- Block random access memory (BRAM) count — 912

The default board resource counts exceed the resource budget and are on the higher end of the cost spectrum. In this example, you can achieve target performance and resource use budget by quantizing the target deep learning network and customizing the bitstream configuration.

Prerequisites

- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library

Load Pretrained Network

To load the pretrained series network, that has been trained on the Modified National Institute Standards of Technology (MNIST) database, enter:

```
snet = getDigitsNetwork;
```

Quantize Network

To quantize the MNIST based digits network, enter:

```
dlquantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
Image = imageDatastore('five_28x28.pgm','Labels','five');
calibrate(dlquantObj,Image);
```

Retrieve `zcu102_int` Bitstream Configuration

To retrieve the `zcu102_int8` bitstream configuration, use the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

```
referencehPC = dlhdl.ProcessorConfig('Bitstream','zcu102_int8')
```

```
referencehPC = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
```

```
SegmentationBlockGeneration: 'on'
            ConvThreadNumber: 64
             InputMemorySize: [227 227 3]
            OutputMemorySize: [227 227 3]
            FeatureSizeLimit: 2048

      Processing Module "fc"
            ModuleGeneration: 'on'
      SoftmaxBlockGeneration: 'off'
      SigmoidBlockGeneration: 'off'
              FCThreadNumber: 16
             InputMemorySize: 25088
            OutputMemorySize: 4096

  Processing Module "custom"
            ModuleGeneration: 'on'
                    Addition: 'on'
              Multiplication: 'on'
                    Resize2D: 'off'
                     Sigmoid: 'off'
                   TanhLayer: 'off'
             InputMemorySize: 40
            OutputMemorySize: 120

Processor Top Level Properties
              RunTimeControl: 'register'
               RunTimeStatus: 'register'
          InputStreamControl: 'register'
         OutputStreamControl: 'register'
                SetupControl: 'register'
           ProcessorDataType: 'int8'

     System Level Properties
              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
             TargetFrequency: 250
               SynthesisTool: 'Xilinx Vivado'
             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
     SynthesisToolChipFamily: 'Zynq UltraScale+'
     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
    SynthesisToolPackageName: ''
     SynthesisToolSpeedValue: ''
```

Estimate Network Performance and Resource Utilization for `zcu102_int8` Bitstream Configuration

To estimate the performance of the digits series network, use the `estimatePerformance` method of the `dlhdl.ProcessorConfig` object. The method returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

```
estimatePerformance(referencehPC,dlquantObj)
```

```
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convo
### The network includes the following layers:
     1   'imageinput'    Image Input             28×28×1 images with 'zerocenter' normalization
     2   'conv_1'        2-D Convolution         8 3×3×1 convolutions with stride [1  1] and padd
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
```

```
     5   'conv_2'        2-D Convolution        16 3×3×8 convolutions with stride [1  1] and pad
     6   'relu_2'        ReLU                   ReLU
     7   'maxpool_2'     2-D Max Pooling        2×2 max pooling with stride [2  2] and padding
     8   'conv_3'        2-D Convolution        32 3×3×16 convolutions with stride [1  1] and pa
     9   'relu_3'        ReLU                   ReLU
    10   'fc'            Fully Connected        10 fully connected layer
    11   'softmax'       Softmax                softmax
    12   'classoutput'   Classification Output  crossentropyex with '0' and 9 other classes

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------             ---------        --
Network                       58101                  0.00023                       1
    conv_1                     4391                  0.00002
    maxpool_1                  2877                  0.00001
    conv_2                     2351                  0.00001
    maxpool_2                  2265                  0.00001
    conv_3                     2651                  0.00001
    fc                        43566                  0.00017
 * The clock frequency of the DL processor is: 250MHz
```

To estimate the resource use of the `zcu102_int8` bitstream, use the `estimateResources` method of the `dlhdl.ProcessorConfig` object. The method returns the estimated DSP slice and BRAM usage.

```
estimateResources(referencehPC)
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912            274080
                        -------------    -------------    -------------
DL_Processor                805( 32%)        386( 43%)     142494( 52%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

The estimated performance is 4303 FPS and the estimated resource use counts are:

- Digital signal processor (DSP) slice count - 797
- Block random access memory (BRAM) count -386

The estimated DSP slice count and BRAM count use exceeds the target device resource budget. Customize the bitstream configuration to reduce resource use.

Create Custom Bitstream Configuration

To create a custom processor configuration, use `dlhdl.ProcessorConfig` class. To learn about the modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

To reduce the resource use for the custom bitstream, modify the `KernelDataType property` for the `conv, fc, and` adder modules. Modify the `ConvThreadNumber` property to reduce DSP slice

count. Reduce the InputMemorySize and OutputMemorySize properties for the conv module to reduce the BRAM count.

```
customhPC = dlhdl.ProcessorConfig;
customhPC.ProcessorDataType = 'int8';
customhPC.setModuleProperty('conv','ConvThreadNumber',4);
customhPC.setModuleProperty('conv','InputMemorySize',[30 30 1]);
customhPC.setModuleProperty('conv','OutputMemorySize',[30 30 1]);
customhPC
```

```
customhPC = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                            ConvThreadNumber: 4
                             InputMemorySize: [30 30 1]
                            OutputMemorySize: [30 30 1]
                            FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                      SigmoidBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                  Processing Module "custom"
                            ModuleGeneration: 'on'
                                    Addition: 'on'
                              Multiplication: 'on'
                                    Resize2D: 'off'
                                     Sigmoid: 'off'
                                   TanhLayer: 'off'
                             InputMemorySize: 40
                            OutputMemorySize: 120

              Processor Top Level Properties
                              RunTimeControl: 'register'
                               RunTimeStatus: 'register'
                          InputStreamControl: 'register'
                         OutputStreamControl: 'register'
                                SetupControl: 'register'
                           ProcessorDataType: 'int8'

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation K
                             TargetFrequency: 200
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

Estimate Network Performance and Resource Utilization for Custom Bitstream Configuration

Estimate the performance of the digits series network for the custom bitstream.

```
estimatePerformance(customhPC,dlquantObj)
```

```
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### The network includes the following layers:
     1   'imageinput'    Image Input             28×28×1 images with 'zerocenter' normalization
     2   'conv_1'        2-D Convolution         8 3×3×1 convolutions with stride [1  1] and padd
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
     5   'conv_2'        2-D Convolution         16 3×3×8 convolutions with stride [1  1] and pad
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     2-D Max Pooling         2×2 max pooling with stride [2  2] and padding
     8   'conv_3'        2-D Convolution         32 3×3×16 convolutions with stride [1  1] and pa
     9   'relu_3'        ReLU                    ReLU
    10   'fc'            Fully Connected         10 fully connected layer
    11   'softmax'       Softmax                 softmax
    12   'classoutput'   Classification Output   crossentropyex with '0' and 9 other classes

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                     433577                  0.00217                       1            4
    conv_1                   26160                  0.00013
    maxpool_1                31888                  0.00016
    conv_2                   44736                  0.00022
    maxpool_2                22337                  0.00011
    conv_3                  265045                  0.00133
    fc                       43411                  0.00022
 * The clock frequency of the DL processor is: 200MHz
```

Estimate the resource use of the custom bitstream.

```
estimateResources(customhPC)
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912            274080
                        -------------    -------------    -------------
DL_Processor               139(  6%)        108( 12%)      56270( 21%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

The estimated performance is 461.3 FPS and the estimated resource use counts are:

- Digital signal processor (DSP) slice count - 131
- Block random access memory (BRAM) count -108

The estimated resources of the customized bitstream match the user target device resource budget and the estimated performance matches the target network performance.

See Also

`dlhdl.ProcessorConfig` | `estimatePerformance` | `estimateResources`

More About

- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10
- “Estimate Performance of Deep Learning Network” on page 8-3

Image Classification Using Neural Network on FPGA

This example shows how to train, compile, and deploy a `dlhdl.Workflow` object that has ResNet-18 neural network to an FPGA and use MATLAB® to retrieve the prediction results.

For this example, you need:

- Deep Learning Toolbox ™
- Deep Learning HDL Toolbox ™
- Deep Learning Toolbox Model for ResNet-18 Network
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- Image Processing Toolbox ™

Load Pretrained Network

Lad the pretrained ResNet-18 network.:

```
snet = resnet18;
```

View the layers of the pretrained network:

```
deepNetworkDesigner(snet);
```

The first layer, the image input layer, requires input images of size 224-by-224-by-3, where three is the number of color channels.

```
inputSize = snet.Layers(1).InputSize;
```

Load Data

This example uses the MathWorks MerchData data set. This is a small data set containing 75 images of MathWorks merchandise, belonging to five different classes (*cap*, *cube*, *playing cards*, *screwdriver*, and *torch*).

```
curDir = pwd;
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
'IncludeSubfolders',true, ...
'LabelSource','foldernames');
```

Specify Training and Validation Sets

Divide the data into training and validation data sets, so that 30% percent of the images go to the training data set and 70% of the images to the validation data set. `splitEachLabel` splits the datastore `imds` into two new datastores, `imdsTrain` and `imdsValidation`.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

Replace Final layers

To retrain ResNet-18 to classify new images, replace the last fully connected layer and final classification layer of the network. In ResNet-18 , these layers have the names `'fc1000'` and `'ClassificationLayer_predictions'`, respectively. The fully connected layer and classification layer of the pretrained network `net` are configured for 1000 classes. These two layers `fc1000` and

`ClassificationLayer_predictions` in ResNet-18, contain information on how to combine the features that the network extracts into class probabilities and predicted labels. These two layers must be fine-tuned for the new classification problem. Extract all the layers, except the last two layers, from the pretrained network.

```
lgraph = layerGraph(snet)
```

```
lgraph = 
  LayerGraph with properties:

       InputNames: {'data'}
      OutputNames: {'ClassificationLayer_predictions'}
           Layers: [71×1 nnet.cnn.layer.Layer]
      Connections: [78×2 table]
```

```
numClasses = numel(categories(imdsTrain.Labels))
```

```
numClasses = 5
```

```
newLearnableLayer = fullyConnectedLayer(numClasses, ...
'Name','new_fc', ...
'WeightLearnRateFactor',10, ...
'BiasLearnRateFactor',10);
lgraph = replaceLayer(lgraph,'fc1000',newLearnableLayer);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassLayer);
```

Prepare Data for Training

The network requires input images of size 224-by-224-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images, such as randomly flipping the training images along the vertical axis and randomly translating them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
'RandXReflection',true, ...
'RandXTranslation',pixelRange, ...
'RandYTranslation',pixelRange);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
'DataAugmentation',imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify Training Options

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. Specify the mini-batch size and validation data. The software validates the network every `ValidationFrequency` iterations during training.

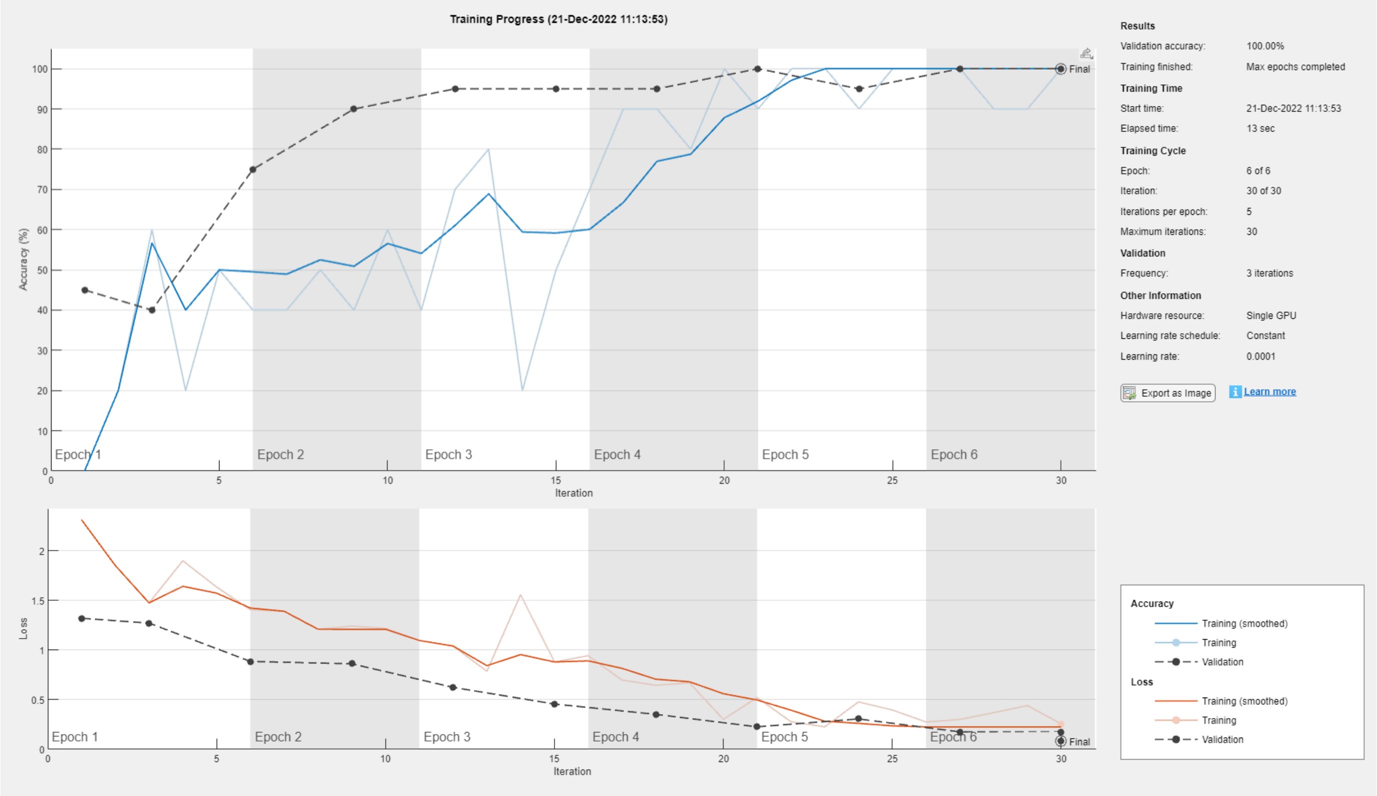
```
options = trainingOptions('sgdm', ...
'MiniBatchSize',10, ...
```

```
'MaxEpochs',6, ...
'InitialLearnRate',1e-4, ...
'Shuffle','every-epoch', ...
'ValidationData',augimdsValidation, ...
'ValidationFrequency',3, ...
'Verbose',false, ...
'Plots','training-progress');
```

Train Network

Train the network that consists of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available. Using this function on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For more information, see “GPU Computing Requirements” (Parallel Computing Toolbox). If a GPU is not available, the network uses a CPU (requires MATLAB® Coder™ Interface for Deep learning). You can also specify the execution environment by using the `ExecutionEnvironment` name-value argument of `trainingOptions`.

```
netTransfer = trainNetwork(augimdsTrain,lgraph,options);
```



Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Create a programming interface with custom name for your target device and an Ethernet interface to connect the target device to the host computer.

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx® Zynq® UltraScale+™ MPSoC ZCU102 board and the bitstream uses the single data type.

```
hW = dlhdl.Workflow(Network=netTransfer,Bitstream='zcu102_single',Target=hTarget);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment.

```
dn =compile(hW,'InputFrameNumberLimit',15)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data'
### The network includes the following layers:
     1   'data'                   Image Input                  224×224×3 images with 'zscore' norm
     2   'conv1'                  2-D Convolution              64 7×7×3 convolutions with stride
     3   'conv1_relu'             ReLU                         ReLU
     4   'pool1'                  2-D Max Pooling              3×3 max pooling with stride [2  2]
     5   'res2a_branch2a'         2-D Convolution              64 3×3×64 convolutions with stride
     6   'res2a_branch2a_relu'    ReLU                         ReLU
     7   'res2a_branch2b'         2-D Convolution              64 3×3×64 convolutions with stride
     8   'res2a'                  Addition                     Element-wise addition of 2 inputs
     9   'res2a_relu'             ReLU                         ReLU
    10   'res2b_branch2a'         2-D Convolution              64 3×3×64 convolutions with stride
    11   'res2b_branch2a_relu'    ReLU                         ReLU
    12   'res2b_branch2b'         2-D Convolution              64 3×3×64 convolutions with stride
    13   'res2b'                  Addition                     Element-wise addition of 2 inputs
    14   'res2b_relu'             ReLU                         ReLU
    15   'res3a_branch2a'         2-D Convolution              128 3×3×64 convolutions with stride
    16   'res3a_branch2a_relu'    ReLU                         ReLU
    17   'res3a_branch2b'         2-D Convolution              128 3×3×128 convolutions with stric
    18   'res3a_branch1'          2-D Convolution              128 1×1×64 convolutions with stride
    19   'res3a'                  Addition                     Element-wise addition of 2 inputs
    20   'res3a_relu'             ReLU                         ReLU
    21   'res3b_branch2a'         2-D Convolution              128 3×3×128 convolutions with stric
    22   'res3b_branch2a_relu'    ReLU                         ReLU
    23   'res3b_branch2b'         2-D Convolution              128 3×3×128 convolutions with stric
    24   'res3b'                  Addition                     Element-wise addition of 2 inputs
    25   'res3b_relu'             ReLU                         ReLU
    26   'res4a_branch2a'         2-D Convolution              256 3×3×128 convolutions with stric
    27   'res4a_branch2a_relu'    ReLU                         ReLU
    28   'res4a_branch2b'         2-D Convolution              256 3×3×256 convolutions with stric
    29   'res4a_branch1'          2-D Convolution              256 1×1×128 convolutions with stric
    30   'res4a'                  Addition                     Element-wise addition of 2 inputs
    31   'res4a_relu'             ReLU                         ReLU
    32   'res4b_branch2a'         2-D Convolution              256 3×3×256 convolutions with stric
    33   'res4b_branch2a_relu'    ReLU                         ReLU
    34   'res4b_branch2b'         2-D Convolution              256 3×3×256 convolutions with stric
    35   'res4b'                  Addition                     Element-wise addition of 2 inputs
    36   'res4b_relu'             ReLU                         ReLU
    37   'res5a_branch2a'         2-D Convolution              512 3×3×256 convolutions with stric
```

```
    38   'res5a_branch2a_relu'   ReLU                        ReLU
    39   'res5a_branch2b'        2-D Convolution             512 3×3×512 convolutions with stric
    40   'res5a_branch1'         2-D Convolution             512 1×1×256 convolutions with stric
    41   'res5a'                 Addition                    Element-wise addition of 2 inputs
    42   'res5a_relu'            ReLU                        ReLU
    43   'res5b_branch2a'        2-D Convolution             512 3×3×512 convolutions with stric
    44   'res5b_branch2a_relu'   ReLU                        ReLU
    45   'res5b_branch2b'        2-D Convolution             512 3×3×512 convolutions with stric
    46   'res5b'                 Addition                    Element-wise addition of 2 inputs
    47   'res5b_relu'            ReLU                        ReLU
    48   'pool5'                 2-D Global Average Pooling  2-D global average pooling
    49   'new_fc'                Fully Connected             5 fully connected layer
    50   'prob'                  Softmax                     softmax
    51   'new_classoutput'       Classification Output       crossentropyex with 'MathWorks Cap

### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'new_classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is i
### Compiling layer group: conv1>>pool1 ...
### Compiling layer group: conv1>>pool1 ... complete.
### Compiling layer group: res2a_branch2a>>res2a_branch2b ...
### Compiling layer group: res2a_branch2a>>res2a_branch2b ... complete.
### Compiling layer group: res2b_branch2a>>res2b_branch2b ...
### Compiling layer group: res2b_branch2a>>res2b_branch2b ... complete.
### Compiling layer group: res3a_branch1 ...
### Compiling layer group: res3a_branch1 ... complete.
### Compiling layer group: res3a_branch2a>>res3a_branch2b ...
### Compiling layer group: res3a_branch2a>>res3a_branch2b ... complete.
### Compiling layer group: res3b_branch2a>>res3b_branch2b ...
### Compiling layer group: res3b_branch2a>>res3b_branch2b ... complete.
### Compiling layer group: res4a_branch1 ...
### Compiling layer group: res4a_branch1 ... complete.
### Compiling layer group: res4a_branch2a>>res4a_branch2b ...
### Compiling layer group: res4a_branch2a>>res4a_branch2b ... complete.
### Compiling layer group: res4b_branch2a>>res4b_branch2b ...
### Compiling layer group: res4b_branch2a>>res4b_branch2b ... complete.
### Compiling layer group: res5a_branch1 ...
### Compiling layer group: res5a_branch1 ... complete.
### Compiling layer group: res5a_branch2a>>res5a_branch2b ...
### Compiling layer group: res5a_branch2a>>res5a_branch2b ... complete.
### Compiling layer group: res5b_branch2a>>res5b_branch2b ...
### Compiling layer group: res5b_branch2a>>res5b_branch2b ... complete.
### Compiling layer group: pool5 ...
### Compiling layer group: pool5 ... complete.
### Compiling layer group: new_fc ...
### Compiling layer group: new_fc ... complete.

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "12.0 MB"
    "OutputResultOffset"        "0x00c00000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01000000"     "8.0 MB"
    "SystemBufferOffset"        "0x01800000"     "28.0 MB"
    "InstructionDataOffset"     "0x03400000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03800000"     "52.0 MB"
    "FCWeightDataOffset"        "0x06c00000"     "4.0 MB"
```

```
    "EndOffset"                 "0x07000000"     "Total: 112.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {{1×2 cell}  [0.0135 0.0162 0.0141 0 0.0135 0.0162 0.0141 0 0.0135 0.0162 (
             ddrInfo: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx® Zynq® UltraScale+ MPSoC ZCU102 hardware, run the `deploy` method of the `dlhdl.Workflow` object. This method programs the FPGA board using the output of the compile method and the programming file, downloads the network weights and biases, displays progress messages, and the time it takes to deploy the network.

```
deploy(hW)
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_single.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_single.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 21-Dec-2022 11:15:51
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 21-Dec-2022 11:15:51
```

Test Network

Load the example image.

```
imgFile = fullfile(pwd,'MerchData','MathWorks Cube','Mathworks cube_0.jpg');
inputImg = imresize(imread(imgFile),[224 224]);
imshow(inputImg)
```

Classify the image on the FPGA by using the `predict` method of the `dlhdl.Workflow` object and display the results.

```
[prediction,speed] = predict(hW,single(inputImg),'Profile','on');
```

```
### Finished writing input activations.
### Running single input activation.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------            -------------              ---------        --
Network                   25200171                  0.11455                       1           252
    data_norm_add           351489                  0.00160
    data_norm               351607                  0.00160
    conv1                  2226639                  0.01012
    pool1                   503083                  0.00229
    res2a_branch2a          973823                  0.00443
    res2a_branch2b          973242                  0.00442
    res2a                   374320                  0.00170
    res2b_branch2a          973556                  0.00443
    res2b_branch2b          973335                  0.00442
    res2b                   374442                  0.00170
    res3a_branch1           539156                  0.00245
    res3a_branch2a          541381                  0.00246
    res3a_branch2b          908821                  0.00413
    res3a                   187225                  0.00085
    res3b_branch2a          908911                  0.00413
    res3b_branch2b          909169                  0.00413
    res3b                   187275                  0.00085
    res4a_branch1           490738                  0.00223
    res4a_branch2a          493429                  0.00224
    res4a_branch2b          894069                  0.00406
    res4a                    93604                  0.00043
```

```
    res4b_branch2a          893888                  0.00406
    res4b_branch2b          893250                  0.00406
    res4b                    93654                  0.00043
    res5a_branch1          1131245                  0.00514
    res5a_branch2a         1133770                  0.00515
    res5a_branch2b         2210255                  0.01005
    res5a                    46862                  0.00021
    res5b_branch2a         2211423                  0.01005
    res5b_branch2b         2211330                  0.01005
    res5b                    46872                  0.00021
    pool5                    73644                  0.00033
    new_fc                   24477                  0.00011
 * The clock frequency of the DL processor is: 220MHz
```

```
[val,idx] = max(prediction);
netTransfer.Layers(end).ClassNames{idx}
```

```
ans =
'MathWorks Cube'
```

See Also

dlhdl.Workflow | dlhdl.Target | compile | deploy | predict

More About

- "Quantization of Deep Neural Networks"

Classify Images on FPGA Using Quantized Neural Network

This example shows how to use Deep Learning HDL Toolbox™ to deploy a quantized deep convolutional neural network (CNN) to an FPGA. In the example you use the pretrained ResNet-18 CNN to perform transfer learning and quantization. You then deploy the quantized network and use MATLAB ® to retrieve the prediction results.

ResNet-18 has been trained on over a million images and can classify images into 1000 object categories, such as keyboard, coffee mug, pencil, and many animals. The network has learned rich feature representations for a wide range of images. The network takes an image as input and outputs a label for the object in the image together with the probabilities for each of the object categories.

For this example, you need:

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model for ResNet-18 Network
- Deep Learning HDL Toolbox™ Support Package for Xilinx® FPGA and SoC Devices
- Image Processing Toolbox™
- Deep Learning Toolbox Model Quantization Library
- MATLAB® Coder™ Interface for Deep Learning

To perform classification on a new set of images, you fine-tune a pretrained ResNet-18 CNN by transfer learning. In transfer learning, you can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights. You can quickly transfer learned features to a new task using a smaller number of training images.

Load Pretrained Network

Load the pretrained ResNet-18 network.

```
net = resnet18;
```

View the layers of the pretrained network.

```
deepNetworkDesigner(net);
```

The first layer, the image input layer, requires input images of size 227-by-227-by-3, where three is the number of color channels.

```
inputSize = net.Layers(1).InputSize;
```

Load Data

This example uses the MathWorks MerchData data set. This is a small data set containing 75 images of MathWorks merchandise, belonging to five different classes (*cap*, *cube*, *playing cards*, *screwdriver*, and *torch*).

```
curDir = pwd;
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
```

```
'IncludeSubfolders',true, ...
'LabelSource','foldernames');
```

Specify Training and Validation Sets

Divide the data into training and validation data sets, so that 30% percent of the images go to the training data set and 70% of the images to the validation data set. `splitEachLabel` splits the datastore `imds` into two new datastores, `imdsTrain` and `imdsValidation`.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

Replace Final layers

To retrain ResNet-18 to classify new images, replace the last fully connected layer and final classification layer of the network. In ResNet-18 , these layers have the names `'fc1000'` and `'ClassificationLayer_predictions'`, respectively. The fully connected layer and classification layer of the pretrained network `net` are configured for 1000 classes. These two layers `fc1000` and `ClassificationLayer_predictions` in ResNet-18, contain information on how to combine the features that the network extracts into class probabilities and predicted labels. These two layers must be fine-tuned for the new classification problem. Extract all the layers, except the last two layers, from the pretrained network.

```
lgraph = layerGraph(net)
```

```
lgraph = 
  LayerGraph with properties:

     InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}
         Layers: [71×1 nnet.cnn.layer.Layer]
    Connections: [78×2 table]
```

```
numClasses = numel(categories(imdsTrain.Labels))
```

```
numClasses = 5
```

```
newLearnableLayer = fullyConnectedLayer(numClasses, ...
'Name','new_fc', ...
'WeightLearnRateFactor',10, ...
'BiasLearnRateFactor',10);
lgraph = replaceLayer(lgraph,'fc1000',newLearnableLayer);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassLayer);
```

Prepare Data for Training

The network requires input images of size 224-by-224-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images, such as randomly flipping the training images along the vertical axis and randomly translating them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
'RandXReflection',true, ...
```

```
'RandXTranslation',pixelRange, ...
'RandYTranslation',pixelRange);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
'DataAugmentation',imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify Training Options

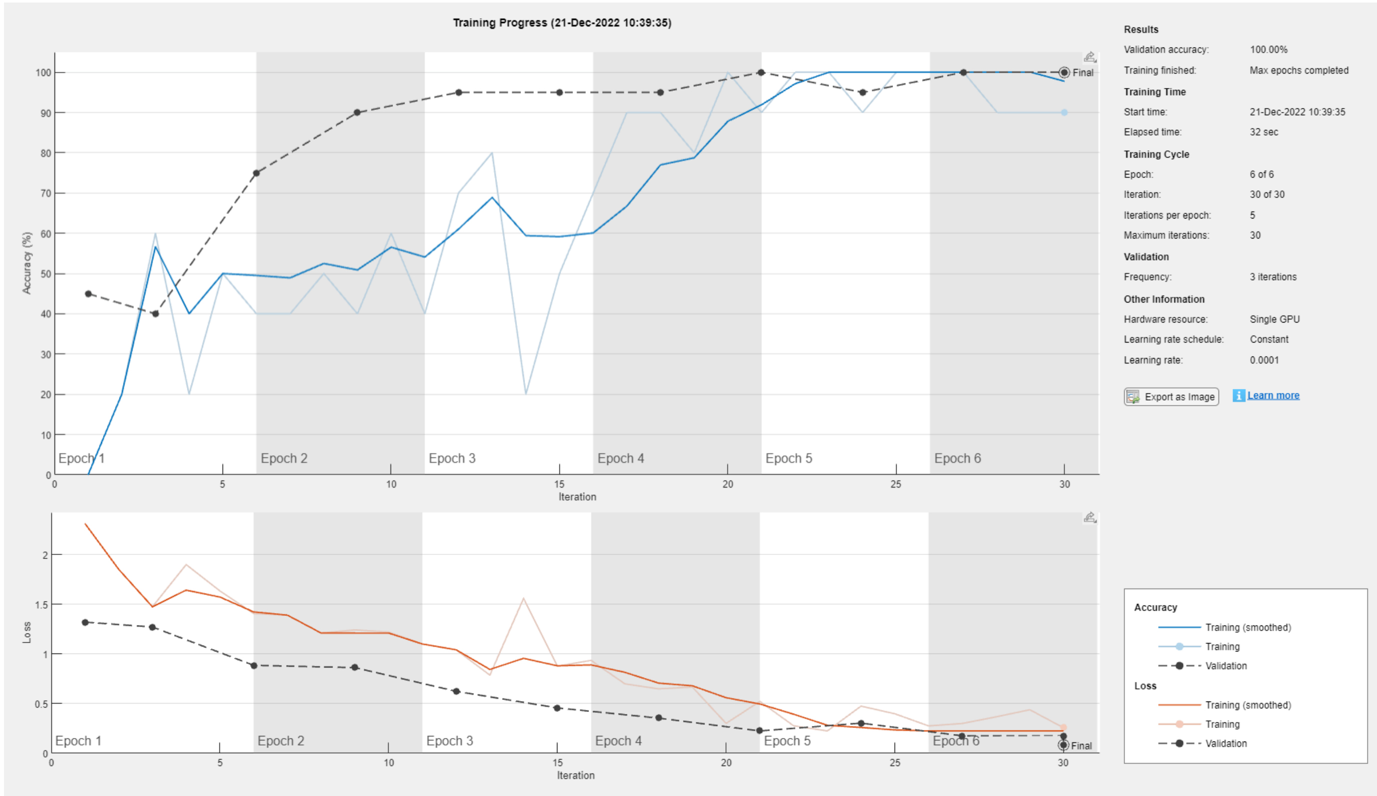
Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. Specify the mini-batch size and validation data. The software validates the network every `ValidationFrequency` iterations during training.

```
options = trainingOptions('sgdm', ...
'MiniBatchSize',10, ...
'MaxEpochs',6, ...
'InitialLearnRate',1e-4, ...
'Shuffle','every-epoch', ...
'ValidationData',augimdsValidation, ...
'ValidationFrequency',3, ...
'Verbose',false, ...
'Plots','training-progress');
```

Train Network

Train the network that consists of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available. Using this function on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For more information, see “GPU Computing Requirements” (Parallel Computing Toolbox). If a GPU is not available, the network uses a CPU (requires MATLAB Coder Interface for Deep learning). You can also specify the execution environment by using the `ExecutionEnvironment` name-value argument of `trainingOptions`.

```
netTransfer = trainNetwork(augimdsTrain,lgraph,options);
```



Quantize Network

Quantize the network using the `dlquantizer` object. Set the target execution environment to FPGA.

```
dlquantObj = dlquantizer(netTransfer,'ExecutionEnvironment','FPGA');
```

Calibrate Quantized Network

Use the `calibrate` function to exercise the network with sample inputs and collect the range information. The `calibrate` function collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns the information as a table, in which each row contains range information for a learnable parameter of the quantized network.

```
calibrate(dlquantObj,augimdsTrain)
```

ans=95×5 table

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv1_Weights' }	{'conv1' }	"Weights"	-0.79143
{'conv1_Bias' }	{'conv1' }	"Bias"	-0.66949
{'res2a_branch2a_Weights'}	{'res2a_branch2a'}	"Weights"	-0.42074
{'res2a_branch2a_Bias' }	{'res2a_branch2a'}	"Bias"	-0.8039
{'res2a_branch2b_Weights'}	{'res2a_branch2b'}	"Weights"	-0.78524
{'res2a_branch2b_Bias' }	{'res2a_branch2b'}	"Bias"	-1.3835
{'res2b_branch2a_Weights'}	{'res2b_branch2a'}	"Weights"	-0.3174
{'res2b_branch2a_Bias' }	{'res2b_branch2a'}	"Bias"	-1.1203
{'res2b_branch2b_Weights'}	{'res2b_branch2b'}	"Weights"	-1.1915

```
    {'res2b_branch2b_Bias'   }    {'res2b_branch2b'}          "Bias"           -0.81928
    {'res3a_branch2a_Weights'}    {'res3a_branch2a'}          "Weights"        -0.19735    (
    {'res3a_branch2a_Bias'   }    {'res3a_branch2a'}          "Bias"           -0.53009    (
    {'res3a_branch2b_Weights'}    {'res3a_branch2b'}          "Weights"        -0.53557    (
    {'res3a_branch2b_Bias'   }    {'res3a_branch2b'}          "Bias"           -0.67756
    {'res3a_branch1_Weights' }    {'res3a_branch1' }          "Weights"        -0.63395    (
    {'res3a_branch1_Bias'    }    {'res3a_branch1' }          "Bias"           -0.95277    (
      ⋮
```

Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Create a programming interface with custom name for your target device and an Ethernet interface to connect the target device to the host computer.

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx® Zynq® UltraScale+™ MPSoC ZCU102 board and the bitstream uses the `int8` data type.

```
hW = dlhdl.Workflow(Network=dlquantObj,Bitstream='zcu102_int8',Target=hTarget);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment.

```
dn = compile(hW,'InputFrameNumberLimit',15)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### The network includes the following layers:
     1   'data'                   Image Input              224×224×3 images with 'zscore' norm
     2   'conv1'                  2-D Convolution          64 7×7×3 convolutions with stride
     3   'conv1_relu'             ReLU                     ReLU
     4   'pool1'                  2-D Max Pooling          3×3 max pooling with stride [2  2]
     5   'res2a_branch2a'         2-D Convolution          64 3×3×64 convolutions with stride
     6   'res2a_branch2a_relu'    ReLU                     ReLU
     7   'res2a_branch2b'         2-D Convolution          64 3×3×64 convolutions with stride
     8   'res2a'                  Addition                 Element-wise addition of 2 inputs
     9   'res2a_relu'             ReLU                     ReLU
    10   'res2b_branch2a'         2-D Convolution          64 3×3×64 convolutions with stride
    11   'res2b_branch2a_relu'    ReLU                     ReLU
    12   'res2b_branch2b'         2-D Convolution          64 3×3×64 convolutions with stride
    13   'res2b'                  Addition                 Element-wise addition of 2 inputs
    14   'res2b_relu'             ReLU                     ReLU
    15   'res3a_branch2a'         2-D Convolution          128 3×3×64 convolutions with stride
    16   'res3a_branch2a_relu'    ReLU                     ReLU
    17   'res3a_branch2b'         2-D Convolution          128 3×3×128 convolutions with stri
    18   'res3a_branch1'          2-D Convolution          128 1×1×64 convolutions with stride
    19   'res3a'                  Addition                 Element-wise addition of 2 inputs
    20   'res3a_relu'             ReLU                     ReLU
```

```
    21   'res3b_branch2a'        2-D Convolution             128 3×3×128 convolutions with stric
    22   'res3b_branch2a_relu'   ReLU                        ReLU
    23   'res3b_branch2b'        2-D Convolution             128 3×3×128 convolutions with stric
    24   'res3b'                 Addition                    Element-wise addition of 2 inputs
    25   'res3b_relu'            ReLU                        ReLU
    26   'res4a_branch2a'        2-D Convolution             256 3×3×128 convolutions with stric
    27   'res4a_branch2a_relu'   ReLU                        ReLU
    28   'res4a_branch2b'        2-D Convolution             256 3×3×256 convolutions with stric
    29   'res4a_branch1'         2-D Convolution             256 1×1×128 convolutions with stric
    30   'res4a'                 Addition                    Element-wise addition of 2 inputs
    31   'res4a_relu'            ReLU                        ReLU
    32   'res4b_branch2a'        2-D Convolution             256 3×3×256 convolutions with stric
    33   'res4b_branch2a_relu'   ReLU                        ReLU
    34   'res4b_branch2b'        2-D Convolution             256 3×3×256 convolutions with stric
    35   'res4b'                 Addition                    Element-wise addition of 2 inputs
    36   'res4b_relu'            ReLU                        ReLU
    37   'res5a_branch2a'        2-D Convolution             512 3×3×256 convolutions with stric
    38   'res5a_branch2a_relu'   ReLU                        ReLU
    39   'res5a_branch2b'        2-D Convolution             512 3×3×512 convolutions with stric
    40   'res5a_branch1'         2-D Convolution             512 1×1×256 convolutions with stric
    41   'res5a'                 Addition                    Element-wise addition of 2 inputs
    42   'res5a_relu'            ReLU                        ReLU
    43   'res5b_branch2a'        2-D Convolution             512 3×3×512 convolutions with stric
    44   'res5b_branch2a_relu'   ReLU                        ReLU
    45   'res5b_branch2b'        2-D Convolution             512 3×3×512 convolutions with stric
    46   'res5b'                 Addition                    Element-wise addition of 2 inputs
    47   'res5b_relu'            ReLU                        ReLU
    48   'pool5'                 2-D Global Average Pooling  2-D global average pooling
    49   'new_fc'                Fully Connected             5 fully connected layer
    50   'prob'                  Softmax                     softmax
    51   'new_classoutput'       Classification Output       crossentropyex with 'MathWorks Cap

### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'new_classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is i
### Compiling layer group: conv1>>pool1 ...
### Compiling layer group: conv1>>pool1 ... complete.
### Compiling layer group: res2a_branch2a>>res2a_branch2b ...
### Compiling layer group: res2a_branch2a>>res2a_branch2b ... complete.
### Compiling layer group: res2b_branch2a>>res2b_branch2b ...
### Compiling layer group: res2b_branch2a>>res2b_branch2b ... complete.
### Compiling layer group: res3a_branch1 ...
### Compiling layer group: res3a_branch1 ... complete.
### Compiling layer group: res3a_branch2a>>res3a_branch2b ...
### Compiling layer group: res3a_branch2a>>res3a_branch2b ... complete.
### Compiling layer group: res3b_branch2a>>res3b_branch2b ...
### Compiling layer group: res3b_branch2a>>res3b_branch2b ... complete.
### Compiling layer group: res4a_branch1 ...
### Compiling layer group: res4a_branch1 ... complete.
### Compiling layer group: res4a_branch2a>>res4a_branch2b ...
### Compiling layer group: res4a_branch2a>>res4a_branch2b ... complete.
### Compiling layer group: res4b_branch2a>>res4b_branch2b ...
### Compiling layer group: res4b_branch2a>>res4b_branch2b ... complete.
### Compiling layer group: res5a_branch1 ...
### Compiling layer group: res5a_branch1 ... complete.
### Compiling layer group: res5a_branch2a>>res5a_branch2b ...
### Compiling layer group: res5a_branch2a>>res5a_branch2b ... complete.
### Compiling layer group: res5b_branch2a>>res5b_branch2b ...
```

```
### Compiling layer group: res5b_branch2a>>res5b_branch2b ... complete.
### Compiling layer group: pool5 ...
### Compiling layer group: pool5 ... complete.
### Compiling layer group: new_fc ...
### Compiling layer group: new_fc ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "8.0 MB"
    "OutputResultOffset"        "0x00800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x01000000"     "28.0 MB"
    "InstructionDataOffset"     "0x02c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03000000"     "16.0 MB"
    "FCWeightDataOffset"        "0x04000000"     "4.0 MB"
    "EndOffset"                 "0x04400000"     "Total: 68.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
             ddrInfo: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

```
deploy(hW)
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_int8.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_int8.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
```

```
### Conv Weights loaded. Current time is 21-Dec-2022 10:45:19
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 21-Dec-2022 10:45:19
```

Test Network

Load the example image.

```
imgFile = fullfile(pwd,'MerchData','MathWorks Cube','Mathworks cube_0.jpg');
inputImg = imresize(imread(imgFile),[224 224]);
imshow(inputImg)
```

Classify the image on the FPGA by using the `predict` method of the `dlhdl.Workflow` object and display the results.

```
[prediction,speed] = predict(hW,single(inputImg),'Profile','on');
```

```
### Finished writing input activations.
### Running single input activation.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    7392114                  0.02957                       1           73
    conv1                  1115165                  0.00446
    pool1                   199164                  0.00080
    res2a_branch2a          270125                  0.00108
    res2a_branch2b          269946                  0.00108
    res2a                   102255                  0.00041
    res2b_branch2a          269792                  0.00108
    res2b_branch2b          269902                  0.00108
    res2b                   102695                  0.00041
    res3a_branch1           155120                  0.00062
    res3a_branch2a          156480                  0.00063
```

```
    res3a_branch2b         244913                  0.00098
    res3a                   51456                  0.00021
    res3b_branch2a         245366                  0.00098
    res3b_branch2b         245123                  0.00098
    res3b                   51286                  0.00021
    res4a_branch1          135535                  0.00054
    res4a_branch2a         136117                  0.00054
    res4a_branch2b         238454                  0.00095
    res4a                   25602                  0.00010
    res4b_branch2a         237909                  0.00095
    res4b_branch2b         238282                  0.00095
    res4b                   26742                  0.00011
    res5a_branch1          324642                  0.00130
    res5a_branch2a         325897                  0.00130
    res5a_branch2b         623521                  0.00249
    res5a                   13881                  0.00006
    res5b_branch2a         624028                  0.00250
    res5b_branch2b         624631                  0.00250
    res5b                   13051                  0.00005
    pool5                   37083                  0.00015
    new_fc                  17764                  0.00007
 * The clock frequency of the DL processor is: 250MHz
```

```
[val,idx] = max(prediction);
dlquantObj.NetworkObject.Layers(end).ClassNames{idx}
```

```
ans =
'MathWorks Cube'
```

Performance Comparison

Compare the performance of the quantized network to the performance of the single data type network.

```
optionsFPGA = dlquantizationOptions('Bitstream','zcu102_int8','Target',hTarget);
predictionFPGA = validate(dlquantObj,imdsValidation,optionsFPGA)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### The network includes the following layers:
     1   'data'                   Image Input                 224×224×3 images with 'zscore' norm
     2   'conv1'                  2-D Convolution             64 7×7×3 convolutions with stride
     3   'conv1_relu'             ReLU                        ReLU
     4   'pool1'                  2-D Max Pooling             3×3 max pooling with stride [2  2]
     5   'res2a_branch2a'         2-D Convolution             64 3×3×64 convolutions with stride
     6   'res2a_branch2a_relu'    ReLU                        ReLU
     7   'res2a_branch2b'         2-D Convolution             64 3×3×64 convolutions with stride
     8   'res2a'                  Addition                    Element-wise addition of 2 inputs
     9   'res2a_relu'             ReLU                        ReLU
    10   'res2b_branch2a'         2-D Convolution             64 3×3×64 convolutions with stride
    11   'res2b_branch2a_relu'    ReLU                        ReLU
    12   'res2b_branch2b'         2-D Convolution             64 3×3×64 convolutions with stride
    13   'res2b'                  Addition                    Element-wise addition of 2 inputs
    14   'res2b_relu'             ReLU                        ReLU
    15   'res3a_branch2a'         2-D Convolution             128 3×3×64 convolutions with stride
    16   'res3a_branch2a_relu'    ReLU                        ReLU
    17   'res3a_branch2b'         2-D Convolution             128 3×3×128 convolutions with stric
    18   'res3a_branch1'          2-D Convolution             128 1×1×64 convolutions with stride
```

```
    19   'res3a'                  Addition                     Element-wise addition of 2 inputs
    20   'res3a_relu'             ReLU                         ReLU
    21   'res3b_branch2a'         2-D Convolution              128 3×3×128 convolutions with stric
    22   'res3b_branch2a_relu'    ReLU                         ReLU
    23   'res3b_branch2b'         2-D Convolution              128 3×3×128 convolutions with stric
    24   'res3b'                  Addition                     Element-wise addition of 2 inputs
    25   'res3b_relu'             ReLU                         ReLU
    26   'res4a_branch2a'         2-D Convolution              256 3×3×128 convolutions with stric
    27   'res4a_branch2a_relu'    ReLU                         ReLU
    28   'res4a_branch2b'         2-D Convolution              256 3×3×256 convolutions with stric
    29   'res4a_branch1'          2-D Convolution              256 1×1×128 convolutions with stric
    30   'res4a'                  Addition                     Element-wise addition of 2 inputs
    31   'res4a_relu'             ReLU                         ReLU
    32   'res4b_branch2a'         2-D Convolution              256 3×3×256 convolutions with stric
    33   'res4b_branch2a_relu'    ReLU                         ReLU
    34   'res4b_branch2b'         2-D Convolution              256 3×3×256 convolutions with stric
    35   'res4b'                  Addition                     Element-wise addition of 2 inputs
    36   'res4b_relu'             ReLU                         ReLU
    37   'res5a_branch2a'         2-D Convolution              512 3×3×256 convolutions with stric
    38   'res5a_branch2a_relu'    ReLU                         ReLU
    39   'res5a_branch2b'         2-D Convolution              512 3×3×512 convolutions with stric
    40   'res5a_branch1'          2-D Convolution              512 1×1×256 convolutions with stric
    41   'res5a'                  Addition                     Element-wise addition of 2 inputs
    42   'res5a_relu'             ReLU                         ReLU
    43   'res5b_branch2a'         2-D Convolution              512 3×3×512 convolutions with stric
    44   'res5b_branch2a_relu'    ReLU                         ReLU
    45   'res5b_branch2b'         2-D Convolution              512 3×3×512 convolutions with stric
    46   'res5b'                  Addition                     Element-wise addition of 2 inputs
    47   'res5b_relu'             ReLU                         ReLU
    48   'pool5'                  2-D Global Average Pooling   2-D global average pooling
    49   'new_fc'                 Fully Connected              5 fully connected layer
    50   'prob'                   Softmax                      softmax
    51   'new_classoutput'        Classification Output        crossentropyex with 'MathWorks Cap

### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'new_classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is i
### Compiling layer group: conv1>>pool1 ...
### Compiling layer group: conv1>>pool1 ... complete.
### Compiling layer group: res2a_branch2a>>res2a_branch2b ...
### Compiling layer group: res2a_branch2a>>res2a_branch2b ... complete.
### Compiling layer group: res2b_branch2a>>res2b_branch2b ...
### Compiling layer group: res2b_branch2a>>res2b_branch2b ... complete.
### Compiling layer group: res3a_branch1 ...
### Compiling layer group: res3a_branch1 ... complete.
### Compiling layer group: res3a_branch2a>>res3a_branch2b ...
### Compiling layer group: res3a_branch2a>>res3a_branch2b ... complete.
### Compiling layer group: res3b_branch2a>>res3b_branch2b ...
### Compiling layer group: res3b_branch2a>>res3b_branch2b ... complete.
### Compiling layer group: res4a_branch1 ...
### Compiling layer group: res4a_branch1 ... complete.
### Compiling layer group: res4a_branch2a>>res4a_branch2b ...
### Compiling layer group: res4a_branch2a>>res4a_branch2b ... complete.
### Compiling layer group: res4b_branch2a>>res4b_branch2b ...
### Compiling layer group: res4b_branch2a>>res4b_branch2b ... complete.
### Compiling layer group: res5a_branch1 ...
### Compiling layer group: res5a_branch1 ... complete.
### Compiling layer group: res5a_branch2a>>res5a_branch2b ...
```

```
### Compiling layer group: res5a_branch2a>>res5a_branch2b ... complete.
### Compiling layer group: res5b_branch2a>>res5b_branch2b ...
### Compiling layer group: res5b_branch2a>>res5b_branch2b ... complete.
### Compiling layer group: pool5 ...
### Compiling layer group: pool5 ... complete.
### Compiling layer group: new_fc ...
### Compiling layer group: new_fc ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "12.0 MB"
    "OutputResultOffset"        "0x00c00000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01000000"     "4.0 MB"
    "SystemBufferOffset"        "0x01400000"     "28.0 MB"
    "InstructionDataOffset"     "0x03000000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03400000"     "16.0 MB"
    "FCWeightDataOffset"        "0x04400000"     "4.0 MB"
    "EndOffset"                 "0x04800000"     "Total: 72.0 MB"

### Network compilation complete.

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 21-Dec-2022 10:46:36
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 21-Dec-2022 10:46:36
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
```

```
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.
### Finished writing input activations.
### Running single input activation.


              Deep Learning Processor Bitstream Build Info

Resource                   Utilized           Total        Percentage
------------------        ----------      ------------    ------------
LUTs (CLB/ALM)*              249703           274080           91.11
DSPs                            391             2520           15.52
Block RAM                       583              912           63.93
* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive L

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data'
### The network includes the following layers:
     1   'data'                   Image Input                  224×224×3 images with 'zscore' norm
     2   'conv1'                  2-D Convolution              64 7×7×3 convolutions with stride
     3   'conv1_relu'             ReLU                         ReLU
     4   'pool1'                  2-D Max Pooling              3×3 max pooling with stride [2  2]
     5   'res2a_branch2a'         2-D Convolution              64 3×3×64 convolutions with stride
     6   'res2a_branch2a_relu'    ReLU                         ReLU
     7   'res2a_branch2b'         2-D Convolution              64 3×3×64 convolutions with stride
     8   'res2a'                  Addition                     Element-wise addition of 2 inputs
     9   'res2a_relu'             ReLU                         ReLU
    10   'res2b_branch2a'         2-D Convolution              64 3×3×64 convolutions with stride
    11   'res2b_branch2a_relu'    ReLU                         ReLU
    12   'res2b_branch2b'         2-D Convolution              64 3×3×64 convolutions with stride
    13   'res2b'                  Addition                     Element-wise addition of 2 inputs
    14   'res2b_relu'             ReLU                         ReLU
    15   'res3a_branch2a'         2-D Convolution              128 3×3×64 convolutions with stride
    16   'res3a_branch2a_relu'    ReLU                         ReLU
    17   'res3a_branch2b'         2-D Convolution              128 3×3×128 convolutions with strid
    18   'res3a_branch1'          2-D Convolution              128 1×1×64 convolutions with stride
    19   'res3a'                  Addition                     Element-wise addition of 2 inputs
    20   'res3a_relu'             ReLU                         ReLU
    21   'res3b_branch2a'         2-D Convolution              128 3×3×128 convolutions with strid
    22   'res3b_branch2a_relu'    ReLU                         ReLU
    23   'res3b_branch2b'         2-D Convolution              128 3×3×128 convolutions with strid
    24   'res3b'                  Addition                     Element-wise addition of 2 inputs
    25   'res3b_relu'             ReLU                         ReLU
    26   'res4a_branch2a'         2-D Convolution              256 3×3×128 convolutions with strid
    27   'res4a_branch2a_relu'    ReLU                         ReLU
    28   'res4a_branch2b'         2-D Convolution              256 3×3×256 convolutions with strid
    29   'res4a_branch1'          2-D Convolution              256 1×1×128 convolutions with strid
    30   'res4a'                  Addition                     Element-wise addition of 2 inputs
    31   'res4a_relu'             ReLU                         ReLU
    32   'res4b_branch2a'         2-D Convolution              256 3×3×256 convolutions with strid
    33   'res4b_branch2a_relu'    ReLU                         ReLU
```

```
    34   'res4b_branch2b'        2-D Convolution            256 3×3×256 convolutions with stric
    35   'res4b'                 Addition                   Element-wise addition of 2 inputs
    36   'res4b_relu'            ReLU                       ReLU
    37   'res5a_branch2a'        2-D Convolution            512 3×3×256 convolutions with stric
    38   'res5a_branch2a_relu'   ReLU                       ReLU
    39   'res5a_branch2b'        2-D Convolution            512 3×3×512 convolutions with stric
    40   'res5a_branch1'         2-D Convolution            512 1×1×256 convolutions with stric
    41   'res5a'                 Addition                   Element-wise addition of 2 inputs
    42   'res5a_relu'            ReLU                       ReLU
    43   'res5b_branch2a'        2-D Convolution            512 3×3×512 convolutions with stric
    44   'res5b_branch2a_relu'   ReLU                       ReLU
    45   'res5b_branch2b'        2-D Convolution            512 3×3×512 convolutions with stric
    46   'res5b'                 Addition                   Element-wise addition of 2 inputs
    47   'res5b_relu'            ReLU                       ReLU
    48   'pool5'                 2-D Global Average Pooling   2-D global average pooling
    49   'new_fc'                Fully Connected            5 fully connected layer
    50   'prob'                  Softmax                    softmax
    51   'new_classoutput'       Classification Output      crossentropyex with 'MathWorks Cap

### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'new_classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is i


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------              ---------        --
Network                   23502752                  0.10683                       1          235
    data_norm_add           210750                  0.00096
    data_norm               210750                  0.00096
    conv1                  2164124                  0.00984
    pool1                   515064                  0.00234
    res2a_branch2a          966221                  0.00439
    res2a_branch2b          966221                  0.00439
    res2a                   210750                  0.00096
    res2b_branch2a          966221                  0.00439
    res2b_branch2b          966221                  0.00439
    res2b                   210750                  0.00096
    res3a_branch1           540861                  0.00246
    res3a_branch2a          540749                  0.00246
    res3a_branch2b          919117                  0.00418
    res3a                   105404                  0.00048
    res3b_branch2a          919117                  0.00418
    res3b_branch2b          919117                  0.00418
    res3b                   105404                  0.00048
    res4a_branch1           503405                  0.00229
    res4a_branch2a          509261                  0.00231
    res4a_branch2b          905421                  0.00412
    res4a                    52724                  0.00024
    res4b_branch2a          905421                  0.00412
    res4b_branch2b          905421                  0.00412
    res4b                    52724                  0.00024
    res5a_branch1          1039437                  0.00472
    res5a_branch2a         1046605                  0.00476
    res5a_branch2b         2005197                  0.00911
    res5a                    26368                  0.00012
    res5b_branch2a         2005197                  0.00911
    res5b_branch2b         2005197                  0.00911
```

```
   res5b                  26368                  0.00012
   pool5                  54594                  0.00025
   new_fc                 22571                  0.00010
 * The clock frequency of the DL processor is: 220MHz



             Deep Learning Processor Bitstream Build Info

Resource                   Utilized           Total        Percentage
------------------        ----------      ------------    ------------
LUTs (CLB/ALM)*              168099          274080           61.33
DSPs                            807            2520           32.02
Block RAM                       453             912           49.67
* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive L

### Finished writing input activations.
### Running single input activation.

predictionFPGA = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: [2×7 table]
```

View the frames per second performance for the quantized network and single-data-type network. The quantized network has a performance of 33.8 frames per second compared to 9.2 frames per second for the single-data-type network. You can use quantization to improve your frames per second performance, however yo could lose accuracy when you quantize your networks.

```
predictionFPGA.Statistics.FramesPerSecond
```

```
ans = 2×1

    9.3606
   33.7719
```

See Also

dlhdl.Workflow | dlhdl.Target | compile | deploy | predict | dlquantizer | dlquantizationOptions | calibrate | validate

More About

- “Quantization of Deep Neural Networks”

Classify ECG Signals Using DAG Network Deployed to FPGA

This example shows how to classify human electrocardiogram (ECG) signals by deploying a transfer learning trained SqueezeNet network `trainedSN` to a Xilinx Zynq Ultrascale+ ZCU102 board.

Required Products

For this example, you need:

- Deep Learning Toolbox ™
- Image Processing Toolbox ™
- Wavelet Toolbox ™
- Deep Learning HDL Toolbox ™
- Deep Learning HDL Toolbox ™ Support Package for Xilinx FPGA and SoC Devices
- Xilinx Zynq Ultrascale+ MPSoC ZCu102

Download Data

Download the data from the GitHub repository. To download the data from the website, click **Clone** and select **Download ZIP**. Save the file `physionet_ECG_data-main.zip` in a folder where you have write permission.

After downloading the data from GitHub, unzip the file in your temporary directory.

```
unzip(fullfile(tempdir,'physionet_ECG_data-main.zip'),tempdir);
```

The ECG data is classified into these labels:

- persons with cardiac arrhythmia (ARR)
- persons with congestive heart failure (CHF)
- persons with normal sinus rhythms (NSR)

The data is collected from these sources:

- MIT-BIH Arrhythmia Database [3][7]
- MIT-BIH Normal Sinus Rhythm Database [3]
- BIDMC Congestive Heart Failure Database [1][3]

Unzipping creates the folder `physionet-ECG_data-main` in your temporary directory.

Unzip `ECGData.zip` in `physionet-ECG_data-main`. Load the ECGData.`mat` data file into your MATLAB workspace.

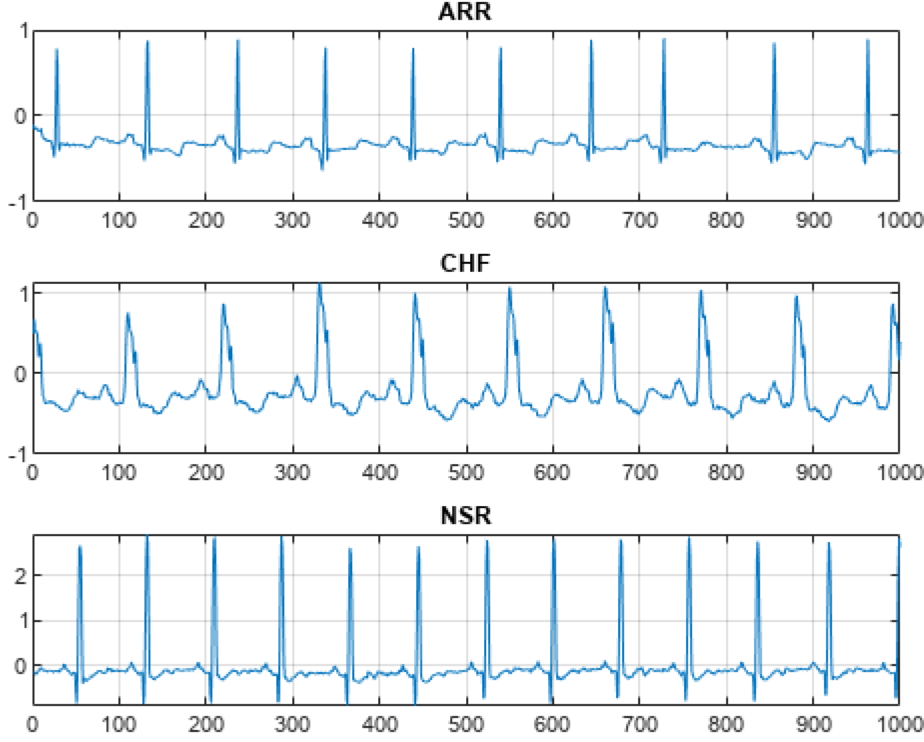
```
unzip(fullfile(tempdir,'physionet_ECG_data-main','ECGData.zip'),...
    fullfile(tempdir,'physionet_ECG_data-main'))
load(fullfile(tempdir,'physionet_ECG_data-main','ECGData.mat'))
```

Create a folder called `dataDir` inside the ECG data directory and then create three directories called `ARR`, `CHF`, and `NSR` inside `dataDir` by using the `helperCreateECGDirectories` function. You can find the source code for this helper function in the Supporting Functions section at the end of this example.

```
% parentDir = tempdir;
parentDir = pwd;
dataDir = 'data';
helperCreateECGDirectories(ECGData,parentDir,dataDir);
```

Plot an ECG that represents each ECG category by using the `helperPlotReps` helper function. does this. You can find the source code for this helper function in the Supporting Functions section at the end of this example.

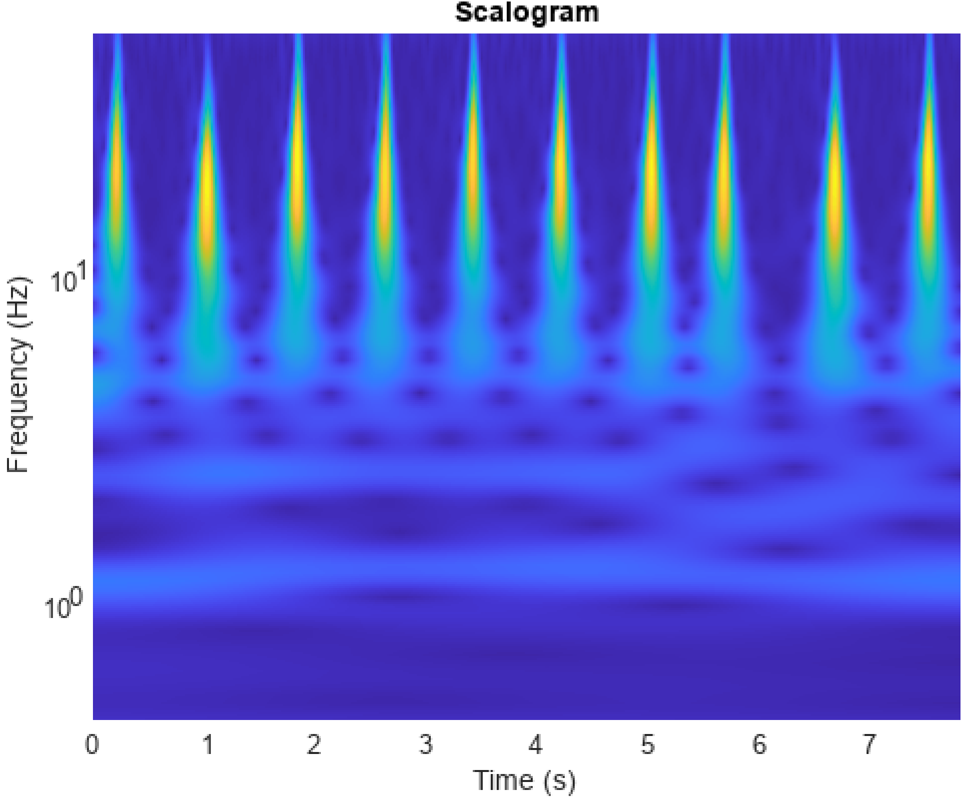
```
helperPlotReps(ECGData)
```



Create Time-Frequency Representations

After making the folders, create time-frequency representations of the ECG signals. Creating time-frequency representations helps with feature extraction. These representations are called scalograms. A scalogram is the absolute value of the continuous wavelet transform (CWT) coefficients of a signal. Create a CWT filter bank using `cwtfilterbank` (Wavelet Toolbox) (Wavelet Toolbox) for a signal with 1000 samples.

```
Fs =128;
fb = cwtfilterbank(SignalLength=1000,...
    SamplingFrequency=Fs,...
    VoicesPerOctave=12);
sig = ECGData.Data(1,1:1000);
[cfs,frq] = wt(fb,sig);
t = (0:999)/Fs;figure;pcolor(t,frq,abs(cfs))
set(gca,'yscale','log');shading interp;axis tight;
title('Scalogram');xlabel('Time (s)');ylabel('Frequency (Hz)')
```



Use the `helperCreateRGBfromTF` helper function to create the scalograms as RGB images and write them to the appropriate subdirectory in `dataDir`. The source code for this helper function is in the Supporting Functions section at the end of this example. To be compatible with the SqueezeNet architecture, each RGB image is an array of size 227-by-227-by-3.

```
helperCreateRGBfromTF(ECGData,parentDir,dataDir)
```

Divide into Training and Validation Data

Load the scalogram images as an image datastore. The `imageDatastore` function automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a CNN.

```
allImages = imageDatastore(fullfile(parentDir,dataDir),...
    'IncludeSubfolders',true,...
    'LabelSource','foldernames');
```

Randomly divide the images into two groups. Use 80% of the images for training, and the remainder for validation. For purposes of reproducibility, we set the random seed to the default value.

```
rng default
[imgsTrain,imgsValidation] = splitEachLabel(allImages,0.8,'randomized');
disp(['Number of training images: ',num2str(numel(imgsTrain.Files))]);
disp(['Number of validation images: ',num2str(numel(imgsValidation.Files))]);
```

Load Transfer Learning Trained Network

Load the transfer learning trained SqueezeNet network `trainedSN`. To create the `trainedSN` network, see "Classify Time Series Using Wavelet Analysis and Deep Learning".

```
load('trainedSN.mat');
```

Configure FPGA Board Interface

Configure the FPGA board interface for the deep learning network deployment and MATLAB communication by using the `dlhdl.Target` class to create a target object with a custom name for your target device and an interface to connect your target device to the host computer. To use JTAG,Install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.b
```

```
hTarget = dlhdl.Target('Xilinx',Interface="Ethernet");
```

Prepare `trainedSN` Network for Deployment

Prepare the `trainedSN` network for deployment by using the `dlhdl.Workflow` class to create an object. When you create the object, specify the network and the bitstream name. Specify `trainedSN` as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx ZCU102 SoC board. The bitstream uses a single data type.

```
hW=dlhdl.Workflow(Network=trainedSN,Bitstream='zcu102_single',Target=hTarget)
```

```
hW = 
  Workflow with properties:

            Network: [1×1 DAGNetwork]
          Bitstream: 'zcu102_single'
    ProcessorConfig: []
             Target: [1×1 dnnfpga.hardware.TargetEthernet]
```

Generate Weights, Biases, and Instructions

Generate weights, biases, and instructions for the `trainedSN` network by using the `compile` method of the `dlhdl.Workflow` object.

```
dn = hW.compile
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### The network includes the following layers:
     1   'data'                      Image Input             227×227×3 images with 'zerocenter
     2   'conv1'                     Convolution             64 3×3×3 convolutions with stride
     3   'relu_conv1'                ReLU                    ReLU
     4   'pool1'                     Max Pooling             3×3 max pooling with stride [2  2
     5   'fire2-squeeze1x1'          Convolution             16 1×1×64 convolutions with strid
     6   'fire2-relu_squeeze1x1'     ReLU                    ReLU
     7   'fire2-expand1x1'           Convolution             64 1×1×16 convolutions with strid
     8   'fire2-relu_expand1x1'      ReLU                    ReLU
     9   'fire2-expand3x3'           Convolution             64 3×3×16 convolutions with strid
    10   'fire2-relu_expand3x3'      ReLU                    ReLU
    11   'fire2-concat'              Depth concatenation     Depth concatenation of 2 inputs
```

```
12   'fire3-squeeze1x1'        Convolution                 16 1×1×128 convolutions with stri
13   'fire3-relu_squeeze1x1'   ReLU                        ReLU
14   'fire3-expand1x1'         Convolution                 64 1×1×16 convolutions with stric
15   'fire3-relu_expand1x1'    ReLU                        ReLU
16   'fire3-expand3x3'         Convolution                 64 3×3×16 convolutions with stric
17   'fire3-relu_expand3x3'    ReLU                        ReLU
18   'fire3-concat'            Depth concatenation         Depth concatenation of 2 inputs
19   'pool3'                   Max Pooling                 3×3 max pooling with stride [2  2
20   'fire4-squeeze1x1'        Convolution                 32 1×1×128 convolutions with stri
21   'fire4-relu_squeeze1x1'   ReLU                        ReLU
22   'fire4-expand1x1'         Convolution                 128 1×1×32 convolutions with stri
23   'fire4-relu_expand1x1'    ReLU                        ReLU
24   'fire4-expand3x3'         Convolution                 128 3×3×32 convolutions with stri
25   'fire4-relu_expand3x3'    ReLU                        ReLU
26   'fire4-concat'            Depth concatenation         Depth concatenation of 2 inputs
27   'fire5-squeeze1x1'        Convolution                 32 1×1×256 convolutions with stri
28   'fire5-relu_squeeze1x1'   ReLU                        ReLU
29   'fire5-expand1x1'         Convolution                 128 1×1×32 convolutions with stri
30   'fire5-relu_expand1x1'    ReLU                        ReLU
31   'fire5-expand3x3'         Convolution                 128 3×3×32 convolutions with stri
32   'fire5-relu_expand3x3'    ReLU                        ReLU
33   'fire5-concat'            Depth concatenation         Depth concatenation of 2 inputs
34   'pool5'                   Max Pooling                 3×3 max pooling with stride [2  2
35   'fire6-squeeze1x1'        Convolution                 48 1×1×256 convolutions with stri
36   'fire6-relu_squeeze1x1'   ReLU                        ReLU
37   'fire6-expand1x1'         Convolution                 192 1×1×48 convolutions with stri
38   'fire6-relu_expand1x1'    ReLU                        ReLU
39   'fire6-expand3x3'         Convolution                 192 3×3×48 convolutions with stri
40   'fire6-relu_expand3x3'    ReLU                        ReLU
41   'fire6-concat'            Depth concatenation         Depth concatenation of 2 inputs
42   'fire7-squeeze1x1'        Convolution                 48 1×1×384 convolutions with stri
43   'fire7-relu_squeeze1x1'   ReLU                        ReLU
44   'fire7-expand1x1'         Convolution                 192 1×1×48 convolutions with stri
45   'fire7-relu_expand1x1'    ReLU                        ReLU
46   'fire7-expand3x3'         Convolution                 192 3×3×48 convolutions with stri
47   'fire7-relu_expand3x3'    ReLU                        ReLU
48   'fire7-concat'            Depth concatenation         Depth concatenation of 2 inputs
49   'fire8-squeeze1x1'        Convolution                 64 1×1×384 convolutions with stri
50   'fire8-relu_squeeze1x1'   ReLU                        ReLU
51   'fire8-expand1x1'         Convolution                 256 1×1×64 convolutions with stri
52   'fire8-relu_expand1x1'    ReLU                        ReLU
53   'fire8-expand3x3'         Convolution                 256 3×3×64 convolutions with stri
54   'fire8-relu_expand3x3'    ReLU                        ReLU
55   'fire8-concat'            Depth concatenation         Depth concatenation of 2 inputs
56   'fire9-squeeze1x1'        Convolution                 64 1×1×512 convolutions with stri
57   'fire9-relu_squeeze1x1'   ReLU                        ReLU
58   'fire9-expand1x1'         Convolution                 256 1×1×64 convolutions with stri
59   'fire9-relu_expand1x1'    ReLU                        ReLU
60   'fire9-expand3x3'         Convolution                 256 3×3×64 convolutions with stri
61   'fire9-relu_expand3x3'    ReLU                        ReLU
62   'fire9-concat'            Depth concatenation         Depth concatenation of 2 inputs
63   'new_dropout'             Dropout                     60% dropout
64   'new_conv'                Convolution                 3 1×1×512 convolutions with stric
65   'relu_conv10'             ReLU                        ReLU
66   'pool10'                  2-D Global Average Pooling  2-D global average pooling
67   'prob'                    Softmax                     softmax
68   'new_classoutput'         Classification Output       crossentropyex with 'ARR' and 2 c
```

```
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data'
### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'new_classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is i
### Compiling layer group: conv1>>fire2-relu_squeeze1x1 ...
### Compiling layer group: conv1>>fire2-relu_squeeze1x1 ... complete.
### Compiling layer group: fire2-expand1x1>>fire2-relu_expand1x1 ...
### Compiling layer group: fire2-expand1x1>>fire2-relu_expand1x1 ... complete.
### Compiling layer group: fire2-expand3x3>>fire2-relu_expand3x3 ...
### Compiling layer group: fire2-expand3x3>>fire2-relu_expand3x3 ... complete.
### Compiling layer group: fire3-squeeze1x1>>fire3-relu_squeeze1x1 ...
### Compiling layer group: fire3-squeeze1x1>>fire3-relu_squeeze1x1 ... complete.
### Compiling layer group: fire3-expand1x1>>fire3-relu_expand1x1 ...
### Compiling layer group: fire3-expand1x1>>fire3-relu_expand1x1 ... complete.
### Compiling layer group: fire3-expand3x3>>fire3-relu_expand3x3 ...
### Compiling layer group: fire3-expand3x3>>fire3-relu_expand3x3 ... complete.
### Compiling layer group: pool3>>fire4-relu_squeeze1x1 ...
### Compiling layer group: pool3>>fire4-relu_squeeze1x1 ... complete.
### Compiling layer group: fire4-expand1x1>>fire4-relu_expand1x1 ...
### Compiling layer group: fire4-expand1x1>>fire4-relu_expand1x1 ... complete.
### Compiling layer group: fire4-expand3x3>>fire4-relu_expand3x3 ...
### Compiling layer group: fire4-expand3x3>>fire4-relu_expand3x3 ... complete.
### Compiling layer group: fire5-squeeze1x1>>fire5-relu_squeeze1x1 ...
### Compiling layer group: fire5-squeeze1x1>>fire5-relu_squeeze1x1 ... complete.
### Compiling layer group: fire5-expand1x1>>fire5-relu_expand1x1 ...
### Compiling layer group: fire5-expand1x1>>fire5-relu_expand1x1 ... complete.
### Compiling layer group: fire5-expand3x3>>fire5-relu_expand3x3 ...
### Compiling layer group: fire5-expand3x3>>fire5-relu_expand3x3 ... complete.
### Compiling layer group: pool5>>fire6-relu_squeeze1x1 ...
### Compiling layer group: pool5>>fire6-relu_squeeze1x1 ... complete.
### Compiling layer group: fire6-expand1x1>>fire6-relu_expand1x1 ...
### Compiling layer group: fire6-expand1x1>>fire6-relu_expand1x1 ... complete.
### Compiling layer group: fire6-expand3x3>>fire6-relu_expand3x3 ...
### Compiling layer group: fire6-expand3x3>>fire6-relu_expand3x3 ... complete.
### Compiling layer group: fire7-squeeze1x1>>fire7-relu_squeeze1x1 ...
### Compiling layer group: fire7-squeeze1x1>>fire7-relu_squeeze1x1 ... complete.
### Compiling layer group: fire7-expand1x1>>fire7-relu_expand1x1 ...
### Compiling layer group: fire7-expand1x1>>fire7-relu_expand1x1 ... complete.
### Compiling layer group: fire7-expand3x3>>fire7-relu_expand3x3 ...
### Compiling layer group: fire7-expand3x3>>fire7-relu_expand3x3 ... complete.
### Compiling layer group: fire8-squeeze1x1>>fire8-relu_squeeze1x1 ...
### Compiling layer group: fire8-squeeze1x1>>fire8-relu_squeeze1x1 ... complete.
### Compiling layer group: fire8-expand1x1>>fire8-relu_expand1x1 ...
### Compiling layer group: fire8-expand1x1>>fire8-relu_expand1x1 ... complete.
### Compiling layer group: fire8-expand3x3>>fire8-relu_expand3x3 ...
### Compiling layer group: fire8-expand3x3>>fire8-relu_expand3x3 ... complete.
### Compiling layer group: fire9-squeeze1x1>>fire9-relu_squeeze1x1 ...
### Compiling layer group: fire9-squeeze1x1>>fire9-relu_squeeze1x1 ... complete.
### Compiling layer group: fire9-expand1x1>>fire9-relu_expand1x1 ...
### Compiling layer group: fire9-expand1x1>>fire9-relu_expand1x1 ... complete.
### Compiling layer group: fire9-expand3x3>>fire9-relu_expand3x3 ...
### Compiling layer group: fire9-expand3x3>>fire9-relu_expand3x3 ... complete.
### Compiling layer group: new_conv>>pool10 ...
### Compiling layer group: new_conv>>pool10 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    ________________
```

```
    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x02000000"     "28.0 MB"
    "InstructionDataOffset"     "0x03c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04000000"     "12.0 MB"
    "EndOffset"                 "0x04c00000"     "Total: 76.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {{}  [-24.2516 -50.7900 -184.4480 0 -24.2516 -50.7900 -184.4480 0 -24.2516
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 hardware, run the deploy function of the
`dlhdl.Workflow`

object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

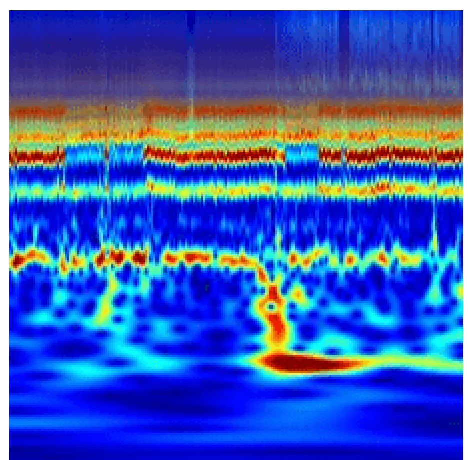
```
hW.deploy
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_single.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_single.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 28-Apr-2022 15:33:54
```

Load Image for Prediction and Run Prediction

Load an image by randomly selecting an image from the validation data store.

```
idx=randi(32);
testim=readimage(imgsValidation,idx);
imshow(testim)
```



Execute the predict method on the `dlhdl.Workflow` object and then show the label in the MATLAB command window.

```
[YPred1,probs1] = classify(trainedSN,testim);
accuracy1 = (YPred1==imgsValidation.Labels);
[YPred2,probs2] = hW.predict(single(testim),'profile','on');
```

```
### Finished writing input activations.
### Running single input activation.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                     9253245                  0.04206                       1           92
    data_norm                361047                  0.00164
    conv1                    672559                  0.00306
    pool1                    509079                  0.00231
    fire2-squeeze1x1         308258                  0.00140
    fire2-expand1x1          305646                  0.00139
    fire2-expand3x3          305085                  0.00139
    fire3-squeeze1x1         627799                  0.00285
    fire3-expand1x1          305241                  0.00139
    fire3-expand3x3          305256                  0.00139
    pool3                    286627                  0.00130
    fire4-squeeze1x1         264151                  0.00120
    fire4-expand1x1          264600                  0.00120
    fire4-expand3x3          264567                  0.00120
    fire5-squeeze1x1         734588                  0.00334
    fire5-expand1x1          264575                  0.00120
    fire5-expand3x3          264719                  0.00120
    pool5                    219725                  0.00100
    fire6-squeeze1x1         194605                  0.00088
    fire6-expand1x1          144199                  0.00066
```

```
    fire6-expand3x3         144819                   0.00066
    fire7-squeeze1x1        288819                   0.00131
    fire7-expand1x1         144285                   0.00066
    fire7-expand3x3         144841                   0.00066
    fire8-squeeze1x1        368116                   0.00167
    fire8-expand1x1         243691                   0.00111
    fire8-expand3x3         243738                   0.00111
    fire9-squeeze1x1        488338                   0.00222
    fire9-expand1x1         243654                   0.00111
    fire9-expand3x3         243683                   0.00111
    new_conv                 93849                   0.00043
    pool10                    2751                   0.00001
 * The clock frequency of the DL processor is: 220MHz
```

```
[val,idx]= max(YPred2);
trainedSN.Layers(end).ClassNames{idx}
```

```
ans =
'ARR'
```

References

1. Baim, D. S., W. S. Colucci, E. S. Monrad, H. S. Smith, R. F. Wright, A. Lanoue, D. F. Gauthier, B. J. Ransil, W. Grossman, and E. Braunwald. "Survival of patients with severe congestive heart failure treated with oral milrinone." *Journal of the American College of Cardiology*. Vol. 7, Number 3, 1986, pp. 661–670.
2. Engin, M. "ECG beat classification using neuro-fuzzy network." *Pattern Recognition Letters*. Vol. 25, Number 15, 2004, pp.1715–1722.
3. Goldberger A. L., L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. "PhysioBank, PhysioToolkit,and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." Circulation. Vol. 101, Number 23: e215–e220. [Circulation Electronic Pages; `http://circ.ahajournals.org/content/101/23/e215.full`]; 2000 (June 13). doi: 10.1161/01.CIR.101.23.e215.
4. Leonarduzzi, R. F., G. Schlotthauer, and M. E. Torres. "Wavelet leader based multifractal analysis of heart rate variability during myocardial ischaemia." In *Engineering in Medicine and Biology Society (EMBC), Annual International Conference of the IEEE*, 110–113. Buenos Aires, Argentina: IEEE, 2010.
5. Li, T., and M. Zhou. "ECG classification using wavelet packet entropy and random forests." *Entropy*. Vol. 18, Number 8, 2016, p.285.
6. Maharaj, E. A., and A. M. Alonso. "Discriminant analysis of multivariate time series: Application to diagnosis based on ECG signals." *Computational Statistics and Data Analysis*. Vol. 70, 2014, pp. 67–87.
7. Moody, G. B., and R. G. Mark. "The impact of the MIT-BIH Arrhythmia Database." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 20. Number 3, May-June 2001, pp. 45–50. (PMID: 11446209)
8. Russakovsky, O., J. Deng, and H. Su et al. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision*. Vol. 115, Number 3, 2015, pp. 211–252.
9. Zhao, Q., and L. Zhang. "ECG feature extraction and classification using wavelet transform and support vector machines." In *IEEE International Conference on Neural Networks and Brain*, 1089–1092. Beijing, China: IEEE, 2005.
10. *ImageNet*. `http://www.image-net.org`

Supporting Functions

helperCreateECGDataDirectories creates a data directory inside a parent directory, then creates three subdirectories inside the data directory. The subdirectories are named after each class of ECG signal found in `ECGData`.

```
function helperCreateECGDirectories(ECGData,parentFolder,dataFolder)

rootFolder = parentFolder;
localFolder = dataFolder;
mkdir(fullfile(rootFolder,localFolder))

folderLabels = unique(ECGData.Labels);
for i = 1:numel(folderLabels)
    mkdir(fullfile(rootFolder,localFolder,char(folderLabels(i))));
end
end
```

helperPlotReps plots the first thousand samples of a representative of each class of ECG signal found in `ECGData`.

```
function helperPlotReps(ECGData)

folderLabels = unique(ECGData.Labels);

for k=1:3
    ecgType = folderLabels{k};
    ind = find(ismember(ECGData.Labels,ecgType));
    subplot(3,1,k)
    plot(ECGData.Data(ind(1),1:1000));
    grid on
    title(ecgType)
end
end
```

helperCreateRGBfromTF uses `cwtfilterbank` (Wavelet Toolbox) to obtain the continuous wavelet transform of the ECG signals and generates the scalograms from the wavelet coefficients. The helper function resizes the scalograms and writes them to disk as jpeg images.

```
function helperCreateRGBfromTF(ECGData,parentFolder,childFolder)

imageRoot = fullfile(parentFolder,childFolder);

data = ECGData.Data;
labels = ECGData.Labels;

[~,signalLength] = size(data);

fb = cwtfilterbank('SignalLength',signalLength,'VoicesPerOctave',12);
r = size(data,1);

for ii = 1:r
    cfs = abs(fb.wt(data(ii,:)));
    im = ind2rgb(im2uint8(rescale(cfs)),jet(128));

    imgLoc = fullfile(imageRoot,char(labels(ii)));
    imFileName = strcat(char(labels(ii)),'_',num2str(ii),'.jpg');
    imwrite(imresize(im,[227 227]),fullfile(imgLoc,imFileName));
```

```
end
end
```

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `compile` | `deploy` | `predict`

More About

- "Quantization of Deep Neural Networks"

Prototype and Verify Deep Learning Networks Without Target Hardware

Rapidly prototype your custom deep learning network and bitstream by visualizing intermediate layer activation results and verifying prediction accuracy without target hardware by emulating the network and bitstream. To emulate the network and bitstream, create a `dlhdl.Simulator` object. Use the `dlhdl.Simulator` object to:

- Retrieve intermediate layer results by using the `activations` function.
- Verify prediction accuracy by using the `predict` function.

In this example, retrieve the intermediate layer activation results and verify the prediction accuracy for the ResNet-18 network and deep learning processor configuration for the `zcu102_single` bitstream.

Prerequisites

- Deep Learning Toolbox ™
- Deep Learning HDL Toolbox ™
- Deep Learning Toolbox Model for ResNet-18 Network
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- Image Processing Toolbox ™
- MATLAB® Coder™ Interface for Deep learning

Load Pretrained SeriesNetwork

To load the pretrained network ResNet-18, enter:

```
snet = resnet18;
```

To view the layers of the pretrained network, enter:

```
analyzeNetwork(snet);
```

The first layer, the image input layer, requires input images of size 224-by-224-by-3, where 3 is the number of color channels.

```
inputSize = snet.Layers(1).InputSize;
```

Define Training and Validation Data Sets

This example uses the MathWorks MerchData data set. This is a small data set containing 75 images of MathWorks merchandise, belonging to five different classes (cap, cube, playing cards, screwdriver, and torch).

```
curDir = pwd;
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
'IncludeSubfolders',true, ...
'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

Replace Final Layers

The fully connected layer and the classification layer of the pretrained network `net` are configured for 1000 classes. These two layers `fc1000` and `ClassificationLayer_predictions` in ResNet-18 contain information on how to combine the features that the network extracts into class probabilities and predicted labels. These layers must be fine-tuned for the new classification problem. Extract all the layers, except the last two layers, from the pretrained network.

```
lgraph = layerGraph(snet)
```

```
lgraph = 
  LayerGraph with properties:

     InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}
         Layers: [71×1 nnet.cnn.layer.Layer]
    Connections: [78×2 table]
```

```
numClasses = numel(categories(imdsTrain.Labels))
```

```
numClasses = 5
```

```
newLearnableLayer = fullyConnectedLayer(numClasses, ...
'Name','new_fc', ...
'WeightLearnRateFactor',10, ...
'BiasLearnRateFactor',10);
lgraph = replaceLayer(lgraph,'fc1000',newLearnableLayer);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassLayer);
```

Train Network

The network requires input images of size 224-by-224-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images, such as randomly flipping the training images along the vertical axis and randomly translating them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
'RandXReflection',true, ...
'RandXTranslation',pixelRange, ...
'RandYTranslation',pixelRange);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

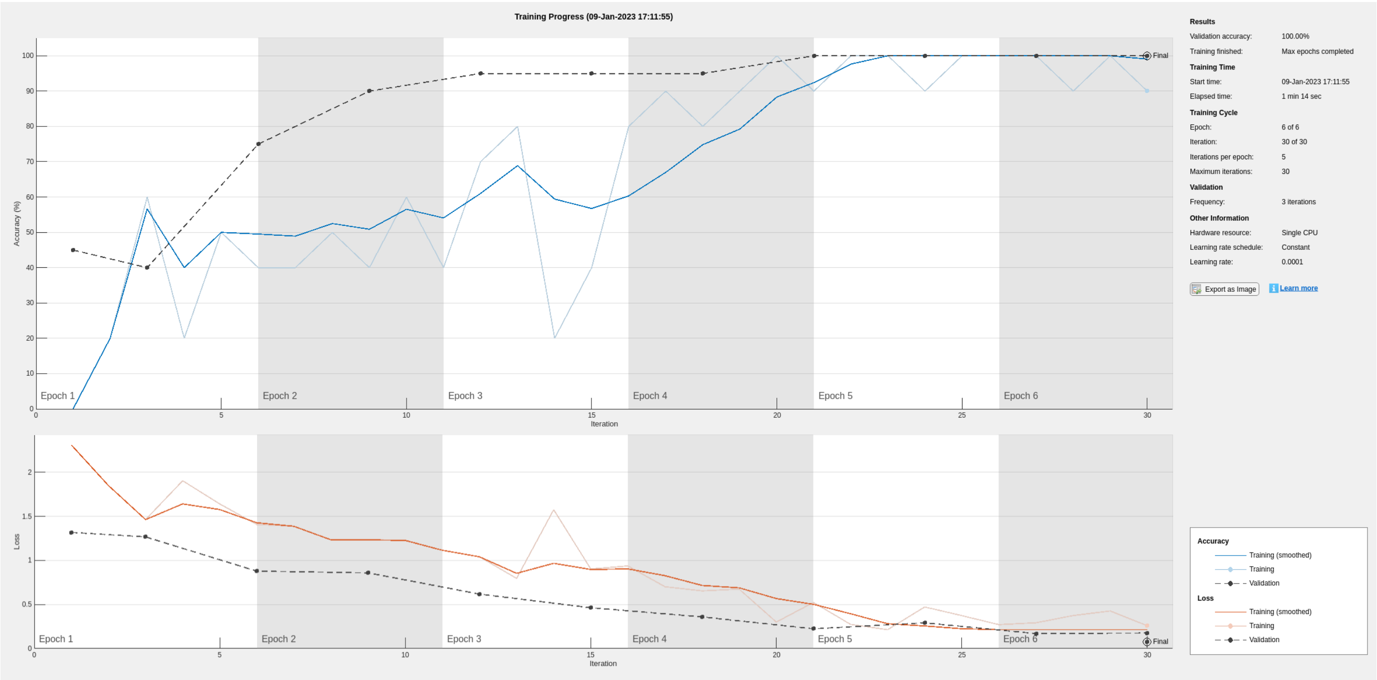
```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
'DataAugmentation',imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. Specify the mini-batch size and validation data. The software validates the network for every `ValidationFrequency` iteration during training.

```
options = trainingOptions('sgdm', ...
'MiniBatchSize',10, ...
'MaxEpochs',6, ...
'InitialLearnRate',1e-4, ...
'Shuffle','every-epoch', ...
'ValidationData',augimdsValidation, ...
'ValidationFrequency',3, ...
'Verbose',false, ...
'Plots','training-progress');
```

Train the network that consists of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a supported GPU device. See “GPU Computing Requirements” (Parallel Computing Toolbox)). Otherwise, the network uses a CPU (requires MATLAB Coder Interface for Deep learning). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value argument of `trainingOptions`.

```
netTransfer = trainNetwork(augimdsTrain,lgraph,options);
```



Retrieve Deep Learning Processor Configuration

Use the `dlhdl.ProcessorConfig` object to retrieve the deep learning processor configuration for the `zcu102_single` bitstream.

```
hPC = dlhdl.ProcessorConfig('Bitstream','zcu102_single');
```

Create Simulator Object

Create a `dlhdl.Simulator` object with ResNet-18 as the network and `hPC` as the `ProcessorConfig` object.

```
simObj = dlhdl.Simulator('Network',netTransfer,'ProcessorConfig',hPC);
```

```
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
Compiling leg: conv1>>pool1 ...
```

```
Compiling leg: conv1>>pool1 ... complete.
Compiling leg: res2a_branch2a>>res2a_branch2b ...
Compiling leg: res2a_branch2a>>res2a_branch2b ... complete.
Compiling leg: res2b_branch2a>>res2b_branch2b ...
Compiling leg: res2b_branch2a>>res2b_branch2b ... complete.
Compiling leg: res3a_branch1 ...
Compiling leg: res3a_branch1 ... complete.
Compiling leg: res3a_branch2a>>res3a_branch2b ...
Compiling leg: res3a_branch2a>>res3a_branch2b ... complete.
Compiling leg: res3b_branch2a>>res3b_branch2b ...
Compiling leg: res3b_branch2a>>res3b_branch2b ... complete.
Compiling leg: res4a_branch1 ...
Compiling leg: res4a_branch1 ... complete.
Compiling leg: res4a_branch2a>>res4a_branch2b ...
Compiling leg: res4a_branch2a>>res4a_branch2b ... complete.
Compiling leg: res4b_branch2a>>res4b_branch2b ...
Compiling leg: res4b_branch2a>>res4b_branch2b ... complete.
Compiling leg: res5a_branch1 ...
Compiling leg: res5a_branch1 ... complete.
Compiling leg: res5a_branch2a>>res5a_branch2b ...
Compiling leg: res5a_branch2a>>res5a_branch2b ... complete.
Compiling leg: res5b_branch2a>>res5b_branch2b ...
Compiling leg: res5b_branch2a>>res5b_branch2b ... complete.
Compiling leg: pool5 ...
Compiling leg: pool5 ... complete.
Compiling leg: new_fc ...
Compiling leg: new_fc ... complete.
```

Load Image for Prediction and Intermediate Layer Activation Results

Load the example image. Save it's size for future use.

```
imgFile = fullfile(pwd,'MerchData','MathWorks Cube','MathWorks cube_0.jpg');
inputImg = imresize(imread(imgFile),inputSize(1:2));
imshow(inputImg)
```

Show Activations of First Maxpool Layer

Investigate features by observing which areas in the convolution layers activate on an image. Compare that image to the corresponding areas in the original images. Each layer of a convolutional neural network consists of many 2-D arrays called *channels*. Pass the image through the network and examine the output activations of the `pool1` layer.

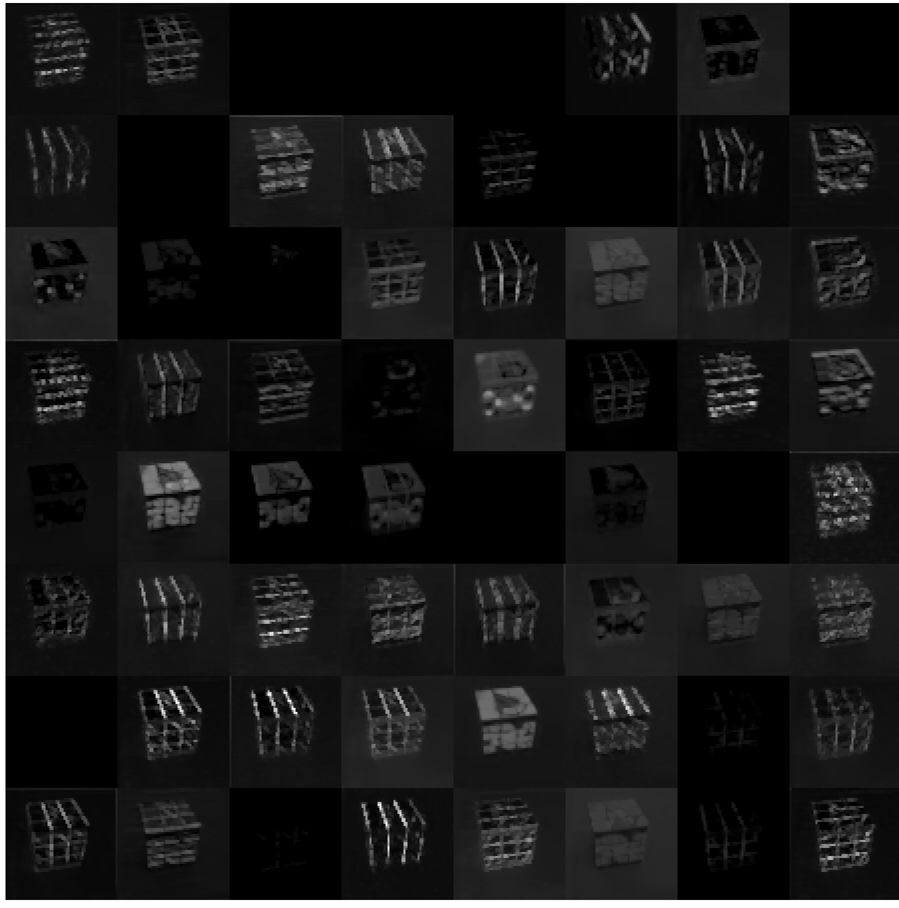
```
act1 = simObj.activations(single(inputImg),'pool1');
```

The activations are returned as a 3-D array, with the third dimension indexing the channel on the `pool1` layer. To show these activations by using the `imtile` function, reshape the array to 4-D. The third dimension in the input to `imtile` represents the image color. Set the third dimension to have size 1 because the activations do not have color. The fourth dimension indexes the channel.

```
sz = size(act1);
act1 = reshape(act1,[sz(1) sz(2) 1 sz(3)]);
```

Display the activations. Each activation can take any value, so normalize the output by using the `mat2gray`. All activations are scaled so that the minimum activation is 0 and the maximum activation is 1. Display the 64 images on an 8-by-8 grid, one for each channel in the layer.

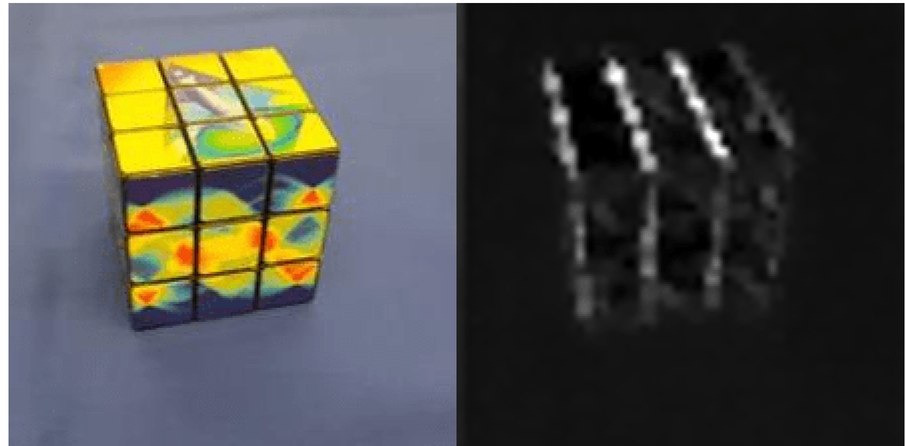
```
I = imtile(mat2gray(act1),'GridSize',[8 8]);
imshow(I)
```



Find Strongest Activation Channel

Find the strongest channels by programmatically investigating channels with large activations. Find the channel that has the largest activation by using the `max` function, resize the channel output, and display the activations.

```
[maxValue,maxValueIndex] = max(max(max(act1)));
act1chMax = act1(:,:,:,maxValueIndex);
act1chMax = mat2gray(act1chMax);
act1chMax = imresize(act1chMax,inputSize(1:2));

I = imtile({inputImg,act1chMax});
imshow(I)
```



Compare the strongest activation channel image to the original image. This channel activates on edges. It activates positively on light left/dark right edges and negatively on dark left/light right edges.

Verify Prediction Results

Verify and display the prediction results of the `dlhdl.Simulator` object by using the `predict` function.

```
prediction = simObj.predict(single(inputImg));
[val, idx] = max(prediction);
netTransfer.Layers(end).ClassNames{idx}
```

```
ans =
'MathWorks Cube'
```

See Also

`dlhdl.Simulator` | `activations` | `predict` | `dlhdl.ProcessorConfig`

Classify Images on FPGA by Using Quantized GoogLeNet Network

This example show how to use the Deep Learning HDL Toolbox™ to deploy a quantized GoogleNet network to classify an image. The example uses the pretrained GoogLeNet network to demonstrate transfer learning, quantization, and deployment for the quantized network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results.

Deploy the quantized GoogLeNet network by creating a `dlhdl.Workflow` object. Use the `dlhdl.Workflow` object to:

- Generate a list of instructions, weights and biases by using the `compile` method.
- Generate a programming file for the FPGA by using the `deploy` method.
- Retrieve the network prediction results and performance by using the `predict` method.

GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories.

Prerequisites

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model for GoogLeNet Network
- Deep Learning HDL Toolbox™ Support Package for Intel FPGA and SoC
- Image Processing Toolbox™
- Intel Arria10 SoC development kit
- Deep Learning Toolbox™ Model Quantization Library support package.
- MATLAB Coder Interface for Deep learning Libraries

Transfer Learning Using GoogLeNet

To perform classification on a new set of images, you fine-tune a pretrained GoogLeNet convolutional neural network by transfer learning. In transfer learning, you can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training images.

Load Pretrained DAG Network

Load the pretrained DAG network, GoogLeNet.

```
net = googlenet;
```

Use the `analyzeNetwork` function to obtain information about the network layers.

```
analyzeNetwork(net);
```

The first layer, the image input layer, requires input images of size 224-by-224-by-3, where 3 is the number of color channels.

```
inputSize = net.Layers(1).InputSize
```

```
inputSize = 1×3

   224   224     3
```

Define Training and Validation Data Sets

This example uses the MathWorks MerchData data set. This is a small data set containing 75 images of MathWorks merchandise, belonging to five different classes (*cap*, *cube*, *playing cards*, *screwdriver*, and *torch*).

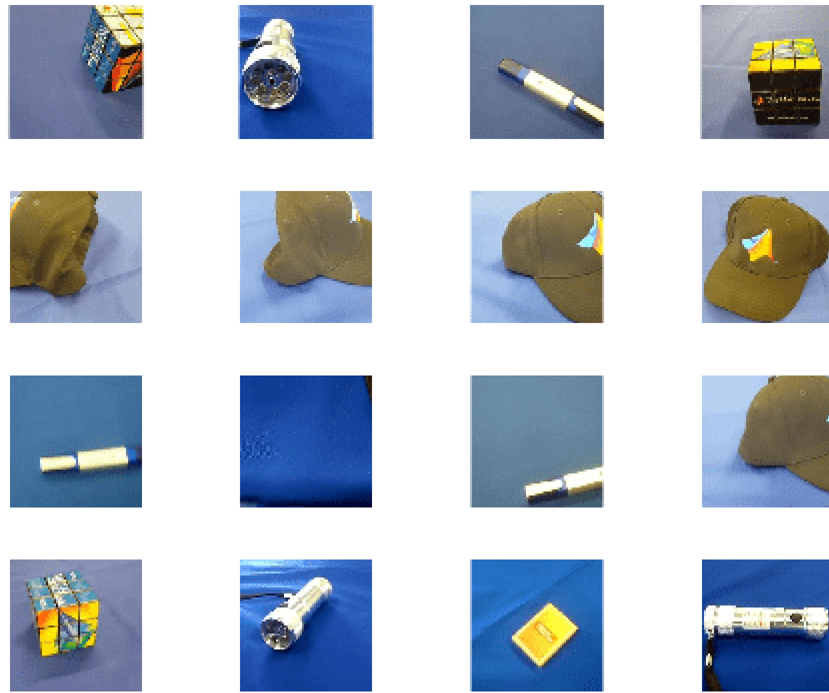
```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. `splitEachLabel` splits the `images` datastore into two new datastores.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

This data set now contains 55 training images and 20 validation images. Display some sample images.

```
numTrainImages = numel(imdsTrain.Labels);
idx = randperm(numTrainImages,16);
figure
for i = 1:16
    subplot(4,4,i)
    I = readimage(imdsTrain,idx(i));
    imshow(I)
end
```



Replace Final Layers

The fully connected layer and classification layer of the pretrained network `net` are configured for 1000 classes. These two layers, `loss3-classifier` and `output` in GoogLeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set.

Extract the layer graph from the trained network.

```
lgraph = layerGraph(net)
```

```
lgraph = 
  LayerGraph with properties:

         Layers: [144×1 nnet.cnn.layer.Layer]
    Connections: [170×2 table]
     InputNames: {'data'}
    OutputNames: {'output'}
```

Replace the fully connected layer with a new fully connected layer that has number of outputs equal to the number of classes. To make learning faster in the new layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the fully connected layer.

```
numClasses = numel(categories(imdsTrain.Labels))
```

```
numClasses = 5
```

Remove 'loss3-classifier', 'prob' and 'output' layers from the lgraph.

```
layers = net.SortedLayers;
for i = 0:2
    lgraph = removeLayers(lgraph,layers(end-i).Name);
end
```

Create three new layers and add them to the lgraph. Ensure the transferred and new layers are properly connected together in the lgraph.

```
newLayers = [
    fullyConnectedLayer(numClasses,'WeightLearnRateFactor',20,'BiasLearnRateFactor',20,'Name','ne
    softmaxLayer('Name','newProb')
    classificationLayer('Name','newClassOutput',"Classes","auto")];

lgraph = addLayers(lgraph,newLayers);
lgraph = connectLayers(lgraph,layers(end-3).Name,'newFC');
```

Train Network

The network requires input images of size 224-by-224-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from over-fitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, the learning rate factors were increased for the fully connected layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size to be 11. The software validates the network every `ValidationFrequency` iterations during training.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',11, ...
    'MaxEpochs',5, ...
    'InitialLearnRate',2e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
```

```
    'ValidationFrequency',3, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network that consists of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a supported GPU device. Otherwise, the network uses a CPU (requires MATLAB Coder Interface for Deep learning Libraries™). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value argument of `trainingOptions`.

```
netTransfer = trainNetwork(augimdsTrain,lgraph,options);
```

Create dlquantizer Object

Create a quantized network by using the `dlquantizer` object. Set the target execution environment to FPGA..

```
dlQuantObj = dlquantizer(netTransfer,'ExecutionEnvironment','FPGA');
```

Calibrate Quantized Network

Use the `calibrate` function to exercise the network by using sample inputs to collect the range information. The `calibrate` function exercises the network and collects the dynamic ranges for the learnable parameters of the convolution and fully connected layers of the network.

For best quantization results, the calibration data must be a representative of actual inputs that are predicted by the network.

```
dlQuantObj.calibrate(augimdsTrain);
```

Set Up Intel Quartus Prime Standard

Set the synthesis tool path to point to an installed Intel® Quartus® Prime Standard Edition 20.1 executable file. You must have already installed Altera® Quartus II.

```
% hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\intel\20.1\quartus\bin\quartus.e
```

Create Target Object

Create a target object with a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet.

```
hTarget = dlhdl.Target('Intel','Interface','JTAG');
```

Generate Bitstream to Run Network

The GoogleNet network consists of multiple Cross Channel Normalization layers. To support this layer on hardware, the 'LRNBlockGeneration' property of the conv module needs to be turned on in the bitstream used for FPGA inference. The shipping arria10soc_int8 bitstream does not have 'LRNBlockGeneration' property turned on. A new bitstream can be generated using the following lines of code. The generated bitstream can be used along with a workflow object for inference.

Update the processor configuration with 'LRNBlockGeneration' property turned on and 'SegmentationBlockGeneration' property turned off. Turn off 'SegmentationBlockGeneration' to fit the Deep Learning IP on the FPGA and avoid overutilization of resources.

```
% hPC = dlhdl.ProcessorConfig('Bitstream', 'arria10soc_int8');
% hPC.setModuleProperty('conv', 'LRNBlockGeneration', 'on');
```

```
% hPC.setModuleProperty('conv', 'SegmentationBlockGeneration', 'off');
% dlhdl.buildProcessor(hPC)
```

To learn how to use the generated bitstream file, see “Generate Custom Bitstream” on page 9-2.

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. Specify `dlQuantObj` as the network. Make sure to use the generated bitstream which enables processing of Cross Channel Normalization layers on FPGA. In this example, the target FPGA board is the Intel Arria10 SOC board and the generated bitstream uses the int8 data type.

```
hW = dlhdl.Workflow('network', dlQuantObj, 'Bitstream', 'dlprocessor.sof','Target',hTarget);
```

Compile Workflow Object

To compile the GoogLeNet network, run the compile function of the `dlhdl.Workflow` object.

```
dn = hW.compile
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream arria10soc_int8.
### The network includes the following layers:
     1   'data'                             Image Input                   224×224×3 images with 'ze
     2   'conv1-7x7_s2'                     Convolution                   64 7×7×3 convolutions wi
     3   'conv1-relu_7x7'                   ReLU                          ReLU
     4   'pool1-3x3_s2'                     Max Pooling                   3×3 max pooling with str
     5   'pool1-norm1'                      Cross Channel Normalization   cross channel normalizat
     6   'conv2-3x3_reduce'                 Convolution                   64 1×1×64 convolutions wi
     7   'conv2-relu_3x3_reduce'            ReLU                          ReLU
     8   'conv2-3x3'                        Convolution                   192 3×3×64 convolutions w
     9   'conv2-relu_3x3'                   ReLU                          ReLU
    10   'conv2-norm2'                      Cross Channel Normalization   cross channel normalizat
    11   'pool2-3x3_s2'                     Max Pooling                   3×3 max pooling with str
    12   'inception_3a-1x1'                 Convolution                   64 1×1×192 convolutions w
    13   'inception_3a-relu_1x1'            ReLU                          ReLU
    14   'inception_3a-3x3_reduce'          Convolution                   96 1×1×192 convolutions w
    15   'inception_3a-relu_3x3_reduce'     ReLU                          ReLU
    16   'inception_3a-3x3'                 Convolution                   128 3×3×96 convolutions w
    17   'inception_3a-relu_3x3'            ReLU                          ReLU
    18   'inception_3a-5x5_reduce'          Convolution                   16 1×1×192 convolutions w
    19   'inception_3a-relu_5x5_reduce'     ReLU                          ReLU
    20   'inception_3a-5x5'                 Convolution                   32 5×5×16 convolutions wi
    21   'inception_3a-relu_5x5'            ReLU                          ReLU
    22   'inception_3a-pool'                Max Pooling                   3×3 max pooling with str
    23   'inception_3a-pool_proj'           Convolution                   32 1×1×192 convolutions w
    24   'inception_3a-relu_pool_proj'      ReLU                          ReLU
    25   'inception_3a-output'              Depth concatenation           Depth concatenation of 4
    26   'inception_3b-1x1'                 Convolution                   128 1×1×256 convolutions
    27   'inception_3b-relu_1x1'            ReLU                          ReLU
    28   'inception_3b-3x3_reduce'          Convolution                   128 1×1×256 convolutions
    29   'inception_3b-relu_3x3_reduce'     ReLU                          ReLU
    30   'inception_3b-3x3'                 Convolution                   192 3×3×128 convolutions
    31   'inception_3b-relu_3x3'            ReLU                          ReLU
    32   'inception_3b-5x5_reduce'          Convolution                   32 1×1×256 convolutions w
    33   'inception_3b-relu_5x5_reduce'     ReLU                          ReLU
    34   'inception_3b-5x5'                 Convolution                   96 5×5×32 convolutions wi
    35   'inception_3b-relu_5x5'            ReLU                          ReLU
```

```
36   'inception_3b-pool'               Max Pooling              3×3 max pooling with stri
37   'inception_3b-pool_proj'          Convolution              64 1×1×256 convolutions w
38   'inception_3b-relu_pool_proj'     ReLU                     ReLU
39   'inception_3b-output'             Depth concatenation      Depth concatenation of 4
40   'pool3-3x3_s2'                    Max Pooling              3×3 max pooling with stri
41   'inception_4a-1x1'                Convolution              192 1×1×480 convolutions
42   'inception_4a-relu_1x1'           ReLU                     ReLU
43   'inception_4a-3x3_reduce'         Convolution              96 1×1×480 convolutions w
44   'inception_4a-relu_3x3_reduce'    ReLU                     ReLU
45   'inception_4a-3x3'                Convolution              208 3×3×96 convolutions w
46   'inception_4a-relu_3x3'           ReLU                     ReLU
47   'inception_4a-5x5_reduce'         Convolution              16 1×1×480 convolutions w
48   'inception_4a-relu_5x5_reduce'    ReLU                     ReLU
49   'inception_4a-5x5'                Convolution              48 5×5×16 convolutions wi
50   'inception_4a-relu_5x5'           ReLU                     ReLU
51   'inception_4a-pool'               Max Pooling              3×3 max pooling with stri
52   'inception_4a-pool_proj'          Convolution              64 1×1×480 convolutions w
53   'inception_4a-relu_pool_proj'     ReLU                     ReLU
54   'inception_4a-output'             Depth concatenation      Depth concatenation of 4
55   'inception_4b-1x1'                Convolution              160 1×1×512 convolutions
56   'inception_4b-relu_1x1'           ReLU                     ReLU
57   'inception_4b-3x3_reduce'         Convolution              112 1×1×512 convolutions
58   'inception_4b-relu_3x3_reduce'    ReLU                     ReLU
59   'inception_4b-3x3'                Convolution              224 3×3×112 convolutions
60   'inception_4b-relu_3x3'           ReLU                     ReLU
61   'inception_4b-5x5_reduce'         Convolution              24 1×1×512 convolutions w
62   'inception_4b-relu_5x5_reduce'    ReLU                     ReLU
63   'inception_4b-5x5'                Convolution              64 5×5×24 convolutions wi
64   'inception_4b-relu_5x5'           ReLU                     ReLU
65   'inception_4b-pool'               Max Pooling              3×3 max pooling with stri
66   'inception_4b-pool_proj'          Convolution              64 1×1×512 convolutions w
67   'inception_4b-relu_pool_proj'     ReLU                     ReLU
68   'inception_4b-output'             Depth concatenation      Depth concatenation of 4
69   'inception_4c-1x1'                Convolution              128 1×1×512 convolutions
70   'inception_4c-relu_1x1'           ReLU                     ReLU
71   'inception_4c-3x3_reduce'         Convolution              128 1×1×512 convolutions
72   'inception_4c-relu_3x3_reduce'    ReLU                     ReLU
73   'inception_4c-3x3'                Convolution              256 3×3×128 convolutions
74   'inception_4c-relu_3x3'           ReLU                     ReLU
75   'inception_4c-5x5_reduce'         Convolution              24 1×1×512 convolutions w
76   'inception_4c-relu_5x5_reduce'    ReLU                     ReLU
77   'inception_4c-5x5'                Convolution              64 5×5×24 convolutions wi
78   'inception_4c-relu_5x5'           ReLU                     ReLU
79   'inception_4c-pool'               Max Pooling              3×3 max pooling with stri
80   'inception_4c-pool_proj'          Convolution              64 1×1×512 convolutions w
81   'inception_4c-relu_pool_proj'     ReLU                     ReLU
82   'inception_4c-output'             Depth concatenation      Depth concatenation of 4
83   'inception_4d-1x1'                Convolution              112 1×1×512 convolutions
84   'inception_4d-relu_1x1'           ReLU                     ReLU
85   'inception_4d-3x3_reduce'         Convolution              144 1×1×512 convolutions
86   'inception_4d-relu_3x3_reduce'    ReLU                     ReLU
87   'inception_4d-3x3'                Convolution              288 3×3×144 convolutions
88   'inception_4d-relu_3x3'           ReLU                     ReLU
89   'inception_4d-5x5_reduce'         Convolution              32 1×1×512 convolutions w
90   'inception_4d-relu_5x5_reduce'    ReLU                     ReLU
91   'inception_4d-5x5'                Convolution              64 5×5×32 convolutions wi
92   'inception_4d-relu_5x5'           ReLU                     ReLU
93   'inception_4d-pool'               Max Pooling              3×3 max pooling with stri
```

```
    94   'inception_4d-pool_proj'         Convolution                 64 1×1×512 convolutions w
    95   'inception_4d-relu_pool_proj'    ReLU                        ReLU
    96   'inception_4d-output'            Depth concatenation         Depth concatenation of 4
    97   'inception_4e-1x1'               Convolution                 256 1×1×528 convolutions
    98   'inception_4e-relu_1x1'          ReLU                        ReLU
    99   'inception_4e-3x3_reduce'        Convolution                 160 1×1×528 convolutions
    100   'inception_4e-relu_3x3_reduce'   ReLU                        ReLU
    101   'inception_4e-3x3'               Convolution                 320 3×3×160 convolutions
    102   'inception_4e-relu_3x3'          ReLU                        ReLU
    103   'inception_4e-5x5_reduce'        Convolution                 32 1×1×528 convolutions
    104   'inception_4e-relu_5x5_reduce'   ReLU                        ReLU
    105   'inception_4e-5x5'               Convolution                 128 5×5×32 convolutions
    106   'inception_4e-relu_5x5'          ReLU                        ReLU
    107   'inception_4e-pool'              Max Pooling                 3×3 max pooling with str
    108   'inception_4e-pool_proj'         Convolution                 128 1×1×528 convolutions
    109   'inception_4e-relu_pool_proj'    ReLU                        ReLU
    110   'inception_4e-output'            Depth concatenation         Depth concatenation of 4
    111   'pool4-3x3_s2'                   Max Pooling                 3×3 max pooling with str
    112   'inception_5a-1x1'               Convolution                 256 1×1×832 convolutions
    113   'inception_5a-relu_1x1'          ReLU                        ReLU
    114   'inception_5a-3x3_reduce'        Convolution                 160 1×1×832 convolutions
    115   'inception_5a-relu_3x3_reduce'   ReLU                        ReLU
    116   'inception_5a-3x3'               Convolution                 320 3×3×160 convolutions
    117   'inception_5a-relu_3x3'          ReLU                        ReLU
    118   'inception_5a-5x5_reduce'        Convolution                 32 1×1×832 convolutions
    119   'inception_5a-relu_5x5_reduce'   ReLU                        ReLU
    120   'inception_5a-5x5'               Convolution                 128 5×5×32 convolutions
    121   'inception_5a-relu_5x5'          ReLU                        ReLU
    122   'inception_5a-pool'              Max Pooling                 3×3 max pooling with str
    123   'inception_5a-pool_proj'         Convolution                 128 1×1×832 convolutions
    124   'inception_5a-relu_pool_proj'    ReLU                        ReLU
    125   'inception_5a-output'            Depth concatenation         Depth concatenation of 4
    126   'inception_5b-1x1'               Convolution                 384 1×1×832 convolutions
    127   'inception_5b-relu_1x1'          ReLU                        ReLU
    128   'inception_5b-3x3_reduce'        Convolution                 192 1×1×832 convolutions
    129   'inception_5b-relu_3x3_reduce'   ReLU                        ReLU
    130   'inception_5b-3x3'               Convolution                 384 3×3×192 convolutions
    131   'inception_5b-relu_3x3'          ReLU                        ReLU
    132   'inception_5b-5x5_reduce'        Convolution                 48 1×1×832 convolutions
    133   'inception_5b-relu_5x5_reduce'   ReLU                        ReLU
    134   'inception_5b-5x5'               Convolution                 128 5×5×48 convolutions
    135   'inception_5b-relu_5x5'          ReLU                        ReLU
    136   'inception_5b-pool'              Max Pooling                 3×3 max pooling with str
    137   'inception_5b-pool_proj'         Convolution                 128 1×1×832 convolutions
    138   'inception_5b-relu_pool_proj'    ReLU                        ReLU
    139   'inception_5b-output'            Depth concatenation         Depth concatenation of 4
    140   'pool5-7x7_s1'                   2-D Global Average Pooling  2-D global average pooli
    141   'pool5-drop_7x7_s1'              Dropout                     40% dropout
    142   'newFC'                          Fully Connected             5 fully connected layer
    143   'newProb'                        Softmax                     softmax
    144   'newClassOutput'                 Classification Output       crossentropyex with 'Ma

### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'newClassOutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is im
### Compiling layer group: conv1-7x7_s2>>pool2-3x3_s2 ...
### Compiling layer group: conv1-7x7_s2>>pool2-3x3_s2 ... complete.
### Compiling layer group: inception_3a-1x1>>inception_3a-relu_1x1 ...
### Compiling layer group: inception_3a-1x1>>inception_3a-relu_1x1 ... complete.
```

```
### Compiling layer group: inception_3a-3x3_reduce>>inception_3a-relu_3x3 ...
### Compiling layer group: inception_3a-3x3_reduce>>inception_3a-relu_3x3 ... complete.
### Compiling layer group: inception_3a-5x5_reduce>>inception_3a-relu_5x5 ...
### Compiling layer group: inception_3a-5x5_reduce>>inception_3a-relu_5x5 ... complete.
### Compiling layer group: inception_3a-pool>>inception_3a-relu_pool_proj ...
### Compiling layer group: inception_3a-pool>>inception_3a-relu_pool_proj ... complete.
### Compiling layer group: inception_3b-1x1>>inception_3b-relu_1x1 ...
### Compiling layer group: inception_3b-1x1>>inception_3b-relu_1x1 ... complete.
### Compiling layer group: inception_3b-3x3_reduce>>inception_3b-relu_3x3 ...
### Compiling layer group: inception_3b-3x3_reduce>>inception_3b-relu_3x3 ... complete.
### Compiling layer group: inception_3b-5x5_reduce>>inception_3b-relu_5x5 ...
### Compiling layer group: inception_3b-5x5_reduce>>inception_3b-relu_5x5 ... complete.
### Compiling layer group: inception_3b-pool>>inception_3b-relu_pool_proj ...
### Compiling layer group: inception_3b-pool>>inception_3b-relu_pool_proj ... complete.
### Compiling layer group: pool3-3x3_s2 ...
### Compiling layer group: pool3-3x3_s2 ... complete.
### Compiling layer group: inception_4a-1x1>>inception_4a-relu_1x1 ...
### Compiling layer group: inception_4a-1x1>>inception_4a-relu_1x1 ... complete.
### Compiling layer group: inception_4a-3x3_reduce>>inception_4a-relu_3x3 ...
### Compiling layer group: inception_4a-3x3_reduce>>inception_4a-relu_3x3 ... complete.
### Compiling layer group: inception_4a-5x5_reduce>>inception_4a-relu_5x5 ...
### Compiling layer group: inception_4a-5x5_reduce>>inception_4a-relu_5x5 ... complete.
### Compiling layer group: inception_4a-pool>>inception_4a-relu_pool_proj ...
### Compiling layer group: inception_4a-pool>>inception_4a-relu_pool_proj ... complete.
### Compiling layer group: inception_4b-1x1>>inception_4b-relu_1x1 ...
### Compiling layer group: inception_4b-1x1>>inception_4b-relu_1x1 ... complete.
### Compiling layer group: inception_4b-3x3_reduce>>inception_4b-relu_3x3 ...
### Compiling layer group: inception_4b-3x3_reduce>>inception_4b-relu_3x3 ... complete.
### Compiling layer group: inception_4b-5x5_reduce>>inception_4b-relu_5x5 ...
### Compiling layer group: inception_4b-5x5_reduce>>inception_4b-relu_5x5 ... complete.
### Compiling layer group: inception_4b-pool>>inception_4b-relu_pool_proj ...
### Compiling layer group: inception_4b-pool>>inception_4b-relu_pool_proj ... complete.
### Compiling layer group: inception_4c-1x1>>inception_4c-relu_1x1 ...
### Compiling layer group: inception_4c-1x1>>inception_4c-relu_1x1 ... complete.
### Compiling layer group: inception_4c-3x3_reduce>>inception_4c-relu_3x3 ...
### Compiling layer group: inception_4c-3x3_reduce>>inception_4c-relu_3x3 ... complete.
### Compiling layer group: inception_4c-5x5_reduce>>inception_4c-relu_5x5 ...
### Compiling layer group: inception_4c-5x5_reduce>>inception_4c-relu_5x5 ... complete.
### Compiling layer group: inception_4c-pool>>inception_4c-relu_pool_proj ...
### Compiling layer group: inception_4c-pool>>inception_4c-relu_pool_proj ... complete.
### Compiling layer group: inception_4d-1x1>>inception_4d-relu_1x1 ...
### Compiling layer group: inception_4d-1x1>>inception_4d-relu_1x1 ... complete.
### Compiling layer group: inception_4d-3x3_reduce>>inception_4d-relu_3x3 ...
### Compiling layer group: inception_4d-3x3_reduce>>inception_4d-relu_3x3 ... complete.
### Compiling layer group: inception_4d-5x5_reduce>>inception_4d-relu_5x5 ...
### Compiling layer group: inception_4d-5x5_reduce>>inception_4d-relu_5x5 ... complete.
### Compiling layer group: inception_4d-pool>>inception_4d-relu_pool_proj ...
### Compiling layer group: inception_4d-pool>>inception_4d-relu_pool_proj ... complete.
### Compiling layer group: inception_4e-1x1>>inception_4e-relu_1x1 ...
### Compiling layer group: inception_4e-1x1>>inception_4e-relu_1x1 ... complete.
### Compiling layer group: inception_4e-3x3_reduce>>inception_4e-relu_3x3 ...
### Compiling layer group: inception_4e-3x3_reduce>>inception_4e-relu_3x3 ... complete.
### Compiling layer group: inception_4e-5x5_reduce>>inception_4e-relu_5x5 ...
### Compiling layer group: inception_4e-5x5_reduce>>inception_4e-relu_5x5 ... complete.
### Compiling layer group: inception_4e-pool>>inception_4e-relu_pool_proj ...
### Compiling layer group: inception_4e-pool>>inception_4e-relu_pool_proj ... complete.
### Compiling layer group: pool4-3x3_s2 ...
### Compiling layer group: pool4-3x3_s2 ... complete.
```

```
### Compiling layer group: inception_5a-1x1>>inception_5a-relu_1x1 ...
### Compiling layer group: inception_5a-1x1>>inception_5a-relu_1x1 ... complete.
### Compiling layer group: inception_5a-3x3_reduce>>inception_5a-relu_3x3 ...
### Compiling layer group: inception_5a-3x3_reduce>>inception_5a-relu_3x3 ... complete.
### Compiling layer group: inception_5a-5x5_reduce>>inception_5a-relu_5x5 ...
### Compiling layer group: inception_5a-5x5_reduce>>inception_5a-relu_5x5 ... complete.
### Compiling layer group: inception_5a-pool>>inception_5a-relu_pool_proj ...
### Compiling layer group: inception_5a-pool>>inception_5a-relu_pool_proj ... complete.
### Compiling layer group: inception_5b-1x1>>inception_5b-relu_1x1 ...
### Compiling layer group: inception_5b-1x1>>inception_5b-relu_1x1 ... complete.
### Compiling layer group: inception_5b-3x3_reduce>>inception_5b-relu_3x3 ...
### Compiling layer group: inception_5b-3x3_reduce>>inception_5b-relu_3x3 ... complete.
### Compiling layer group: inception_5b-5x5_reduce>>inception_5b-relu_5x5 ...
### Compiling layer group: inception_5b-5x5_reduce>>inception_5b-relu_5x5 ... complete.
### Compiling layer group: inception_5b-pool>>inception_5b-relu_pool_proj ...
### Compiling layer group: inception_5b-pool>>inception_5b-relu_pool_proj ... complete.
### Compiling layer group: pool5-7x7_s1 ...
### Compiling layer group: pool5-7x7_s1 ... complete.
### Compiling layer group: newFC ...
### Compiling layer group: newFC ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "12.0 MB"
    "OutputResultOffset"        "0x00c00000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01000000"     "4.0 MB"
    "SystemBufferOffset"        "0x01400000"     "28.0 MB"
    "InstructionDataOffset"     "0x03000000"     "8.0 MB"
    "ConvWeightDataOffset"      "0x03800000"     "32.0 MB"
    "FCWeightDataOffset"        "0x05800000"     "4.0 MB"
    "EndOffset"                 "0x05c00000"     "Total: 92.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Intel Arria10 SoC hardware, run the deploy function of the `dlhdl.Workflow` object . This function uses the output of the compile function to program the FPGA board by using the programming file. The function also downloads the network weights and biases. The deploy function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

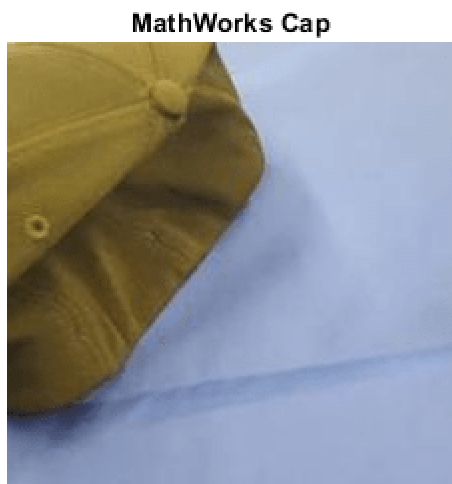
```
hW.deploy
```

```
### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 11-Jun-2021 22:20:12
```

```
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 11-Jun-2021 22:20:12
```

Load Example Image

```
I = imresize(readimage(imdsValidation,1),[224 224]);
figure
imshow(I)
```



Retrieve Image Prediction

Execute the predict function of the `dlhdl.Workflow object` and display the prediction results.

```
[prediction, speed] = hW.predict(single(I),'Profile','off');
```

```
### Finished writing input activations.
### Running single input activation.
```

```
[val, index] = max(prediction);
label = netTransfer.Layers(end).ClassNames{index}
```

```
label =
'MathWorks Cap'
```

```
title(string(label));
```

Retrieve Deployed Network Performance

View the performance of the deployed network by using the `predict` method with the `Profile` argument set to `on`.

```
[~, speed] = hW.predict(single(I),'Profile','on')
```

```
### Finished writing input activations.
### Running single input activation.
```

```
              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------            -------------              ---------        --
Network                  15836394                  0.10558                       1          158
    conv1-7x7_s2          1139964                  0.00760
    pool1-3x3_s2           268928                  0.00179
    pool1-norm1            310985                  0.00207
    conv2-3x3_reduce       278740                  0.00186
    conv2-3x3              823735                  0.00549
    conv2-norm2            952105                  0.00635
    pool2-3x3_s2           273479                  0.00182
    inception_3a-1x1       198078                  0.00132
    inception_3a-3x3_reduce    280845                  0.00187
    inception_3a-3x3       196410                  0.00131
    inception_3a-5x5_reduce     73846                  0.00049
    inception_3a-5x5        35295                  0.00024
    inception_3a-pool       94554                  0.00063
    inception_3a-pool_proj    115223                  0.00077
    inception_3b-1x1       619945                  0.00413
    inception_3b-3x3_reduce    620509                  0.00414
    inception_3b-3x3       367297                  0.00245
    inception_3b-5x5_reduce    207909                  0.00139
    inception_3b-5x5       178552                  0.00119
    inception_3b-pool      179959                  0.00120
    inception_3b-pool_proj    344959                  0.00230
    pool3-3x3_s2           293640                  0.00196
    inception_4a-1x1       332992                  0.00222
    inception_4a-3x3_reduce    181829                  0.00121
    inception_4a-3x3        83777                  0.00056
    inception_4a-5x5_reduce     55639                  0.00037
    inception_4a-5x5        14500                  0.00010
    inception_4a-pool       77187                  0.00051
    inception_4a-pool_proj    130965                  0.00087
    inception_4b-1x1       300254                  0.00200
    inception_4b-3x3_reduce    220515                  0.00147
    inception_4b-3x3       101764                  0.00068
    inception_4b-5x5_reduce     73096                  0.00049
    inception_4b-5x5        25720                  0.00017
    inception_4b-pool       82277                  0.00055
    inception_4b-pool_proj    139530                  0.00093
    inception_4c-1x1       246715                  0.00164
    inception_4c-3x3_reduce    246987                  0.00165
    inception_4c-3x3       129291                  0.00086
    inception_4c-5x5_reduce     72855                  0.00049
    inception_4c-5x5        25444                  0.00017
    inception_4c-pool       82661                  0.00055
    inception_4c-pool_proj    139761                  0.00093
    inception_4d-1x1       220154                  0.00147
    inception_4d-3x3_reduce    273136                  0.00182
    inception_4d-3x3       159811                  0.00107
    inception_4d-5x5_reduce     86719                  0.00058
    inception_4d-5x5        32485                  0.00022
    inception_4d-pool       82309                  0.00055
    inception_4d-pool_proj    139464                  0.00093
    inception_4e-1x1       474515                  0.00316
    inception_4e-3x3_reduce    309661                  0.00206
```

```
    inception_4e-3x3        193442                 0.00129
    inception_4e-5x5_reduce     88661                 0.00059
    inception_4e-5x5         62881                 0.00042
    inception_4e-pool        85098                 0.00057
    inception_4e-pool_proj    254234                 0.00169
    pool4-3x3_s2            164072                 0.00109
    inception_5a-1x1        385821                 0.00257
    inception_5a-3x3_reduce    250827                 0.00167
    inception_5a-3x3         99439                 0.00066
    inception_5a-5x5_reduce     69697                 0.00046
    inception_5a-5x5         32465                 0.00022
    inception_5a-pool        53624                 0.00036
    inception_5a-pool_proj    205084                 0.00137
    inception_5b-1x1        567107                 0.00378
    inception_5b-3x3_reduce    295819                 0.00197
    inception_5b-3x3        139308                 0.00093
    inception_5b-5x5_reduce     92415                 0.00062
    inception_5b-5x5         46311                 0.00031
    inception_5b-pool        53882                 0.00036
    inception_5b-pool_proj    205632                 0.00137
    pool5-7x7_s1             69837                 0.00047
    newFC                    23215                 0.00015
 * The clock frequency of the DL processor is: 150MHz

speed=75×5 table
                                   Latency(cycles)    Latency(seconds)    NumFrames    Total Late
                                   _______________    ________________    _________    __________

    Network                           1.5836e+07          0.10558            "1"          "1584
    ____conv1-7x7_s2                    1.14e+06        0.0075998            ""           ""
    ____pool1-3x3_s2                  2.6893e+05        0.0017929            ""           ""
    ____pool1-norm1                   3.1098e+05        0.0020732            ""           ""
    ____conv2-3x3_reduce              2.7874e+05        0.0018583            ""           ""
    ____conv2-3x3                     8.2374e+05        0.0054916            ""           ""
    ____conv2-norm2                    9.521e+05        0.0063474            ""           ""
    ____pool2-3x3_s2                  2.7348e+05        0.0018232            ""           ""
    ____inception_3a-1x1              1.9808e+05        0.0013205            ""           ""
    ____inception_3a-3x3_reduce       2.8084e+05        0.0018723            ""           ""
    ____inception_3a-3x3              1.9641e+05        0.0013094            ""           ""
    ____inception_3a-5x5_reduce            73846       0.00049231            ""           ""
    ____inception_3a-5x5                   35295        0.0002353            ""           ""
    ____inception_3a-pool                  94554       0.00063036            ""           ""
    ____inception_3a-pool_proj        1.1522e+05       0.00076815            ""           ""
    ____inception_3b-1x1              6.1994e+05         0.004133            ""           ""
        ⋮
```

The `speed` table contains the latency information for every layer, total network latency, and the overall network performance in frames per second (FPS). For more information, see “Profile Inference Run” on page 5-4.

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `compile` | `deploy` | `predict` | `dlquantizer` | `dlquantizationOptions` | `calibrate` | `validate`

More About

- "Quantization of Deep Neural Networks"

Estimate Resource Utilization for Custom Board and Reference Design

Rapidly prototype the deployment of deep learning networks to your custom board by using the `estimateResources` function. Estimate the resource utilization of the deep learning processor configuration for your custom board. Optimize the integration of custom IP cores and reference design into your system by using the `estimateResources` function to estimate the resource utilization of your reference design. The synthesis tool that you use must be in the list of tools supported by the `SynthesisTool` property of the `dlhdl.ProcessorConfig` object. For a list of supported tools and device families, see “SynthesisTool” and “SynthesisToolChipFamily”.

In this example, estimate the resource utilization for your custom board that has the Kintex® Ultrascale+™ chip family. Also estimate the resource utilization of the reference design for the Xilinx® Zynq® Ultrascale+™ MPSoC ZCU102 board.

Prerequisites

- Deep Learning HDL Toolbox™
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- HDL Coder™

Estimate Resource Utilization for Kintex® Ultrascale™ Board

To estimate the resource utilization for your custom board that has a Kintex® Ultrascale™ chip family , use the `estimateResource` function of the `dlhdl.ProcessorConfig` object.

1. Add the `dlhdl_device_registration.m` file to the MATLAB® search path.
2. Create a `dlhdl.ProcessorConfig` object.
3. Update the `SynthesisToolChipFamily` and `SynthesisToolDeviceName` properties of the `dlhdl.ProcessorConfig` object.
4. Use the `estimateResources` function to retrieve the resource utilization for your custom board.

Deep Learning HDL Toolbox™ does not support lookup table (LUT) estimation for custom boards.

```
hPC = dlhdl.ProcessorConfig;
hPC.SynthesisToolChipFamily = 'KintexU';
hPC.SynthesisToolDeviceName = 'xcku040-ffva1156-2-e';
hPC.estimateResources
```

```
Warning: Device family "KintexU" is not currently supported for LUT Estimation. Supported familie

              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    1920             600            242400
                        -------------    -------------    -------------
DL_Processor                 381( 20%)        508( 85%)           0(  0%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

Estimate Resource Utilization for Custom Reference Design

Estimate the resource utilization for a reference design that you want to integrate into your system that has a Xilinx® Zynq® Ultrascale+™ MPSoC ZCU102 board. Use the `estimateResource` function with the `IncludeReferenceDesign` name-value argument. The `estimateResources` function uses the `ResourcesUsed.LogicElements`, `ResourcesUsed.DSP`, and `ResourcesUsed.RAM` information in the reference design plugin file to perform the resource estimation. To estimate resource utilization for your custom reference design, you must populate your reference design file with values for `ResourcesUsed.LogicElements`, `ResourcesUsed.DSP`, and `ResourcesUsed.RAM`. See "ResourcesUsed". The reference design used in this code is located at `$supportpackageinstallationfolder/Xilinx/boards/+DLZCU102/+matlab_libiio_3axi4_master_2019_1/plugin_rd.m`.

```
hPC_referencedesign = dlhdl.ProcessorConfig;
hPC_referencedesign.estimateResources('IncludeReferenceDesign',true)
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912              274080
                        -------------    -------------    -------------
Total                    384( 16%)        586( 65%)       251119( 92%)
ReferenceDesign            3(  1%)         78(  9%)        35000( 13%)
DL_Processor             381( 16%)        508( 56%)       216119( 79%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

The `estimateResources` function returns the resource utilization for the reference design and for the deep learning processor configuration.

Supporting Files

Device Registration File

Use the `dlhdl_device_registration.m` file to register a custom device family. Estimate the resource utilization of the custom device by using the `estimateResources` function.

```
type dlhdl_device_registration.m
```

```
function hFPGADeviceFamily = dlhdl_device_registration
% Register a new device family by providing the following details:
% 1. Device Family Name
% 2. Vendor(Intel/Xilinx)
% 3. DSP Width
% 4. RAM Width
% 5. RAM Depth
% 6. SplitDSP Width(Optional) - alternative DSP Width supported by the DSP macro
% 7. SplitRAM Width(Optional) - alternative RAM Width supported by the RAM macro

hFPGADeviceFamily = { ...
    kintex_ultrascale();...
    };
end

function hFPGADeviceFamily = kintex_ultrascale()
    % Datasheets :
    % https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf
```

```
    % https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources
    hFPGADeviceFamily = hdlcoder.FPGADeviceInfo('Name', 'KintexU');
    hFPGADeviceFamily.Vendor = 'Xilinx';
    hFPGADeviceFamily.DSPWidth = [27, 18];
    hFPGADeviceFamily.RAMWidth = 36;
    hFPGADeviceFamily.SplitRAMWidth = 18;
    hFPGADeviceFamily.RAMDepth = 1024;
end
```

See Also

`dlhdl.ProcessorConfig` | `estimatePerformance` | `estimateResources`

More About

- "Estimate Resource Utilization for Custom Processor Configuration" on page 8-10
- "Estimate Performance of Deep Learning Network" on page 8-3

Speech Command Recognition by Using FPGA

This example shows how to deploy a custom pretrained series network that detects the presence of speech commands in audio to a Xilinx™ Zynq® UltraScale+™ MPSoC ZCU102 Evaluation Kit. This example uses the pretrained network that was trained by using the Speech Commands Dataset [1]. To create the pretrained network, see "Train Speech Command Recognition Model Using Deep Learning".

Prerequisites

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Audio Toolbox™
- Xilinx™ Zynq® UltraScale+™ MPSoC ZCU102 Evaluation Kit

Load Speech Commands Data Set

This example uses the Google Speech Commands Dataset [1]. Download the dataset and untar the downloaded file. Set `PathToDatabase` to the location of the data.

```
url = 'https://ssd.mathworks.com/supportfiles/audio/google_speech.zip';
downloadFolder = tempdir;
dataFolder = fullfile(downloadFolder,'google_speech');

if ~exist(dataFolder,'dir')
    disp('Downloading data set (1.4 GB) ...')
    unzip(url,downloadFolder)
end
```

Load Pretrained Speech Recognition Network

The pretrained network `trainedAudioNet` is a simple series network made up of 24 layers. The network uses max pooling layers to downsample the feature maps "spatially" (that is, in time and frequency) and a final max pooling layer that pools the input feature map globally over time. This enforces (approximate) time-translation invariance in the input spectrograms, allowing the network to perform the same classification independent of the exact position of the speech in time. Global pooling also significantly reduces the number of parameters in the final fully connected layer. To reduce the possibility of the network memorizing specific features of the training data, add a small amount of dropout to the input to the last fully connected layer.

The network is small, as it has only five convolutional layers with few filters. `numF` controls the number of filters in the convolutional layers. To increase the accuracy of the network, try increasing the network depth by adding identical blocks of convolutional, batch normalization, and ReLU layers. You can also try increasing the number of convolutional filters by increasing `numF`.

Use a weighted cross entropy classification loss. The `weightedClassificationLayer` function creates a custom classification layer that calculates the cross entropy loss with observations weighted by `classWeights`. Specify the class weights in the same order as the classes appear in `categories(YTrain)`. To give each class equal total weight in the loss, use class weights that are inversely proportional to the number of training examples in each class. When using the Adam optimizer to train the network, the training algorithm is independent of the overall normalization of the class weights. Load the pretrained network `trainedAudioNet`.

```
load('trainedAudioNet.mat');
```

Create Training and Validation Datastore

Create an `audioDataStore` that points to the training and validation data sets. See `audioDatastore` (Audio Toolbox).

```
ads = audioDatastore(fullfile(dataFolder, 'train'), ...
    'IncludeSubfolders',true, ...
    'FileExtensions','.wav', ...
    'LabelSource','foldernames');
```

Specify the words that you want your model to recognize as commands. Label words that are not commands as `unknown`. Labeling words that are not commands as `unknown` creates a group of words that approximates the distribution of all words other than the commands. The network uses this group to learn the difference between commands and all other words.

To reduce the class imbalance between the known and unknown words and speed up processing, include only a fraction of the unknown words in the training set.

To create a datastore that contains only the commands and the subset of unknown words, Use `subset` (Audio Toolbox) (Audio Toolbox). Count the number of examples belonging to each category.

```
commands = categorical(["yes","no","up","down","left","right","on","off","stop","go"]);

isCommand = ismember(ads.Labels,commands);
isUnknown = ~isCommand;

includeFraction = 0.2;
mask = rand(numel(ads.Labels),1) < includeFraction;
isUnknown = isUnknown & mask;
ads.Labels(isUnknown) = categorical("unknown");

adsTrain = subset(ads,isCommand|isUnknown);
countEachLabel(adsTrain)
```

```
ans=11×2 table
     Label      Count
    _______     _____

    down        1842
    go          1861
    left        1839
    no          1853
    off         1839
    on          1864
    right       1852
    stop        1885
    unknown     4390
    up          1843
    yes         1860
```

```
ads = audioDatastore(fullfile(dataFolder, 'validation'), ...
    'IncludeSubfolders',true, ...
    'FileExtensions','.wav', ...
    'LabelSource','foldernames');
```

```
isCommand = ismember(ads.Labels,commands);
isUnknown = ~isCommand;

includeFraction = 0.2;
mask = rand(numel(ads.Labels),1) < includeFraction;
isUnknown = isUnknown & mask;
ads.Labels(isUnknown) = categorical("unknown");

adsValidation = subset(ads,isCommand|isUnknown);
countEachLabel(adsValidation)
```

```
ans=11×2 table
     Label     Count
    _______    _____

    down        264
    go          260
    left        247
    no          270
    off         256
    on          257
    right       256
    stop        246
    unknown     618
    up          260
    yes         261
```

To train the network with the entire dataset and achieve the highest possible accuracy, set `reduceDataset` to `false`. To run this example quickly, set `reduceDataset` to `true`.

```
reduceDataset = false;
if reduceDataset
    numUniqueLabels = numel(unique(adsTrain.Labels));
    % Reduce the dataset by a factor of 20
    adsTrain = splitEachLabel(adsTrain,round(numel(adsTrain.Files) / numUniqueLabels / 20));
    adsValidation = splitEachLabel(adsValidation,round(numel(adsValidation.Files) / numUniqueLabe
end
```

Compute Auditory Spectograms

To prepare the data for efficient training of a convolutional neural network, convert the speech waveforms to auditory-based spectrograms.

Define the parameters of the feature extraction. The `segmentDuration` variable is the duration of each speech clip (in seconds). The `frameDuration` variable is the duration of each frame for spectrum calculation. The `hopDuration` variable is the time step between each spectrum. `numBands` is the number of filters in the auditory spectrogram.

To perform the feature extraction, create an `audioFeatureExtractor` (Audio Toolbox) (Audio Toolbox) object.

```
fs = 16e3; % Known sample rate of the data set.

segmentDuration = 1;
frameDuration = 0.025;
hopDuration = 0.010;
```

```
segmentSamples = round(segmentDuration*fs);
frameSamples = round(frameDuration*fs);
hopSamples = round(hopDuration*fs);
overlapSamples = frameSamples - hopSamples;

FFTLength = 512;
numBands = 50;

afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'FFTLength',FFTLength, ...
    'Window',hann(frameSamples,'periodic'), ...
    'OverlapLength',overlapSamples, ...
    'barkSpectrum',true);
setExtractorParams(afe,'barkSpectrum','NumBands',numBands,'WindowNormalization',false);
```

Read a file from the dataset. Training a convolutional neural network requires input to be a consistent size. Some files in the data set are less than 1 second long. Apply zero-padding to the front and back of the audio signal so that it is of length `segmentSamples`.

```
x = read(adsTrain);

numSamples = size(x,1);

numToPadFront = floor( (segmentSamples - numSamples)/2 );
numToPadBack = ceil( (segmentSamples - numSamples)/2 );

xPadded = [zeros(numToPadFront,1,'like',x);x;zeros(numToPadBack,1,'like',x)];
```

To extract audio features, call `extract`. The output is a Bark spectrum with time across rows.

```
features = extract(afe,xPadded);
[numHops,numFeatures] = size(features)
```

```
numHops = 98

numFeatures = 50
```

In this example, you post-process the auditory spectrogram by applying a logarithm. Taking a log of small numbers can lead to roundoff error.

To speed up processing, you can distribute the feature extraction across multiple workers by using `parfor`.

First, determine the number of partitions for the dataset. If you do not have Parallel Computing Toolbox™, use a single partition.

```
if ~isempty(ver('parallel')) && ~reduceDataset
    pool = gcp;
    numPar = numpartitions(adsTrain,pool);
else
    numPar = 1;
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

For each partition, read from the datastore, zero-pad the signal, and then extract the features.

```
parfor ii = 1:numPar
    subds = partition(adsTrain,numPar,ii);
    XTrain = zeros(numHops,numBands,1,numel(subds.Files));
    for idx = 1:numel(subds.Files)
        x = read(subds);
        xPadded = [zeros(floor((segmentSamples-size(x,1))/2),1);x;zeros(ceil((segmentSamples-size
        XTrain(:,:,:,idx) = extract(afe,xPadded);
    end
    XTrainC{ii} = XTrain;
end
```

Convert the output to a four-dimensional array that has auditory spectrograms along the fourth dimension.

```
XTrain = cat(4,XTrainC{:});

[numHops,numBands,numChannels,numSpec] = size(XTrain)
```

```
numHops = 98

numBands = 50

numChannels = 1

numSpec = 22928
```

To obtain data that has a smoother distribution, take the logarithm of the spectrograms by using a small offset.

```
epsil = 1e-6;
XTrain = log10(XTrain + epsil);
```

Perform the feature extraction steps described above for the validation set.

```
if ~isempty(ver('parallel'))
    pool = gcp;
    numPar = numpartitions(adsValidation,pool);
else
    numPar = 1;
end
parfor ii = 1:numPar
    subds = partition(adsValidation,numPar,ii);
    XValidation = zeros(numHops,numBands,1,numel(subds.Files));
    for idx = 1:numel(subds.Files)
        x = read(subds);
        xPadded = [zeros(floor((segmentSamples-size(x,1))/2),1);x;zeros(ceil((segmentSamples-size
        XValidation(:,:,:,idx) = extract(afe,xPadded);
    end
    XValidationC{ii} = XValidation;
end
XValidation = cat(4,XValidationC{:});
XValidation = log10(XValidation + epsil);
```

Isolate the train and validation labels. Remove empty categories.

```
YTrain = removecats(adsTrain.Labels);
YValidation = removecats(adsValidation.Labels);
```

Add Background Noise Data

The network must be able to recognize different spoken words and also to detect if the input contains silence or background noise.

To create samples of one-second clips of background noise, use the audio files in the `_background_` folder. Create an equal number of background clips from each background noise file. You can also create your own recordings of background noise and add them to the `_background_` folder. Before calculating the spectrograms, the function rescales each audio clip by using a factor sampled from a log-uniform distribution in the range provided by `volumeRange`.

```
adsBkg = audioDatastore(fullfile(dataFolder, 'background'));
numBkgClips = 4000;
if reduceDataset
    numBkgClips = numBkgClips/20;
end
volumeRange = log10([1e-4,1]);

numBkgFiles = numel(adsBkg.Files);
numClipsPerFile = histcounts(1:numBkgClips,linspace(1,numBkgClips,numBkgFiles+1));
Xbkg = zeros(size(XTrain,1),size(XTrain,2),1,numBkgClips,'single');
bkgAll = readall(adsBkg);
ind = 1;

for count = 1:numBkgFiles
    bkg = bkgAll{count};
    idxStart = randi(numel(bkg)-fs,numClipsPerFile(count),1);
    idxEnd = idxStart+fs-1;
    gain = 10.^((volumeRange(2)-volumeRange(1))*rand(numClipsPerFile(count),1) + volumeRange(1))
    for j = 1:numClipsPerFile(count)

        x = bkg(idxStart(j):idxEnd(j))*gain(j);

        x = max(min(x,1),-1);

        Xbkg(:,:,:,ind) = extract(afe,x);

        if mod(ind,1000)==0
            disp("Processed " + string(ind) + " background clips out of " + string(numBkgClips))
        end
        ind = ind + 1;
    end
end
```

```
Processed 1000 background clips out of 4000
Processed 2000 background clips out of 4000
Processed 3000 background clips out of 4000
Processed 4000 background clips out of 4000
```

```
Xbkg = log10(Xbkg + epsil);
```

Split the spectrograms of background noise among the training, validation, and test sets. Because the `_background_noise_` folder contains only about five and a half minutes of background noise, the background samples in the different data sets are highly correlated. To increase the variation in the background noise, you can create your own background files and add them to the folder. To increase the robustness of the network to noise, you can also try mixing background noise into the speech files.

```
numTrainBkg = floor(0.85*numBkgClips);
numValidationBkg = floor(0.15*numBkgClips);

XTrain(:,:,:,end+1:end+numTrainBkg) = Xbkg(:,:,:,1:numTrainBkg);
YTrain(end+1:end+numTrainBkg) = "background";

XValidation(:,:,:,end+1:end+numValidationBkg) = Xbkg(:,:,:,numTrainBkg+1:end);
YValidation(end+1:end+numValidationBkg) = "background";
```

Create Target Object

Create a target object for your target device that has a vendor name and an interface to connect your target device to the host computer. Interface options are JTAG (default) and Ethernet. Vendor options are Intel or Xilinx. Use the installed Xilinx Vivado Design Suite over an Ethernet connection to program the device.

```
hT = dlhdl.Target('Xilinx', Interface = 'Ethernet');
```

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pretrained series network `trainedAudioNet` as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Zynq UltraScale+ MPSoC ZCU102 board. The bitstream uses a single data type.

```
hW = dlhdl.Workflow(Network = trainedNet, Bitstream = 'zcu102_single', Target = hT);
```

Compile `trainedAudioNet` Network

To compile the `trainedAudioNet` series network, run the compile function of the `dlhdl.Workflow` object.

```
compile(hW)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### The network includes the following layers:
     1   'imageinput'    Image Input             98×50×1 images with 'zerocenter' normalization
     2   'conv_1'        Convolution             12 3×3×1 convolutions with stride [1  1] and pad
     3   'batchnorm_1'   Batch Normalization     Batch normalization with 12 channels
     4   'relu_1'        ReLU                    ReLU
     5   'maxpool_1'     Max Pooling             3×3 max pooling with stride [2  2] and padding
     6   'conv_2'        Convolution             24 3×3×12 convolutions with stride [1  1] and pa
     7   'batchnorm_2'   Batch Normalization     Batch normalization with 24 channels
     8   'relu_2'        ReLU                    ReLU
     9   'maxpool_2'     Max Pooling             3×3 max pooling with stride [2  2] and padding
    10   'conv_3'        Convolution             48 3×3×24 convolutions with stride [1  1] and pa
    11   'batchnorm_3'   Batch Normalization     Batch normalization with 48 channels
    12   'relu_3'        ReLU                    ReLU
    13   'maxpool_3'     Max Pooling             3×3 max pooling with stride [2  2] and padding
    14   'conv_4'        Convolution             48 3×3×48 convolutions with stride [1  1] and pa
    15   'batchnorm_4'   Batch Normalization     Batch normalization with 48 channels
    16   'relu_4'        ReLU                    ReLU
    17   'conv_5'        Convolution             48 3×3×48 convolutions with stride [1  1] and pa
    18   'batchnorm_5'   Batch Normalization     Batch normalization with 48 channels
    19   'relu_5'        ReLU                    ReLU
    20   'maxpool_4'     Max Pooling             13×1 max pooling with stride [1  1] and padding
```

```
    21   'dropout'       Dropout                  20% dropout
    22   'fc'            Fully Connected          12 fully connected layer
    23   'softmax'       Softmax                  softmax
    24   'classoutput'   Classification Output    Weighted cross entropy

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'imageinput' of type 'ImageInputLayer' is split into 'imageinput' and 'imag
### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'weightedClassificationLayer' is implemented in sof
### Compiling layer group: conv_1>>relu_5 ...
### Compiling layer group: conv_1>>relu_5 ... complete.
### Compiling layer group: maxpool_4 ...
### Compiling layer group: maxpool_4 ... complete.
### Compiling layer group: fc ...
### Compiling layer group: fc ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "28.0 MB"
    "InstructionDataOffset"     "0x02800000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02c00000"     "4.0 MB"
    "FCWeightDataOffset"        "0x03000000"     "4.0 MB"
    "EndOffset"                 "0x03400000"     "Total: 52.0 MB"

### Network compilation complete.

ans = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {{}  [4.4809 0 0 0 4.4809 0 0 0 4.4809 0 0 0 4.4809 0 0 0 4.4809 0 0 0 4.4
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Zynq® UltraScale+™ MPSoC ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file.The function also downloads the network weights and biases. The deploy function verifies the Xilinx Vivado tool and the supported tool version. It then starts programming the FPGA device by using the bitstream, displays progress messages, and the time it takes to deploy the network.

```
deploy(hW)
```

```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
```

```
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'

Downloading target FPGA device configuration over Ethernet to SD card done. The system will now r


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 11-Nov-2021 15:15:18
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 11-Nov-2021 15:15:18
```

Run Prediction on Audio Files

Classify five inputs from the validation data set and compare the prediction results to the classification results from the Deep Learning Toolbox™. `YPred` is the classification result from the Deep learning Toolbox™. The `fpga_prediction` variable is the classification result from the FPGA.

```
numtestFrames = size(XValidation,4);
numView = 5;
listIndex = randperm(numtestFrames,numView);
testDataBatch = XValidation(:,:,:,listIndex);
YPred = classify(trainedNet,testDataBatch);
[scores,speed] = predict(hW,testDataBatch, Profile ='on');
```

```
### Finished writing input activations.
### Running in multi-frame mode with 5 inputs.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    353130                  0.00161                       5           15
    imageinput_norm         52668                  0.00024
    conv_1                  21136                  0.00010
    maxpool_1               47686                  0.00022
    conv_2                  37475                  0.00017
    maxpool_2               45278                  0.00021
    conv_3                  21260                  0.00010
    maxpool_3               38857                  0.00018
    conv_4                  16171                  0.00007
    conv_5                  27011                  0.00012
    maxpool_4               27632                  0.00013
    fc                      17923                  0.00008
 * The clock frequency of the DL processor is: 220MHz
```

```
[~,idx] = max(scores,[],2);
fpga_prediction = trainedNet.Layers(end).Classes(idx);
```

Compare the prediction results from Deep Learning Toolbox™ and the FPGA side by side. The prediction results from the FPGA match the prediction results from Deep Learning Toolbox™. In this table, the ground truth prediction is the Deep Learning Toolbox™ prediction.

```
fprintf('%12s %24s\n','Ground Truth','FPGA Prediction');for i= 1:size(fpga_prediction,1)
    fprintf('%s %24s\n',YPred(i),fpga_prediction(i)); end
```

```
Ground Truth          FPGA Prediction
```

```
no                          no
unknown                     unknown
yes                         yes
no                          no
yes                         yes
```

References

[1] Warden P. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available at http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed under the Creative Commons Attribution 4.0 license, available here: https://creativecommons.org/licenses/by/4.0/legalcode.

MathWorks, Inc.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `classify`

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2

Modulation Classification by Using FPGA

This example shows how to deploy a pretrained convolutional neural network (CNN) for modulation classification to the Xilinx™ Zynq® UltraScale+™ MPSoC ZCU102 Evaluation Kit. The pretrained network is trained by using generated synthetic, channel-impaired waveforms. To train the `trainedNet` network, see “Modulation Classification with Deep Learning”.

Prerequisites

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Communications Toolbox™
- Xilinx™ Zynq® UltraScale+™ MPSoC ZCU102 Evaluation Kit

Predict Modulation Type by Using CNN

The trained CNN in this example recognizes these eight digital and three analog modulation types:

- Binary phase shift keying (BPSK)
- Quadrature phase shift keying (QPSK)
- 8-ary phase shift keying (8-PSK)
- 16-ary quadrature amplitude modulation (16-QAM)
- 64-ary quadrature amplitude modulation (64-QAM)
- 4-ary pulse amplitude modulation (PAM4)
- Gaussian frequency shift keying (GFSK)
- Continuous phase frequency shift keying (CPFSK)
- Broadcast FM (B-FM)
- Double sideband amplitude modulation (DSB-AM)
- Single sideband amplitude modulation (SSB-AM)

```
modulationTypes = categorical(["BPSK", "QPSK", "8PSK", ...
  "16QAM", "64QAM", "PAM4", "GFSK", "CPFSK", ...
  "B-FM", "DSB-AM", "SSB-AM"]);
```

Load the trained network.

```
load trainedModulationClassificationNetwork
trainedNet
```

```
trainedNet = 
  SeriesNetwork with properties:

         Layers: [28×1 nnet.cnn.layer.Layer]
     InputNames: {'Input Layer'}
    OutputNames: {'Output'}
```

The trained CNN takes 1024 channel-impaired samples and predicts the modulation type of each frame. Generate several PAM4 frames that have Rician multipath fading, center frequency and

sampling time drift, and AWGN. To generate synthetic signals to test the CNN, use the following functions. Then use the CNN to predict the modulation type of the frames.

- `randi`: Generate random bits
- `pammod` (Communications Toolbox) (Communications Toolbox) PAM4-modulate the bits
- `rcosdesign` (Signal Processing Toolbox) (Signal Processing Toolbox): Design a square-root raised cosine pulse shaping filter
- `filter`: Pulse shape the symbols
- `comm.RicianChannel` (Communications Toolbox) (Communications Toolbox): Apply Rician multipath channel
- `comm.PhaseFrequencyOffset` (Communications Toolbox) (Communications Toolbox): Apply phase and frequency shift due to clock offset
- `interp1`: Apply timing drift due to clock offset
- `awgn` (Communications Toolbox) (Communications Toolbox): Add AWGN

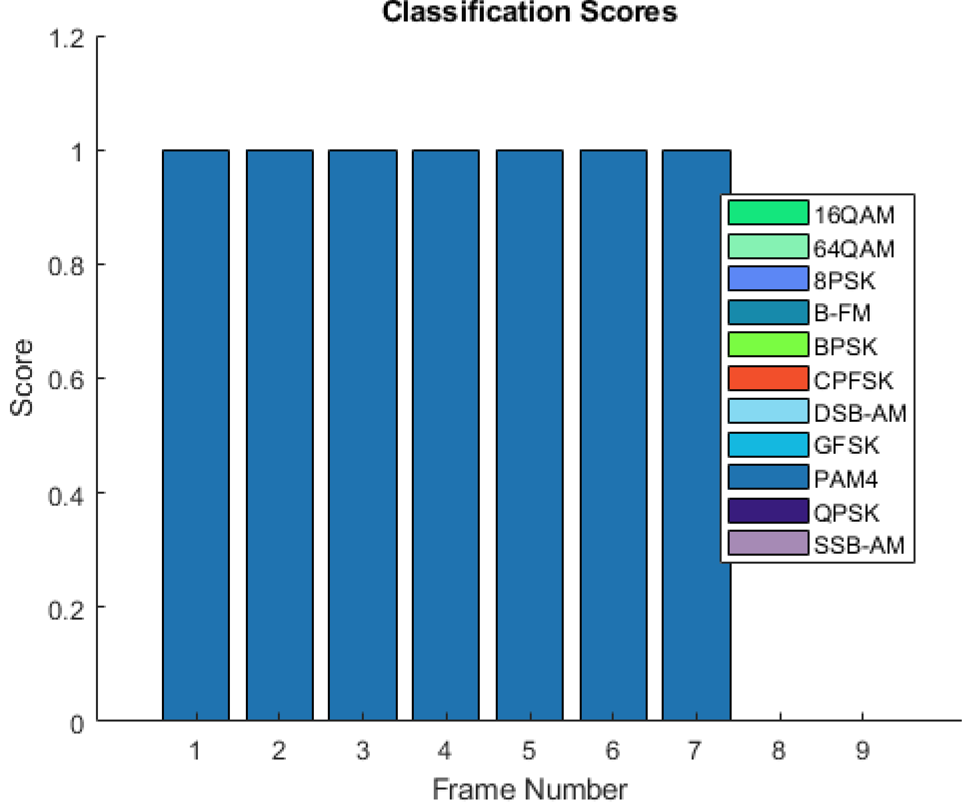
```
% Set the random number generator to a known state to be able to regenerate
% the same frames every time the simulation is run
rng(123456)
% Random bits
d = randi([0 3], 1024, 1);
% PAM4 modulation
syms = pammod(d,4);
% Square-root raised cosine filter
filterCoeffs = rcosdesign(0.35,4,8);
tx = filter(filterCoeffs,1,upsample(syms,8));

% Channel
SNR = 30;
maxOffset = 5;
fc = 902e6;
fs = 200e3;
multipathChannel = comm.RicianChannel(...
    'SampleRate', fs, ...
    'PathDelays', [0 1.8 3.4] / 200e3, ...
    'AveragePathGains', [0 -2 -10], ...
    'KFactor', 4, ...
    'MaximumDopplerShift', 4);

frequencyShifter = comm.PhaseFrequencyOffset(...
    'SampleRate', fs);

% Apply an independent multipath channel
reset(multipathChannel)
outMultipathChan = multipathChannel(tx);

% Determine clock offset factor
clockOffset = (rand() * 2*maxOffset) - maxOffset;
C = 1 + clockOffset / 1e6;

% Add frequency offset
frequencyShifter.FrequencyOffset = -(C-1)*fc;
outFreqShifter = frequencyShifter(outMultipathChan);

% Add sampling time drift
```

```
t = (0:length(tx)-1)' / fs;
newFs = fs * C;
tp = (0:length(tx)-1)' / newFs;
outTimeDrift = interp1(t, outFreqShifter, tp);

% Add noise
rx = awgn(outTimeDrift,SNR,0);

% Frame generation for classification
unknownFrames = helperModClassGetNNFrames(rx);

% Classification
[prediction1,score1] = classify(trainedNet,unknownFrames);
```

Return the classifier predictions, which are analogous to hard decisions. The network correctly identifies the frames as PAM4 frames. For details on the generation of the modulated signals, see the helperModClassGetModulator function.

The classifier also returns a vector of scores for each frame. The score corresponds to the probability that each frame has the predicted modulation type. Plot the scores.

```
helperModClassPlotScores(score1,modulationTypes)
```



Waveform Generation for Training

Generate 10,000 frames for each modulation type, where 80% of the frames are used for training, 10% are used for validation and 10% are used for testing. Use the training and validation frames

during the network training phase. You obtain the final classification accuracy by using test frames. Each frame is 1024 samples long and has a sample rate of 200 kHz. For digital modulation types, eight samples represent a symbol. The network makes each decision based on single frames rather than on multiple consecutive frames (as in video). Assume a center frequency of 902 MHz and 100 MHz for the digital and analog modulation types, respectively.

```
numFramesPerModType = 10000;
percentTrainingSamples = 80;
percentValidationSamples = 10;
percentTestSamples = 10;

sps = 8;                % Samples per symbol
spf = 1024;             % Samples per frame
symbolsPerFrame = spf / sps;
fs = 200e3;             % Sample rate
fc = [902e6 100e6];     % Center frequencies
```

Create Channel Impairments

Pass each frame through a channel by using:

- AWGN
- Rician multipath fading
- Clock offset, resulting in center frequency offset and sampling time drift

Because the network in this example makes decisions based on single frames, each frame must pass through an independent channel **AWGN**.

The channel adds AWGN by using an SNR of 30 dB. Implement the channel by using the `awgn` (Communications Toolbox) (Communications Toolbox) function.

Rician Multipath

The channel passes the signals through a Rician multipath fading channel by using the `comm.RicianChannel` (Communications Toolbox) (Communications Toolbox) System object. Assume a delay profile of [0 1.8 3.4] samples that have corresponding average path gains of [0 -2 -10] dB. The K-factor is 4 and the maximum Doppler shift is 4 Hz, which is equivalent to a walking speed at 902 MHz. Implement the channel by using the following settings.

Clock Offset

Clock offset occurs because of the inaccuracies of internal clock sources of transmitters and receivers. Clock offset causes the center frequency, which is used to downconvert the signal to baseband, and the digital-to-analog converter sampling rate to differ from theoretical values. The channel simulator uses the clock offset factor C, expressed as C=1+Δclock106, where Δclock is the clock offset. For each frame, the channel generates a random Δclock value from a uniformly distributed set of values in the range [−maxΔclock maxΔclock], where maxΔclock is the maximum clock offset. Clock offset is measured in parts per million (ppm). For this example, assume a maximum clock offset of 5 ppm.

```
maxDeltaOff = 5;
deltaOff = (rand()*2*maxDeltaOff) - maxDeltaOff;
C = 1 + (deltaOff/1e6);
```

Frequency Offset

Subject each frame to a frequency offset based on clock offset factor C and the center frequency. Implement the channel by using the `comm.PhaseFrequencyOffset` (Communications Toolbox) (Communications Toolbox).

Sampling Rate Offset

Subject each frame to a sampling rate offset based on clock offset factor C. Implement the channel by using the `interp1` function to resample the frame at the new rate of $C \times fs$.

Combined Channel

To apply all three channel impairments to the frames, use the helperModClassTestChannel object.

```
channel = helperModClassTestChannel(...
  'SampleRate', fs, ...
  'SNR', SNR, ...
  'PathDelays', [0 1.8 3.4] / fs, ...
  'AveragePathGains', [0 -2 -10], ...
  'KFactor', 4, ...
  'MaximumDopplerShift', 4, ...
  'MaximumClockOffset', 5, ...
  'CenterFrequency', 902e6)
```

```
channel = 
  helperModClassTestChannel with properties:

                    SNR: 30
        CenterFrequency: 902000000
             SampleRate: 200000
             PathDelays: [0 9.0000e-06 1.7000e-05]
       AveragePathGains: [0 -2 -10]
                KFactor: 4
    MaximumDopplerShift: 4
     MaximumClockOffset: 5
```

You can view basic information about the channel by using the info object function.

```
chInfo = info(channel)
```

```
chInfo = struct with fields:
               ChannelDelay: 6
     MaximumFrequencyOffset: 4510
    MaximumSampleRateOffset: 1
```

Waveform Generation

Create a loop that generates channel-impaired frames for each modulation type and stores the frames with their corresponding labels in MAT files. By saving the data into files, you do not have to eliminate the need to generate the data every time you run this example. You can also share the data more effectively.

Remove a random number of samples from the beginning of each frame to remove transients and to make sure that the frames have a random starting point with respect to the symbol boundaries.

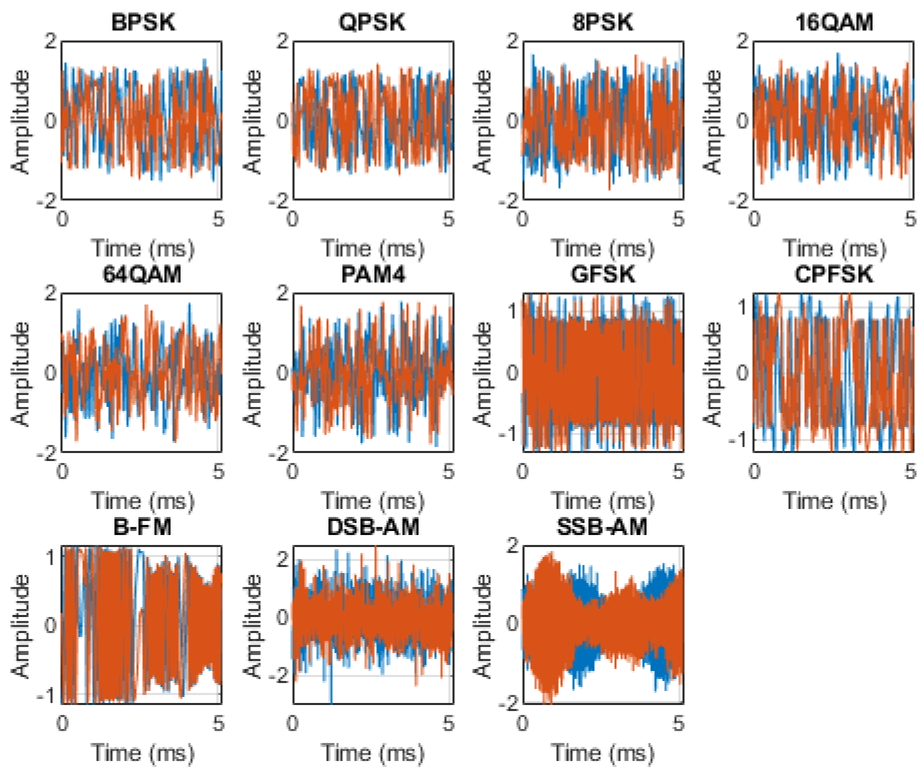
```
% Set the random number generator to a known state to be able to regenerate
% the same frames every time the simulation is run
```

```
rng(1235)
tic

numModulationTypes = length(modulationTypes);

channelInfo = info(channel);
transDelay = 50;
dataDirectory = fullfile(tempdir,"ModClassDataFiles");
disp("Data file directory is " + dataDirectory);

fileNameRoot = "frame";

% Check if data files exist
dataFilesExist = false;
if exist(dataDirectory,'dir')
  files = dir(fullfile(dataDirectory,sprintf("%s*",fileNameRoot)));
  if length(files) == numModulationTypes*numFramesPerModType
    dataFilesExist = true;
  end
end

if ~dataFilesExist
  disp("Generating data and saving in data files...")
  [success,msg,msgID] = mkdir(dataDirectory);
  if ~success
    error(msgID,msg)
  end
  for modType = 1:numModulationTypes
    elapsedTime = seconds(toc);
    elapsedTime.Format = 'hh:mm:ss';
    fprintf('%s - Generating %s frames\n', ...
      elapsedTime, modulationTypes(modType))

    label = modulationTypes(modType);
    numSymbols = (numFramesPerModType / sps);
    dataSrc = helperModClassGetSource(modulationTypes(modType), sps, 2*spf, fs);
    modulator = helperModClassGetModulator(modulationTypes(modType), sps, fs);
    if contains(char(modulationTypes(modType)), {'B-FM','DSB-AM','SSB-AM'})
      % Analog modulation types use a center frequency of 100 MHz
      channel.CenterFrequency = 100e6;
    else
      % Digital modulation types use a center frequency of 902 MHz
      channel.CenterFrequency = 902e6;
    end

    for p=1:numFramesPerModType
      % Generate random data
      x = dataSrc();

      % Modulate
      y = modulator(x);

      % Pass through independent channels
      rxSamples = channel(y);

      % Remove transients from the beginning, trim to size, and normalize
      frame = helperModClassFrameGenerator(rxSamples, spf, spf, transDelay, sps);
```

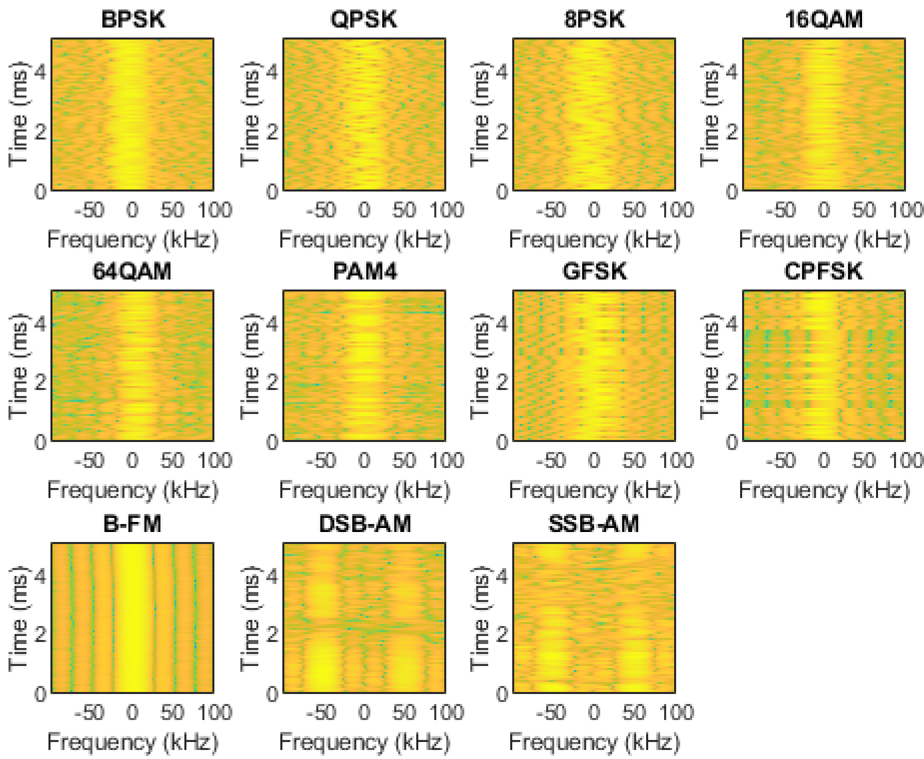
```
        % Save data file
        fileName = fullfile(dataDirectory,...
          sprintf("%s%s%03d",fileNameRoot,modulationTypes(modType),p));
        save(fileName,"frame","label")
      end
    end
else
  disp("Data files exist. Skip data generation.")
end
```

```
Data files exist. Skip data generation.
```

```
% Plot the amplitude of the real and imaginary parts of the example frames
% against the sample number
helperModClassPlotTimeDomain(dataDirectory,modulationTypes,fs)
```




```
% Plot the spectrogram of the example frames
helperModClassPlotSpectrogram(dataDirectory,modulationTypes,fs,sps)
```



Create a Datastore

To manage the files that contain the generated complex waveforms, use a `signalDatastore` object. Datastores are especially useful when each individual file fits in memory, but the entire collection does not necessarily fit.

```
frameDS = signalDatastore(dataDirectory,'SignalVariableNames',["frame","label"]);
```

Transform Complex Signals to Real Arrays

The deep learning network in this example looks for real inputs while the received signal has complex baseband samples. Transform the complex signals into real-valued 4-D arrays. The output frames have size 1-by-spf-by-2-by-N, where the first page (3rd dimension) is in-phase samples and the second page is quadrature samples. When the convolutional filters are of size 1-by-spf, this approach ensures that the information in the I and Q is mixed even in the convolutional layers and makes better use of the phase information. See helperModClassIQAsPages.

```
frameDSTrans = transform(frameDS,@helperModClassIQAsPages);
```

Split into Training, Validation, and Test

Divide the frames into training, validation, and test data. See helperModClassSplitData.

```
splitPercentages = [percentTrainingSamples,percentValidationSamples,percentTestSamples];
[trainDSTrans,validDSTrans,testDSTrans] = helperModClassSplitData(frameDSTrans,splitPercentages)
```

Import Data Into Memory

Neural network training is iterative. At every iteration, the datastore reads data from files and transforms the data before updating the network coefficients. If the data fits into the memory of your computer, importing the data from the files into the memory enables faster training by eliminating this repeated read from file and transform process. Instead, the data is read from the files and transformed once. Training this network using data files on disk takes about 110 minutes while training using in-memory data takes about 50 minutes.

Import the data in the files into memory. The files have two variables: `frame` and `label`. Each `read` call to the datastore returns a cell array, where the first element is the `frame` and the second element is the `label`. To read frames and labels, use the `transform` functions helperModClassReadFrame and helperModClassReadLabel. Use `readall` with the `"UseParallel"` option set to `true` to enable parallel processing of the transform functions, if you have Parallel Computing Toolbox license. Because the `readall` function, by default, concatenates the output of the `read` function over the first dimension, return the frames in a cell array and manually concatenate over the fourth dimension.

```
% Read the training and validation frames into the memory
pctExists = parallelComputingLicenseExists();
trainFrames = transform(trainDSTrans, @helperModClassReadFrame);
rxTrainFrames = readall(trainFrames,"UseParallel",pctExists);
rxTrainFrames = cat(4, rxTrainFrames{:});
validFrames = transform(validDSTrans, @helperModClassReadFrame);
rxValidFrames = readall(validFrames,"UseParallel",pctExists);
rxValidFrames = cat(4, rxValidFrames{:});

% Read the training and validation labels into the memory
trainLabels = transform(trainDSTrans, @helperModClassReadLabel);
rxTrainLabels = readall(trainLabels,"UseParallel",pctExists);
validLabels = transform(validDSTrans, @helperModClassReadLabel);
rxValidLabels = readall(validLabels,"UseParallel",pctExists);
testFrames = transform(testDSTrans, @helperModClassReadFrame);
rxTestFrames = readall(testFrames,"UseParallel",pctExists);
rxTestFrames = cat(4, rxTestFrames{:});

% Read the test labels into the memory
YPred = transform(testDSTrans, @helperModClassReadLabel);
rxTestLabels = readall(YPred,"UseParallel",pctExists);
```

Create Target Object

Create a target object for your target device that has a vendor name and an interface to connect your target device to the host computer. Interface options are JTAG (default) and Ethernet. Vendor options are Intel or Xilinx. To program the device, use the installed Xilinx Vivado Design Suite over an Ethernet connection.

```
hT = dlhdl.Target('Xilinx', Interface = 'Ethernet');
```

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pretrained series network `trainedAudioNet` as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Zynq UltraScale+ MPSoC ZCU102 board. The bitstream uses a single data type.

```
hW = dlhdl.Workflow(Network = trainedNet, Bitstream = 'zcu102_single', Target = hT);
```

Compile `trainedModulationClassification` Network

To compile the `trainedNet` series network, run the compile function of the `dlhdl.Workflow` object.

```
compile(hW)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### The network includes the following layers:
     1   'Input Layer'   Image Input             1×1024×2 images
     2   'CNN1'          Convolution             16 1×8×2 convolutions with stride [1  1] and pad
     3   'BN1'           Batch Normalization     Batch normalization with 16 channels
     4   'ReLU1'         ReLU                    ReLU
     5   'MaxPool1'      Max Pooling             1×2 max pooling with stride [1  2] and padding
     6   'CNN2'          Convolution             24 1×8×16 convolutions with stride [1  1] and pa
     7   'BN2'           Batch Normalization     Batch normalization with 24 channels
     8   'ReLU2'         ReLU                    ReLU
     9   'MaxPool2'      Max Pooling             1×2 max pooling with stride [1  2] and padding
    10   'CNN3'          Convolution             32 1×8×24 convolutions with stride [1  1] and pa
    11   'BN3'           Batch Normalization     Batch normalization with 32 channels
    12   'ReLU3'         ReLU                    ReLU
    13   'MaxPool3'      Max Pooling             1×2 max pooling with stride [1  2] and padding
    14   'CNN4'          Convolution             48 1×8×32 convolutions with stride [1  1] and pa
    15   'BN4'           Batch Normalization     Batch normalization with 48 channels
    16   'ReLU4'         ReLU                    ReLU
    17   'MaxPool4'      Max Pooling             1×2 max pooling with stride [1  2] and padding
    18   'CNN5'          Convolution             64 1×8×48 convolutions with stride [1  1] and pa
    19   'BN5'           Batch Normalization     Batch normalization with 64 channels
    20   'ReLU5'         ReLU                    ReLU
    21   'MaxPool5'      Max Pooling             1×2 max pooling with stride [1  2] and padding
    22   'CNN6'          Convolution             96 1×8×64 convolutions with stride [1  1] and pa
    23   'BN6'           Batch Normalization     Batch normalization with 96 channels
    24   'ReLU6'         ReLU                    ReLU
    25   'AP1'           Average Pooling         1×32 average pooling with stride [1  1] and padd
    26   'FC1'           Fully Connected         11 fully connected layer
    27   'SoftMax'       Softmax                 softmax
    28   'Output'        Classification Output   crossentropyex with '16QAM' and 10 other classes

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool1' made symmetric as it p
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool2' made symmetric as it p
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool3' made symmetric as it p
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool4' made symmetric as it p
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool5' made symmetric as it p
### Notice: The layer 'Input Layer' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in
### Notice: The layer 'SoftMax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'Output' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemente
### Compiling layer group: CNN1>>ReLU6 ...
### Compiling layer group: CNN1>>ReLU6 ... complete.
### Compiling layer group: AP1 ...
### Compiling layer group: AP1 ... complete.
### Compiling layer group: FC1 ...
### Compiling layer group: FC1 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
```

```
    _______________________     ______________     ________________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "28.0 MB"
    "InstructionDataOffset"     "0x02800000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02c00000"     "4.0 MB"
    "FCWeightDataOffset"        "0x03000000"     "4.0 MB"
    "EndOffset"                 "0x03400000"     "Total: 52.0 MB"

### Network compilation complete.

ans = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Zynq® UltraScale+™ MPSoC ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file.The function also downloads the network weights and biases. The deploy function verifies the Xilinx Vivado tool and the supported tool version. It then starts programming the FPGA device by using the bitstream, displays progress messages, and the time it takes to deploy the network.

```
deploy(hW)
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 11-Nov-2021 15:39:14
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 11-Nov-2021 15:39:14
```

Results

Classify five inputs from the test data set and compare the prediction results to the classification results from the Deep Learning Toolbox™. The `YPred` variable is the classification results from the Deep learning Toolbox™. The `fpga_prediction` variable is the classification result from the FPGA.

```
numtestFrames = size(rxTestFrames,4);
numView = 5;
listIndex = randperm(numtestFrames,numView);
testDataBatch = rxTestFrames(:,:,:,listIndex);
YPred = classify(trainedNet,testDataBatch);
[scores,speed] = predict(hW,testDataBatch, Profile ='on');
```

```
### Finished writing input activations.
### Running in multi-frame mode with 5 inputs.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
```

```
                          -------------          -------------          ---------        --
Network                   656546                 0.00298                    5            32
    CNN1                   11922                 0.00005
    MaxPool1               33524                 0.00015
    CNN2                   16136                 0.00007
    MaxPool2               74772                 0.00034
    CNN3                   11929                 0.00005
    MaxPool3               79074                 0.00036
    CNN4                    8185                 0.00004
    MaxPool4              112135                 0.00051
    CNN5                    6866                 0.00003
    MaxPool5              145626                 0.00066
    CNN6                    5077                 0.00002
    AP1                   144501                 0.00066
    FC1                     6763                 0.00003
 * The clock frequency of the DL processor is: 220MHz
```

```
[~,idx] = max(scores, [],2);
fpga_prediction = trainedNet.Layers(end).Classes(idx);
```

Compare the prediction results from Deep Learning Toolbox™ and the FPGA side by side. The prediction results from the FPGA match the prediction results from Deep Learning Toolbox™. In this table, the ground truth prediction is the Deep Learning Toolbox™ prediction.

```
fprintf('%12s %24s\n','Ground Truth','FPGA Prediction');for i= 1:size(fpga_prediction,1)
fprintf('%s %24s\n',YPred(i),fpga_prediction(i)); end
```

```
Ground Truth          FPGA Prediction

PAM4                     PAM4
BPSK                     BPSK
DSB-AM                   DSB-AM
SSB-AM                   SSB-AM
8PSK                     8PSK
```

References

1 O'Shea, T. J., J. Corgan, and T. C. Clancy. "Convolutional Radio Modulation Recognition Networks." Preprint, submitted June 10, 2016. https://arxiv.org/abs/1602.04105

2 O'Shea, T. J., T. Roy, and T. C. Clancy. "Over-the-Air Deep Learning Based Radio Signal Classification." IEEE Journal of Selected Topics in Signal Processing. Vol. 12, Number 1, 2018, pp. 168–179.

3 Liu, X., D. Yang, and A. E. Gamal. "Deep Neural Network Architectures for Modulation Classification." Preprint, submitted January 5, 2018. https://arxiv.org/abs/1712.00443v3

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `classify`

More About

- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2

Deploy Simple Adder Network by using MATLAB Deployment Script and Deployment Instructions File

This example shows how to create a `.dln` file for deploying a pretrained adder network. Deploy and initialize the generated deep learning processor IP core and adder network by using a MATLAB® deployment utility script to parse the generated .dln file.

Prerequisites

- Intel® Arria®10 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Intel® FPGA and SoC
- Deep Learning HDL Toolbox™
- Deep learning Toolbox™
- HDL Verifier™

Introduction

Generate a file that has instructions to communicate with the deployed deep learning processor IP core by using Deep Learning HDL Toolbox™. Initialize the deployed deep learning processor IP core without a MATLAB® connection by using a utility to parse and execute the instructions in the generated file. Use the example MATLAB® utility, `MATLABDeploymentUtility.m` to create your own custom utility. To deploy and initialize the generated deep learning processor IP core:

1. Create a `.dln` binary file.
2. Deploy the `.dln` file by using the MATLAB® utility script file.
3. Retrieve the prediction results by using MATLAB and the `predict` method.

Create Binary File

Create a `dlhdl.Target` object to deploy to a file. Provide the file name with '.dln' extension. `Filename` is an optional parameter here. If `FileName` is not provided, the generated file name is the same as the name of the object in the `Bitstream` argument of the `dlhdl.Workflow` object.

```
hTargetFile = dlhdl.Target('Intel','Interface','File','Filename','AdderNWdeploymentData.dln');
```

Create a simple adder network and an object of the `dlhdl.Workflow` class.

```
image = randi(255, [3,3,4]);
% create adder only network
inLayer = imageInputLayer(size(image), 'Name', 'data', 'Normalization', 'none');
addLayer = additionLayer(2, 'Name', 'add');
outLayer = regressionLayer('Name','output');
lgraph = layerGraph([inLayer, addLayer, outLayer]);
lgraph = connectLayers(lgraph, 'data', 'add/in2');
snet = assembleNetwork(lgraph);
hW = dlhdl.Workflow('network', snet, 'Bitstream', 'arria10soc_single','Target',hTargetFile);
```

Generate the network weights and biases, deployment instructions by using the `compile` method of the `dlhdl.Workflow` object.

```
hW.compile
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream arria10soc_single.
### The network includes the following layers:
     1   'data'     Image Input         3×3×4 images                           (SW Layer)
     2   'add'      Addition            Element-wise addition of 2 inputs  (HW Layer)
     3   'output'   Regression Output   mean-squared-error                 (SW Layer)

### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "0.0 MB"
    "SystemBufferOffset"        "0x00800000"     "20.0 MB"
    "InstructionDataOffset"     "0x01c00000"     "4.0 MB"
    "EndOffset"                 "0x02000000"     "Total: 32.0 MB"

### Network compilation complete.

ans = struct with fields:
             weights: []
        instructions: [1×1 struct]
           registers: []
    syncInstructions: [1×1 struct]
        constantData: {}
```

To generate .dln file use the `deploy` method of the `dlhdl.Workflow` object.

```
hW.deploy
```

```
WR@ADDR: 0x00000000 Len: 1: 0x00000001
WR@ADDR: 0x00000008 Len: 1: 0x80000000
WR@ADDR: 0x0000000c Len: 1: 0x80000000
WR@ADDR: 0x00000010 Len: 1: 0x80000000
WR@ADDR: 0x00000014 Len: 1: 0x80000000
WR@ADDR: 0x00000018 Len: 1: 0x80000000
WR@ADDR: 0x00000340 Len: 1: 0x01C00000
WR@ADDR: 0x00000348 Len: 1: 0x00000006
WR@ADDR: 0x00000338 Len: 1: 0x00000001
WR@ADDR: 0x0000033c Len: 1: 0x00C00000
WR@ADDR: 0x00000308 Len: 1: 0x01C00018
WR@ADDR: 0x0000030c Len: 1: 0x00000070
WR@ADDR: 0x00000224 Len: 1: 0x00000000
WR@ADDR: 0x81c00000 Len: 118: 0x00000001
WR@ADDR: 0x00000228 Len: 1: 0x00000000
WR@ADDR: 0x00000228 Len: 1: 0x00000001
WR@ADDR: 0x00000228 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0000000B
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0x5EECE9BF
WR@ADDR: 0x00000160 Len: 1: 0x00000001
```

```
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0001000B
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0xA6607FD1
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0002000B
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0xE69958D6
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0003000B
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0xCE9B0C98
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0000000A
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0xE306BC8E
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0001000A
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0x6D1D3062
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0002000A
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0x5E0BE35F
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0003000A
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0x8E5097FB
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0004000A
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0xE9C840AC
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0005000A
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0x742F745C
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0006000A
WR@ADDR: 0x00000164 Len: 1: 0x00000000
```

```
WR@ADDR: 0x00000168 Len: 1: 0x725F612A
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
WR@ADDR: 0x00000140 Len: 1: 0x00000001
WR@ADDR: 0x0000014c Len: 1: 0x0007000A
WR@ADDR: 0x00000164 Len: 1: 0x00000000
WR@ADDR: 0x00000168 Len: 1: 0x7014FDA9
WR@ADDR: 0x00000160 Len: 1: 0x00000001
WR@ADDR: 0x00000160 Len: 1: 0x00000000
```

The generated .dln file is a binary file. All the data inside the file is in hex format.

Structure of Generated .dln File

The data inside the binary file is of strings and uint32 bit format. All the strings are NULL terminated. This image shows a section of the generated .dln file.

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	4d	57	44	4c	4e	56	30	32	00	30	37	2d	44	65	63	2d	MWDLNV02.07-Dec-
00000010	32	30	32	31	20	31	38	3a	33	35	3a	35	31	00	44	65	2021 18:35:51.De
00000020	65	70	20	4c	65	61	72	6e	69	6e	67	20	48	44	4c	20	ep Learning HDL
00000030	54	6f	6f	6c	62	6f	78	00	31	2e	33	00	28	52	32	30	Toolbox.1.3.(R20
00000040	32	32	61	29	00	31	34	2d	4f	63	74	2d	32	30	32	31	22a).14-Oct-2021
00000050	00	00	00	00	00	00	00	01	00	00	00	00	80	00	00	00	............€...
00000060	80	49	6e	74	65	6c	00	6d	77	69	70	63	6f	72	65	5f	€Intel.mwipcore_
00000070	64	6c	30	3a	6d	6d	77	72	30	00	6d	77	69	70	63	6f	dl0:mmwr0.mwipco
00000080	72	65	5f	64	6c	30	3a	6d	6d	72	64	30	00	6d	77	69	re_dl0:mmrd0.mwi
00000090	70	63	6f	72	65	5f	64	64	72	30	3a	6d	6d	32	73	30	pcore_ddr0:mm2s0
000000a0	00	6d	77	69	70	63	6f	72	65	5f	64	64	72	30	3a	73	.mwipcore_ddr0:s
000000b0	32	6d	6d	30	00	53	4f	44	00	57	52	44	00	57	52	40	2mm0.SOD.WRD.WR@
000000c0	41	44	44	52	3a	20	30	78	30	30	30	30	30	30	30	30	ADDR: 0x00000000
000000d0	20	4c	65	6e	3a	20	31	00	00	00	00	00	01	00	00	00	Len: 1........
000000e0	01	00	00	00	57	52	44	00	57	52	40	41	44	44	52	3a	....WRD.WR@ADDR:
000000f0	20	30	78	30	30	30	30	30	30	30	38	20	4c	65	6e	3a	0x00000008 Len:
00000100	20	31	00	08	00	00	00	01	00	00	00	00	00	00	80	57	1...........€W
00000110	52	44	00	57	52	40	41	44	44	52	3a	20	30	78	30	30	RD.WR@ADDR: 0x00

The binary file consists of:

- Header section which consists of Date and time the file was generated and some important information like DL IP Base address, DL IP Address range, DDR Base address and DDR address range.
- Start of Data(SOD) section indicates start of instructions to read and write data.
- Data section with AXI read and write transactions.
- An End of data(EOD) command indicates the end of the file.

For more information about the binary file structure, see "Initialize Deployed Deep Learning Processor Without Using a MATLAB Connection" on page 5-9.

Deploy Bitstream and .dln file using MATLAB deployment utility

Setup Xilinx vivado tool path before programming the bitstream. To use JTAG,Install Xilinx™ Vivado™ Design Suite 2020.1. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.1\bin\vivado.b
hTarget1 = dlhdl.Target('Intel','Interface','JTAG');
hW1 = dlhdl.Workflow('network', snet, 'Bitstream', 'arria10soc_single','Target',hTarget1);
% Program BitStream at this point. Because to transfer data using FPGAIO i.e, Write/Read FPGA mus
hW1.deploy('ProgramBitstream',true,'ProgramNetwork',false);
```

```
### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
```

Deploy .dln file using MATLAB deployment utility:

Use the MATLAB deployment utility script to extract the instructions from the binary file(generated .dln file) and program the FPGA. The deployment utility script:

- Reads the header details of the .dln file until detection of the 'SOD' command. Line 1 to 35 in the `MATLABDeploymentUtility.m script file read in the header information` .Once SOD is detected actual read and write instructions of compiled network will starts.
- Reads data by extracting the address, length of data to be read and data to read information from the read packet structure. Use the extracted address, length of data to be read and data to read as input arguments to the `readmemory` function.
- Write data by extracting the write data address and data to write information from the write packet structure. Use the extracted write data address and data to write as input arguments to the `writememory` function.
- Detects the end of data (EOD) command and closes the generated file.

```
MATLABDeploymentUtility('AdderNWdeploymentData.dln');
```

```
WR@ADDR: 0x00000000 Len: 1: 0X00000001
WR@ADDR: 0x00000008 Len: 1: 80000000
WR@ADDR: 0x0000000c Len: 1: 80000000
WR@ADDR: 0x00000010 Len: 1: 80000000
WR@ADDR: 0x00000014 Len: 1: 80000000
WR@ADDR: 0x00000018 Len: 1: 80000000
WR@ADDR: 0x00000340 Len: 1: 0X01C00000
WR@ADDR: 0x00000348 Len: 1: 0X00000006
WR@ADDR: 0x00000338 Len: 1: 0X00000001
WR@ADDR: 0x0000033c Len: 1: 0X00C00000
WR@ADDR: 0x00000308 Len: 1: 0X01C00018
WR@ADDR: 0x0000030c Len: 1: 0X00000070
WR@ADDR: 0x00000224 Len: 1: 0X00000000
WR@ADDR: 0x81c00000 Len: 118: 0X00000001
WR@ADDR: 0x00000228 Len: 1: 0X00000000
WR@ADDR: 0x00000228 Len: 1: 0X00000001
WR@ADDR: 0x00000228 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0000000B
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: 5EECE9BF
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
```

```
WR@ADDR: 0x0000014c Len: 1: 0X0001000B
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: A6607FD1
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0002000B
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: E69958D6
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0003000B
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: CE9B0C98
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0000000A
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: E306BC8E
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0001000A
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: 6D1D3062
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0002000A
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: 5E0BE35F
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0003000A
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: 8E5097FB
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0004000A
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: E9C840AC
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0005000A
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: 742F745C
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0006000A
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: 725F612A
WR@ADDR: 0x00000160 Len: 1: 0X00000001
```

```
WR@ADDR: 0x00000160 Len: 1: 0X00000000
WR@ADDR: 0x00000140 Len: 1: 0X00000001
WR@ADDR: 0x0000014c Len: 1: 0X0007000A
WR@ADDR: 0x00000164 Len: 1: 0X00000000
WR@ADDR: 0x00000168 Len: 1: 7014FDA9
WR@ADDR: 0x00000160 Len: 1: 0X00000001
WR@ADDR: 0x00000160 Len: 1: 0X00000000
```

Retrieve Prediction Results

Deploy the generated deep learning processor IP core and network by using the MATLAB deployment utility. Retrieve the prediction results from the deployed deep learning processor and compare them with the prediction results from the Deep Learning Toolbox™.

```
hW2 = dlhdl.Workflow('network', snet, 'Bitstream', 'arria10soc_single','Target',hTarget1);
[prediction, ~] = hW2.predict(image,'ProgramBitstream',false,'ProgramNetwork',true);
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream arria10soc_single.
### The network includes the following layers:
     1   'data'     Image Input         3×3×4 images                       (SW Layer)
     2   'add'      Addition            Element-wise addition of 2 inputs  (HW Layer)
     3   'output'   Regression Output   mean-squared-error                 (SW Layer)

### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    ________________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "0.0 MB"
    "SystemBufferOffset"        "0x00800000"     "20.0 MB"
    "InstructionDataOffset"     "0x01c00000"     "4.0 MB"
    "EndOffset"                 "0x02000000"     "Total: 32.0 MB"

### Network compilation complete.

### Finished writing input activations.
### Running single input activation.
```

Even though we provide 'ProgramNetwork' as 'true' in the above prediction function. FPGA remains programmed with the network instructions deployed through MATLAB deployment utility only. This is because during Programming network we look for network checksum, if checksum matches with the previous checksum, network will not be reprogrammed.

```
% Get DL toolbox output.
DLToolboxSimulationOutp = snet.predict(image, 'ExecutionEnvironment', 'cpu');

% Verify DL Toolbox prediction result with prediction results for the deployment
% done using MATLAB deployment utility Script
isequal(DLToolboxSimulationOutp,prediction)
```

```
ans = logical
   1
```

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `classify`

More About

- “Initialize Deployed Deep Learning Processor Without Using a MATLAB Connection” on page 5-9

Human Pose Estimation by Using Segmentation DAG Network Deployed to FPGA

This example shows how to create, compile, and deploy a `dlhdl.Workflow` object by using the Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC. The `Workflow` object has a custom trained human pose estimation network as the network object. The network detects and outputs poses of people present in an input image of size 256-by-192. To train the network, see Estimate Body Pose Using Deep Learning.

The goal of body pose estimation is to identify the location of people in an image and the orientation of their body parts. When multiple people are present in a scene, pose estimation can be more difficult because of occlusion, body contact, and proximity of similar body parts. Rapidly prototype and verify the accuracy and performance of your custom trained human pose estimation network by using Deep Learning HDL Toolbox™ to deploy the network to your target FPGA board and using MATLAB® to retrieve the prediction results.

Prerequisites

- Zynq® Ultrascale+™ MPSoC ZCU102 Evaluation Kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx™ FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Load Pretrained Pose Estimation Network

To load the pretrained Directed Acyclic Graph (DAG) network, enter:

```
net = getPoseEstimationNetwork
```

```
Fetching PoseEstimationNetwork.zip (55 MB)...
Fetching PoseEstimationNetwork.zip (55 MB)

net = 
  DAGNetwork with properties:

         Layers: [75×1 nnet.cnn.layer.Layer]
    Connections: [82×2 table]
     InputNames: {'data'}
    OutputNames: {'RegressionLayer_conv15_fwd'}
```

Use the `analyzeNetwork` function to obtain information about the 75 layers in the DAG network.

```
analyzeNetwork(net)
```

Create Target Object

Use the `dlhdl.Target` class to create a target object that has a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG (default) and Ethernet. To use JTAG, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.b
```

```
hTarget = dlhdl.Target('Xilinx', Interface = 'Ethernet');
```

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. Specify the saved pretrained pose estimation network, `net`, as the network object. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx ZCU102 SoC board and the bitstream uses the single data type.

```
hW = dlhdl.Workflow(Network = net, Bitstream = 'zcu102_single', Target = hTarget);
```

Compile Workflow Object

To compile the Pose Estimation Network, run the compile function of the `dlhdl.Workflow` object.

```
dn = compile(hW);
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### The network includes the following layers:
     1   'data'                        Image Input             256×192×3 images with 'zscore' n
     2   'conv1'                       Convolution             64 7×7×3 convolutions with strid
     3   'bn_conv1'                    Batch Normalization     Batch normalization with 64 chan
     4   'conv1_relu'                  ReLU                    ReLU
     5   'pool1'                       Max Pooling             3×3 max pooling with stride [2
     6   'res2a_branch2a'              Convolution             64 3×3×64 convolutions with stri
     7   'bn2a_branch2a'               Batch Normalization     Batch normalization with 64 chan
     8   'res2a_branch2a_relu'         ReLU                    ReLU
     9   'res2a_branch2b'              Convolution             64 3×3×64 convolutions with stri
    10   'bn2a_branch2b'               Batch Normalization     Batch normalization with 64 chan
    11   'res2a'                       Addition                Element-wise addition of 2 input
    12   'res2a_relu'                  ReLU                    ReLU
    13   'res2b_branch2a'              Convolution             64 3×3×64 convolutions with stri
    14   'bn2b_branch2a'               Batch Normalization     Batch normalization with 64 chan
    15   'res2b_branch2a_relu'         ReLU                    ReLU
    16   'res2b_branch2b'              Convolution             64 3×3×64 convolutions with stri
    17   'bn2b_branch2b'               Batch Normalization     Batch normalization with 64 chan
    18   'res2b'                       Addition                Element-wise addition of 2 input
    19   'res2b_relu'                  ReLU                    ReLU
    20   'res3a_branch2a'              Convolution             128 3×3×64 convolutions with str
    21   'bn3a_branch2a'               Batch Normalization     Batch normalization with 128 cha
    22   'res3a_branch2a_relu'         ReLU                    ReLU
    23   'res3a_branch2b'              Convolution             128 3×3×128 convolutions with st
    24   'bn3a_branch2b'               Batch Normalization     Batch normalization with 128 cha
    25   'res3a'                       Addition                Element-wise addition of 2 input
    26   'res3a_relu'                  ReLU                    ReLU
    27   'res3a_branch1'               Convolution             128 1×1×64 convolutions with str
    28   'bn3a_branch1'                Batch Normalization     Batch normalization with 128 cha
    29   'res3b_branch2a'              Convolution             128 3×3×128 convolutions with st
    30   'bn3b_branch2a'               Batch Normalization     Batch normalization with 128 cha
    31   'res3b_branch2a_relu'         ReLU                    ReLU
    32   'res3b_branch2b'              Convolution             128 3×3×128 convolutions with st
    33   'bn3b_branch2b'               Batch Normalization     Batch normalization with 128 cha
    34   'res3b'                       Addition                Element-wise addition of 2 input
    35   'res3b_relu'                  ReLU                    ReLU
    36   'res4a_branch2a'              Convolution             256 3×3×128 convolutions with st
    37   'bn4a_branch2a'               Batch Normalization     Batch normalization with 256 cha
    38   'res4a_branch2a_relu'         ReLU                    ReLU
```

```
    39   'res4a_branch2b'              Convolution              256 3×3×256 convolutions with st
    40   'bn4a_branch2b'               Batch Normalization      Batch normalization with 256 cha
    41   'res4a'                       Addition                 Element-wise addition of 2 input
    42   'res4a_relu'                  ReLU                     ReLU
    43   'res4a_branch1'               Convolution              256 1×1×128 convolutions with st
    44   'bn4a_branch1'                Batch Normalization      Batch normalization with 256 cha
    45   'res4b_branch2a'              Convolution              256 3×3×256 convolutions with st
    46   'bn4b_branch2a'               Batch Normalization      Batch normalization with 256 cha
    47   'res4b_branch2a_relu'         ReLU                     ReLU
    48   'res4b_branch2b'              Convolution              256 3×3×256 convolutions with st
    49   'bn4b_branch2b'               Batch Normalization      Batch normalization with 256 cha
    50   'res4b'                       Addition                 Element-wise addition of 2 input
    51   'res4b_relu'                  ReLU                     ReLU
    52   'res5a_branch2a'              Convolution              512 3×3×256 convolutions with st
    53   'bn5a_branch2a'               Batch Normalization      Batch normalization with 512 cha
    54   'res5a_branch2a_relu'         ReLU                     ReLU
    55   'res5a_branch2b'              Convolution              512 3×3×512 convolutions with st
    56   'bn5a_branch2b'               Batch Normalization      Batch normalization with 512 cha
    57   'res5a'                       Addition                 Element-wise addition of 2 input
    58   'res5a_relu'                  ReLU                     ReLU
    59   'res5a_branch1'               Convolution              512 1×1×256 convolutions with st
    60   'bn5a_branch1'                Batch Normalization      Batch normalization with 512 cha
    61   'res5b_branch2a'              Convolution              512 3×3×512 convolutions with st
    62   'bn5b_branch2a'               Batch Normalization      Batch normalization with 512 cha
    63   'res5b_branch2a_relu'         ReLU                     ReLU
    64   'res5b_branch2b'              Convolution              512 3×3×512 convolutions with st
    65   'bn5b_branch2b'               Batch Normalization      Batch normalization with 512 cha
    66   'res5b'                       Addition                 Element-wise addition of 2 input
    67   'res5b_relu'                  ReLU                     ReLU
    68   'transposed-conv_1'           Transposed Convolution   256 4×4×512 transposed convoluti
    69   'relu_1'                      ReLU                     ReLU
    70   'transposed-conv_2'           Transposed Convolution   256 4×4×256 transposed convoluti
    71   'relu_2'                      ReLU                     ReLU
    72   'transposed-conv_3'           Transposed Convolution   256 4×4×256 transposed convoluti
    73   'relu_3'                      ReLU                     ReLU
    74   'conv2d_final'                Convolution              17 1×1×256 convolutions with str
    75   'RegressionLayer_conv15_fwd'  Regression Output        mean-squared-error

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convo
### Notice: The layer 'transposed-conv_1' of type 'nnet.cnn.layer.TransposedConvolution2DLayer' i
### Notice: The layer 'transposed-conv_2' of type 'nnet.cnn.layer.TransposedConvolution2DLayer' i
### Notice: The layer 'transposed-conv_3' of type 'nnet.cnn.layer.TransposedConvolution2DLayer' i
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data'
### Notice: The layer 'RegressionLayer_conv15_fwd' with type 'nnet.cnn.layer.RegressionOutputLaye
### Compiling layer group: conv1>>pool1 ...
### Compiling layer group: conv1>>pool1 ... complete.
### Compiling layer group: res2a_branch2a>>res2a_branch2b ...
### Compiling layer group: res2a_branch2a>>res2a_branch2b ... complete.
### Compiling layer group: res2b_branch2a>>res2b_branch2b ...
### Compiling layer group: res2b_branch2a>>res2b_branch2b ... complete.
### Compiling layer group: res3a_branch1 ...
### Compiling layer group: res3a_branch1 ... complete.
### Compiling layer group: res3a_branch2a>>res3a_branch2b ...
### Compiling layer group: res3a_branch2a>>res3a_branch2b ... complete.
### Compiling layer group: res3b_branch2a>>res3b_branch2b ...
### Compiling layer group: res3b_branch2a>>res3b_branch2b ... complete.
### Compiling layer group: res4a_branch1 ...
### Compiling layer group: res4a_branch1 ... complete.
```

```
### Compiling layer group: res4a_branch2a>>res4a_branch2b ...
### Compiling layer group: res4a_branch2a>>res4a_branch2b ... complete.
### Compiling layer group: res4b_branch2a>>res4b_branch2b ...
### Compiling layer group: res4b_branch2a>>res4b_branch2b ... complete.
### Compiling layer group: res5a_branch1 ...
### Compiling layer group: res5a_branch1 ... complete.
### Compiling layer group: res5a_branch2a>>res5a_branch2b ...
### Compiling layer group: res5a_branch2a>>res5a_branch2b ... complete.
### Compiling layer group: res5b_branch2a>>res5b_branch2b ...
### Compiling layer group: res5b_branch2a>>res5b_branch2b ... complete.
### Compiling layer group: transposed-conv_1_insertZeros ...
### Compiling layer group: transposed-conv_1_insertZeros ... complete.
### Compiling layer group: transposed-conv_1>>relu_1 ...
### Compiling layer group: transposed-conv_1>>relu_1 ... complete.
### Compiling layer group: transposed-conv_2_insertZeros ...
### Compiling layer group: transposed-conv_2_insertZeros ... complete.
### Compiling layer group: transposed-conv_2>>relu_2 ...
### Compiling layer group: transposed-conv_2>>relu_2 ... complete.
### Compiling layer group: transposed-conv_3_insertZeros ...
### Compiling layer group: transposed-conv_3_insertZeros ... complete.
### Compiling layer group: transposed-conv_3>>conv2d_final ...
### Compiling layer group: transposed-conv_3>>conv2d_final ... complete.

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"        
    "OutputResultOffset"        "0x01800000"     "8.0 MB"         
    "SchedulerDataOffset"       "0x02000000"     "8.0 MB"         
    "SystemBufferOffset"        "0x02800000"     "28.0 MB"        
    "InstructionDataOffset"     "0x04400000"     "8.0 MB"         
    "ConvWeightDataOffset"      "0x04c00000"     "220.0 MB"       
    "EndOffset"                 "0x12800000"     "Total: 296.0 MB"

### Network compilation complete.
```

Program Bitstream Into FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. The function also downloads the network weights and biases. The deploy function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

```
deploy(hW)
```

```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'

Downloading target FPGA device configuration over Ethernet to SD card done. The system will now r
```

```
System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 19-Jan-2022 20:13:32
```

Load Test Image

Read a test image, then crop an image of a person and resize it to the network input size

```
I = imread('visionteam1.jpg');
bbox = [182 74 303 404];
Iin = imresize(imcrop(I, bbox), [256, 192]);
```

Run Prediction for One Image

Execute the predict function of the `dlhdl.Workflow` object.

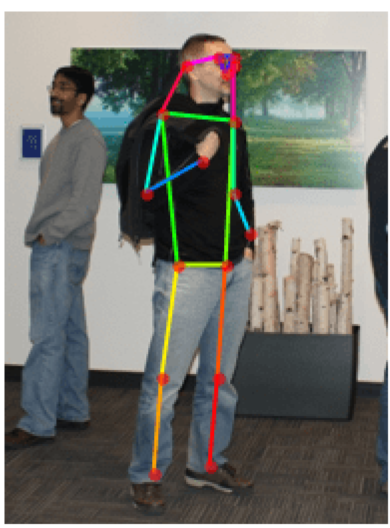
```
[prediction, speed] = predict(hW, single(Iin), Profile = 'on');
```

```
### Finished writing input activations.
### Running single input activation.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------              ---------        --
Network                  106379104                  0.48354                       1          1063
    data_norm_add           344327                  0.00157
    data_norm               344408                  0.00157
    conv1                  2193504                  0.00997
    pool1                   518554                  0.00236
    res2a_branch2a          961197                  0.00437
    res2a_branch2b          960769                  0.00437
    res2a                   366754                  0.00167
    res2b_branch2a          961107                  0.00437
    res2b_branch2b          960940                  0.00437
    res2b                   366715                  0.00167
    res3a_branch1           549086                  0.00250
    res3a_branch2a          542269                  0.00246
    res3a_branch2b          894520                  0.00407
    res3a                   183362                  0.00083
    res3b_branch2a          894609                  0.00407
    res3b_branch2b          894473                  0.00407
    res3b                   183403                  0.00083
    res4a_branch1           485003                  0.00220
    res4a_branch2a          485309                  0.00221
    res4a_branch2b          877978                  0.00399
    res4a                    91703                  0.00042
    res4b_branch2a          878002                  0.00399
    res4b_branch2b          878177                  0.00399
    res4b                    91743                  0.00042
    res5a_branch1          1063237                  0.00483
    res5a_branch2a         1063292                  0.00483
    res5a_branch2b         2064743                  0.00939
    res5a                    45904                  0.00021
    res5b_branch2a         2064047                  0.00938
```

```
    res5b_branch2b         2064894                  0.00939
    res5b                    45894                  0.00021
    transposed-conv_1_insertZeros    219876                  0.00100
    transposed-conv_1      6587071                  0.02994
    transposed-conv_2_insertZeros    261960                  0.00119
    transposed-conv_2     16585251                  0.07539
    transposed-conv_3_insertZeros   1058301                  0.00481
    transposed-conv_3     55919081                  0.25418
    conv2d_final           1427387                  0.00649
 * The clock frequency of the DL processor is: 220MHz
```

The output data has 17 channels. Each channel corresponds to a heatmap for a unique body part. To obtain keypoints from the heatmaps, use `heatmaps2Keypoints` helper function. To visualize the results, superimpose the detected keypoints on the original image by using the `visualizeKeyPoints` helper function. The functions are attached to the example as supporting files.

```
keypoints = heatmaps2Keypoints(prediction);
J = visualizeKeyPoints(Iin, keypoints);
imshow(J);
```



See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `classify`

Semantic Segmentation of Multispectral Images by Using Quantized U-Net on FPGA

This example show how to use the Deep Learning HDL Toolbox™ to deploy a quantized U-Net to perform semantic segmentation on multispectral images. The example uses the pretrained U-Net network to demonstrate quantization and deployment of the quantized network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases, and activations of network layers to 8-bit scaled integer data types. To retrieve the prediction results, use MATLAB®.

Deploy the quantized U-Net network by creating a `dlhdl.Workflow` object. Use the `dlhdl.Workflow` object to:

- Generate a list of instructions, weights and biases by using the `compile` method.
- Generate a programming file for the FPGA by using the `deploy` method.
- Retrieve the network prediction results and performance by using the `predict` method.

The quantized network takes in a multispectral input image of size 256-by-256 that has six channels and outputs a segmentation map where each pixel corresponds to one of 18 classes. This network is taken from the Semantic Segmentation of Multispectral Images Using Deep Learning example from the Computer Vision Toolbox™. To train the network, see "Semantic Segmentation of Multispectral Images Using Deep Learning" (Image Processing Toolbox).

Prerequisites

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning HDL Toolbox™ Support Package for Intel FPGA and SoC
- Intel Arria10 SoC Development Kit
- Deep Learning Toolbox™ Model Quantization Library Support Package.
- MATLAB Coder Interface for Deep learning

Load Pretrained U-Net Network

Load the pretrained Directed Acyclic Graph (DAG) network U-Net using the `downloadTrainedUnet` helper function. This function is attached to the example as a supporting file.

```
imageDir = tempdir;
trainedUNetURL = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat';
downloadTrainedUnet(trainedUNetURL, imageDir);
load(fullfile(imageDir, 'trainedUnet', 'multispectralUnet.mat'));
```

To obtain information about the 58 layers in the DAG network, use the `analyzeNetwork` function.

```
analyzeNetwork(net)
```

Download Data

The pretrained network was trained on a high-resolution multispectral data set [1 on page 10-238]. The image set was captured using a drone over Hamlin Beach State Park, NY. The data contains

labeled training, validation, and test set that have 18 object class labels. The size of the data file is ~3.0 GB. For calibration and testing of the network, use parts of the training data set.

Download the MAT-file version of the data set by using the `downloadHamlinBeachMSIData` helper function. This function is attached to the example as a supporting file.

```
imageDir = tempdir;
url = 'https://home.cis.rit.edu/~cnspci/other/data/rit18_data.mat';
downloadHamlinBeachMSIData(url, imageDir);
```

Create Calibration Data

The pretrained U-Net network accepts inputs of size 256-by-256-by-6. The training data in the downloaded MAT file has a size of 7-by-9393-by-5642. Use the `extractMultispectralData` helper function to extract patches of size 256-by-256-by-6 and store them in MAT files for calibration. The seventh channel in the training data is a binary mask and is not used by the pretrained network for inference.

For best quantization results, the calibration data must be representative of actual inputs that are predicted by the U-Net network. Expedite the calibration process by reducing the calibration data set to six images. Choose the six images so that they form a 2-by-3 grid to represent a large continuous image.

```
foldername = 'CalibData';
dataPath = fullfile(imageDir, 'rit18_data', 'rit18_data.mat');
im = extractMultispectralData(foldername, dataPath, 2, 3);
```

The first three channels of the multispectral training data contain RGB information. Display a histogram-equalized version of the extracted data.

```
im = histeq(im(:,:,[3 2 1]));
montage({im});
```

Create an `imageDatastore` object to use for calibration. The patches are loaded from the folder 'CalibData'.

```
imds = imageDatastore('CalibData', FileExtensions = '.mat', ReadFcn = @matReader);
```

Create dlquantizer Object

Create a quantized network object by using `dlquantizer`. Set the target execution environment to FPGA.

```
dlQuantObj = dlquantizer(net, ExecutionEnvironment = 'FPGA');
```

Calibrate Quantized Network

Use the `calibrate` function to exercise the network by using sample inputs and collect the range information. The `calibrate` function exercises the network. The function collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and in the dynamic ranges of the activations in all layers of the network. The `calibrate` function returns a table.

```
calibrate(dlQuantObj, imds)
```

```
ans=103×5 table
        Optimized Layer Name               Network Layer Name          Learnables / Activat
    ____________________________    ______________________    ________________
```

```
    {'Encoder-Section-1-Conv-1_Weights'}    {'Encoder-Section-1-Conv-1'}           "Weights"
    {'Encoder-Section-1-Conv-1_Bias'   }    {'Encoder-Section-1-Conv-1'}           "Bias"
    {'Encoder-Section-1-Conv-2_Weights'}    {'Encoder-Section-1-Conv-2'}           "Weights"
    {'Encoder-Section-1-Conv-2_Bias'   }    {'Encoder-Section-1-Conv-2'}           "Bias"
    {'Encoder-Section-2-Conv-1_Weights'}    {'Encoder-Section-2-Conv-1'}           "Weights"
    {'Encoder-Section-2-Conv-1_Bias'   }    {'Encoder-Section-2-Conv-1'}           "Bias"
    {'Encoder-Section-2-Conv-2_Weights'}    {'Encoder-Section-2-Conv-2'}           "Weights"
    {'Encoder-Section-2-Conv-2_Bias'   }    {'Encoder-Section-2-Conv-2'}           "Bias"
    {'Encoder-Section-3-Conv-1_Weights'}    {'Encoder-Section-3-Conv-1'}           "Weights"
    {'Encoder-Section-3-Conv-1_Bias'   }    {'Encoder-Section-3-Conv-1'}           "Bias"
    {'Encoder-Section-3-Conv-2_Weights'}    {'Encoder-Section-3-Conv-2'}           "Weights"
    {'Encoder-Section-3-Conv-2_Bias'   }    {'Encoder-Section-3-Conv-2'}           "Bias"
    {'Encoder-Section-4-Conv-1_Weights'}    {'Encoder-Section-4-Conv-1'}           "Weights"
    {'Encoder-Section-4-Conv-1_Bias'   }    {'Encoder-Section-4-Conv-1'}           "Bias"
    {'Encoder-Section-4-Conv-2_Weights'}    {'Encoder-Section-4-Conv-2'}           "Weights"
    {'Encoder-Section-4-Conv-2_Bias'   }    {'Encoder-Section-4-Conv-2'}           "Bias"
      ⋮
```

Create Target Object

Set the synthesis tool path to point to an installed Intel® Quartus® Prime Standard Edition 20.1 executable file. You must have already installed Altera® Quartus II.

```
% hdlsetuptoolpath(ToolName = 'Altera Quartus II', ToolPath = 'C:\intel\20.1\quartus\bin\quartus
```

Create a target object that has a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG (default) and Ethernet.

```
hTarget = dlhdl.Target('Intel', Interface = 'JTAG');
```

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. Specify the network and bitstream name. Specify the quantized network object `dlQuantObj` as the network. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Intel Arria 10 SoC board. The bitstream uses an int8 data type.

```
hW = dlhdl.Workflow(Network = dlQuantObj, Bitstream = 'arria10soc_int8', Target = hTarget);
```

Compile Workflow Object

To compile the U-Net network, run the `compile` function of the `dlhdl.Workflow` object.

```
dn = compile(hW)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream arria10soc_int8.
### The network includes the following layers:
     1   'ImageInputLayer'                          Image Input                 256×256×6 images w
     2   'Encoder-Section-1-Conv-1'                 Convolution                 64 3×3×6 convoluti
     3   'Encoder-Section-1-ReLU-1'                 ReLU                        ReLU
     4   'Encoder-Section-1-Conv-2'                 Convolution                 64 3×3×64 convolut
     5   'Encoder-Section-1-ReLU-2'                 ReLU                        ReLU
     6   'Encoder-Section-1-MaxPool'                Max Pooling                 2×2 max pooling wi
     7   'Encoder-Section-2-Conv-1'                 Convolution                 128 3×3×64 convolu
     8   'Encoder-Section-2-ReLU-1'                 ReLU                        ReLU
     9   'Encoder-Section-2-Conv-2'                 Convolution                 128 3×3×128 convo
```

```
    10   'Encoder-Section-2-ReLU-2'               ReLU                        ReLU
    11   'Encoder-Section-2-MaxPool'              Max Pooling                 2×2 max pooling wi
    12   'Encoder-Section-3-Conv-1'               Convolution                 256 3×3×128 convo
    13   'Encoder-Section-3-ReLU-1'               ReLU                        ReLU
    14   'Encoder-Section-3-Conv-2'               Convolution                 256 3×3×256 convo
    15   'Encoder-Section-3-ReLU-2'               ReLU                        ReLU
    16   'Encoder-Section-3-MaxPool'              Max Pooling                 2×2 max pooling wi
    17   'Encoder-Section-4-Conv-1'               Convolution                 512 3×3×256 convo
    18   'Encoder-Section-4-ReLU-1'               ReLU                        ReLU
    19   'Encoder-Section-4-Conv-2'               Convolution                 512 3×3×512 convo
    20   'Encoder-Section-4-ReLU-2'               ReLU                        ReLU
    21   'Encoder-Section-4-DropOut'              Dropout                     50% dropout
    22   'Encoder-Section-4-MaxPool'              Max Pooling                 2×2 max pooling wi
    23   'Mid-Conv-1'                             Convolution                 1024 3×3×512 conv
    24   'Mid-ReLU-1'                             ReLU                        ReLU
    25   'Mid-Conv-2'                             Convolution                 1024 3×3×1024 con
    26   'Mid-ReLU-2'                             ReLU                        ReLU
    27   'Mid-DropOut'                            Dropout                     50% dropout
    28   'Decoder-Section-1-UpConv'               Transposed Convolution      512 2×2×1024 tran
    29   'Decoder-Section-1-UpReLU'               ReLU                        ReLU
    30   'Decoder-Section-1-DepthConcatenation'   Depth concatenation         Depth concatenati
    31   'Decoder-Section-1-Conv-1'               Convolution                 512 3×3×1024 conv
    32   'Decoder-Section-1-ReLU-1'               ReLU                        ReLU
    33   'Decoder-Section-1-Conv-2'               Convolution                 512 3×3×512 convo
    34   'Decoder-Section-1-ReLU-2'               ReLU                        ReLU
    35   'Decoder-Section-2-UpConv'               Transposed Convolution      256 2×2×512 trans
    36   'Decoder-Section-2-UpReLU'               ReLU                        ReLU
    37   'Decoder-Section-2-DepthConcatenation'   Depth concatenation         Depth concatenati
    38   'Decoder-Section-2-Conv-1'               Convolution                 256 3×3×512 convo
    39   'Decoder-Section-2-ReLU-1'               ReLU                        ReLU
    40   'Decoder-Section-2-Conv-2'               Convolution                 256 3×3×256 convo
    41   'Decoder-Section-2-ReLU-2'               ReLU                        ReLU
    42   'Decoder-Section-3-UpConv'               Transposed Convolution      128 2×2×256 trans
    43   'Decoder-Section-3-UpReLU'               ReLU                        ReLU
    44   'Decoder-Section-3-DepthConcatenation'   Depth concatenation         Depth concatenati
    45   'Decoder-Section-3-Conv-1'               Convolution                 128 3×3×256 convo
    46   'Decoder-Section-3-ReLU-1'               ReLU                        ReLU
    47   'Decoder-Section-3-Conv-2'               Convolution                 128 3×3×128 convo
    48   'Decoder-Section-3-ReLU-2'               ReLU                        ReLU
    49   'Decoder-Section-4-UpConv'               Transposed Convolution      64 2×2×128 transp
    50   'Decoder-Section-4-UpReLU'               ReLU                        ReLU
    51   'Decoder-Section-4-DepthConcatenation'   Depth concatenation         Depth concatenati
    52   'Decoder-Section-4-Conv-1'               Convolution                 64 3×3×128 convol
    53   'Decoder-Section-4-ReLU-1'               ReLU                        ReLU
    54   'Decoder-Section-4-Conv-2'               Convolution                 64 3×3×64 convolu
    55   'Decoder-Section-4-ReLU-2'               ReLU                        ReLU
    56   'Final-ConvolutionLayer'                 Convolution                 18 1×1×64 convolu
    57   'Softmax-Layer'                          Softmax                     softmax
    58   'Segmentation-Layer'                     Pixel Classification Layer  Cross-entropy los

### Notice: The layer 'Decoder-Section-1-UpConv' of type 'nnet.cnn.layer.TransposedConvolution2D
### Notice: The layer 'Decoder-Section-2-UpConv' of type 'nnet.cnn.layer.TransposedConvolution2D
### Notice: The layer 'Decoder-Section-3-UpConv' of type 'nnet.cnn.layer.TransposedConvolution2D
### Notice: The layer 'Decoder-Section-4-UpConv' of type 'nnet.cnn.layer.TransposedConvolution2D
### Notice: The layer 'ImageInputLayer' with type 'nnet.cnn.layer.ImageInputLayer' is implemente
### Notice: The layer 'Softmax-Layer' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in 
### Notice: The layer 'Segmentation-Layer' with type 'nnet.cnn.layer.PixelClassificationLayer' i
### Compiling layer group: Encoder-Section-1-Conv-1>>Encoder-Section-1-ReLU-2 ...
```

```
### Compiling layer group: Encoder-Section-1-Conv-1>>Encoder-Section-1-ReLU-2 ... complete.
### Compiling layer group: Encoder-Section-1-MaxPool>>Encoder-Section-2-ReLU-2 ...
### Compiling layer group: Encoder-Section-1-MaxPool>>Encoder-Section-2-ReLU-2 ... complete.
### Compiling layer group: Encoder-Section-2-MaxPool>>Encoder-Section-3-ReLU-2 ...
### Compiling layer group: Encoder-Section-2-MaxPool>>Encoder-Section-3-ReLU-2 ... complete.
### Compiling layer group: Encoder-Section-3-MaxPool>>Encoder-Section-4-ReLU-2 ...
### Compiling layer group: Encoder-Section-3-MaxPool>>Encoder-Section-4-ReLU-2 ... complete.
### Compiling layer group: Encoder-Section-4-MaxPool>>Mid-ReLU-2 ...
### Compiling layer group: Encoder-Section-4-MaxPool>>Mid-ReLU-2 ... complete.
### Compiling layer group: Decoder-Section-1-UpConv_insertZeros ...
### Compiling layer group: Decoder-Section-1-UpConv_insertZeros ... complete.
### Compiling layer group: Decoder-Section-1-UpConv>>Decoder-Section-1-UpReLU ...
### Compiling layer group: Decoder-Section-1-UpConv>>Decoder-Section-1-UpReLU ... complete.
### Compiling layer group: Decoder-Section-1-Conv-1>>Decoder-Section-1-ReLU-2 ...
### Compiling layer group: Decoder-Section-1-Conv-1>>Decoder-Section-1-ReLU-2 ... complete.
### Compiling layer group: Decoder-Section-2-UpConv_insertZeros ...
### Compiling layer group: Decoder-Section-2-UpConv_insertZeros ... complete.
### Compiling layer group: Decoder-Section-2-UpConv>>Decoder-Section-2-UpReLU ...
### Compiling layer group: Decoder-Section-2-UpConv>>Decoder-Section-2-UpReLU ... complete.
### Compiling layer group: Decoder-Section-2-Conv-1>>Decoder-Section-2-ReLU-2 ...
### Compiling layer group: Decoder-Section-2-Conv-1>>Decoder-Section-2-ReLU-2 ... complete.
### Compiling layer group: Decoder-Section-3-UpConv_insertZeros ...
### Compiling layer group: Decoder-Section-3-UpConv_insertZeros ... complete.
### Compiling layer group: Decoder-Section-3-UpConv>>Decoder-Section-3-UpReLU ...
### Compiling layer group: Decoder-Section-3-UpConv>>Decoder-Section-3-UpReLU ... complete.
### Compiling layer group: Decoder-Section-3-Conv-1>>Decoder-Section-3-ReLU-2 ...
### Compiling layer group: Decoder-Section-3-Conv-1>>Decoder-Section-3-ReLU-2 ... complete.
### Compiling layer group: Decoder-Section-4-UpConv_insertZeros ...
### Compiling layer group: Decoder-Section-4-UpConv_insertZeros ... complete.
### Compiling layer group: Decoder-Section-4-UpConv>>Decoder-Section-4-UpReLU ...
### Compiling layer group: Decoder-Section-4-UpConv>>Decoder-Section-4-UpReLU ... complete.
### Compiling layer group: Decoder-Section-4-Conv-1>>Final-ConvolutionLayer ...
### Compiling layer group: Decoder-Section-4-Conv-1>>Final-ConvolutionLayer ... complete.

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "16.0 MB"        
    "OutputResultOffset"        "0x01000000"     "48.0 MB"        
    "SchedulerDataOffset"       "0x04000000"     "24.0 MB"        
    "SystemBufferOffset"        "0x05800000"     "28.0 MB"        
    "InstructionDataOffset"     "0x07400000"     "36.0 MB"        
    "ConvWeightDataOffset"      "0x09800000"     "540.0 MB"       
    "EndOffset"                 "0x2b400000"     "Total: 692.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
```

Program Bitstream into FPGA and Download Network Weights

To deploy the network on the Intel Arria10 SoC hardware, run the `deploy` function of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board by using the programming file. The function also loads the network weights and biases into the device. The `deploy` function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

```
deploy(hW)
```

```
### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 14-Dec-2021 23:40:29
```

Load Example Images

Extract patches for inference on FPGA by using the `extractMultispectralData` helper function and store them in MAT files. Create 20 patches of size 256-by-256-by-6 so that they form a 4-by-5 grid to represent a large input image.

```
foldername = 'TestData';
dataPath = fullfile(imageDir, 'rit18_data', 'rit18_data.mat');
extractMultispectralData(foldername, dataPath, 4, 5);
```

Load the extracted data into `testData` by using the `helperConcatenateMultispectralData` helper function. It concatenates inputs along the fourth dimension for multiframe prediction by using the `dlhdl.Workflow` object. The function is attached to the example as a supporting file.

```
testData = helperConcatenateMultispectralData(foldername);
```

Run Prediction

Execute the `predict` function of the `dlhdl.Workflow object` and display the prediction results for `testData`. Because the input is concatenated along the fourth dimension, the predictions occur simultaneously.

```
[prediction, speed] = predict(hW, testData(:,:,1:6,:), 'Profile', 'on');
```

```
### Finished writing input activations.
### Running in multi-frame mode with 20 inputs.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                  175391449                  1.16928                      20         35078
    Encoder-Section-1-Conv-1   1216888                  0.00811
    Encoder-Section-1-Conv-2   2898182                  0.01932
    Encoder-Section-1-MaxPool   5225243                  0.03483
    Encoder-Section-2-Conv-1    689902                  0.00460
    Encoder-Section-2-Conv-2   2604963                  0.01737
    Encoder-Section-2-MaxPool   4862763                  0.03242
    Encoder-Section-3-Conv-1    416523                  0.00278
    Encoder-Section-3-Conv-2   2406534                  0.01604
    Encoder-Section-3-MaxPool   6432961                  0.04289
    Encoder-Section-4-Conv-1    345878                  0.00231
```

```
    Encoder-Section-4-Conv-2   4062950                  0.02709
    Encoder-Section-4-MaxPool   7270617                  0.04847
    Mid-Conv-1           1298161                0.00865
    Mid-Conv-2          14902377                0.09935
    Decoder-Section-1-UpConv_insertZeros  14894578                  0.09930
    Decoder-Section-1-UpConv   6431694                  0.04288
    Decoder-Section-1-Conv-1   1842230                  0.01228
    Decoder-Section-1-Conv-2   9572771                  0.06382
    Decoder-Section-2-UpConv_insertZeros  10785828                  0.07191
    Decoder-Section-2-UpConv   4863034                  0.03242
    Decoder-Section-2-Conv-1   3103690                  0.02069
    Decoder-Section-2-Conv-2  10455339                  0.06970
    Decoder-Section-3-UpConv_insertZeros  10361041                  0.06907
    Decoder-Section-3-UpConv   5225305                  0.03484
    Decoder-Section-3-Conv-1   4555619                  0.03037
    Decoder-Section-3-Conv-2  11171105                  0.07447
    Decoder-Section-4-UpConv_insertZeros  11466232                  0.07644
    Decoder-Section-4-UpConv   5907915                  0.03939
    Decoder-Section-4-Conv-1   2673353                  0.01782
    Decoder-Section-4-Conv-2   1539401                  0.01026
    Final-ConvolutionLayer   5908123                  0.03939
 * The clock frequency of the DL processor is: 150MHz
```

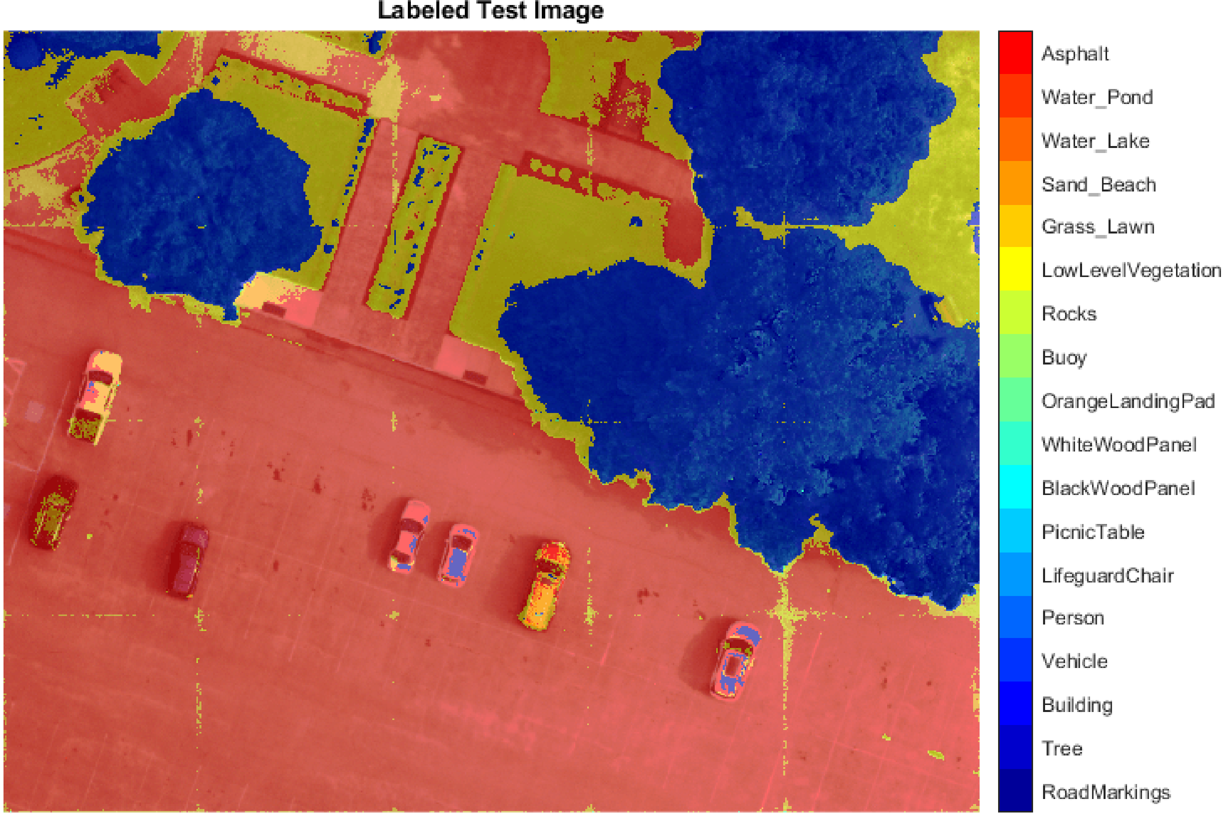
The output of `hW.predict` is of shape 256-by-256-by-18-by-20, where the outputs are concatenated along the fourth dimension. The 20 test images were created from a 1024-by-1280-by-6 section of the training data. The inputs and outputs are rearranged by using `helperArrangeInput` and `helperArrangeOutput` functions to display the prediction results. The functions are attached to the example as supporting files.

```
testImage = helperArrangeInput(testData, 4, 5);
segmentedImage = helperArrangeOutput(prediction, 4, 5);
```

Display the Prediction Results

Overlay the segmented image on the histogram-equalized RGB test image and display the prediction results.

```
classNames = [ ...
    "RoadMarkings", "Tree", "Building", "Vehicle", "Person", ...
    "LifeguardChair", "PicnicTable", "BlackWoodPanel", ...
    "WhiteWoodPanel", "OrangeLandingPad", "Buoy", "Rocks", ...
    "LowLevelVegetation", "Grass_Lawn", "Sand_Beach", ...
    "Water_Lake", "Water_Pond", "Asphalt"];

cmap = jet(numel(classNames));
N = numel(classNames);
ticks = 1/(N*2):1/N:1;

B = labeloverlay(histeq(testImage(:,:,[3 2 1])), medfilt2(segmentedImage), Transparency = 0.4, Co

figure
imshow(B);
title('Labeled Test Image')
colorbar('TickLabels', cellstr(classNames), 'Ticks', ticks, 'TickLength', 0, 'TickLabelInterprete
colormap(cmap)
```



References

[1] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918. 2017.

[2] Kemker, Ronald, Carl Salvaggio, and Christopher Kanan. "Algorithms for Semantic Segmentation of Multispectral Remote Sensing Imagery Using Deep Learning." ISPRS Journal of Photogrammetry and Remote Sensing, Deep Learning RS Data, 145 (November 1, 2018): 60-77. https://doi.org/10.1016/j.isprsjprs.2018.04.014.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `dlquantizer` | `calibrate` | `compile` | `deploy` | `predict` | `classify`

Optimize Deep Learning Processor Configuration for Network Performance

This example shows how to generate a deep learning processor configuration and estimate the performance of a pretrained network. Generate a deep learning processor configuration optimized for the target frames-per-second value of the network, then generate a custom bitstream by using the optimized processor configuration.

Load Pretrained Network and Create Processor Configuration

To load a pretrained ResNet-18 network, enter:

```
net = resnet18;
```

Create a custom deep learning processor configuration. For more information, see `dlhdl.ProcessorConfig`.

```
hPC = dlhdl.ProcessorConfig;
```

Estimate Network Performance

Establish the baseline performance of the network, by estimating the performance of the ResNet-18 network. Estimate the performance, by using the `estimatePerformance` method of the `dlhdl.ProcessorConfig` object. The method returns the estimated layer latency, network latency, and network performance in frames per second.

```
estimatePerformance(hPC,net);
```

```
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data'
### The network includes the following layers:
     1   'data'                              Image Input                 224×224×3 images with
     2   'conv1'                             2-D Convolution             64 7×7×3 convolutions w
     3   'conv1_relu'                        ReLU                        ReLU
     4   'pool1'                             2-D Max Pooling             3×3 max pooling with st
     5   'res2a_branch2a'                    2-D Convolution             64 3×3×64 convolutions
     6   'res2a_branch2a_relu'               ReLU                        ReLU
     7   'res2a_branch2b'                    2-D Convolution             64 3×3×64 convolutions
     8   'res2a'                             Addition                    Element-wise addition o
     9   'res2a_relu'                        ReLU                        ReLU
    10   'res2b_branch2a'                    2-D Convolution             64 3×3×64 convolutions
    11   'res2b_branch2a_relu'               ReLU                        ReLU
    12   'res2b_branch2b'                    2-D Convolution             64 3×3×64 convolutions
    13   'res2b'                             Addition                    Element-wise addition o
    14   'res2b_relu'                        ReLU                        ReLU
    15   'res3a_branch2a'                    2-D Convolution             128 3×3×64 convolutions
    16   'res3a_branch2a_relu'               ReLU                        ReLU
    17   'res3a_branch2b'                    2-D Convolution             128 3×3×128 convolution
    18   'res3a_branch1'                     2-D Convolution             128 1×1×64 convolutions
    19   'res3a'                             Addition                    Element-wise addition o
    20   'res3a_relu'                        ReLU                        ReLU
    21   'res3b_branch2a'                    2-D Convolution             128 3×3×128 convolution
    22   'res3b_branch2a_relu'               ReLU                        ReLU
    23   'res3b_branch2b'                    2-D Convolution             128 3×3×128 convolution
    24   'res3b'                             Addition                    Element-wise addition o
```

```
    25   'res3b_relu'                        ReLU                          ReLU
    26   'res4a_branch2a'                    2-D Convolution               256 3×3×128 convolutior
    27   'res4a_branch2a_relu'               ReLU                          ReLU
    28   'res4a_branch2b'                    2-D Convolution               256 3×3×256 convolutior
    29   'res4a_branch1'                     2-D Convolution               256 1×1×128 convolutior
    30   'res4a'                             Addition                      Element-wise addition c
    31   'res4a_relu'                        ReLU                          ReLU
    32   'res4b_branch2a'                    2-D Convolution               256 3×3×256 convolutior
    33   'res4b_branch2a_relu'               ReLU                          ReLU
    34   'res4b_branch2b'                    2-D Convolution               256 3×3×256 convolutior
    35   'res4b'                             Addition                      Element-wise addition c
    36   'res4b_relu'                        ReLU                          ReLU
    37   'res5a_branch2a'                    2-D Convolution               512 3×3×256 convolutior
    38   'res5a_branch2a_relu'               ReLU                          ReLU
    39   'res5a_branch2b'                    2-D Convolution               512 3×3×512 convolutior
    40   'res5a_branch1'                     2-D Convolution               512 1×1×256 convolutior
    41   'res5a'                             Addition                      Element-wise addition c
    42   'res5a_relu'                        ReLU                          ReLU
    43   'res5b_branch2a'                    2-D Convolution               512 3×3×512 convolutior
    44   'res5b_branch2a_relu'               ReLU                          ReLU
    45   'res5b_branch2b'                    2-D Convolution               512 3×3×512 convolutior
    46   'res5b'                             Addition                      Element-wise addition c
    47   'res5b_relu'                        ReLU                          ReLU
    48   'pool5'                             2-D Global Average Pooling    2-D global average pool
    49   'fc1000'                            Fully Connected               1000 fully connected la
    50   'prob'                              Softmax                       softmax
    51   'ClassificationLayer_predictions'   Classification Output         crossentropyex with 'te

### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification(


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    21328236                  0.10664                       1          213
    ____data_norm_add        210750                  0.00105
    ____data_norm            210750                  0.00105
    ____conv1               2164124                  0.01082
    ____pool1                515064                  0.00258
    ____res2a_branch2a       966221                  0.00483
    ____res2a_branch2b       966221                  0.00483
    ____res2a                210750                  0.00105
    ____res2b_branch2a       966221                  0.00483
    ____res2b_branch2b       966221                  0.00483
    ____res2b                210750                  0.00105
    ____res3a_branch1        540861                  0.00270
    ____res3a_branch2a       540749                  0.00270
    ____res3a_branch2b       919117                  0.00460
    ____res3a                105404                  0.00053
    ____res3b_branch2a       919117                  0.00460
    ____res3b_branch2b       919117                  0.00460
    ____res3b                105404                  0.00053
    ____res4a_branch1        503405                  0.00252
    ____res4a_branch2a       509261                  0.00255
    ____res4a_branch2b       905421                  0.00453
    ____res4a                 52724                  0.00026
```

```
    ____res4b_branch2a      905421                  0.00453
    ____res4b_branch2b      905421                  0.00453
    ____res4b                52724                  0.00026
    ____res5a_branch1       744525                  0.00372
    ____res5a_branch2a      751693                  0.00376
    ____res5a_branch2b     1415373                  0.00708
    ____res5a                26368                  0.00013
    ____res5b_branch2a     1415373                  0.00708
    ____res5b_branch2b     1415373                  0.00708
    ____res5b                26368                  0.00013
    ____pool5                54594                  0.00027
    ____fc1000              207351                  0.00104
 * The clock frequency of the DL processor is: 200MHz
```

The estimated frames-per-second performance is 9.4 frames per second. To improve the network performance, you can modify the properties of the custom deep learning processor configuration `hPC` or use the `optimizeConfigurationForNetwork` method. In this example, you use the `optimizeConfigurationForNetwork` method. To learn about modifying the properties manually, see “Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization” on page 8-17.

Generate Optimized Processor Configuration

Optimize the processor configuration by using the `optimizeConfigurationForNetwork` method. Use the optional `FramesPerSecond` name-value argument.

```
hPC_optimized = optimizeConfigurationForNetwork(hPC,net,FramesPerSecond=10);
```

```
### Optimizing processor configuration for deep learning network...


              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912            274080
                        -------------    -------------    -------------
Total                        438( 18%)        600( 66%)     270396( 99%)
ReferenceDesign                3(  1%)         78(  9%)      35000( 13%)
DL_Processor                 435( 18%)        522( 58%)     235396( 86%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
### Note: Processing module "conv" property "InputMemorySize" changed from "[227 227 3]" to "[21
### Note: Processing module "conv" property "OutputMemorySize" changed from "[227 227 3]" to "[2
### Note: Processing module "conv" property "SegmentationBlockGeneration" changed from "true" to
### Note: Processing module "fc" property "FCThreadNumber" changed from "4" to "8".
### Note: Processing module "fc" property "WeightAXIDataBitwidth" changed from "128" to "256".
### Note: Processing module "fc" property "SoftmaxBlockGeneration" changed from "false" to "true"

                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'off'
                             ConvThreadNumber: 16
                              InputMemorySize: [217 217 3]
                             OutputMemorySize: [217 217 3]
                             FeatureSizeLimit: 2048

                      Processing Module "fc"
```

```
                    ModuleGeneration: 'on'
              SoftmaxBlockGeneration: 'on'
              SigmoidBlockGeneration: 'off'
                      FCThreadNumber: 8
                     InputMemorySize: 25088
                    OutputMemorySize: 4096

          Processing Module "custom"
                    ModuleGeneration: 'on'
                            Addition: 'on'
                      Multiplication: 'on'
                            Resize2D: 'off'
                             Sigmoid: 'off'
                           TanhLayer: 'off'
                     InputMemorySize: 40
                    OutputMemorySize: 120

      Processor Top Level Properties
                      RunTimeControl: 'register'
                       RunTimeStatus: 'register'
                  InputStreamControl: 'register'
                 OutputStreamControl: 'register'
                        SetupControl: 'register'
                   ProcessorDataType: 'single'

             System Level Properties
                      TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Ki
                     TargetFrequency: 200
                       SynthesisTool: 'Xilinx Vivado'
                     ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
             SynthesisToolChipFamily: 'Zynq UltraScale+'
             SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
            SynthesisToolPackageName: ''
             SynthesisToolSpeedValue: ''

### Optimizing processor configuration for deep learning network complete.
```

Estimate performance of the ResNet-18 network by using the new optimized deep learning processor configuration.

```
estimatePerformance(hPC_optimized,net);
```

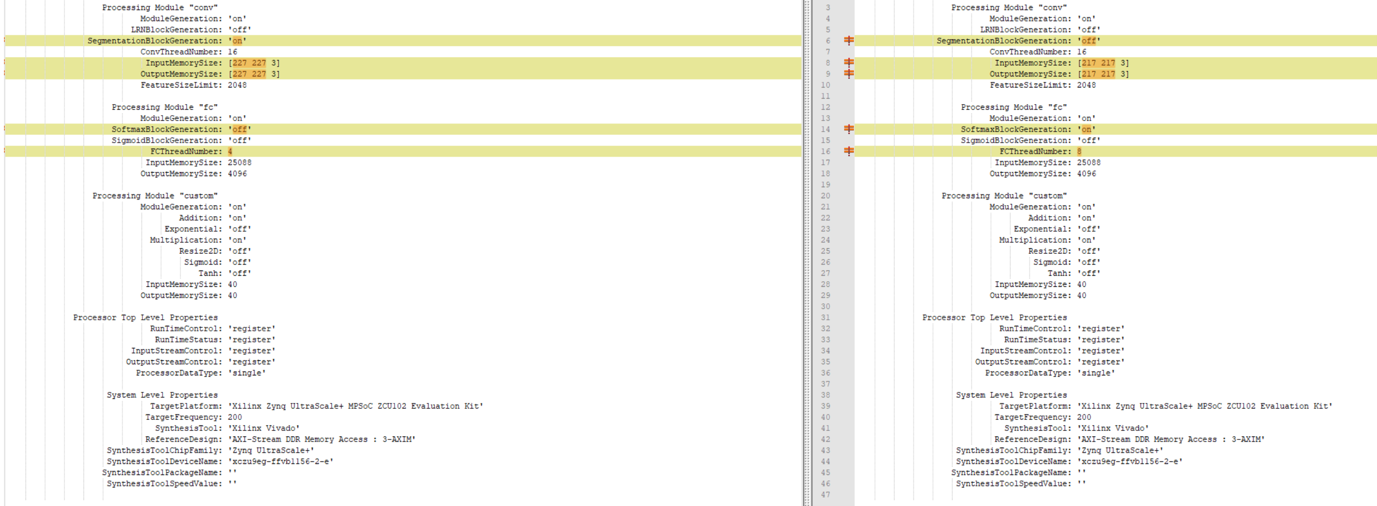
```
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data'
### The network includes the following layers:
     1   'data'                                Image Input              224×224×3 images with
     2   'conv1'                               2-D Convolution          64 7×7×3 convolutions w
     3   'conv1_relu'                          ReLU                     ReLU
     4   'pool1'                               2-D Max Pooling          3×3 max pooling with st
     5   'res2a_branch2a'                      2-D Convolution          64 3×3×64 convolutions
     6   'res2a_branch2a_relu'                 ReLU                     ReLU
     7   'res2a_branch2b'                      2-D Convolution          64 3×3×64 convolutions
     8   'res2a'                               Addition                 Element-wise addition o
     9   'res2a_relu'                          ReLU                     ReLU
    10   'res2b_branch2a'                      2-D Convolution          64 3×3×64 convolutions
    11   'res2b_branch2a_relu'                 ReLU                     ReLU
    12   'res2b_branch2b'                      2-D Convolution          64 3×3×64 convolutions
    13   'res2b'                               Addition                 Element-wise addition o
```

```
    14   'res2b_relu'                        ReLU                         ReLU
    15   'res3a_branch2a'                    2-D Convolution              128 3×3×64 convolutions
    16   'res3a_branch2a_relu'               ReLU                         ReLU
    17   'res3a_branch2b'                    2-D Convolution              128 3×3×128 convolution
    18   'res3a_branch1'                     2-D Convolution              128 1×1×64 convolutions
    19   'res3a'                             Addition                     Element-wise addition o
    20   'res3a_relu'                        ReLU                         ReLU
    21   'res3b_branch2a'                    2-D Convolution              128 3×3×128 convolution
    22   'res3b_branch2a_relu'               ReLU                         ReLU
    23   'res3b_branch2b'                    2-D Convolution              128 3×3×128 convolution
    24   'res3b'                             Addition                     Element-wise addition o
    25   'res3b_relu'                        ReLU                         ReLU
    26   'res4a_branch2a'                    2-D Convolution              256 3×3×128 convolution
    27   'res4a_branch2a_relu'               ReLU                         ReLU
    28   'res4a_branch2b'                    2-D Convolution              256 3×3×256 convolution
    29   'res4a_branch1'                     2-D Convolution              256 1×1×128 convolution
    30   'res4a'                             Addition                     Element-wise addition o
    31   'res4a_relu'                        ReLU                         ReLU
    32   'res4b_branch2a'                    2-D Convolution              256 3×3×256 convolution
    33   'res4b_branch2a_relu'               ReLU                         ReLU
    34   'res4b_branch2b'                    2-D Convolution              256 3×3×256 convolution
    35   'res4b'                             Addition                     Element-wise addition o
    36   'res4b_relu'                        ReLU                         ReLU
    37   'res5a_branch2a'                    2-D Convolution              512 3×3×256 convolution
    38   'res5a_branch2a_relu'               ReLU                         ReLU
    39   'res5a_branch2b'                    2-D Convolution              512 3×3×512 convolution
    40   'res5a_branch1'                     2-D Convolution              512 1×1×256 convolution
    41   'res5a'                             Addition                     Element-wise addition o
    42   'res5a_relu'                        ReLU                         ReLU
    43   'res5b_branch2a'                    2-D Convolution              512 3×3×512 convolution
    44   'res5b_branch2a_relu'               ReLU                         ReLU
    45   'res5b_branch2b'                    2-D Convolution              512 3×3×512 convolution
    46   'res5b'                             Addition                     Element-wise addition o
    47   'res5b_relu'                        ReLU                         ReLU
    48   'pool5'                             2-D Global Average Pooling   2-D global average pool
    49   'fc1000'                            Fully Connected              1000 fully connected la
    50   'prob'                              Softmax                      softmax
    51   'ClassificationLayer_predictions'   Classification Output        crossentropyex with 'te

### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   19966252                  0.09983                       1          199
    ____data_norm_add       210750                  0.00105
    ____data_norm           210750                  0.00105
    ____conv1              2224339                  0.01112
    ____pool1               632402                  0.00316
    ____res2a_branch2a     1038708                  0.00519
    ____res2a_branch2b     1038708                  0.00519
    ____res2a               210750                  0.00105
    ____res2b_branch2a     1038708                  0.00519
    ____res2b_branch2b     1038708                  0.00519
    ____res2b               210750                  0.00105
    ____res3a_branch1       630228                  0.00315
```

```
    ____res3a_branch2a      625092                  0.00313
    ____res3a_branch2b      919117                  0.00460
    ____res3a               105404                  0.00053
    ____res3b_branch2a      919117                  0.00460
    ____res3b_branch2b      919117                  0.00460
    ____res3b               105404                  0.00053
    ____res4a_branch1       503405                  0.00252
    ____res4a_branch2a      509261                  0.00255
    ____res4a_branch2b      905421                  0.00453
    ____res4a                52724                  0.00026
    ____res4b_branch2a      905421                  0.00453
    ____res4b_branch2b      905421                  0.00453
    ____res4b                52724                  0.00026
    ____res5a_branch1       506957                  0.00253
    ____res5a_branch2a      514125                  0.00257
    ____res5a_branch2b      940237                  0.00470
    ____res5a                26368                  0.00013
    ____res5b_branch2a      940237                  0.00470
    ____res5b_branch2b      940237                  0.00470
    ____res5b                26368                  0.00013
    ____pool5                54594                  0.00027
    ____fc1000              103438                  0.00052
    ____prob                  1262                  0.00001
 * The clock frequency of the DL processor is: 200MHz
```

The new estimated frames per second performance is 10 frames per second.

This image shows the comparison between the original processor configuration and the optimized processor configuration:



The optimized processor configuration has:

- `SegmentationBlockGeneration` turned off.
- `InputMemorySize` and `OutputMemorySize` reduced to [217 217 3].
- `SoftMaxBlockGeneration` turned on.
- `FCThreadNumber` increased to 8.

Generate Optimized Custom Bitstream

Use the optimized custom deep learning processor configuration to build and generate a custom bitstream. Use the custom bitstream to deploy the pretrained ResNet-18 network to your target FPGA board.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.bat
dlhdl.buildProcessor(hPC_optimized);
```

See Also

dlhdl.ProcessorConfig | estimatePerformance

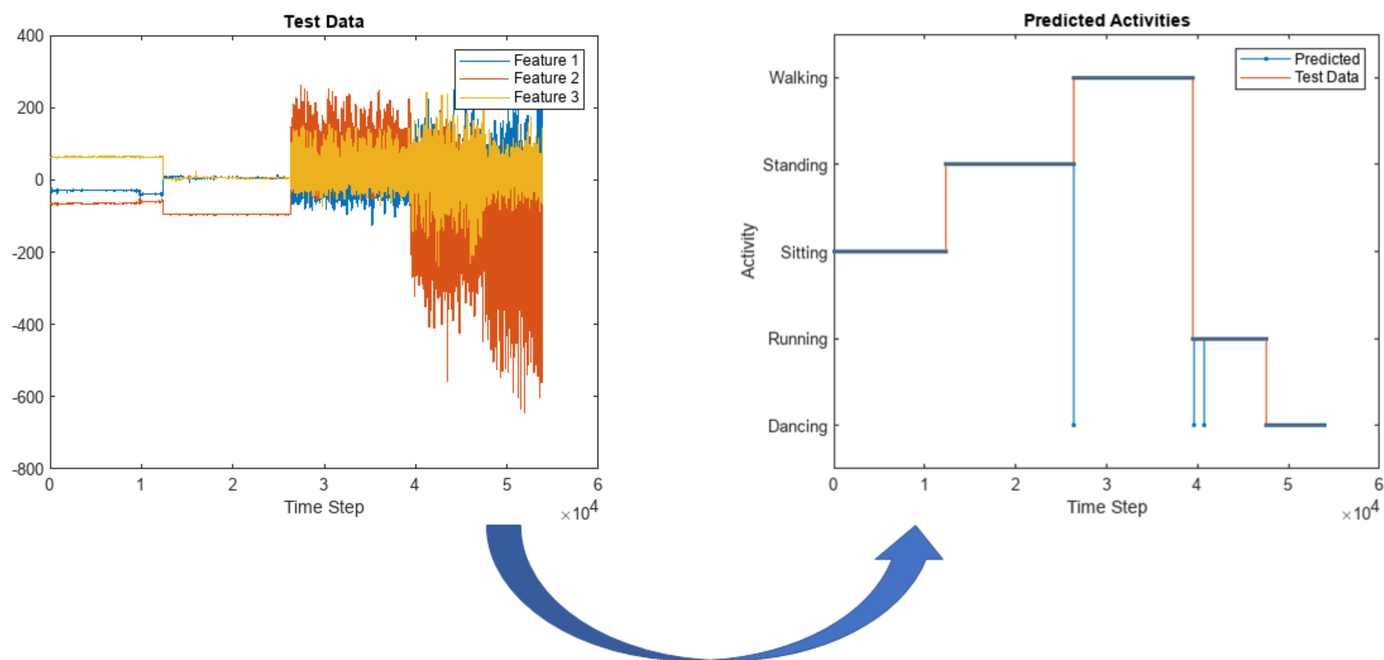
More About

- “Generate Custom Bitstream” on page 9-2
- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10

Run Sequence-to-Sequence Classification on FPGAs by Using Deep Learning HDL Toolbox

This example shows how to create, compile, and deploy a long short-term memory (LSTM) network trained on accelerometer data from human movement by using the Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC. Use the deployed network to classify human activity based on sequence input data. Use MATLAB® to retrieve the prediction results from the target device.

The network attached to this example was trained using the “Sequence-to-Sequence Classification Using Deep Learning”. This example uses sensor data obtained from a smartphone worn on the body. This example deploys an LSTM network trained to recognize the activity of the wearer given time series data that represents accelerometer readings in three different directions. The graphs below show the raw data for these accelerometer readings over time and the resulting classifications. The training data contains time series data for seven people. Each sequence has three features and varies in length. The data set contains six training observations and one test observation.



Prerequisites

- Xilinx® Zynq® Ultrascale+™ ZCU102 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Load the Pretrained Network

To load the pretrained human body movement network, enter:

```
load SequenceToSequenceClassification
```

View the layers of the network by using the `analyzeNetwork` function. The function returns a graphical representation of the network and detailed parameter settings of the layers in the network.

```
analyzeNetwork(net)
```

ANALYSIS RESULT

	Name	Type	Activations	Learnable Prope...	States
1	sequenceinput Sequence input with 3 dimensions	Sequence Input	3(C) × 1(B) × 1(T)	-	-
2	lstm LSTM with 200 hidden units	LSTM	200(C) × 1(B) × 1(T)	InputWeigh… 800 ×… RecurrentW… 800 ×… Bias 800 ×…	HiddenSta… 200 ×… CellState 200 ×…
3	fc 5 fully connected layer	Fully Connected	5(C) × 1(B) × 1(T)	Weights 5 × 200 Bias 5 × 1	-
4	softmax softmax	Softmax	5(C) × 1(B) × 1(T)	-	-
5	classoutput crossentropyex with 'Dancing' and 4 oth…	Classification Output	5(C) × 1(B) × 1(T)	-	-

Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Specify that the interface is for a Xilinx board with an Ethernet interface.

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

To use the JTAG interface, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado tool path, enter:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and FPGA board. In this example the target FPGA board is the Xilinx ZCU102 SOC board. The bitstream uses a single data type.

```
hW = dlhdl.Workflow('network', net, 'Bitstream', 'zcu102_lstm_single','Target',hTarget);
```

To run the example in a Xilinx ZC706 board, enter:

```
hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'zc706_lstm_single','Target',hTarget);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment. The total number of frames exceeds the default

value of 30. Set the `InputFrameNumberLimit` name-value argument to `10000` to run predictions in chunks of 10,000 frames to prevent timeouts.

```
dn = compile(hW,'InputFrameNumberLimit',10000)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_lstm_single.
### The network includes the following layers:
     1   'sequenceinput'   Sequence Input          Sequence input with 3 dimensions
     2   'lstm'            LSTM                    LSTM with 200 hidden units
     3   'fc'              Fully Connected         5 fully connected layer
     4   'softmax'         Softmax                 softmax
     5   'classoutput'     Classification Output   crossentropyex with 'Dancing' and 4 other clas

### Notice: The layer 'sequenceinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented i
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple
### Compiling layer group: lstm.wi ...
### Compiling layer group: lstm.wi ... complete.
### Compiling layer group: lstm.wo ...
### Compiling layer group: lstm.wo ... complete.
### Compiling layer group: lstm.wg ...
### Compiling layer group: lstm.wg ... complete.
### Compiling layer group: lstm.wf ...
### Compiling layer group: lstm.wf ... complete.
### Compiling layer group: fc ...
### Compiling layer group: fc ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "20.0 MB"
    "InstructionDataOffset"     "0x02000000"     "4.0 MB"
    "FCWeightDataOffset"        "0x02400000"     "4.0 MB"
    "EndOffset"                 "0x02800000"     "Total: 40.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` method of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board and download the network weights and biases. The `deploy` function starts programming the FPGA device and displays progress messages, and the required time to deploy the network.

```
deploy(hW)
```

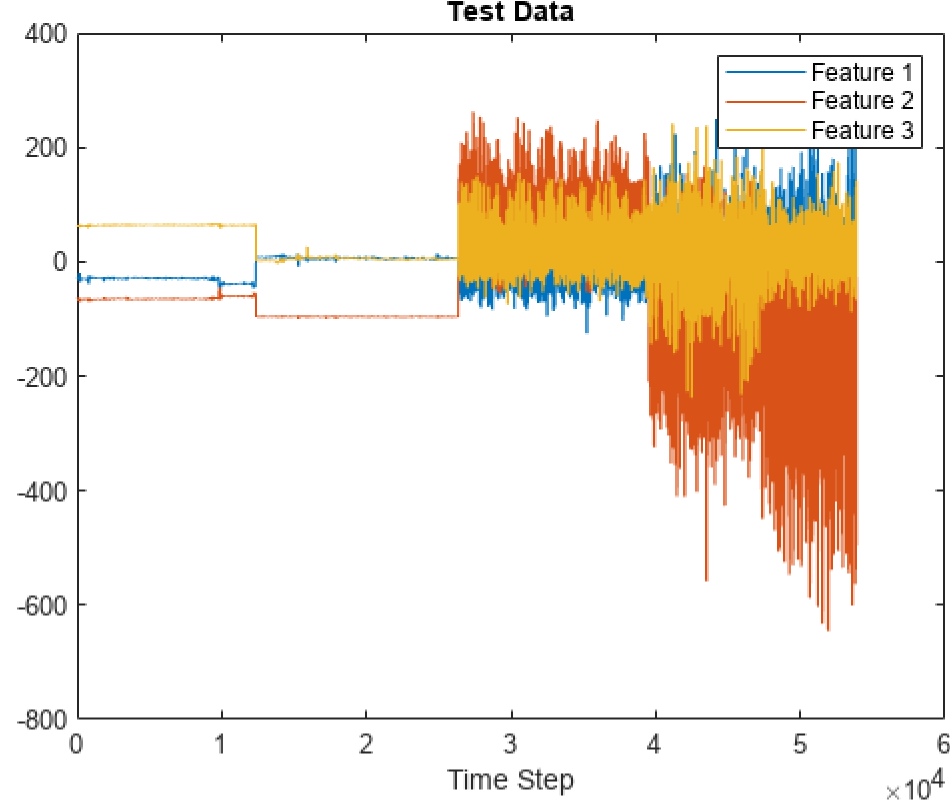
```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Resetting network state.
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 30-Jun-2022 13:41:44
```

Load Human Activity Test Data

Load the test data and classify the activity at each time step. Each sequence has three features and varies in length. The three features correspond to the accelerometer readings in three different directions.

Load the human activity test data. `XTest` contains a single sequence of dimension 3. `YTest` contains a sequence of categorical labels that correspond to the activity at each time step.

```
load HumanActivityTest
numFeatures = 3;
figure
plot(XTest{1}')
xlabel("Time Step")
legend("Feature " + (1:numFeatures))
title("Test Data")
```



Run the Prediction

Classify the test data by using the `classify` function.

```
YPred = classify(hW.Network, XTest{1});
```

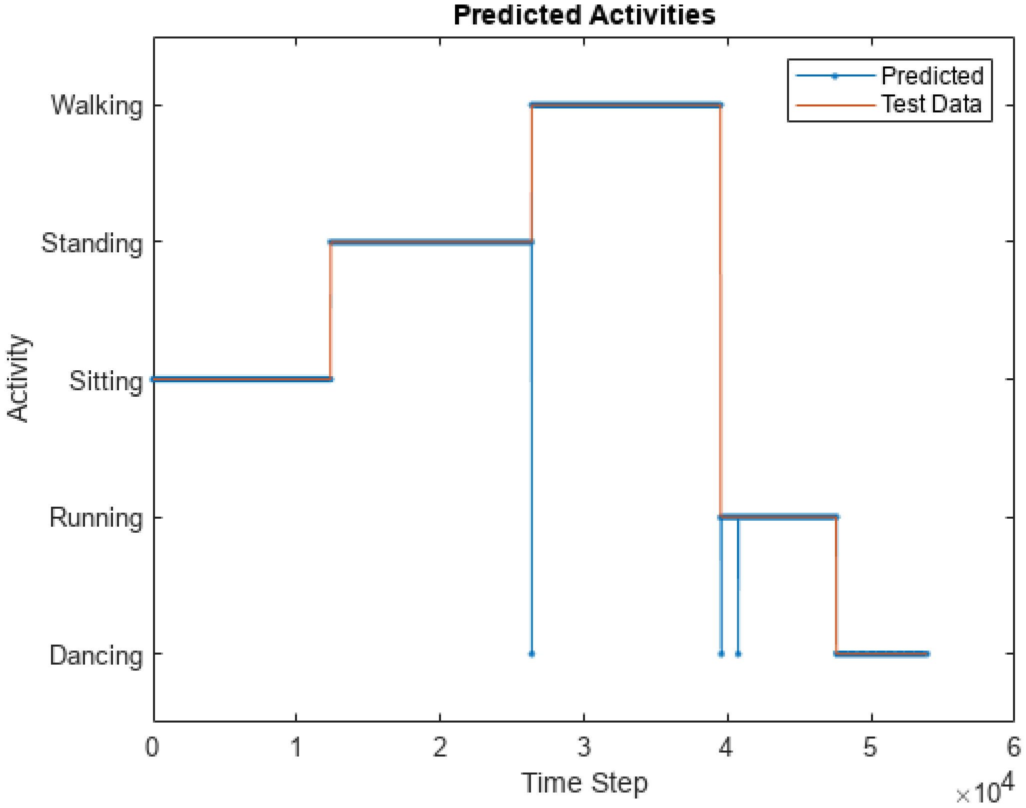
Calculate the accuracy of the prediction.

```
acc = sum(YPred == YTest{1})./numel(YTest{1})
```

```
acc = 0.9995
```

Compare the predictions with the test data by using a plot.

```
figure
plot(YPred,'.-')
hold on
plot(YTest{1})
hold off

xlabel("Time Step")
ylabel("Activity")
title("Predicted Activities")
legend(["Predicted" "Test Data"])
```



Compare this graph to the output of the `predict` method.

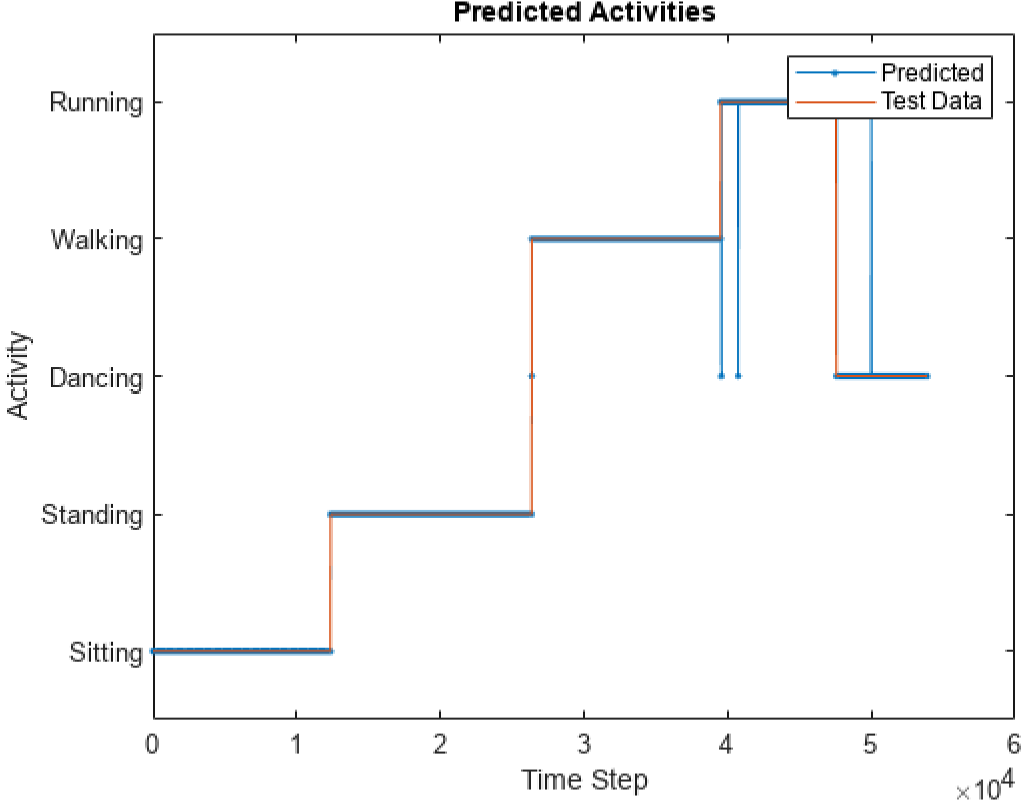
Run the `predict` method of the `dlhdl.Workflow` object, to retrieve the hardware prediction results.

```
predictions = hW.predict(XTest{1}(:,1:10000));
predictions = horzcat(predictions, hW.predict(XTest{1}(:,10001:20000)));
predictions = horzcat(predictions, hW.predict(XTest{1}(:,20001:30000)));
predictions = horzcat(predictions, hW.predict(XTest{1}(:,30001:40000)));
```

```
predictions = horzcat(predictions, hW.predict(XTest{1}(:,40001:50000)));
predictions = horzcat(predictions, hW.predict(XTest{1}(:,50001:end)));
save("hardwarepredictions.mat","predictions")
indices = [];
actions = [];
for x = 1:length(YPred)
    [r,i] = max(predictions(:,x));
    indices = [indices i];
    switch i
        case 1
            actions = [actions categorical("Dancing")];
        case 2
            actions = [actions categorical("Running")];
        case 5
            actions = [actions categorical("Walking")];
        case 4
            actions = [actions categorical("Standing")];
        case 3
            actions = [actions categorical("Sitting")];
    end
end
```

Plot the comparison between the FPGA board predictions and test data.

```
figure
plot(actions,'.-')
hold on
plot(YTest{1})
hold off

xlabel("Time Step")
ylabel("Activity")
title("Predicted Activities")
legend(["Predicted" "Test Data"])
```



The hardware-predicted activities are similar to the activities classified by the `classify` function.

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `compile` | `deploy` | `predict` | `predictAndUpdateState` | `resetState`

More About

- "Support for Long Short-Term Memory Networks" on page 13-2
- "How Deep Learning HDL Toolbox Compiles the LSTM Layer" on page 13-5

Generate Word-By-Word Text on FPGAs by Using Deep Learning HDL Toolbox

This example shows how to deploy a long short-term memory (LSTM) network to generate text word-by-word on an FPGA by using the Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC. Use MATLAB® to retrieve the prediction results from the target device.

This example reads text from the Project Gutenberg website, parses the HTML code to extract the relevant text, then uses a custom mini-batch datastore, `documentGenerationDatastore` to input the documents to the network as mini-batches of sequence data. The datastore converts documents to sequences of numeric word indices. The deep learning network is an LSTM network that contains a word embedding layer.

To train a deep learning network for word-by-word text generation, train a sequence-to-sequence LSTM network to predict the next word in a sequence of words. To train the network to predict the next word, specify the responses as the input sequences shifted by one time step. This example uses the pretrained network from the “Word-By-Word Text Generation Using Deep Learning” example.

Prerequisites

- Xilinx® Zynq® Ultrascale+™ ZCU102 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Text Analytics Toolbox™

Load Training Data

Load the training data. Read the HTML code from Alice's Adventures in Wonderland by Lewis Carroll from Project Gutenberg.

```
url = "https://www.gutenberg.org/files/11/11-h/11-h.htm";
code = webread(url);
```

Parse HTML Code

The HTML code contains the relevant text inside paragraph elements. Extract the relevant text by parsing the HTML code using `htmlTree` and then finding the elements with element name `"p"`.

```
tree = htmlTree(code);
selector = "p";
subtrees = findElement(tree,selector);
```

Extract the text data from the HTML subtrees by using `extractHTMLText` and view the first 10 paragraphs.

```
textData = extractHTMLText(subtrees);
textData(1:10)
```

```
ans = 10×1 string
    "Alice was beginning to get very tired of sitting by her sister on the bank, and of having no
    "So she was considering in her own mind (as well as she could, for the hot day made her feel
    "There was nothing so very remarkable in that; nor did Alice think it so very much out of the
```

```
    "In another moment down went Alice after it, never once considering how in the world she was
    "The rabbit-hole went straight on like a tunnel for some way, and then dipped suddenly down,
    "Either the well was very deep, or she fell very slowly, for she had plenty of time as she we
    "“Well!” thought Alice to herself, “after such a fall as this, I shall think nothing of tumbl
    "Down, down, down. Would the fall never come to an end? “I wonder how many miles I’ve fallen
    "Presently she began again. “I wonder if I shall fall right through the earth! How funny it’l
    "Down, down, down. There was nothing else to do, so Alice soon began talking again. “Dinah’l
```

Remove the empty paragraphs and view the first 10 remaining paragraphs.

```
textData(textData == "") = [];
textData(1:10)
```

```
ans = 10×1 string
    "Alice was beginning to get very tired of sitting by her sister on the bank, and of having no
    "So she was considering in her own mind (as well as she could, for the hot day made her feel
    "There was nothing so very remarkable in that; nor did Alice think it so very much out of the
    "In another moment down went Alice after it, never once considering how in the world she was
    "The rabbit-hole went straight on like a tunnel for some way, and then dipped suddenly down,
    "Either the well was very deep, or she fell very slowly, for she had plenty of time as she we
    "“Well!” thought Alice to herself, “after such a fall as this, I shall think nothing of tumbl
    "Down, down, down. Would the fall never come to an end? “I wonder how many miles I’ve fallen
    "Presently she began again. “I wonder if I shall fall right through the earth! How funny it’l
    "Down, down, down. There was nothing else to do, so Alice soon began talking again. “Dinah’l
```

Visualize the text data in a word cloud.

```
figure
wordcloud(textData);
title("Alice's Adventures in Wonderland")
```

Alice's Adventures in Wonderland

Prepare Data for Training

Create a datastore that contains the data for training by using the `documentGenerationDatastore` function. For the predictors, the function converts the documents into sequences of word indices by using a word encoding. The first word index for each document corresponds to a "start of text" token. The "start of text" token is given by the string `"startOfText"`. For the responses, the datastore returns categorical sequences of the words shifted by one.

Tokenize the text data by using the `tokenizedDocument` function..

```
documents = tokenizedDocument(textData);
```

Create a document generation datastore using the tokenized documents.

```
ds = documentGenerationDatastore(documents);
```

To reduce the amount of padding added to the sequences, sort the documents in the datastore by sequence length.

```
ds = sort(ds);
```

Load the Pretrained Network

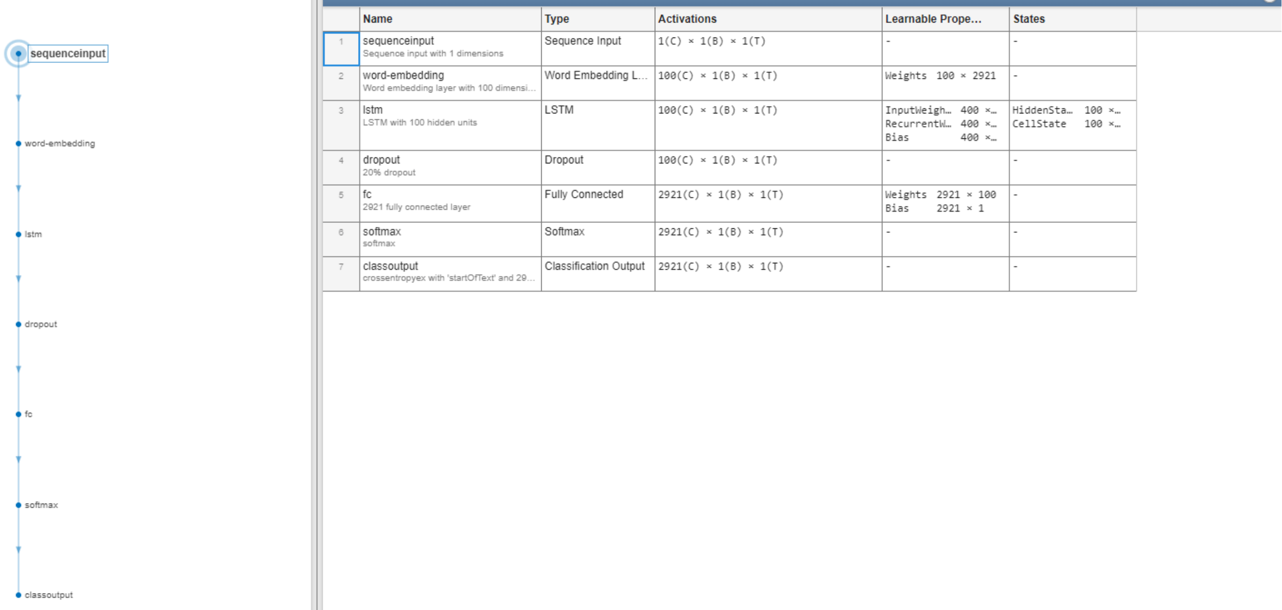
To load the LSTM network, enter:

```
if ~isfile('WordByWordTextgenerationModel.zip')
        url = 'https://ssd.mathworks.com/supportfiles/dlhdl/WordByWordTextGenerationModel.zip';
```

```
        websave('WordByWordTextGenerationModel.zip',url);
end
unzip('WordByWordTextGenerationModel.zip')
load WordByWordTextgenerationModel.mat
```

View the layers of the network by using the `analyzeNetwork` function. The function returns a graphical representation of the network and detailed parameter settings of the layers in the network.

```
analyzeNetwork(net)
```



	Name	Type	Activations	Learnable Prope...	States
1	sequenceinput Sequence input with 1 dimensions	Sequence Input	1(C) × 1(B) × 1(T)	-	-
2	word-embedding Word embedding layer with 100 dimensi...	Word Embedding L...	100(C) × 1(B) × 1(T)	Weights 100 × 2921	-
3	lstm LSTM with 100 hidden units	LSTM	100(C) × 1(B) × 1(T)	InputWeigh… 400 ×… RecurrentW… 400 ×… Bias 400 ×…	HiddenSta… 100 ×… CellState 100 ×…
4	dropout 20% dropout	Dropout	100(C) × 1(B) × 1(T)	-	-
5	fc 2921 fully connected layer	Fully Connected	2921(C) × 1(B) × 1(T)	Weights 2921 × 100 Bias 2921 × 1	-
6	softmax softmax	Softmax	2921(C) × 1(B) × 1(T)	-	-
7	classoutput crossentropyex with 'startOfText' and 29...	Classification Output	2921(C) × 1(B) × 1(T)	-	-

Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Specify that the interface is for a Xilinx board with an Ethernet interface.

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

To use the JTAG interface, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado tool path, enter:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
hTarget = dlhdl.Target('Xilinx','Interface','JTAG');
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and the bitstream name. Ensure that the bitstream name matches the data type and the FPGA board. In this example, the target FPGA board is the Xilinx ZCU102 SOC board. The bitstream uses a single data type.

```
hW = dlhdl.Workflow('network', net, 'Bitstream', 'zcu102_lstm_single','Target',hTarget);
```

Tu run the example on the Xilinx ZC706 board, enter:

```
hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'zc706_lstm_single','Target',hTarget);
```

Compile the LSTM Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment.

```
dn = compile(hW)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_lstm_single.
### The network includes the following layers:
     1   'sequenceinput'    Sequence Input          Sequence input with 1 dimensions
     2   'word-embedding'   Word Embedding Layer    Word embedding layer with 100 dimensions and
     3   'lstm'             LSTM                    LSTM with 100 hidden units
     4   'fc'               Fully Connected         2921 fully connected layer
     5   'softmax'          Softmax                 softmax
     6   'classoutput'      Classification Output   crossentropyex with 'startOfText' and 2920 o

### Notice: The layer 'sequenceinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented i
### Notice: The layer 'word-embedding' with type 'dnnfpga.layer.wordEmbeddingLayerDLP' is impleme
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwa
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple
### Compiling layer group: lstm.wi ...
### Compiling layer group: lstm.wi ... complete.
### Compiling layer group: lstm.wo ...
### Compiling layer group: lstm.wo ... complete.
### Compiling layer group: lstm.wg ...
### Compiling layer group: lstm.wg ... complete.
### Compiling layer group: lstm.wf ...
### Compiling layer group: lstm.wf ... complete.
### Compiling layer group: fc ...
### Compiling layer group: fc ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "20.0 MB"
    "InstructionDataOffset"     "0x02000000"     "4.0 MB"
    "FCWeightDataOffset"        "0x02400000"     "4.0 MB"
    "EndOffset"                 "0x02800000"     "Total: 40.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` function of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board by using the programming file and downloads the network weights and biases. The `deploy` function starts programming the FPGA device and displays progress messages, and the time it takes to deploy the network.

```
deploy(hW)
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_lstm_single.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_lstm_single.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Resetting network state.
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 22-Sep-2022 15:09:17
```

Generate New Text Using Hardware

Generate the first word of the text by sampling a word from a probability distribution according to first words of the text in the training data. Generate the remaining words by using the deployed LSTM network to predict the next time step using the current sequence of generated text. Keep generating words one-by-one until the network predicts the "end of text" word.

To make the first prediction using the network, input the index that represents the "start of text" token. Find the index by using the `word2ind` function with the word encoding used by the document datastore.

```
enc = ds.Encoding;
wordIndex = word2ind(enc,"startOfText");
```

For the remaining predictions, sample the next word according to the prediction scores of the network. The prediction scores represent the probability distribution of the next word. Sample the words from the vocabulary given by the class names of the output layer of the network.

```
vocabulary = string(net.Layers(end).Classes);
```

Make predictions word by word using the `predictAndUpdateState` function. For each prediction, input the index of the previous word. Stop predicting when the network predicts the end of text word or when the generated text is 200 characters long. To generate multiple pieces of text, reset the network state between generations by using the `resetState` function.

```
generatedText = "";
maxLength = 200;
```

```
resetState(hW);
while strlength(generatedText) < maxLength
    [wordScores,~] = predictAndUpdateState(hW,wordIndex);
    newWord = datasample(vocabulary,1,'Weights',wordScores);

    if newWord == "EndOfText"
        break
    end

    generatedText = generatedText + " " + newWord;
    wordIndex = word2ind(enc,newWord);
end
```

```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
```

```
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
```

The generation process introduces whitespace characters between each prediction, which means that some punctuation characters appear with unnecessary spaces before and after. Reconstruct the generated text by removing the spaces before and after the appropriate punctuation characters.

Remove the spaces that appear before the specified punctuation characters.

```
punctuationCharacters = ["." "," "’" ")" ":" "?" "!"];
generatedText = replace(generatedText," " + punctuationCharacters,punctuationCharacters);
```

Remove the spaces that appear after the specified punctuation characters.

```
punctuationCharacters = ["(" "‘"];
generatedText = replace(generatedText,punctuationCharacters + " ",punctuationCharacters)
```

```
generatedText = 
" The Mock Turtle sighed deeply, and began watching running from him. “ Off with her head! ” Thos
```

See Also

dlhdl.Workflow | dlhdl.Target | compile | deploy | predict | predictAndUpdateState | resetState

More About

- “Support for Long Short-Term Memory Networks” on page 13-2
- “How Deep Learning HDL Toolbox Compiles the LSTM Layer” on page 13-5

Run Sequence Forecasting on FPGA by Using Deep Learning HDL Toolbox

This example shows how to create, compile, and deploy a long short-term memory (LSTM) network trained on waveform data by using the Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC. Use the deployed network to predict future values by using open-loop and closed-loop forecasting. Use MATLAB® to retrieve the prediction results from the target device.

Waveform Data Network

The network attached to this example was trained using the "Time Series Forecasting Using Deep Learning". This example uses the `WaveformData.mat` data set, which contains 2000 synthetically generated waveforms of varying lengths with three channels. This example uses a trained LSTM network to forecast future values of the waveforms given the values from the previous time steps using both closed loop and open loop forecasting.

Prerequisites

- Xilinx® Zynq® Ultrascale+™ ZCU102 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

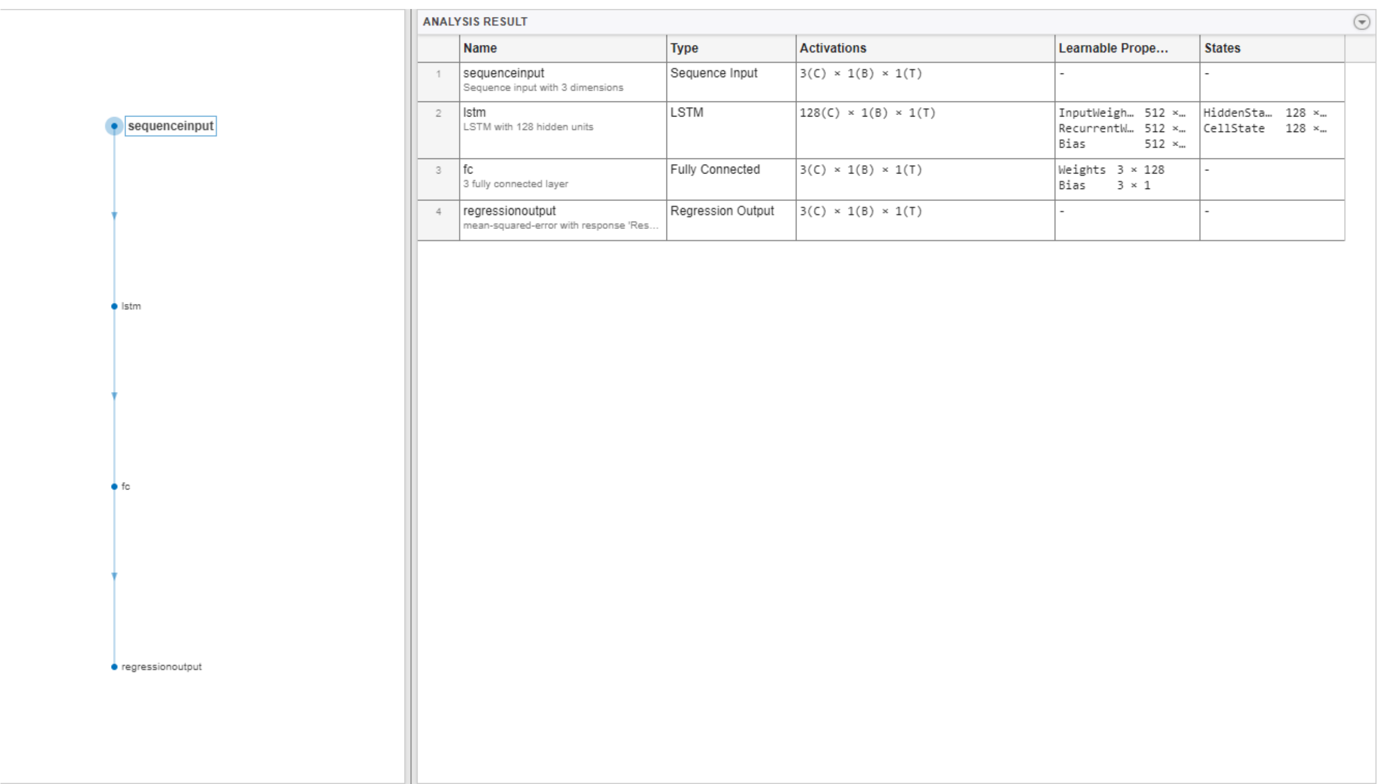
Load the Pretrained Network

To load the LSTM network enter:

```
load WaveformForcastingNet
```

Use the `analyzeNetwork` function to obtain information about the network layers. the function returns a graphical representation of the network that contains detailed parameter information for every layer in the network.

```
analyzeNetwork(net)
```



Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Specify that the interface is for a Xilinx board with an Ethernet interface.

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

To use the JTAG interface, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
hTarget = dlhdl.Target('Xilinx','Interface','JTAG');
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and the bitstream name. Ensure that the bitstream name matches the data type and the FPGA board. In this example the target FPGA board is the Xilinx ZCU102 SOC board. The bitstream uses a single data type.

```
hW = dlhdl.Workflow('network', net, 'Bitstream', 'zcu102_lstm_single','Target',hTarget);
```

Tu run the example on the Xilinx ZC706 board, enter:

```
hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'zc706_lstm_single','Target',hTarget);
```

Compile the LSTM Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment. The total number of frames exceeds the default value of 30. Set the `InputFrameNumberLimit` name-value argument to `1000` to run predictions in chunks of 1000 frames to prevent timeouts.

```
dn = compile(hW,'InputFrameNumberLimit',1000)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_lstm_single.
### The network includes the following layers:
     1   'sequenceinput'      Sequence Input      Sequence input with 3 dimensions             (S
     2   'lstm'               LSTM                LSTM with 128 hidden units                   (H
     3   'fc'                 Fully Connected     3 fully connected layer                      (H
     4   'regressionoutput'   Regression Output   mean-squared-error with response 'Response'  (S

### Notice: The layer 'sequenceinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented i
### Notice: The layer 'regressionoutput' with type 'nnet.cnn.layer.RegressionOutputLayer' is imp
### Compiling layer group: lstm.wi ...
### Compiling layer group: lstm.wi ... complete.
### Compiling layer group: lstm.wo ...
### Compiling layer group: lstm.wo ... complete.
### Compiling layer group: lstm.wg ...
### Compiling layer group: lstm.wg ... complete.
### Compiling layer group: lstm.wf ...
### Compiling layer group: lstm.wf ... complete.
### Compiling layer group: fc ...
### Compiling layer group: fc ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "20.0 MB"
    "InstructionDataOffset"     "0x02000000"     "4.0 MB"
    "FCWeightDataOffset"        "0x02400000"     "4.0 MB"
    "EndOffset"                 "0x02800000"     "Total: 40.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` function of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The

deploy function starts programming the FPGA device and displays progress messages, and the required time to deploy the network.

```
deploy(hW)
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Resetting network state.
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 09-Nov-2022 09:35:06
```

Test Network

Prepare the test data for prediction. Normalize the test data using the statistics calculated from the training data. To forecast the values of future time steps of a sequence, specify the targets as the test sequences with values shifted by one time step. In other words, at each time step of the input sequence, the LSTM network learns to predict the value of the next time step. The predictors as the test sequences without the final time step.

```
load Waveformdata
numChannels = size(data{1},1);
numObservations = numel(data);

idxTrain = 1:floor(0.9*numObservations);
idxTest = floor(0.9*numObservations)+1:numObservations;
dataTrain = data(idxTrain);
dataTest = data(idxTest);

for n = 1:numel(dataTrain)
    X = dataTrain{n};
    XTrain{n} = X(:,1:end-1);
    TTrain{n} = X(:,2:end);
end

muX = mean(cat(2,XTrain{:}),2);
sigmaX = std(cat(2,XTrain{:}),0,2);
muT = mean(cat(2,TTrain{:}),2);
sigmaT = std(cat(2,TTrain{:}),0,2);

for n = 1:size(dataTest,1)
    X = dataTest{n};
    XTest{n} = (X(:,1:end-1) - muX) ./ sigmaX;
    TTest{n} = (X(:,2:end) - muT) ./ sigmaT;
end
```

Make predictions using the test data.

```
YTest = hW.predict(XTest{1},Profile ='on');
```

```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 115.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------              ---------        --
Network                       38755                  0.00018                      115           44
```

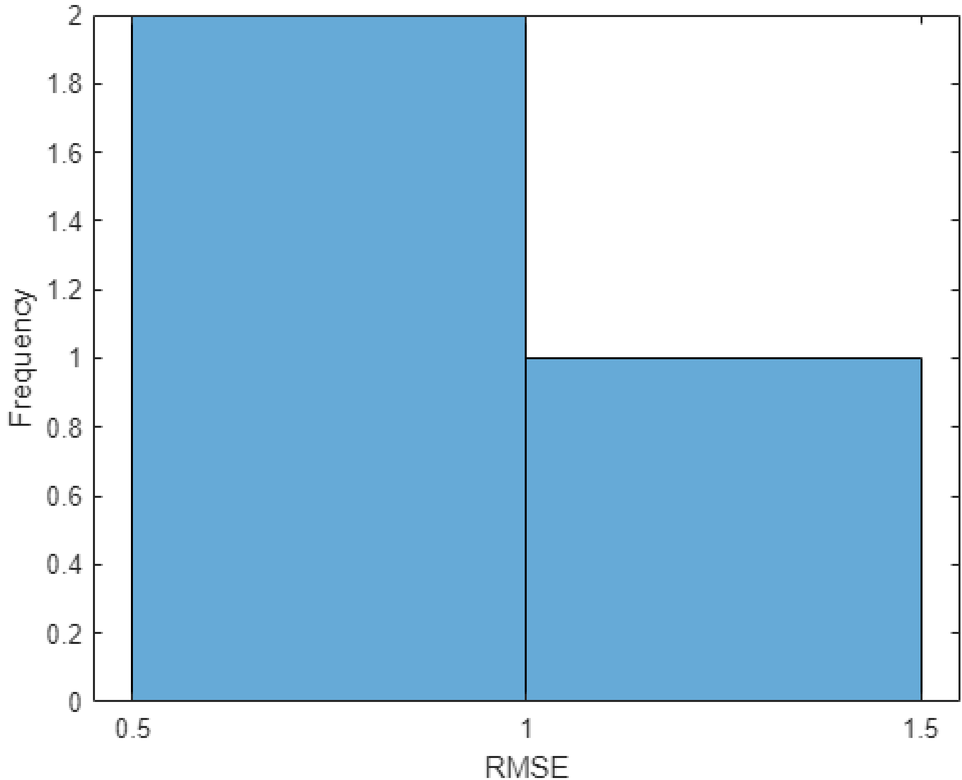
```
    memSeparator_0              88                  0.00000
    lstm.wi                   7478                  0.00003
    lstm.wo                   7549                  0.00003
    lstm.wg                   7619                  0.00003
    lstm.wf                   7519                  0.00003
    lstm.sigmoid_1             222                  0.00000
    lstm.sigmoid_3             224                  0.00000
    lstm.tanh_1                204                  0.00000
    lstm.sigmoid_2             224                  0.00000
    lstm.multiplication_2       294                  0.00000
    lstm.multiplication_1       314                  0.00000
    lstm.c_add                 308                  0.00000
    lstm.tanh_2                229                  0.00000
    lstm.multiplication_3       287                  0.00000
    fc                        6196                  0.00003
 * The clock frequency of the DL processor is: 220MHz
```

To evaluate the accuracy, calculate the root mean squared error (RMSE) between the predictions and the target for each test sequence.

```
for i = 1:size(YTest,1)
    rmse(i) = sqrt(mean((YTest(i) - TTest{1}(i)).^2,"all"));
end
```

Visualize the errors in a histogram. Lower values indicate greater accuracy.

```
figure
histogram(rmse)
xlabel("RMSE")
ylabel("Frequency")
```



Calculate the mean RMSE over all test observations.

```
mean(rmse)
```

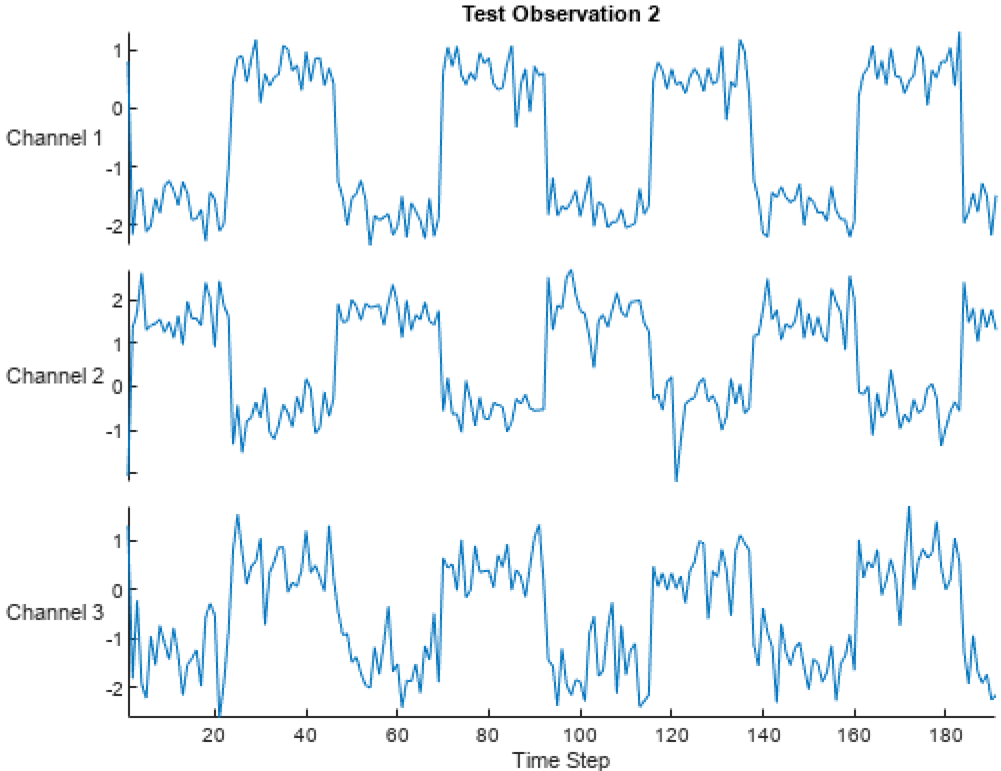
```
ans = single
    0.8385
```

Forecast Future Time Steps

To forecast the values of multiple future time steps, when given an input time series or sequence, use the `predictAndUpdateState` function. This function predicts time steps one at a time and updates the network state at each prediction. For each prediction, use the previous prediction as the input to the function.

Visualize one of the test sequences in a plot.

```
idx = 2;
X = XTest{idx};
T = TTest{idx};

figure
stackedplot(X',DisplayLabels="Channel " + (1:numChannels))
xlabel("Time Step")
title("Test Observation " + idx)
```



Open-Loop Forecasting

Open-loop forecasting predicts the next time step in a sequence using only the input data. When making predictions for subsequent time steps, you collect the true values form your data source and use those as input. For example, suppose that you want to predict the value for time step t of a sequence by using data collected in time steps 1 through $t-1$. To make predictions for time step $t+1$, wait until you record the true value for time step t and use that value as input to make the next prediction. Use open-loop forecasting when you have true values to provide to the network before making the next prediction.

Initialize the network state by resetting the state using the `resetState` function, then make an initial prediction using the first few time steps of the input data. Update the network state by using the first 75 time steps of the input data.

```
resetState(hW)
offset = 75;
[~,~] = hW.predictAndUpdateState(X(:,1:offset));
```

```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 75.
```

To forecast further predictions, loop over time steps and update the network state by using the `predictAndUpdateState` function. Forecast values for the remaining time steps of the test observation by looping over the time steps of the input data and using them as input to the network. The first prediction is the value that corresponds to the time step `offset + 1`.

```
numTimeSteps = size(X,2);
numPredictionTimeSteps = numTimeSteps - offset;
Y = zeros(numChannels,numPredictionTimeSteps);

for t = 1:numPredictionTimeSteps
    Xt = X(:,offset+t);
    Y(:,t) = predictAndUpdateState(hW,Xt);
end
```

```
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

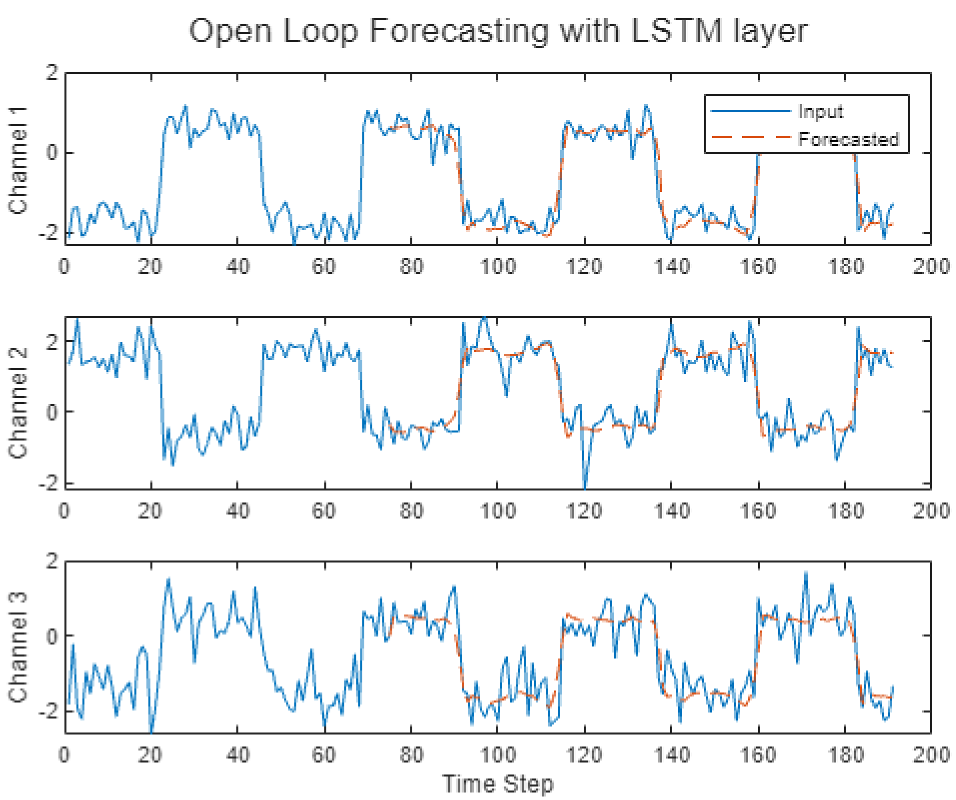
```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
```

Compare the predictions with the target values.

```
figure
t = tiledlayout(numChannels,1);
title(t,"Open Loop Forecasting with LSTM layer")

for i = 1:numChannels
    nexttile
    plot(T(i,:))
    hold on
    plot(offset:numTimeSteps,[T(i,offset) Y(i,:)],'--')
    ylabel("Channel " + i)
end

xlabel("Time Step")
nexttile(1)
legend(["Input" "Forecasted"])
```



Closed-Loop Forecasting

Closed-loop forecasting predicts subsequent time steps in a sequence by using the previous predictions as input. In this case, the model does not require the true values to make the prediction. For example, suppose that you want to predict the value for time steps t through $t + k$ of the sequence by using data collected in time steps 1 through $t - 1$. To make predictions for time step i, use the predicted value for time step $i - 1$ as input. Use closed-loop forecasting to forecast multiple subsequent time steps or when you do not have true values to provide to the network before making the next prediction.

Initialize the network state by resetting the state using the `resetState` function, then make an initial prediction, `Z,` using the first few time steps of the input data. Update the network state by using the first 75 time steps of the input data.

```
resetState(hW)
offset = size(X,2);
[Z, ~] = predictAndUpdateState(hW,X);
```

```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 191.
```

To forecast further predictions, loop over time steps and update the network state by using the `predictAndUpdateState` function. Forecast the next 200 time steps by iteratively passing the previously predicted value to the network. Because the network does not require the input data to make any further predictions, you can specify any number of time steps to forecast.

```
numPredictionTimeSteps = 200;
Xt = Z(:,end);
Y = zeros(numChannels,numPredictionTimeSteps);

for t = 1:numPredictionTimeSteps
    [Y(:,t),~] =  predictAndUpdateState(hW,Xt);
    Xt = Y(:,t);
end
```

```
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

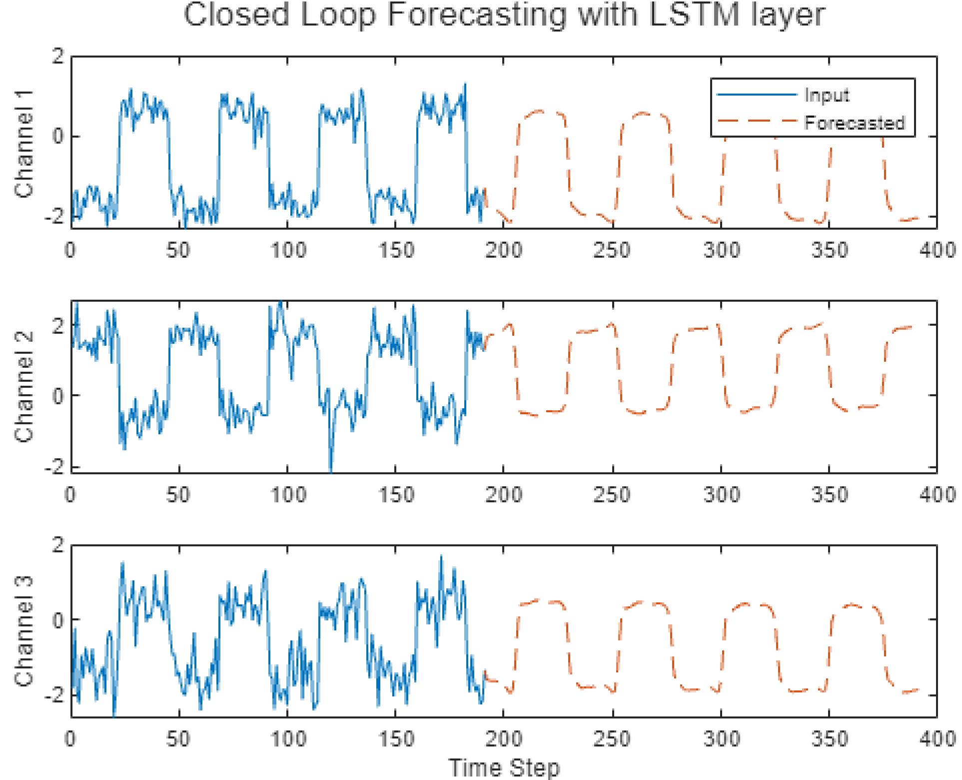
```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
```

Visualize the forecasted values in a plot.

```
numTimeSteps = offset + numPredictionTimeSteps;

figure
t = tiledlayout(numChannels,1);
title(t,"Closed Loop Forecasting with LSTM layer")

for i = 1:numChannels
    nexttile
    plot(T(i,1:offset))
    hold on
    plot(offset:numTimeSteps,[T(i,offset) Y(i,:)],'--')
    ylabel("Channel " + i)
end

xlabel("Time Step")
nexttile(1)
legend(["Input" "Forecasted"])
```



Closed-loop forecasting allows you to forecast an arbitrary number of time steps, but can be less accurate when compared to open-loop forecasting because the network does not have access to the true values during the forecasting process.

See Also

`dlhdl.Workflow` | `dlhdl.Target` | `compile` | `deploy` | `predict` | `predictAndUpdateState` | `resetState`

More About

- “Support for Long Short-Term Memory Networks” on page 13-2
- “How Deep Learning HDL Toolbox Compiles the LSTM Layer” on page 13-5

Detect Objects Using YOLO v3 Network Deployed to FPGA

This example shows how to deploy a trained you only look once (YOLO) v3 object detector to a target FPGA board. You then use MATLAB to retrieve the object classification from the FPGA board.

Compared to YOLO v2 networks, YOLO v3 networks have additional detection heads that help detect smaller objects.

Create YOLO v3 Detector Object

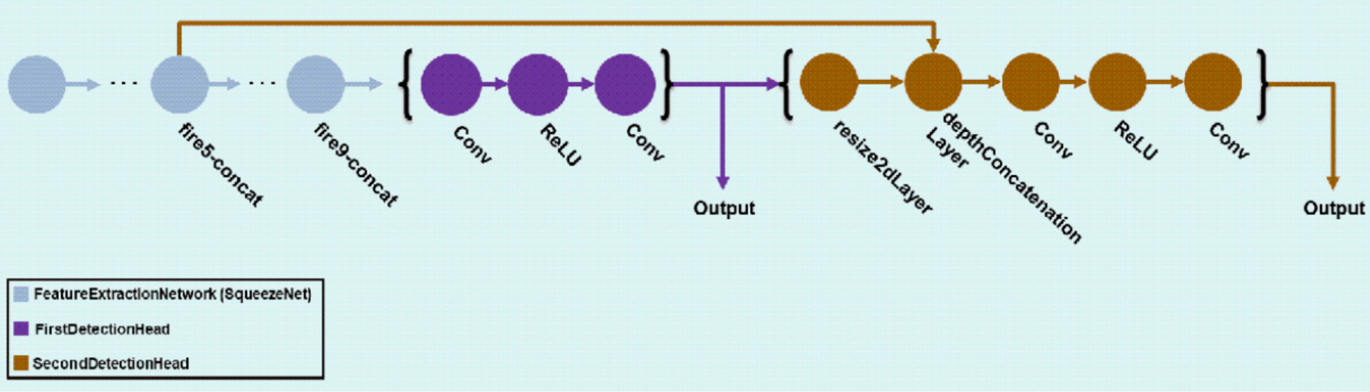
In this example, you use a pretrained YOLO v3 object detector. To construct and train a custom YOLO v3 detector, see “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox).

Use the `downloadPretrainedYOLOv3Detector` function to generate a `dlnetwork` object. To get the code for this function, see the downloadPretrainedYOLOv3Detector Function on page 10-289 section.

```
preTrainedDetector = downloadPretrainedYOLOv3Detector;
```

```
Downloaded pretrained detector
```

The generated network uses training data to estimate the anchor boxes, which help the detector learn to predict the boxes. For more information about anchor boxes, see “Anchor Boxes for Object Detection” (Computer Vision Toolbox). The `downloadPretrainedYOLOv3Detector` function creates this YOLO v3 network:



Load the Pretrained network

Extract the network from the pretrained YOLO v3 detector object.

```
yolov3Detector = preTrainedDetector;
net = yolov3Detector.Network;
```

Extract the attributes of the network as variables.

```
anchorBoxes = yolov3Detector.AnchorBoxes;
outputNames = yolov3Detector.Network.OutputNames;
inputSize = yolov3Detector.InputSize;
classNames = yolov3Detector.ClassNames;
```

Use the `analyzeNetwork` function to obtain information about the network layers. the function returns a graphical representation of the network that contains detailed parameter information for every layer in the network.

```
analyzeNetwork(net);
```

Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Create a programming interface with custom name for your target device and an Ethernet interface to connect the target device to the host computer.

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx® Zynq® UltraScale+™ MPSoC ZCU102 board and the bitstream uses the single data type.

```
hW = dlhdl.Workflow('Network',net,'Bitstream','zcu102_single','Target',hTarget);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment.

```
dn = compile(hW);
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### An output layer called 'Output1_customOutputConv1' of type 'nnet.cnn.layer.RegressionOutputLa
### An output layer called 'Output2_customOutputConv2' of type 'nnet.cnn.layer.RegressionOutputLa
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### The network includes the following layers:
     1   'data'                         Image Input                  227×227×3 images
     2   'conv1'                        2-D Convolution              64 3×3×3 convolutions with
     3   'relu_conv1'                   ReLU                         ReLU
     4   'pool1'                        2-D Max Pooling              3×3 max pooling with stride
     5   'fire2-squeeze1x1'             2-D Convolution              16 1×1×64 convolutions with
     6   'fire2-relu_squeeze1x1'        ReLU                         ReLU
     7   'fire2-expand1x1'              2-D Convolution              64 1×1×16 convolutions with
     8   'fire2-relu_expand1x1'         ReLU                         ReLU
     9   'fire2-expand3x3'              2-D Convolution              64 3×3×16 convolutions with
    10   'fire2-relu_expand3x3'         ReLU                         ReLU
    11   'fire2-concat'                 Depth concatenation          Depth concatenation of 2 in
    12   'fire3-squeeze1x1'             2-D Convolution              16 1×1×128 convolutions wi
    13   'fire3-relu_squeeze1x1'        ReLU                         ReLU
    14   'fire3-expand1x1'              2-D Convolution              64 1×1×16 convolutions with
    15   'fire3-relu_expand1x1'         ReLU                         ReLU
    16   'fire3-expand3x3'              2-D Convolution              64 3×3×16 convolutions with
    17   'fire3-relu_expand3x3'         ReLU                         ReLU
    18   'fire3-concat'                 Depth concatenation          Depth concatenation of 2 in
    19   'pool3'                        2-D Max Pooling              3×3 max pooling with stride
    20   'fire4-squeeze1x1'             2-D Convolution              32 1×1×128 convolutions wi
    21   'fire4-relu_squeeze1x1'        ReLU                         ReLU
    22   'fire4-expand1x1'              2-D Convolution              128 1×1×32 convolutions wi
```

```
    23   'fire4-relu_expand1x1'        ReLU                          ReLU
    24   'fire4-expand3x3'             2-D Convolution               128 3×3×32 convolutions wit
    25   'fire4-relu_expand3x3'        ReLU                          ReLU
    26   'fire4-concat'                Depth concatenation           Depth concatenation of 2 in
    27   'fire5-squeeze1x1'            2-D Convolution               32 1×1×256 convolutions wit
    28   'fire5-relu_squeeze1x1'       ReLU                          ReLU
    29   'fire5-expand1x1'             2-D Convolution               128 1×1×32 convolutions wit
    30   'fire5-relu_expand1x1'        ReLU                          ReLU
    31   'fire5-expand3x3'             2-D Convolution               128 3×3×32 convolutions wit
    32   'fire5-relu_expand3x3'        ReLU                          ReLU
    33   'fire5-concat'                Depth concatenation           Depth concatenation of 2 in
    34   'pool5'                       2-D Max Pooling               3×3 max pooling with stride
    35   'fire6-squeeze1x1'            2-D Convolution               48 1×1×256 convolutions wit
    36   'fire6-relu_squeeze1x1'       ReLU                          ReLU
    37   'fire6-expand1x1'             2-D Convolution               192 1×1×48 convolutions wit
    38   'fire6-relu_expand1x1'        ReLU                          ReLU
    39   'fire6-expand3x3'             2-D Convolution               192 3×3×48 convolutions wit
    40   'fire6-relu_expand3x3'        ReLU                          ReLU
    41   'fire6-concat'                Depth concatenation           Depth concatenation of 2 in
    42   'fire7-squeeze1x1'            2-D Convolution               48 1×1×384 convolutions wit
    43   'fire7-relu_squeeze1x1'       ReLU                          ReLU
    44   'fire7-expand1x1'             2-D Convolution               192 1×1×48 convolutions wit
    45   'fire7-relu_expand1x1'        ReLU                          ReLU
    46   'fire7-expand3x3'             2-D Convolution               192 3×3×48 convolutions wit
    47   'fire7-relu_expand3x3'        ReLU                          ReLU
    48   'fire7-concat'                Depth concatenation           Depth concatenation of 2 in
    49   'fire8-squeeze1x1'            2-D Convolution               64 1×1×384 convolutions wit
    50   'fire8-relu_squeeze1x1'       ReLU                          ReLU
    51   'fire8-expand1x1'             2-D Convolution               256 1×1×64 convolutions wit
    52   'fire8-relu_expand1x1'        ReLU                          ReLU
    53   'fire8-expand3x3'             2-D Convolution               256 3×3×64 convolutions wit
    54   'fire8-relu_expand3x3'        ReLU                          ReLU
    55   'fire8-concat'                Depth concatenation           Depth concatenation of 2 in
    56   'fire9-squeeze1x1'            2-D Convolution               64 1×1×512 convolutions wit
    57   'fire9-relu_squeeze1x1'       ReLU                          ReLU
    58   'fire9-expand1x1'             2-D Convolution               256 1×1×64 convolutions wit
    59   'fire9-relu_expand1x1'        ReLU                          ReLU
    60   'fire9-expand3x3'             2-D Convolution               256 3×3×64 convolutions wit
    61   'fire9-relu_expand3x3'        ReLU                          ReLU
    62   'fire9-concat'                Depth concatenation           Depth concatenation of 2 in
    63   'customConv1'                 2-D Convolution               1024 3×3×512 convolutions w
    64   'customRelu1'                 ReLU                          ReLU
    65   'customOutputConv1'           2-D Convolution               18 1×1×1024 convolutions wi
    66   'featureConv2'                2-D Convolution               128 1×1×512 convolutions wi
    67   'featureRelu2'                ReLU                          ReLU
    68   'Output1_customOutputConv1'   Regression Output             mean-squared-error
    69   'featureResize2'              dnnfpga.custom.Resize2DLayer  dnnfpga.custom.Resize2DLaye
    70   'depthConcat2'                Depth concatenation           Depth concatenation of 2 in
    71   'customConv2'                 2-D Convolution               256 3×3×384 convolutions wi
    72   'customRelu2'                 ReLU                          ReLU
    73   'customOutputConv2'           2-D Convolution               18 1×1×256 convolutions wit
    74   'Output2_customOutputConv2'   Regression Output             mean-squared-error

### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'Output1_customOutputConv1' with type 'nnet.cnn.layer.RegressionOutputLayer
### Notice: The layer 'Output2_customOutputConv2' with type 'nnet.cnn.layer.RegressionOutputLayer
### Compiling layer group: conv1>>fire2-relu_squeeze1x1 ...
### Compiling layer group: conv1>>fire2-relu_squeeze1x1 ... complete.
```

```
### Compiling layer group: fire2-expand1x1>>fire2-relu_expand1x1 ...
### Compiling layer group: fire2-expand1x1>>fire2-relu_expand1x1 ... complete.
### Compiling layer group: fire2-expand3x3>>fire2-relu_expand3x3 ...
### Compiling layer group: fire2-expand3x3>>fire2-relu_expand3x3 ... complete.
### Compiling layer group: fire3-squeeze1x1>>fire3-relu_squeeze1x1 ...
### Compiling layer group: fire3-squeeze1x1>>fire3-relu_squeeze1x1 ... complete.
### Compiling layer group: fire3-expand1x1>>fire3-relu_expand1x1 ...
### Compiling layer group: fire3-expand1x1>>fire3-relu_expand1x1 ... complete.
### Compiling layer group: fire3-expand3x3>>fire3-relu_expand3x3 ...
### Compiling layer group: fire3-expand3x3>>fire3-relu_expand3x3 ... complete.
### Compiling layer group: pool3>>fire4-relu_squeeze1x1 ...
### Compiling layer group: pool3>>fire4-relu_squeeze1x1 ... complete.
### Compiling layer group: fire4-expand1x1>>fire4-relu_expand1x1 ...
### Compiling layer group: fire4-expand1x1>>fire4-relu_expand1x1 ... complete.
### Compiling layer group: fire4-expand3x3>>fire4-relu_expand3x3 ...
### Compiling layer group: fire4-expand3x3>>fire4-relu_expand3x3 ... complete.
### Compiling layer group: fire5-squeeze1x1>>fire5-relu_squeeze1x1 ...
### Compiling layer group: fire5-squeeze1x1>>fire5-relu_squeeze1x1 ... complete.
### Compiling layer group: fire5-expand1x1>>fire5-relu_expand1x1 ...
### Compiling layer group: fire5-expand1x1>>fire5-relu_expand1x1 ... complete.
### Compiling layer group: fire5-expand3x3>>fire5-relu_expand3x3 ...
### Compiling layer group: fire5-expand3x3>>fire5-relu_expand3x3 ... complete.
### Compiling layer group: pool5>>fire6-relu_squeeze1x1 ...
### Compiling layer group: pool5>>fire6-relu_squeeze1x1 ... complete.
### Compiling layer group: fire6-expand1x1>>fire6-relu_expand1x1 ...
### Compiling layer group: fire6-expand1x1>>fire6-relu_expand1x1 ... complete.
### Compiling layer group: fire6-expand3x3>>fire6-relu_expand3x3 ...
### Compiling layer group: fire6-expand3x3>>fire6-relu_expand3x3 ... complete.
### Compiling layer group: fire7-squeeze1x1>>fire7-relu_squeeze1x1 ...
### Compiling layer group: fire7-squeeze1x1>>fire7-relu_squeeze1x1 ... complete.
### Compiling layer group: fire7-expand1x1>>fire7-relu_expand1x1 ...
### Compiling layer group: fire7-expand1x1>>fire7-relu_expand1x1 ... complete.
### Compiling layer group: fire7-expand3x3>>fire7-relu_expand3x3 ...
### Compiling layer group: fire7-expand3x3>>fire7-relu_expand3x3 ... complete.
### Compiling layer group: fire8-squeeze1x1>>fire8-relu_squeeze1x1 ...
### Compiling layer group: fire8-squeeze1x1>>fire8-relu_squeeze1x1 ... complete.
### Compiling layer group: fire8-expand1x1>>fire8-relu_expand1x1 ...
### Compiling layer group: fire8-expand1x1>>fire8-relu_expand1x1 ... complete.
### Compiling layer group: fire8-expand3x3>>fire8-relu_expand3x3 ...
### Compiling layer group: fire8-expand3x3>>fire8-relu_expand3x3 ... complete.
### Compiling layer group: fire9-squeeze1x1>>fire9-relu_squeeze1x1 ...
### Compiling layer group: fire9-squeeze1x1>>fire9-relu_squeeze1x1 ... complete.
### Compiling layer group: fire9-expand1x1>>fire9-relu_expand1x1 ...
### Compiling layer group: fire9-expand1x1>>fire9-relu_expand1x1 ... complete.
### Compiling layer group: fire9-expand3x3>>fire9-relu_expand3x3 ...
### Compiling layer group: fire9-expand3x3>>fire9-relu_expand3x3 ... complete.
### Compiling layer group: customConv1>>customOutputConv1 ...
### Compiling layer group: customConv1>>customOutputConv1 ... complete.
### Compiling layer group: featureConv2>>featureRelu2 ...
### Compiling layer group: featureConv2>>featureRelu2 ... complete.
### Compiling layer group: customConv2>>customOutputConv2 ...
### Compiling layer group: customConv2>>customOutputConv2 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________
```

```
    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x02000000"     "28.0 MB"
    "InstructionDataOffset"     "0x03c00000"     "8.0 MB"
    "ConvWeightDataOffset"      "0x04400000"     "104.0 MB"
    "EndOffset"                 "0x0ac00000"     "Total: 172.0 MB"

### Network compilation complete.
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx® Zynq® UltraScale+ MPSoC ZCU102 hardware, run the `deploy` method of the `dlhdl.Workflow` object. This method programs the FPGA board using the output of the compile method and the programming file, downloads the network weights and biases, displays progress messages, and the time it takes to deploy the network.

```
deploy(hW);
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_single.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_single.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 27-Oct-2022 13:44:50
```

Test Network

Load the example image and convert the image into a `dlarray`. Then classify the image on the FPGA by using the `predict` method of the `dlhdl.Workflow` object and display the results.

```
img = imread('vehicle_image.jpg');
I = single(rescale(img));
I = imresize(I, yolov3Detector.InputSize(1:2));
dlX = dlarray(I,'SSC');
```

Store the output of each detection head of the network in the `features` variable. Pass `features` to the post-processing function `processYOLOv3Ouputs` to combine the multiple outputs and compute the final results. To get the code for this function, see the processYOLOv3Output Function on page 10-289 section.

```
features = cell(size(net.OutputNames'));
[features{:}] = hW.predict(dlX, 'Profiler', 'on');
```

```
### Finished writing input activations.
### Running single input activation.
```

```
             Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    34469645                  0.15668                      1          344
    conv1                    673148                  0.00306
    pool1                    509022                  0.00231
    fire2-squeeze1x1         308280                  0.00140
    fire2-expand1x1          305546                  0.00139
    fire2-expand3x3          305227                  0.00139
    fire3-squeeze1x1         628018                  0.00285
    fire3-expand1x1          305219                  0.00139
    fire3-expand3x3          305220                  0.00139
    pool3                    286781                  0.00130
    fire4-squeeze1x1         264346                  0.00120
    fire4-expand1x1          264777                  0.00120
    fire4-expand3x3          264750                  0.00120
    fire5-squeeze1x1         749166                  0.00341
    fire5-expand1x1          264800                  0.00120
    fire5-expand3x3          264880                  0.00120
    pool5                    219686                  0.00100
    fire6-squeeze1x1         195193                  0.00089
    fire6-expand1x1          145091                  0.00066
    fire6-expand3x3          145075                  0.00066
    fire7-squeeze1x1         290001                  0.00132
    fire7-expand1x1          144830                  0.00066
    fire7-expand3x3          145390                  0.00066
    fire8-squeeze1x1         369605                  0.00168
    fire8-expand1x1          245085                  0.00111
    fire8-expand3x3          245208                  0.00111
    fire9-squeeze1x1         490784                  0.00223
    fire9-expand1x1          244864                  0.00111
    fire9-expand3x3          245458                  0.00112
    customConv1            17592876                  0.07997
    customOutputConv1        952889                  0.00433
    featureConv2             913457                  0.00415
    featureResize2            57819                  0.00026
    customConv2             5600648                  0.02546
    customOutputConv2        526143                  0.00239
 * The clock frequency of the DL processor is: 220MHz
```

```
[bboxes, scores, labels] = processYOLOv3Output(anchorBoxes, inputSize, classNames, features, I);
resultImage = insertObjectAnnotation(I,'rectangle',bboxes,scores);
imshow(resultImage)
```

The FPGA returns a score prediction of `0.89605` with a bounding box drawn around the object in the image. The FPGA also returns a prediction of vehicle to the `labels` variable.

`downloadPretrainedYOLOv3Detector` Function

The `downloadPretrainedYOLOv3Detector` function to download the pretrained YOLO v3 detector network

```
function detector = downloadPretrainedYOLOv3Detector
if ~exist('yolov3SqueezeNetVehicleExample_21aSPKG.mat', 'file')
    if ~exist('yolov3SqueezeNetVehicleExample_21aSPKG.zip', 'file')
        zipFile = matlab.internal.examples.downloadSupportFile('vision/data', 'yolov3SqueezeNetVe
        copyfile(zipFile);
    end
    unzip('yolov3SqueezeNetVehicleExample_21aSPKG.zip');
end
pretrained = load("yolov3SqueezeNetVehicleExample_21aSPKG.mat");
detector = pretrained.detector;
disp('Downloaded pretrained detector');
end
```

`processYOLOv3Output` Function

The `processYOLOv3Output` function is attached as a helper file in this example's directory. This function converts the feature maps from multiple detection heads to bounding boxes, scores and labels. A code snippet of the function is shown below.

```
function [bboxes, scores, labels] = processYOLOv3Output(anchorBoxes, inputSize, classNames, featu
% This function converts the feature maps from multiple detection heads to bounding boxes, score
% processYOLOv3Output is C code generatable

% Breaks down the raw output from predict function into Confidence score, X, Y, Width,
% Height and Class probabilities for each output from detection head
predictions = iYolov3Transform(features, anchorBoxes);
```

```
% Initialize parameters for post-processing
inputSize2d = inputSize(1:2);
info.PreprocessedImageSize = inputSize2d(1:2);
info.ScaleX = size(img,1)/inputSize2d(1);
info.ScaleY = size(img,2)/inputSize2d(1);
params.MinSize = [1 1];
params.MaxSize = size(img(:,:,1));
params.Threshold = 0.5;
params.FractionDownsampling = 1;
params.DetectionInputWasBatchOfImages = false;
params.NetworkInputSize = inputSize;
params.DetectionPreprocessing = "none";
params.SelectStrongest = 1;
bboxes = [];
scores = [];
labels = [];

% Post-process the predictions to get bounding boxes, scores and labels
[bboxes, scores, labels] = iPostprocessMultipleDetection(anchorBoxes, inputSize, classNames, pred
end

function [bboxes, scores, labels] = iPostprocessMultipleDetection (anchorBoxes, inputSize, classN
% Post-process the predictions to get bounding boxes, scores and labels

% YpredData is a (x,8) cell array, where x = number of detection heads
% Information in each column is:
% column 1 -> confidence scores
% column 2 to column 5 -> X offset, Y offset, Width, Height of anchor boxes
% column 6 -> class probabilities
% column 7-8 -> copy of width and height of anchor boxes

% Initialize parameters for post-processing
classes = classNames;
predictions = YPredData;
extractPredictions = cell(size(predictions));
% Extract dlarray data
for i = 1:size(extractPredictions,1)
    for j = 1:size(extractPredictions,2)
        extractPredictions{i,j} = extractdata(predictions{i,j});
    end
end

% Storing the values of columns 2 to 5 of extractPredictions
% Columns 2 to 5 represent information about X-coordinate, Y-coordinate, Width and Height of pred
extractedCoordinates = cell(size(predictions,1),4);
for i = 1:size(predictions,1)
    for j = 2:5
        extractedCoordinates{i,j-1} = extractPredictions{i,j};
    end
end

% Convert predictions from grid cell coordinates to box coordinates.
boxCoordinates = anchorBoxGenerator(anchorBoxes, inputSize, classNames, extractedCoordinates, par
% Replace grid cell coordinates in extractPredictions with box coordinates
for i = 1:size(YPredData,1)
    for j = 2:5
        extractPredictions{i,j} = single(boxCoordinates{i,j-1});
    end
```

```
end

% 1. Convert bboxes from spatial to pixel dimension
% 2. Combine the prediction from different heads.
% 3. Filter detections based on threshold.

% Reshaping the matrices corresponding to confidence scores and  bounding boxes
detections = cell(size(YPredData,1),6);
for i = 1:size(detections,1)
    for j = 1:5
        detections{i,j} = reshapePredictions(extractPredictions{i,j});
    end
end
% Reshaping the matrices corresponding to class probablities
numClasses = repmat({numel(classes)},[size(detections,1),1]);
for i = 1:size(detections,1)
    detections{i,6} = reshapeClasses(extractPredictions{i,6},numClasses{i,1});
end

% cell2mat converts the cell of matrices into one matrix, this combines the
% predictions of all detection heads
detections = cell2mat(detections);

% Getting the most probable class and corresponding index
[classProbs, classIdx] = max(detections(:,6:end),[],2);
detections(:,1) = detections(:,1).*classProbs;
detections(:,6) = classIdx;

% Keep detections whose confidence score is greater than threshold.
detections = detections(detections(:,1) >= params.Threshold,:);

[bboxes, scores, labels] = iPostProcessDetections(detections, classes, info, params);
end

function [bboxes, scores, labels] = iPostProcessDetections(detections, classes, info, params)
% Resizes the anchor boxes, filters anchor boxes based on size and apply
% NMS to eliminate overlapping anchor boxes
if ~isempty(detections)

    % Obtain bounding boxes and class data for pre-processed image
    scorePred = detections(:,1);
    bboxesTmp = detections(:,2:5);
    classPred = detections(:,6);
    inputImageSize = ones(1,2);
    inputImageSize(2) = info.ScaleX.*info.PreprocessedImageSize(2);
    inputImageSize(1) = info.ScaleY.*info.PreprocessedImageSize(1);
    % Resize boxes to actual image size.
    scale = [inputImageSize(2) inputImageSize(1) inputImageSize(2) inputImageSize(1)];
    bboxPred = bboxesTmp.*scale;
    % Convert x and y position of detections from centre to top-left.
    bboxPred = iConvertCenterToTopLeft(bboxPred);

    % Filter boxes based on MinSize, MaxSize.
    [bboxPred, scorePred, classPred] = filterBBoxes(params.MinSize, params.MaxSize, bboxPred, sc

    % Apply NMS to eliminate boxes having significant overlap
    if params.SelectStrongest
        [bboxes, scores, classNames] = selectStrongestBboxMulticlass(bboxPred, scorePred, classPr
```

```
            'RatioType', 'Union', 'OverlapThreshold', 0.4);
    else
        bboxes = bboxPred;
        scores = scorePred;
        classNames = classPred;
    end

    % Limit width detections
    detectionsWd = min((bboxes(:,1) + bboxes(:,3)),inputImageSize(1,2));
    bboxes(:,3) = detectionsWd(:,1) - bboxes(:,1);

    % Limit height detections
    detectionsHt = min((bboxes(:,2) + bboxes(:,4)),inputImageSize(1,1));
    bboxes(:,4) = detectionsHt(:,1) - bboxes(:,2);
    bboxes(bboxes<1) = 1;

    % Convert classId to classNames.
    labels = categorical(classes,cellstr(classes));
    labels = labels(classNames);

else
    % If detections are empty then bounding boxes, scores and labels should
    % be empty
    bboxes = zeros(0,4,'single');
    scores = zeros(0,1,'single');
    labels = categorical(classes);
end
end

function x = reshapePredictions(pred)
% Reshapes the matrices corresponding to scores, X, Y, Width and Height to
% make them compatible for combining the outputs of different detection
% heads
[h,w,c,n] = size(pred);
x = reshape(pred,h*w*c,1,n);
end

function x = reshapeClasses(pred,numClasses)
% Reshapes the matrices corresponding to the class probabilities, to make it
% compatible for combining the outputs of different detection heads
[h,w,c,n] = size(pred);
numAnchors = c/numClasses;
x = reshape(pred,h*w,numClasses,numAnchors,n);
x = permute(x,[1,3,2,4]);
[h,w,c,n] = size(x);
x = reshape(x,h*w,c,n);
end

function bboxes = iConvertCenterToTopLeft(bboxes)
% Convert x and y position of detections from centre to top-left.
bboxes(:,1) = bboxes(:,1) - bboxes(:,3)/2 + 0.5;
bboxes(:,2) = bboxes(:,2) - bboxes(:,4)/2 + 0.5;
bboxes = floor(bboxes);
bboxes(bboxes<1) = 1;
end

function tiledAnchors = anchorBoxGenerator(anchorBoxes, inputSize, classNames,YPredCell,inputImag
% Convert grid cell coordinates to box coordinates.
```

```
% Generate tiled anchor offset.
tiledAnchors = cell(size(YPredCell));
for i = 1:size(YPredCell,1)
    anchors = anchorBoxes{i,:};
    [h,w,~,n] = size(YPredCell{i,1});
    [tiledAnchors{i,2},tiledAnchors{i,1}] = ndgrid(0:h-1,0:w-1,1:size(anchors,1),1:n);
    [~,~,tiledAnchors{i,3}] = ndgrid(0:h-1,0:w-1,anchors(:,2),1:n);
    [~,~,tiledAnchors{i,4}] = ndgrid(0:h-1,0:w-1,anchors(:,1),1:n);
end

for i = 1:size(YPredCell,1)
    [h,w,~,~] = size(YPredCell{i,1});
    tiledAnchors{i,1} = double((tiledAnchors{i,1} + YPredCell{i,1})./w);
    tiledAnchors{i,2} = double((tiledAnchors{i,2} + YPredCell{i,2})./h);
    tiledAnchors{i,3} = double((tiledAnchors{i,3}.*YPredCell{i,3})./inputImageSize(2));
    tiledAnchors{i,4} = double((tiledAnchors{i,4}.*YPredCell{i,4})./inputImageSize(1));
end
end

function predictions = iYolov3Transform(YPredictions, anchorBoxes)
% This function breaks down the raw output from predict function into Confidence score, X, Y, Wid
% Height and Class probabilities for each output from detection head

predictions = cell(size(YPredictions,1),size(YPredictions,2) + 2);

for idx = 1:size(YPredictions,1)
    % Get the required info on feature size.
    numChannelsPred = size(YPredictions{idx},3);  %number of channels in a feature map
    numAnchors = size(anchorBoxes{idx},1);    %number of anchor boxes per grid
    numPredElemsPerAnchors = numChannelsPred/numAnchors;
    channelsPredIdx = 1:numChannelsPred;
    predictionIdx = ones([1,numAnchors.*5]);

    % X positions.
    startIdx = 1;
    endIdx = numChannelsPred;
    stride = numPredElemsPerAnchors;
    predictions{idx,2} = YPredictions{idx}(:,:,startIdx:stride:endIdx,:);
    predictionIdx = [predictionIdx startIdx:stride:endIdx];

    % Y positions.
    startIdx = 2;
    endIdx = numChannelsPred;
    stride = numPredElemsPerAnchors;
    predictions{idx,3} = YPredictions{idx}(:,:,startIdx:stride:endIdx,:);
    predictionIdx = [predictionIdx startIdx:stride:endIdx];

    % Width.
    startIdx = 3;
    endIdx = numChannelsPred;
    stride = numPredElemsPerAnchors;
    predictions{idx,4} = YPredictions{idx}(:,:,startIdx:stride:endIdx,:);
    predictionIdx = [predictionIdx startIdx:stride:endIdx];

    % Height.
    startIdx = 4;
    endIdx = numChannelsPred;
    stride = numPredElemsPerAnchors;
```

```
        predictions{idx,5} = YPredictions{idx}(:,:,startIdx:stride:endIdx,:);
        predictionIdx = [predictionIdx startIdx:stride:endIdx];

        % Confidence scores.
        startIdx = 5;
        endIdx = numChannelsPred;
        stride = numPredElemsPerAnchors;
        predictions{idx,1} = YPredictions{idx}(:,:,startIdx:stride:endIdx,:);
        predictionIdx = [predictionIdx startIdx:stride:endIdx];

        % Class probabilities.
        classIdx = setdiff(channelsPredIdx,predictionIdx);
        predictions{idx,6} = YPredictions{idx}(:,:,classIdx,:);
    end

    for i = 1:size(predictions,1)
        predictions{i,7} = predictions{i,4};
        predictions{i,8} = predictions{i,5};
    end

    % Apply activation to the predicted cell array
    % Apply sigmoid activation to columns 1-3 (Confidence score, X, Y)
    for i = 1:size(predictions,1)
        for j = 1:3
            predictions{i,j} = sigmoid(predictions{i,j});
        end
    end
    % Apply exponentiation to columns 4-5 (Width, Height)
    for i = 1:size(predictions,1)
        for j = 4:5
            predictions{i,j} = exp(predictions{i,j});
        end
    end
    % Apply sigmoid activation to column 6 (Class probabilities)
    for i = 1:size(predictions,1)
        for j = 6
            predictions{i,j} = sigmoid(predictions{i,j});
        end
    end
    end

    function [bboxPred, scorePred, classPred] = filterBBoxes(minSize, maxSize, bboxPred, scorePred, c
    % Filter boxes based on MinSize, MaxSize
    [bboxPred, scorePred, classPred] = filterSmallBBoxes(minSize, bboxPred, scorePred, classPred);
    [bboxPred, scorePred, classPred] = filterLargeBBoxes(maxSize, bboxPred, scorePred, classPred);
    end

    function varargout = filterSmallBBoxes(minSize, varargin)
    % Filter boxes based on MinSize
    bboxes = varargin{1};
    tooSmall = any((bboxes(:,[4 3]) < minSize),2);
    for ii = 1:numel(varargin)
        varargout{ii} = varargin{ii}(~tooSmall,:);
    end
    end

    function varargout = filterLargeBBoxes(maxSize, varargin)
    % Filter boxes based on MaxSize
```

```
bboxes = varargin{1};
tooBig = any((bboxes(:,[4 3]) > maxSize),2);
for ii = 1:numel(varargin)
    varargout{ii} = varargin{ii}(~tooBig,:);
end
end

function m = cell2mat(c)
% Converts the cell of matrices into one matrix by concatenating
% the output corresponding to each feature map

elements = numel(c);
% If number of elements is 0 return an empty array
if elements == 0
    m = [];
    return
end
% If number of elements is 1, return same element as matrix
if elements == 1
    if isnumeric(c{1}) || ischar(c{1}) || islogical(c{1}) || isstruct(c{1})
        m = c{1};
        return
    end
end
% Error out for unsupported cell content
ciscell = iscell(c{1});
cisobj = isobject(c{1});
if cisobj || ciscell
    disp('CELL2MAT does not support cell arrays containing cell arrays or objects.');
end
% If input input is struct, extract field names of structure into a cell
if isstruct(c{1})
    cfields = cell(elements,1);
    for n = 1:elements
        cfields{n} = fieldnames(c{n});
    end
    if ~isequal(cfields{:})
        disp('The field names of each cell array element must be consistent and in consistent orc
    end
end
% If number of dimensions is 2
if ndims(c) == 2
    rows = size(c,1);
    cols = size(c,2);
    if (rows < cols)
        % If rows is less than columns first concatenate each column into 1
        % row then concatenate all the rows
        m = cell(rows,1);
        for n = 1:rows
            m{n} = cat(2,c{n,:});
        end
        m = cat(1,m{:});
    else
        % If columns is less than rows, first concatenate each corresponding
        % row into columns, then combine all columns into 1
        m = cell(1,cols);
        for n = 1:cols
            m{n} = cat(1,c{:,n});
```

```
        end
        m = cat(2,m{:});
    end
    return
end
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. “YOLOv3: An Incremental Improvement.” Preprint, submitted April 8, 2018. https://arxiv.org/abs/1804.02767.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `classify`

Run Sequence-to-Sequence Regression on FPGAs

This example shows how to create, compile, and deploy a long short-term memory (LSTM) network trained on remaining useful life (RUL) of engines. Use the deployed network to predict the RUL for an engine. Use MATLAB® to retrieve the prediction results from the target device.

This example uses the turbofan engine degradation data used in [1]. The example uses an LSTM network to predict the remaining useful life of an engine measured in cycles when given time series data representing various sensors in the engine. The training data contains simulated time series data for 100 engines. Each sequence varies in length and corresponds to a full run to failure (RTF) instance. The test data contains 100 partial sequences and the corresponding values for the remaining useful life at the end of each sequence.

The data set contains 100 training observations and 100 test observations.

To learn more about how to train this network, see "Sequence-to-Sequence Regression Using Deep Learning". Fore this example, you must have a Xilinx® Zynq® Ultrascale+™ ZCU102 SoC development kit.

Download Data

Download and unzip the turbofan engine degradation simulation data set.

Each time series in the turbofan engine degradation simulation data set represents a different engine. Each engine starts with unknown degrees of initial wear and manufacturing variation. The engine operates normally at the start of each time series, and develops a fault at some point during the series. In the training set, the fault grows in magnitude until system failure.

The data contains ZIP-compressed text files with 26 columns of numbers, separated by spaces. Each row is a snapshot of data taken during a single operational cycle, and each column is a different variable. The columns correspond to the following:

- Column 1 — Unit number
- Column 2 — Time in cycles
- Columns 3–5 — Operational settings
- Columns 6–26 — Sensor measurements 1–21

Create a directory to store the turbofan engine degradation Simulation data set.

```
dataFolder = fullfile(tempdir,"turbofan");
if ~exist(dataFolder,'dir')
    mkdir(dataFolder);
end
```

Download and extract the turbofan engine degradation simulation data set.

```
filename = matlab.internal.examples.downloadSupportFile("nnet","data/TurbofanEngineDegradationSi
unzip(filename,dataFolder)
```

Prepare Test Data

Load the test data using the `processTurboFanDataTest` function attached to this example. The `processTurboFanDataTest` function extracts the data from `filenamePredictors` and

filenameResponses and returns the cell arrays XTest and YTest, which contain the test predictor and response sequences, respectively.

```
filenamePredictors = fullfile(pwd,"test_FD001.txt");
filenameResponses = fullfile(pwd,"RUL_FD001.txt");
[XTest,YTest] = processTurboFanDataTest(filenamePredictors,filenameResponses);
```

Remove features with constant values using idxConstant calculated from the training data. Normalize the test predictors using the same parameters as in the training data. Clip the test responses at the same threshold used for the training data.

```
filenamePredictors = fullfile(pwd,"train_FD001.txt");
[XTrain,YTrain] = processTurboFanDataTrain(filenamePredictors);
```

Remove Features with Constant Values

Features that remain constant for all time steps can negatively impact the training. Find the rows of data that have the same minimum and maximum values, and remove the rows. Then use these values to clean up the test dataset.

```
m = min([XTrain{:}],[],2);
M = max([XTrain{:}],[],2);
idxConstant = M == m;

for i = 1:numel(XTrain)
    XTrain{i}(idxConstant,:) = [];
end

numFeatures = size(XTrain{1},1);
mu = mean([XTrain{:}],2);
sig = std([XTrain{:}],0,2);

for i = 1:numel(XTrain)
    XTrain{i} = (XTrain{i} - mu) ./ sig;
end

thr = 150; %threshold
for i = 1:numel(XTest)
    XTest{i}(idxConstant,:) = [];
    XTest{i} = (XTest{i} - mu) ./ sig;
    YTest{i}(YTest{i} > thr) = thr;
end
```

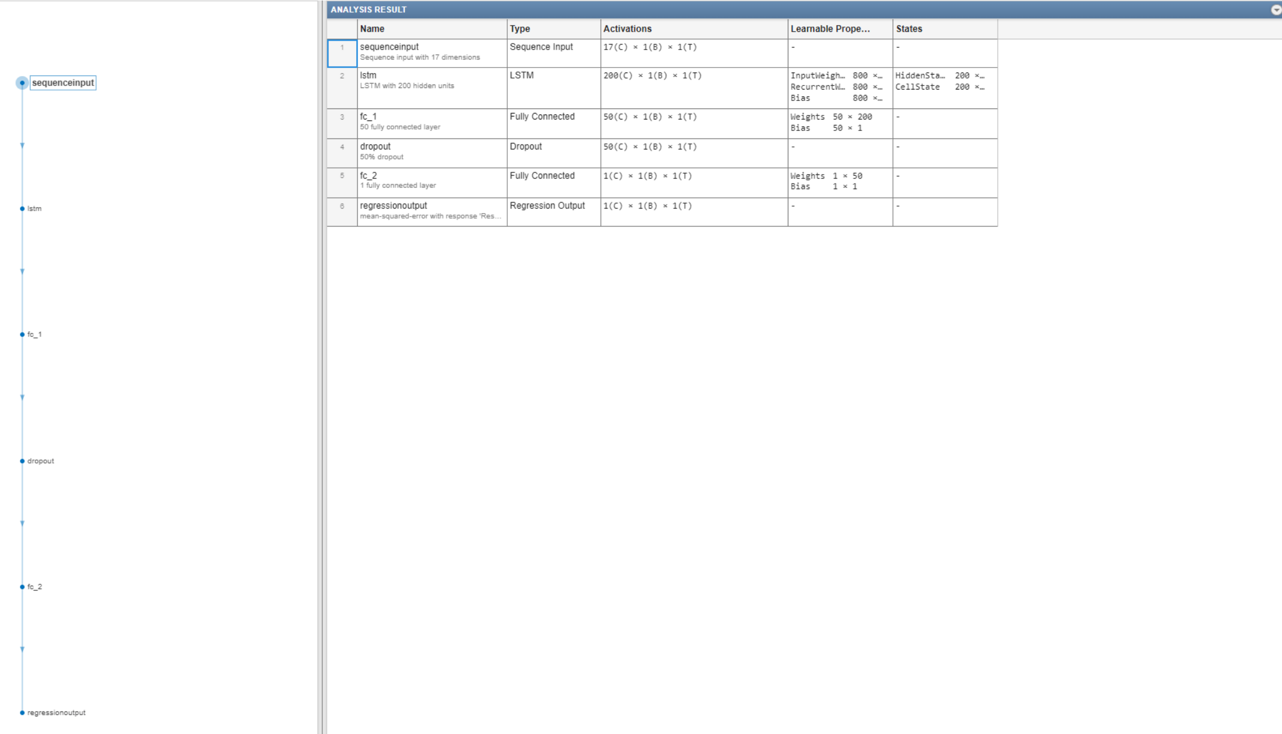
Load the Pretrained Network

Load the LSTM network . This network was trained on NASA CMAPSS Data described in [1], enter:

```
load CMAPSSDataNetwork
```

View the layers of the network by using the analyzeNetwork function. The function returns a graphical representation of the network and the parameter settings for the layers in the network.

```
analyzeNetwork(net)
```



Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Specify that the interface is for a Xilinx board with an Ethernet interface.

To create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

Alternatively, to use the JTAG interface, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
hTarget = dlhdl.Target('Xilinx','Interface','JTAG');
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and the bitstream name. Ensure that the bitstream name matches the data type of your FPGA board. In this example, the target board is the Xilinx ZCU102 SOC. The bitstream uses a single data type.

```
hW = dlhdl.Workflow('network', net, 'Bitstream', 'zcu102_lstm_single','Target',hTarget);
```

Alternatively, to run the example on the Xilinx ZC706 board, enter:

```
hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'zc706_lstm_single','Target',hTarget);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment. The total number of frames exceeds the default

value of 30. Set the `InputFrameNumberLimit` name-value argument to 500 to run predictions in chunks of 500 frames to prevent timeouts.

```
dn = compile(hW,'InputFrameNumberLimit',500)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_lstm_single.
### The network includes the following layers:
     1   'sequenceinput'      Sequence Input      Sequence input with 17 dimensions            (S
     2   'lstm'               LSTM                LSTM with 200 hidden units                   (H
     3   'fc_1'               Fully Connected     50 fully connected layer                     (H
     4   'fc_2'               Fully Connected     1 fully connected layer                      (H
     5   'regressionoutput'   Regression Output   mean-squared-error with response 'Response'  (S

### Notice: The layer 'sequenceinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented i
### Notice: The layer 'regressionoutput' with type 'nnet.cnn.layer.RegressionOutputLayer' is impl
### Compiling layer group: lstm.wi ...
### Compiling layer group: lstm.wi ... complete.
### Compiling layer group: lstm.wo ...
### Compiling layer group: lstm.wo ... complete.
### Compiling layer group: lstm.wg ...
### Compiling layer group: lstm.wg ... complete.
### Compiling layer group: lstm.wf ...
### Compiling layer group: lstm.wf ... complete.
### Compiling layer group: fc_1>>fc_2 ...
### Compiling layer group: fc_1>>fc_2 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "20.0 MB"
    "InstructionDataOffset"     "0x02000000"     "4.0 MB"
    "FCWeightDataOffset"        "0x02400000"     "4.0 MB"
    "EndOffset"                 "0x02800000"     "Total: 40.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` function of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board by using the programming file. The `deploy` function starts programming the FPGA device and displays progress messages, and the required time to deploy the network.

```
hW.deploy
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_lstm_single.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_lstm_single.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Resetting network state.
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 01-Sep-2022 12:16:55
```

Predict Remaining Useful Life

Run the `predict` method of the `dlhdl.Workflow` object, to make predictions on the test data.

```
for i = 1:numel(XTest)
 YPred{i} = hW.predict(XTest{i});
end
```

```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 31.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 49.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 126.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 106.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 98.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 105.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 160.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 166.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 55.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 192.
```

```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 83.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 217.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 195.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 46.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 76.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 113.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 165.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 133.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 135.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 184.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 148.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 39.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 130.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 186.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 48.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 76.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 140.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 158.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 171.
### Resetting network state.
```

```
### Finished writing input activations.
### Running a sequence of length 143.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 196.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 145.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 50.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 203.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 198.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 126.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 121.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 125.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 37.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 133.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 123.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 156.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 172.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 54.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 152.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 146.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 73.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 78.
### Resetting network state.
### Finished writing input activations.
```

```
### Running a sequence of length 303.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 74.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 144.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 189.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 164.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 121.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 113.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 136.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 160.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 176.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 94.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 147.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 159.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 232.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 155.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 168.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 71.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 147.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 71.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 187.
```

```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 54.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 152.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 68.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 131.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 112.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 137.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 88.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 205.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 162.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 72.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 101.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 133.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 213.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 162.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 73.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 172.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 34.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 110.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 56.
### Resetting network state.
```

```
### Finished writing input activations.
### Running a sequence of length 68.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 177.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 146.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 234.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 150.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 244.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 133.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 89.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 97.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 134.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 121.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 97.
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 198.
```

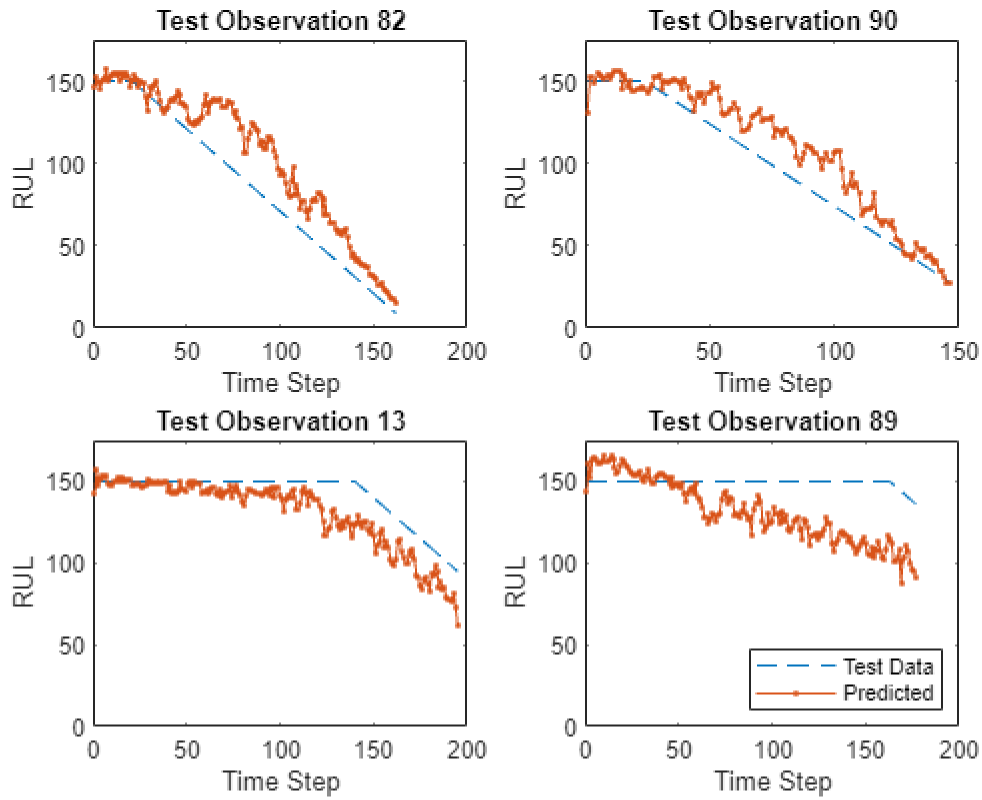
The LSTM network makes predictions on the partial sequence one time step at a time. At each time step, the network makes predictions using the value at this time step, and the network state calculated from the previous time steps. The network updates its state between each prediction. The `predict` function returns a sequence of these predictions. The last element of the prediction corresponds to the predicted RUL for the partial sequence.

Alternatively, you can make predictions one time step at a time by using `predictAndUpdateState`. This function is useful when you have the values of the time steps in a stream. Usually, it is faster to make predictions on full sequences when compared to making predictions one time step at a time. For an example showing how to forecast future time steps by updating the network between single time step predictions, see “Time Series Forecasting Using Deep Learning”.

Visualize some of the predictions in a plot.

```
idx = randperm(numel(YPred),4);
figure
for i = 1:numel(idx)
    subplot(2,2,i)
```

```
    plot(YTest{idx(i)},'--')
    hold on
    plot(YPred{idx(i)},'.-')
    hold off

    ylim([0 thr + 25])
    title("Test Observation " + idx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted"],'Location','southeast')
```

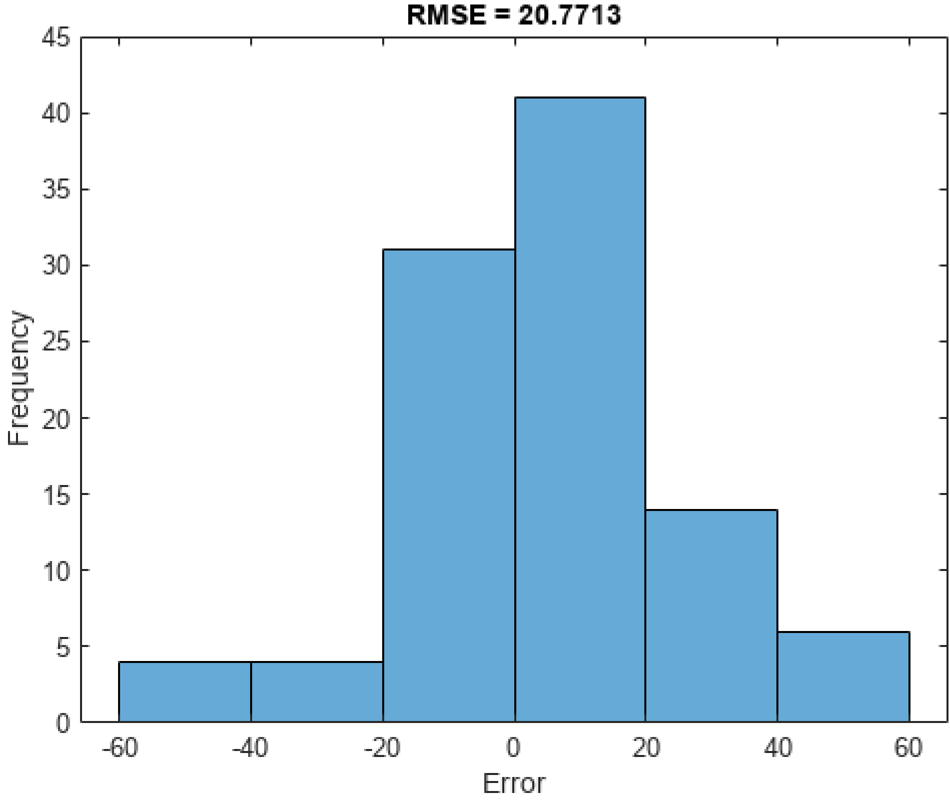


For a given partial sequence, the predicted current RUL is the last element of the predicted sequences. Calculate the root-mean-square error (RMSE) of the predictions and visualize the prediction error in a histogram.

```
for i = 1:numel(YTest)
    YTestLast(i) = YTest{i}(end);
    YPredLast(i) = YPred{i}(end);
end
figure
rmse = sqrt(mean((YPredLast - YTestLast).^2))
```

```
rmse = single
    20.7713
```

```
histogram(YPredLast - YTestLast)
title("RMSE = " + rmse)
ylabel("Frequency")
xlabel("Error")
```



References

1 Saxena, Abhinav, Kai Goebel, Don Simon, and Neil Eklund. "Damage propagation modeling for aircraft engine run-to-failure simulation." *2008 International Conference on Prognostics and Health Management* (2008): 1–9. https://doi.org/10.1109/PHM.2008.4711414.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `classify`

Deploy and Verify YOLO v2 Vehicle Detector on FPGA

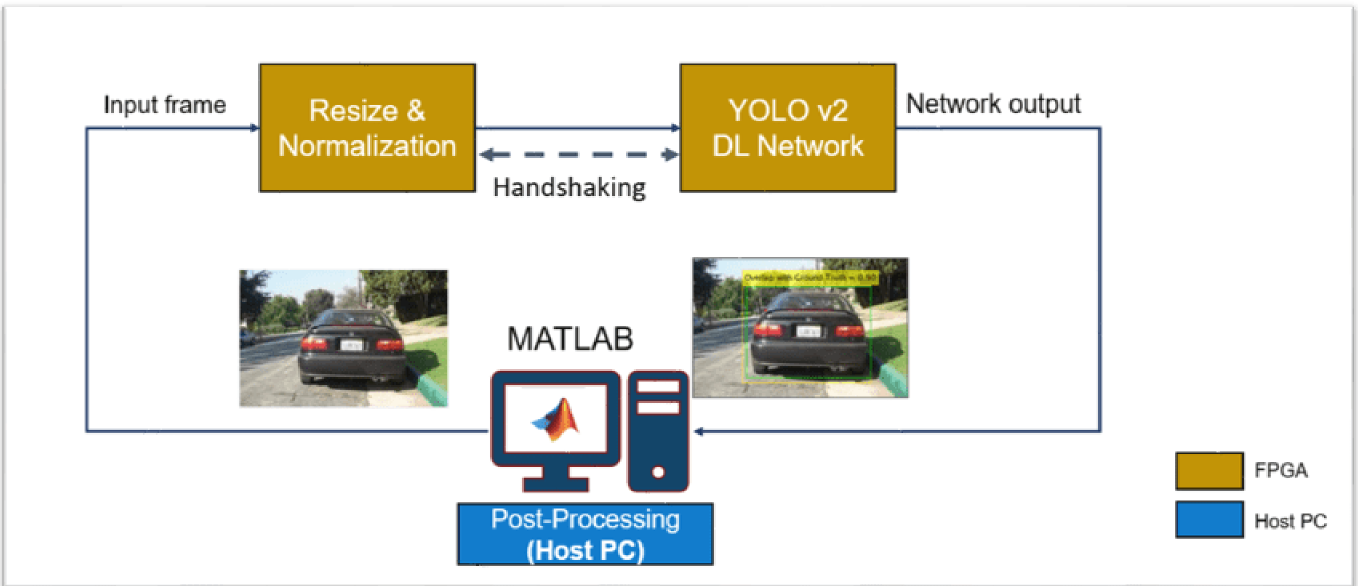
This example shows how to deploy a you only look once (YOLO) v2 vehicle detector on FPGA and verify the end-to-end application using MATLAB.

The end-to-end application includes preprocessing steps, image resize and normalization, followed by a YOLO v2 vehicle detection network.

The example deploys the algorithm to a Xilinx® Zynq® Ultrascale+(TM) MPSoC ZCU102 board. Set up the board's SD card using "Guided SD Card Set Up" (Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices).

Introduction

A YOLO v2 vehicle detection application is composed of three main modules. The first module, preprocessing, accepts the input image frame and performs image resize and normalization. In the second module, the preprocessed data is consumed by the YOLO v2 vehicle detection network, which internally comprises a feature extraction network followed by a detection network. In the third module, the network output is postprocessed for identifying the strongest bounding boxes and the resulting bounding box is overlaid on the input image. In this example, as shown in the below block diagram, the first two modules are deployed on the FPGA and the postprocessing is done in MATLAB.



This example shows how to:

1. Configure the deep learning processor and generate IP core.
2. Model the design under test (DUT) that includes preprocessing modules (resize and normalization) and handshaking logic with the deep learning processor.
3. Generate and deploy bitstream to the FPGA.
4. Compile and deploy YOLO v2 deep learning network.
5. Verify the deployed YOLO v2 vehicle detector using MATLAB.

Configure Deep Learning Processor and Generate IP Core

The deep learning processor IP core accesses the preprocessed input from the DDR memory, performs the vehicle detection, and loads the output back into the memory. To generate a deep learning processor IP core that has the required interfaces, create a deep learning processor configuration by using the `dlhdl.ProcessorConfig` class. In the processor configuration, set the `InputRunTimeControl` and `OutputRunTimeControl` parameters. These parameters indicate the interface type for interfacing between the input and output of the deep learning processor. To learn about these parameters, see "Interface with the Deep Learning Processor IP Core" on page 12-17. In this example, the deep learning processor uses the `register` mode for input and output runtime control.

```
hPC = dlhdl.ProcessorConfig;
hPC.InputRunTimeControl = "register";
hPC.OutputRunTimeControl = "register";
```

Specify the `TargetPlatform` property of the processor configuration object as `Generic Deep Learning Processor`. This option generates a custom generic deep learning processor IP core.

```
hPC.TargetPlatform = 'Generic Deep Learning Processor';
```

Use the `setModuleProperty` method to set the properties of the `conv` module of the deep learning processor. These properties can be tuned based on the design choice to ensure that the design fits on the FPGA. To learn more about these parameters, see `setModuleProperty`. In this example, `LRNBlockGeneration` is turned on and `SegmentationBlockGeneration` is turned off to support YOLOv2 vehicle detection network. `ConvThreadNumber` is set to 9.

```
hPC.setModuleProperty('conv','LRNBlockGeneration', 'on');
hPC.setModuleProperty('conv','SegmentationBlockGeneration', 'off');
hPC.setModuleProperty('conv','ConvThreadNumber',9);
```

This example uses the Xilinx ZCU102 board to deploy the deep learning processor. Use the `hdlsetuptoolpath` function to add the Xilinx Vivado synthesis tool path to the system path.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

Use the `dlhdl.buildProcessor` function with the `hPC` object to generate the deep learning IP core. It takes some time to generate the deep learning processor IP core.

```
dlhdl.buildProcessor(hPC);
```

The generated IP core contains a standard set of registers and the generated IP core report. The IP core report is generated in the same folder as ip core with the name `testbench_ip_core_report.html`.

IP core name	dlprocessor
IP core version	1.0
IP core folder	dlhdl_prj\ipcore\dlprocessor_v1_0
IP core zip file name	dlprocessor_v1_0.zip
Target platform	Generic Deep Learning Processor Xilinx
Target tool	Xilinx Vivado
Target language	VHDL
Model	testbench

`IP core name` and `IP core folder` are required in a subsequent step in 'Set Target Reference Design' task of the IP core generation workflow of the DUT. The IP core report also has the address map of the registers that are needed for handshaking with input and output of deep learning processor IP core.

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping
InputNext	Inport	boolean	AXI4	x"350"
OutputNext	Inport	boolean	AXI4	x"360"
StreamingMode	Inport	boolean	AXI4	x"34C"
InputStop	Inport	boolean	AXI4	x"374"
inputStart	Inport	boolean	AXI4	x"224"
FrameCount	Inport	uint32	AXI4	x"24C"
InputValid	Outport	boolean	AXI4	x"354"
InputAddr	Outport	uint32	AXI4	x"358"
InputSize	Outport	uint32	AXI4	x"35C"
OutputValid	Outport	boolean	AXI4	x"364"
OutputAddr	Outport	uint32	AXI4	x"368"
OutputSize	Outport	uint32	AXI4	x"36C"

The registers `InputValid`, `InputAddr`, and `InputSize` contain the values of the corresponding handshaking signals that are required to write the preprocessed frame into DDR memory. The register `inputNext` is used by the DUT to pulse the inputNext signal after the data is written into memory. These register addresses are setup in the `helperSLYOLOv2PreprocessSetup.m` script. The other registers listed in the report are read/written using MATLAB. For more details on interface signals, see the Design Processing Mode Interface Signals section of "Interface with the Deep Learning Processor IP Core" on page 12-17.

Model Design Under Test (DUT)

This section describes the design of the preprocessing modules (image resize and image normalization) and the handshaking logic in a DUT.

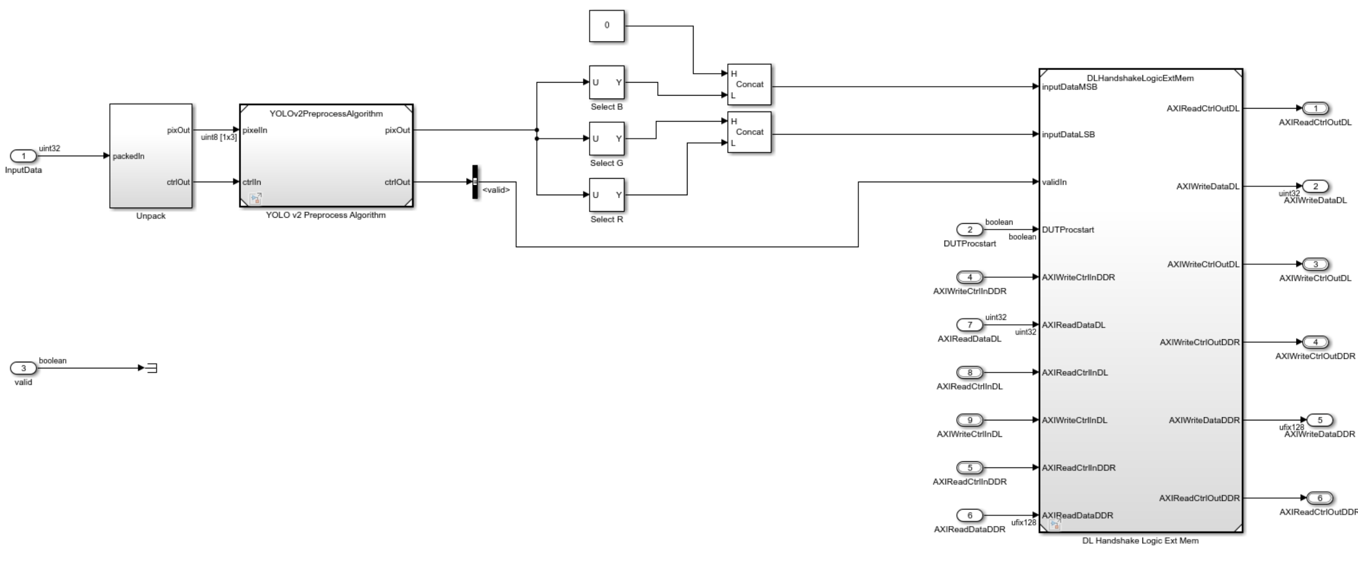
```
open_system('YOLOv2PreprocessTestbench');
```

YOLO v2 DUT - Preprocess with deep learning hand shake logic

Copyright 2021-2022 The MathWorks, Inc.

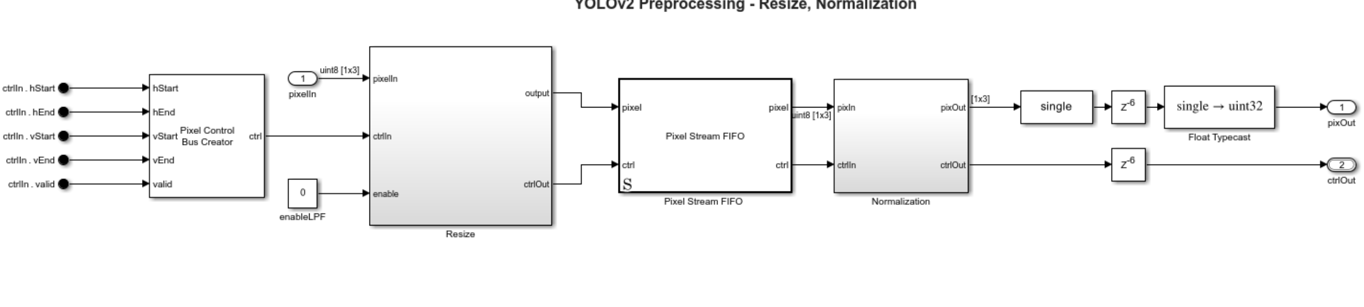
The figure shows the top level view of the `YOLOv2PreprocessTestbench.slx` model. The `InitFcn` callback of the model configures the required workspace variables for the model using `helperSLYOLOv2PreprocessSetup.m` script. The `Select Image` subsystem selects the input frame from the `Input Images` block. A `Frame To Pixels` block converts the input image frame from the `Select Image` block to a pixel stream and `pixelcontrol` bus. The `Pack` subsystem concatenates the R, G, B components of the pixel stream and the five control signals of the `pixelcontrol` bus to form `uint32` data. The packed data is fed to the `YOLO v2 Preprocess DUT` for resizing and normalization. This preprocessed data is then written to the DDR using the handshaking signals from deep learning IP core. The DDR memory and the deep learning processor IP core are modeled as `PL DDR` and `DL IP core` subsystems. The model also includes a `Verify Output` subsystem which logs the signals required for the verification of the preprocessed data being written to memory using `preprocessDUTVerify.m` script.

```
open_system('YOLOv2PreprocessTestbench/YOLO v2 Preprocess DUT');
```



The `YOLO v2 Preprocess DUT` contains subsystems for unpacking, preprocessing (resize and normalization) and handshaking logic. The `Unpack` subsystem returns the packed input to the pixel stream and `pixelcontrol` bus. In the `YOLO v2 Preprocess Algorithm` subsystem, the input pixel stream is resized and rescaled as required by the deep learning network. This preprocessed frame is then passed to the `DL Handshake Logic Ext Mem` subsystem to be written into the PL DDR. This example models two AXI4 Master interfaces to write the preprocessed frame to the DDR memory and to read and write the registers of deep learning IP Core.

```
open_system('YOLOv2PreprocessDUT/YOLO v2 Preprocess Algorithm');
```



The `YOLO v2 Preprocess Algorithm` subsystem comprises of resizing, and normalization operations. The pixel stream is passed to the `Resize` subsystem for resizing to the dimensions expected by the deep learning network. The input image dimensions and the network input dimensions are setup using `helperSLYOLOv2PreprocessSetup.m` script. The resized input is passed to `Normalization` subsystem for rescaling the pixel values to [0, 1] range. The resize and normalization algorithms used in this example are described in the “Change Image Size” (Vision HDL Toolbox) and “Image Normalization Using External Memory” (Vision HDL Toolbox) examples respectively.

```
open_system('YOLOv2PreprocessDUT/DL Handshake Logic Ext Mem');
```

Deep Learning hand shake logic with external memory

Copyright 2022 The MathWorks, Inc.

The `DL Handshake Logic Ext Mem` subsystem contains the finite state machine (FSM) logic for handshaking with DL IP and a subsystem to write the frame to DDR. The `Read DL Registers` subsystem has the FSM logic to read the handshaking signals (InputValid, InputAddr, and InputSize) from the DL IP core for multiple frames. The `Write to DDR` subsystem uses these handshaking signals to write the preprocessed frame to the memory using AXI stream protocol. The output write control bus from the DDR memory contains a signal `wr_done` which indicates that the frame write operation is done successfully. The `TriggerDLInputNext` subsystem pulses the `inputNext` signal after the preprocessed frame is written into the DDR to indicate to the DL IP core that the input data frame is available for processing.

In the next section, the IP core is generated for the `YOLO v2 Preprocess DUT` subsystem and is integrated into the reference design.

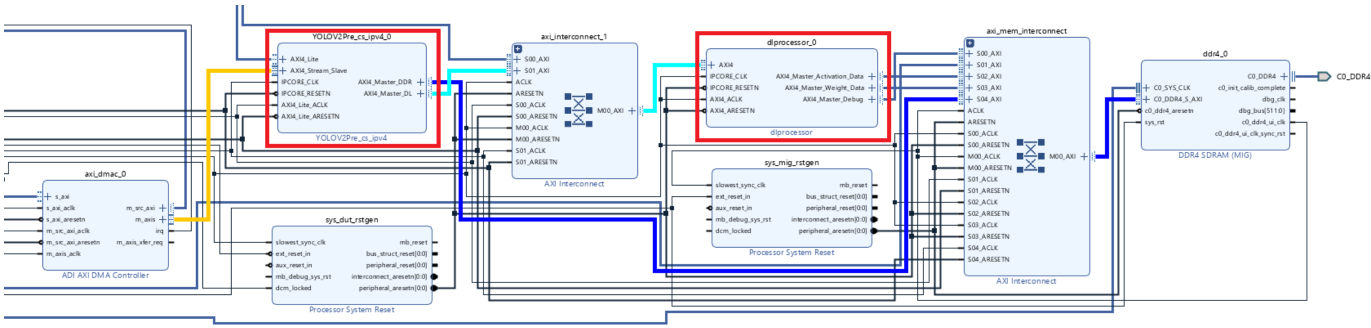
Generate and Deploy Bitstream to FPGA

This example uses the `Deep Learning with Preprocessing Interface` reference design that is provided by the Vision HDL Toolbox™ Support Package for Xilinx® Zynq®-Based Hardware.

```
pathToRefDesign = fullfile(...
    matlabshared.supportpkg.getSupportPackageRoot,...
    "toolbox","shared","supportpackages","visionzynq","target",....
    "+visionzynq", "+ZCU102", "plugin_rd.m");
if (~exist(pathToRefDesign, 'file'))
    error(['This example requires you to download and install '...
```

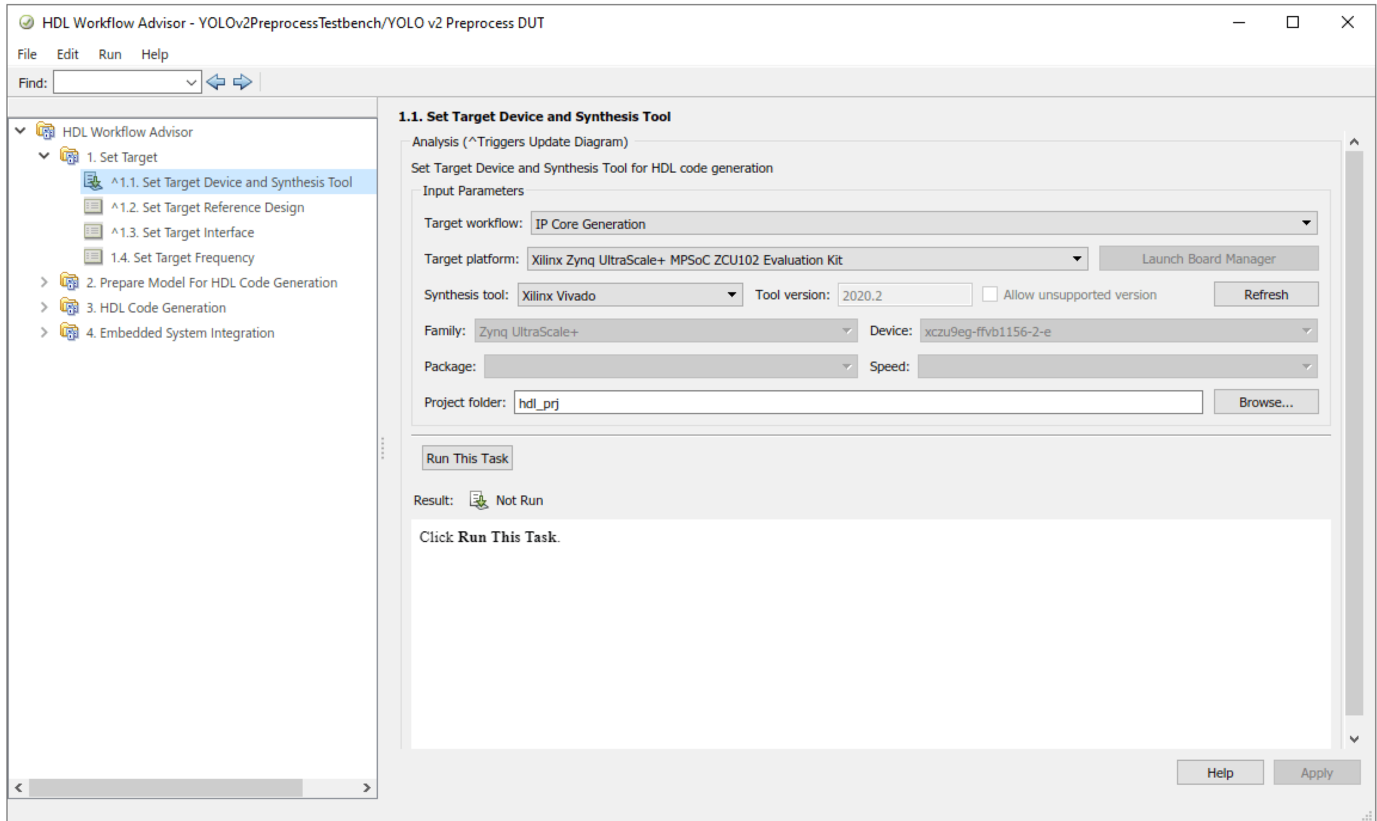
```
        'Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware']);
end
```

The reference design contains the ADI AXI DMA Controller to move the data from processor to FPGA fabric. The data is sent from the ARM processing system, through the DMA controller and AXI4-Stream interface, to the generated DUT Preprocessing IP core. The DUT contains two AXI Master interfaces. One AXI interface is connected to the Deep Learning Processor IP core and the other is connected to the DDR memory interface generator (MIG) IP.

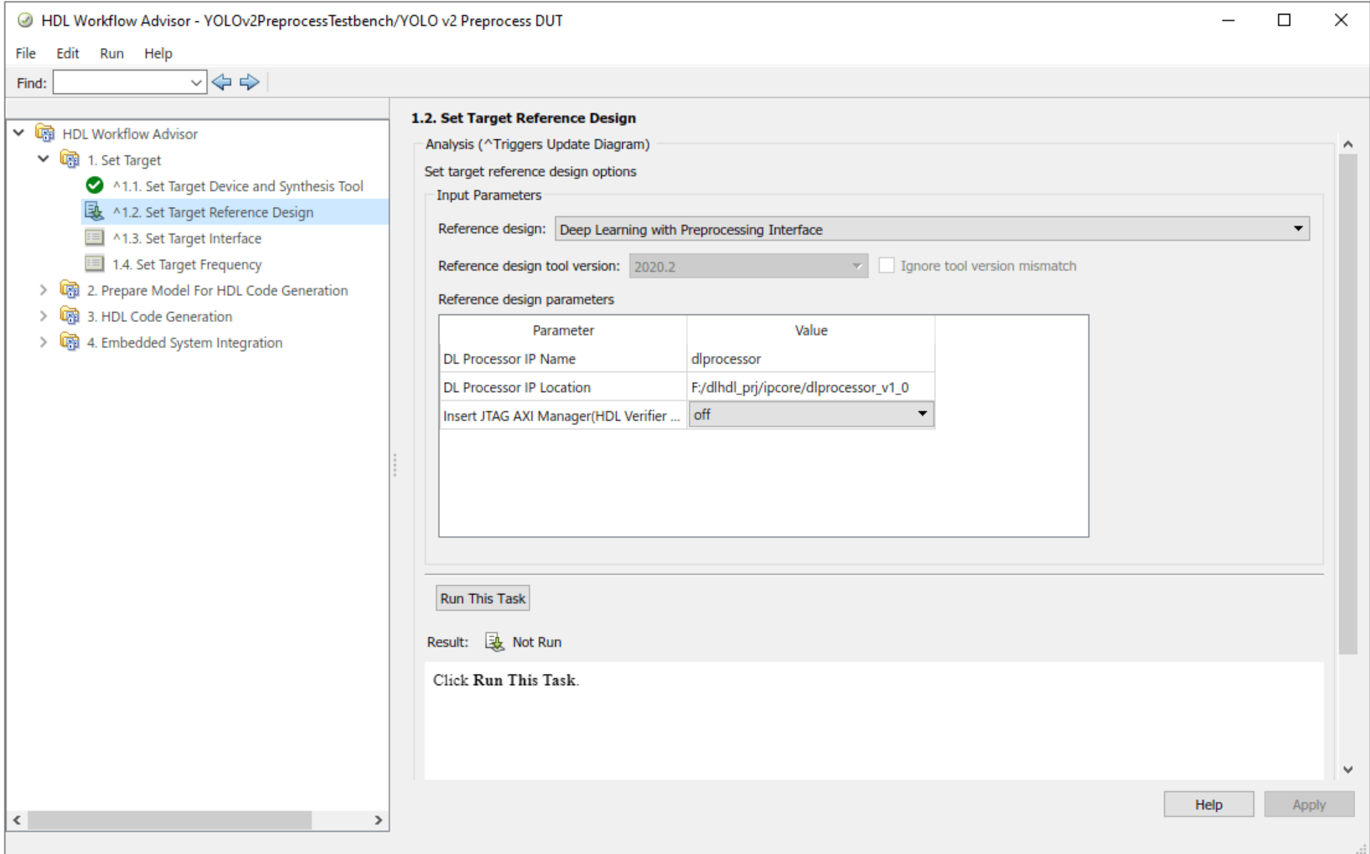


Start the targeting workflow by right clicking the `YOLO v2 Preprocess DUT` subsystem and selecting HDL Code > HDL Workflow Advisor.

- In step 1.1, select IP Core Generation workflow and the platform 'Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit'.



- In step 1.2, the reference design is set to "Deep Learning with Preprocessing Interface". The DL Processor IP name and the DL Processor IP location specify the name and location of the generated deep learning processor IP core, and are obtained from the IP core report.

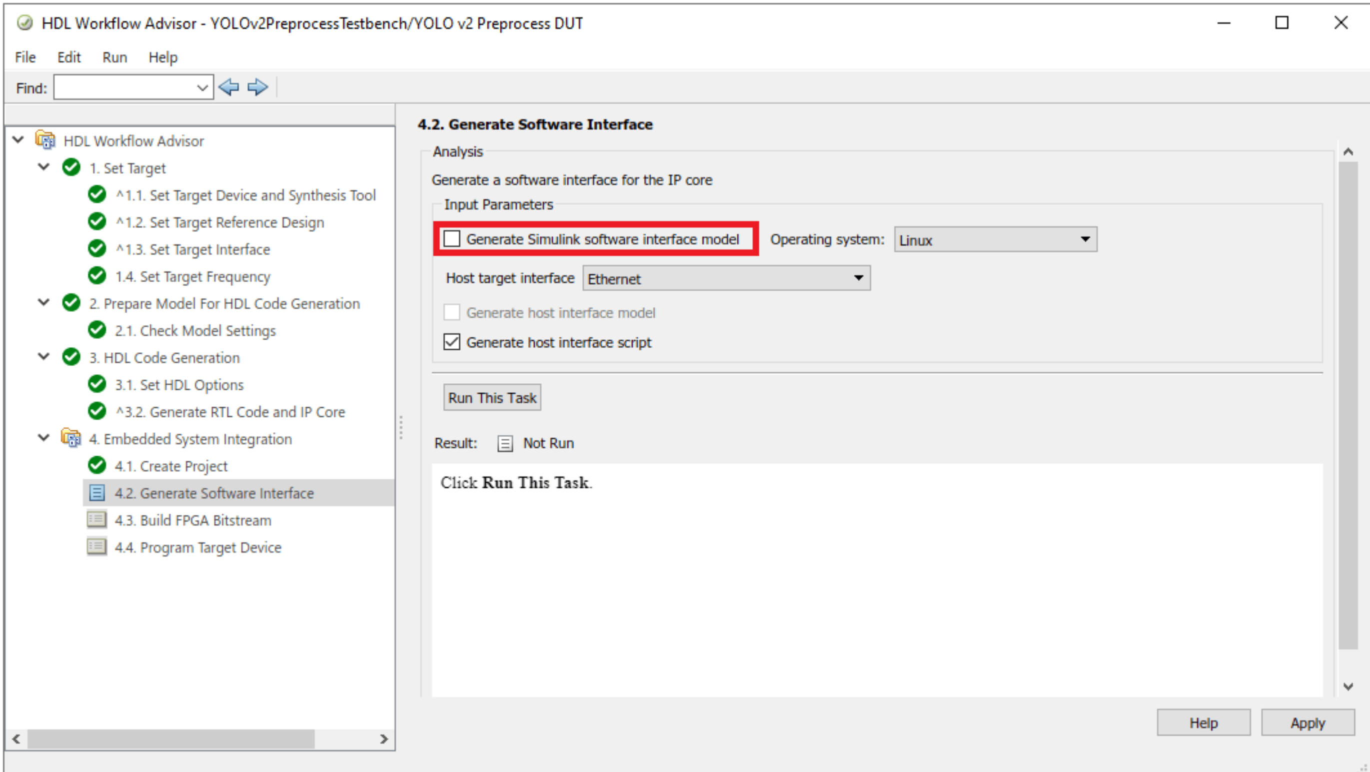


- In step 1.3, map the target platform interfaces to the input and output ports of the DUT.

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
inputData	Inport	uint32	AXI4-Stream Slave	Data	
DUTProcstart	Inport	boolean	AXI4-Lite	x"100"	Options...
valid	Inport	boolean	AXI4-Stream Slave	Valid	
AXIWriteCtrlInDDR	Inport	bus	AXI4 Master DDR Write	Write Slave to Master B	
AXIReadCtrlInDDR	Inport	bus	AXI4 Master DDR Read	Read Slave to Master Bu	
AXIReadDataDDR	Inport	ufix128	AXI4 Master DDR Read	Data	
AXIReadDataDL	Inport	uint32	AXI4 Master DL Read	Data	
AXIReadCtrlInDL	Inport	bus	AXI4 Master DL Read	Read Slave to Master Bu	
AXIWriteCtrlInDL	Inport	bus	AXI4 Master DL Write	Write Slave to Master B	
AXIReadCtrlOutDL	Outport	bus	AXI4 Master DL Read	Read Master to Slave Bu	
AXIWriteDataDL	Outport	uint32	AXI4 Master DL Write	Data	
AXIWriteCtrlOutDL	Outport	bus	AXI4 Master DL Write	Write Master to Slave B	
AXIWriteCtrlOutDDR	Outport	bus	AXI4 Master DDR Write	Write Master to Slave B	
AXIWriteDataDDR	Outport	ufix128	AXI4 Master DDR Write	Data	
AXIReadCtrlOutDDR	Outport	bus	AXI4 Master DDR Read	Read Master to Slave Bu	

- **AXI4-Stream Slave interface**: The `inputData` and `valid` ports of the DUT are mapped to the data and valid ports of the AXI4-Stream Slave interface respectively.
- **AXI4-Lite Interface**: The `DUTProcstart` register is mapped to the AXI4-Lite register. When this register is written, it triggers the process of input handshaking logic. Choosing the AXI4-Lite interface directs HDL Coder to generate a memory-mapped register in the FPGA fabric. You can access this register from software running on the ARM processor.
- **AXI4 Master DDR interface**: The `AXIWriteCtrlInDDR`, `AXIReadCtrlInDDR`, `AXIReadDataDDR`, `AXIWriteCtrlOutDDR`, `AXIWriteDataDDR` and `AXIReadCtrlOutDDR` ports of DUT are mapped to AXI4 Master DDR interface. The **Read Channel** of the AXI4 Master DDR interface is mapped to the AXI4 Master DDR Read interface, and the **Write Channel** of the AXI4 Master DDR interface is mapped to the AXI4 Master DDR Write interface. This interface is used for the data transfer between the Preprocess DUT and the PL DDR. Using the Write Channel of this interface, the preprocessed data is written to the PL DDR which can then be accessed by the Deep Learning Processor IP.
- **AXI4 Master DL interface**: The `AXIReadDataDL`, `AXIReadCtrlInDL`, `AXIWriteCtrlInDL`, `AXIReadCtrlOutDL`, `AXIWriteDataDL` and `AXIWriteCtrlOutDL` ports of DUT are mapped to AXI4 Master DL interface. The **Read Channel** of the AXI4 Master DL interface is mapped to the AXI4 Master DL Read interface, and the **Write Channel** of the AXI4 Master DL interface is mapped to the AXI4 Master DL Write interface. This interface is used for the communication between Preprocess DUT and the Deep Learning Processor IP. In this example, this interface is used for implementing input handshaking logic with Deep Learning Processor IP.
- Step 2 prepares the design for hdl code generation.
- Step 3 generates HDL code for the IP core.
- Step 4.1 integrates the newly generated IP core into the reference design.
- In step 4.2, the host interface script and Zynq software interface model is created. Since this example uses the interface script, and not the model, uncheck **Generate Simulink software**

interface model. The host interface script, `gs_YOLOv2PreprocessTestbench_interface`, generated in this step is parameterized and provided as `setupPreprocessIPInterfaces.m` function as part of this example.



- Step 4.3 generates the bitstream. The bit file is named `block_design_wrapper.bit` and located at hdl_prj\vivado_ip_prj\vivado_prj.runs\impl_1. This bitstream is downloaded to FPGA in the next section.

Compile and Deploy Yolo v2 Deep Learning Network

Now that the bitstream is generated for the IP core of the DUT integrated with the reference design that contains the DL IP core, you can deploy the end to end deep learning application onto an FPGA.

Create a target object to connect your target device to the host computer. Use the installed Xilinx Vivado Design Suite over an Ethernet connection to program the device.

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet','IpAddr','192.168.1.101');
```

Load the pretrained YOLO v2 object detection network.

```
vehicleDetector = load('yolov2VehicleDetector.mat');
detector = vehicleDetector.detector;
net = detector.Network;
```

Update the bitstream build information in the MAT file generated during the IP core generation. The name of the MAT file is `dlprocessor.mat` and is located in cwd\dlhdl_prj\, where cwd is your current working folder. Copy the file to the present working folder. This MAT file generated using the target platfom `Generic Deep Learning Processor` does not contain the Board/Vendor

information. Use `updateBitstreamBuildInfo.m` function to update the Board/Vendor information and generate a new MAT file with the same name as generated bitstream.

```
bitstreamName = 'block_design_wrapper';
updateBitstreamBuildInfo('dlprocessor.mat',[bitstreamName,'.mat']);
```

Create a deep learning HDL workflow object using the `dlhdl.Workflow` class.

```
hW = dlhdl.Workflow('Network',net,'Bitstream',[bitstreamName,'.bit'],'Target',hTarget);
```

Compile the network, `net` using the `dlhdl.Workflow` object.

```
frameBufferCount = 3;
compile(hW, 'InputFrameNumberLimit', frameBufferCount);
```

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
hSOC = xilinxsoc('192.168.1.101', 'root', 'root');
```

Call the `xilinxsoc` object function `ProgramFPGA` to program the FPGA and set the device tree to use the processor on the SoC board.

```
programFPGA(hSOC, [bitstreamName,'.bit'], 'devicetree_vision_dlhdl.dtb');
```

Run the `deploy` function of the `dlhdl.Workflow` object to download the network weights and biases on the Zynq UltraScale+ MPSoC ZCU102 board.

```
deploy(hW, 'ProgramBitStream', false);
```

Clear the DLHDL workflow object and hardware target.

```
clear hW;
clear hTarget;
```

Verify Deployed YOLO v2 Vehicle Detector Using MATLAB

The function `YOLOv2DeployAndVerifyDetector` takes `hSOC` object as input and performs vehicle detection using the YOLO v2 network deployed on FPGA and verifies the end-to-end application using MATLAB.

```
YOLOv2DeployAndVerifyDetector(hSOC);
```

This flowchart shows the operations performed in the function.

This section describes the steps in the flowchart in detail.

Load the vehicle data set

```
datasetLocation = [matlabroot, filesep, 'examples', filesep, 'deeplearning_shared', filesep, 'da
unzip([datasetLocation, filesep, 'vehicleDatasetImages.zip']);
data = load([datasetLocation, filesep, 'vehicleDatasetGroundTruth.mat']);
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes. Add the fullpath to the local vehicle data folder.

```
vehicleDataset.imageFilename = fullfile(pwd,vehicleDataset.imageFilename);
```

Select images from the vehicle dataset. Each image present in `inputDataTbl` has 224 rows and 340 columns.

```
inputDataTbl = vehicleDataset(153:259,:);
```

Setup deep learning and preprocessing interfaces

Connect to the FPGA on-board the SoC board by using the fpga function. Use the processor hardware object `hSOC` as an input to the `fpga` function.

```
hFPGA = fpga(hSOC);
```

Get network input and output size. The `networkOutputSize` is the output size of `yolov2ClassConv` obtained from `analyzeNetwork(net)`.

```
networkInputSize = net.Layers(1, 1).InputSize;
networkOutputSize = [16,16,24];
```

The deep learning processor writes the `yolov2ClassConv` layer output to the external memory in a specified data format. This data format depends on the chosen `ConvThreadNumber` of the deep learning processor. `readLengthFromDLIP` contains the output data size. For more information, see "External Memory Data Format" on page 12-9

```
readLengthFromDLIP = (networkOutputSize(1)*networkOutputSize(2)*networkOutputSize(3)*4)/3;
```

Setup the deep learning IP interfaces using `setupDLIPInterfaces.m` function. This function uses `BitstreamManager` class to obtain the address map of the deep learning IP core registers.

```
addrMap = setupDLIPInterfaces(hFPGA, [bitstreamName,'.bit'], readLengthFromDLIP);
ddrbaseAddr = dec2hex(addrMap('ddrbase'));
```

Get image dimensions and create `visionhdl.FrameToPixels` System object™

```
frm = imread(inputDataTbl.imageFilename{1});

frmActivePixels = size(frm,2);
frmActiveLines = size(frm,1);

frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',size(frm,3),...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+10,...
      'TotalVideoLines',frmActiveLines+10,...
      'StartingActiveLine',6,...
      'FrontPorch',5);
```

Setup the preprocess IP interfaces using `setupPreprocessIPInterfaces.m` function.

```
inputFrameLength = frm2pix.TotalPixelsPerLine * frm2pix.TotalVideoLines;
setupPreprocessIPInterfaces(hFPGA, inputFrameLength);
```

Configure deep learning IP core

Set data processing mode to continuous streaming mode by setting `StreamingMode` register to `true` and `FrameCount` register to `0`.

```
writePort(hFPGA, "StreamingMode", 1);
writePort(hFPGA, "FrameCount", 0);
```

Pulse the `inputStart` signal to indicate to the deep learning IP core to start processing the data.

```
writePort(hFPGA, "inputStart", 0);
writePort(hFPGA, "inputStart", 1);
writePort(hFPGA, "inputStart", 0);
```

Assert `DUTProcStart` to signal preprocess DUT to start writing the preprocessed data to the DDR.

```
writePort(hFPGA, "DUTProcStart", 1);
```

Send input video frame to YOLO v2 Preprocess DUT

Use `packPixelAndControlBus.m` function to pack the pixel and control bus data. The `frm2pix` object converts the input video frame to a pixel stream and `pixelcontrol` bus. Then, the R, G, B components of the pixel data and the `hStart`, `hEnd`, `vStart`, `vEnd`, and `valid` signals of the `pixelcontrol` bus are packed to generate 32 bit data, as shown.

This packed input is fed to the `YOLO v2 Preprocess DUT` using the `writePort` function of `fpga` object. The input is preprocessed and written to the memory by the DUT. The deep learning IP core reads the data from memory, performs the vehicle detection, and writes the output back to the memory.

```
writePort(hFPGA, "InputData", inputImagePacked);
```

Read output data and perform postprocessing

The deep learning IP core returns handshaking signals indicating address, size, and validity of the output. When the `outputValid` signal becomes `true`, the script reads the processed output data frame using the `outputAddr` and `outputSize` signals. The `readDataFromPLDDR.m` function reads the output data using the `readPort` function of `fpga` object.

```
outputValid = readPort(hFPGA, "OutputValid");
while(outputValid~=1)
    pause(0.1);
    outputValid = readPort(hFPGA, "OutputValid");
end

outData = readDataFromPLDDR(hFPGA, ddrbaseAddr);
```

After reading the output data from DDR, pulse the OutputNext signal by using the `hFPGA` object

```
writePort(hFPGA, "OutputNext", 0);
writePort(hFPGA, "OutputNext", 1);
writePort(hFPGA, "OutputNext", 0);
```

The `yolov2TransformLayerAndPostProcess.m` function performs the transform layer processing and postprocessing on the `outData`, and returns the bounding boxes.

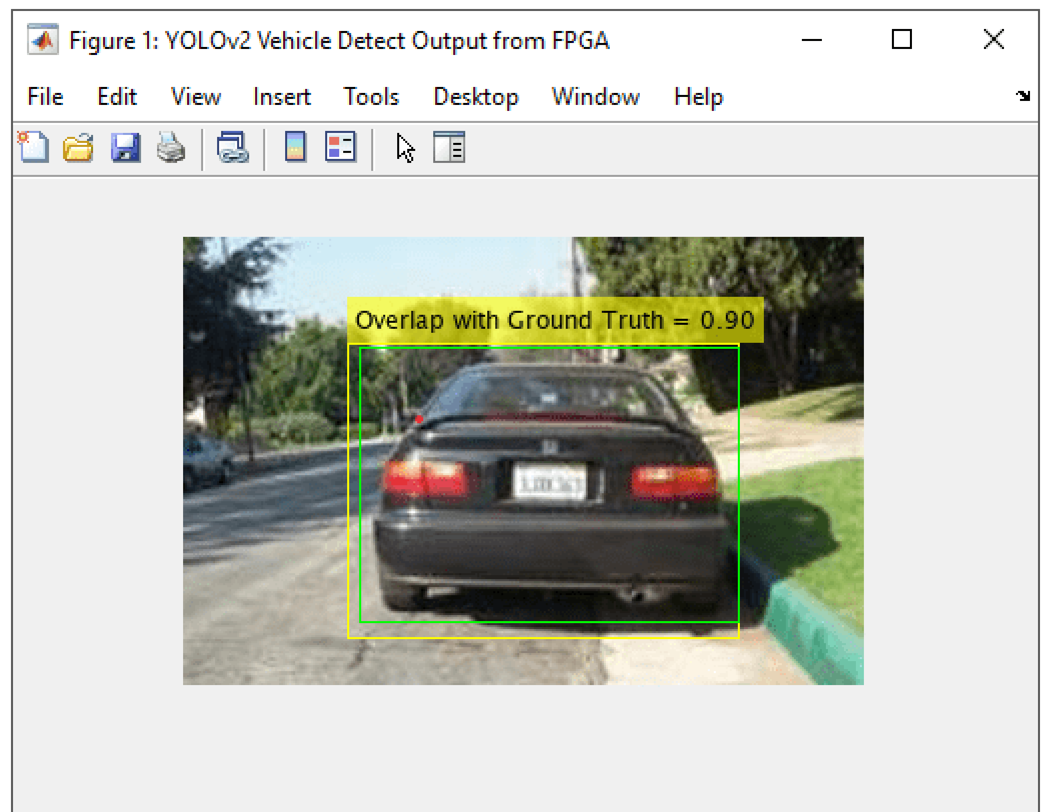
```
anchorBoxes = detector.AnchorBoxes;
[bboxes, scores] = yolov2TransformLayerAndPostProcess(outData, inputImage, networkInputSize, net
```

Verify postprocessed output

The bounding boxes obtained from the post-processing and the ground truth are overlaid on the input image along with the overlap ratio.

```
bboxesGT = inputDataTbl.vehicle{imIndex};
```

```
overlapRatio = bboxOverlapRatio(bboxes, bboxesGT);

bbOverlap = sprintf("Overlap with Ground Truth = %0.2f", overlapRatio);
outputImage = insertObjectAnnotation(inputImage,'rectangle',bboxes,bbOverlap);
outputImage = insertObjectAnnotation(outputImage,'rectangle',bboxesGT, '', 'Color', 'green');
imshow(outputImage);
```



Pulse inputStop signal

After processing all the frames, pulse the inputStop signal by using the `hFPGA` object.

```
writePort(hFPGA, "InputStop", 0);
writePort(hFPGA, "InputStop", 1);
writePort(hFPGA, "InputStop", 0);
```

Conclusion

This example deployed the YOLO v2 vehicle detector application comprising of preprocessing steps (image resize and normalization) and handshaking logic on FPGA, performed vehicle detection, and verified the results using MATLAB.

For information about debugging the design deployed on the FPGA, see the “Debug YOLO v2 Vehicle Detector on FPGA” (Vision HDL Toolbox) example. This example shows how to use FPGA data capture and AXI manager features of the HDL Verifier™ product to capture the required data for debugging from the FPGA.

Deploy Semantic Segmentation Network Using Dilated Convolutions on FPGA

This example shows how to deploy a trained semantic segmentation network that uses dilated convolutions to a Xilinx® Zynq® Ultrascale+™ ZCU102 SoC development kit. Semantic segmentation networks like DeepLab [1] make extensive use of dilated convolutions, also known as `atrous convolutions` because they can increase the receptive field of the layer without increasing the number of parameters or computations.

A semantic segmentation network classifies every pixel in an image,which results in an image that is segmented by class. Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis.

The network attached to this example was created in the example “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

To obtain an improved frames per second (FPS) performance, you can quantize the network. This example shows how to calibrate and deploy a quantized network and then compare the performance between the quantized network and a single data type network.

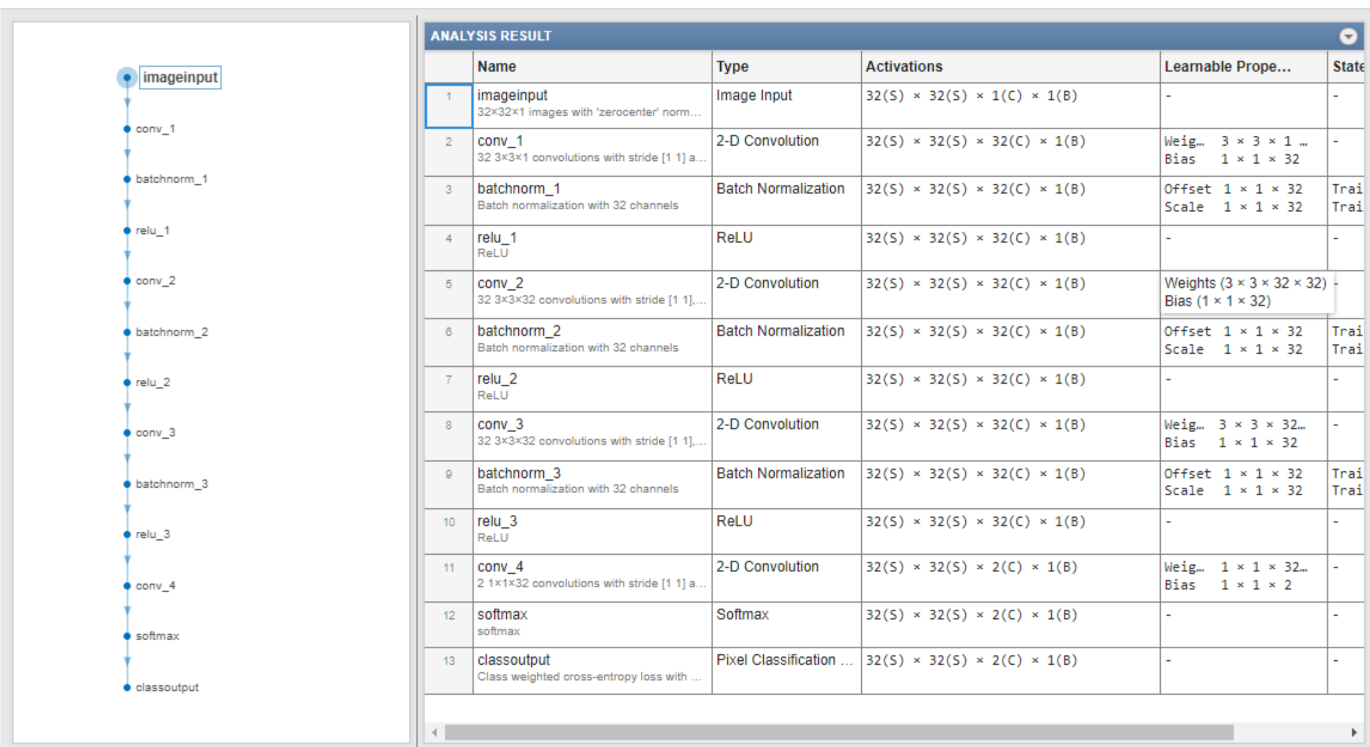
Load the Pretrained Network

To load the pretrained semantic segmentation network, enter:

```
load("trainedSemanticSegmentationNet.mat")
```

Use the `analyzeNetwork` function to view a graphical representation of the network and detailed parameter settings for the layers in the network.

```
analyzeNetwork(net)
```



	Name	Type	Activations	Learnable Prope...	State
1	imageinput 32×32×1 images with 'zerocenter' norm...	Image Input	32(S) × 32(S) × 1(C) × 1(B)	-	-
2	conv_1 32 3×3×1 convolutions with stride [1 1] a...	2-D Convolution	32(S) × 32(S) × 32(C) × 1(B)	Weig… 3 × 3 × 1 … Bias 1 × 1 × 32	-
3	batchnorm_1 Batch normalization with 32 channels	Batch Normalization	32(S) × 32(S) × 32(C) × 1(B)	Offset 1 × 1 × 32 Scale 1 × 1 × 32	Trai Trai
4	relu_1 ReLU	ReLU	32(S) × 32(S) × 32(C) × 1(B)	-	-
5	conv_2 32 3×3×32 convolutions with stride [1 1],...	2-D Convolution	32(S) × 32(S) × 32(C) × 1(B)	Weights (3 × 3 × 32 × 32) Bias (1 × 1 × 32)	-
6	batchnorm_2 Batch normalization with 32 channels	Batch Normalization	32(S) × 32(S) × 32(C) × 1(B)	Offset 1 × 1 × 32 Scale 1 × 1 × 32	Trai Trai
7	relu_2 ReLU	ReLU	32(S) × 32(S) × 32(C) × 1(B)	-	-
8	conv_3 32 3×3×32 convolutions with stride [1 1],...	2-D Convolution	32(S) × 32(S) × 32(C) × 1(B)	Weig… 3 × 3 × 32… Bias 1 × 1 × 32	-
9	batchnorm_3 Batch normalization with 32 channels	Batch Normalization	32(S) × 32(S) × 32(C) × 1(B)	Offset 1 × 1 × 32 Scale 1 × 1 × 32	Trai Trai
10	relu_3 ReLU	ReLU	32(S) × 32(S) × 32(C) × 1(B)	-	-
11	conv_4 2 1×1×32 convolutions with stride [1 1] a...	2-D Convolution	32(S) × 32(S) × 2(C) × 1(B)	Weig… 1 × 1 × 32… Bias 1 × 1 × 2	-
12	softmax softmax	Softmax	32(S) × 32(S) × 2(C) × 1(B)	-	-
13	classoutput Class weighted cross-entropy loss with ...	Pixel Classification ...	32(S) × 32(S) × 2(C) × 1(B)	-	-

Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Specify that the interface is for a Xilinx board with an Ethernet interface.

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
```

To use the JTAG interface, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado tool path, enter:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object and specifying the network and bitstream name. Ensure that the bitstream name matches the data type and FPGA board. In this example, the target FPGA board is the Xilinx ZCU102 SOC board and the bitstream uses a single data type.

```
wfObj = dlhdl.Workflow('network', net, 'Bitstream', 'zcu102_single','Target',hTarget);
```

To run the example on a Xilinx ZC706 board, enter:

```
hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'zc706_single','Target',hTarget);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment. Because the total number of frames exceeds the default value of 30, set the `InputFrameNumberLimit` to `64` to run predictions in chunks of 64 frames to prevent timeouts.

```
dn = compile(wfObj,'InputFrameNumberLimit',64)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'imageinput' of type 'ImageInputLayer' is split into an image input layer
### The network includes the following layers:
     1   'imageinput'    Image Input                  32×32×1 images with 'zerocenter' normaliza
     2   'conv_1'        2-D Convolution              32 3×3×1 convolutions with stride [1  1] an
     3   'relu_1'        ReLU                         ReLU
     4   'conv_2'        2-D Convolution              32 3×3×32 convolutions with stride [1  1],
     5   'relu_2'        ReLU                         ReLU
     6   'conv_3'        2-D Convolution              32 3×3×32 convolutions with stride [1  1],
     7   'relu_3'        ReLU                         ReLU
     8   'conv_4'        2-D Convolution              2 1×1×32 convolutions with stride [1  1] an
     9   'softmax'       Softmax                      softmax
    10   'classoutput'   Pixel Classification Layer   Class weighted cross-entropy loss with clas

### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwa
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.PixelClassificationLayer' is implem
### Compiling layer group: conv_1>>conv_4 ...
### Compiling layer group: conv_1>>conv_4 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "28.0 MB"
    "InstructionDataOffset"     "0x02800000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02c00000"     "4.0 MB"
    "EndOffset"                 "0x03000000"     "Total: 48.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {{}  [-247.8812 0 0 0 -247.8812 0 0 0 -247.8812 0 0 0 -247.8812 0 0 0 -247
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` method of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board and download the network weights and biases. The `deploy` function programs the FPGA device and displays progress messages, and the required time to deploy the network.

```
deploy(wfObj)
```

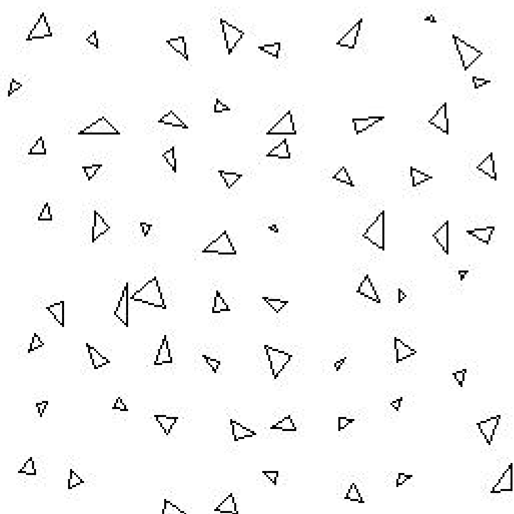
```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
```

```
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_single.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_single.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 09-Nov-2022 12:03:53
```

Load Test Image

Read the example image

```
imgTest = imread('triangleTest.jpg');
figure
imshow(imgTest)
```



Run Prediction for Test Image

Segment the test image using `semanticseg_FPGA` and display the results using `labeloverlay`.

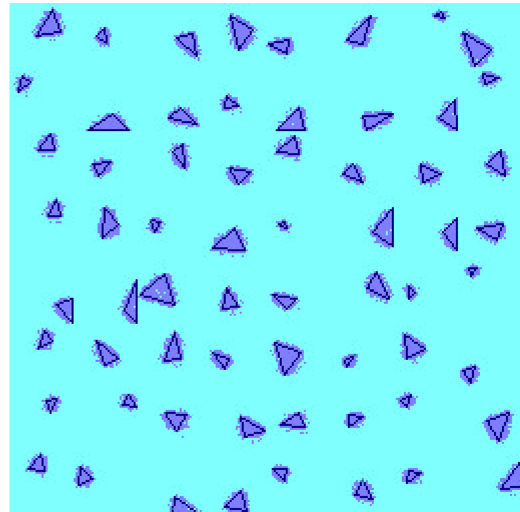
```
networkInputSize = net.Layers(1).InputSize(1:2);
imgTestSize = size(imgTest);

assert(all(mod(imgTestSize(1:2), networkInputSize)) == 0, 'The 2D image input size should be a mu

numberOfBlocks = imgTestSize./networkInputSize;
```

```
totalBlocks = prod(numberOfBlocks);

splitImage = mat2cell(imgTest, networkInputSize(1)*ones(1, numberOfBlocks(1)), networkInputSize(1
multiFrameInput = zeros([networkInputSize 1 totalBlocks]);
for i=1:totalBlocks
    multiFrameInput(:,:,:,i) = splitImage{i};
end


result = semanticseg_FPGA(multiFrameInput, wfObj.Network, wfObj);
```

```
### Finished writing input activations.
### Running in multi-frame mode with 64 inputs.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                     254070                  0.00115                      64          162
    imageinput_norm           7574                  0.00003
    conv_1                   26062                  0.00012
    conv_2                   97124                  0.00044
    conv_3                   97116                  0.00044
    conv_4                   26175                  0.00012
 * The clock frequency of the DL processor is: 220MHz
```

The performance of the single data type network is 865.6 frames per second. Concatenate the images to highlight the triangles identified in the input image.

```
concatenatedResult = [];
for i=1:numberOfBlocks(2)
    subset = result(:,:,numberOfBlocks(1)*(i-1)+1:i*numberOfBlocks(1));
    verticalConcatenation = [];
    for j=1:numberOfBlocks(1)
        verticalConcatenation = [verticalConcatenation; subset(:,:,j)];
    end
    concatenatedResult = [concatenatedResult verticalConcatenation];
end

croppedFinal = labeloverlay(imgTest, concatenatedResult);
figure
imshow(croppedFinal)
```



Prepare the Quantized Network for Deployment

Load the data. The data set includes 32-by-32 triangle images.

```
dataFolder = fullfile(toolboxdir('vision'),'visiondata','triangleImages');
imageFolderTrain = fullfile(dataFolder,'trainingImages');
```

Create an image datastore for the images by using an `imageDatastore` object.

```
imdsTrain = imageDatastore(imageFolderTrain);
```

Create a quantized network by using `dlquantizer`. Set the target execution environment to `FPGA`.

```
dlQuantObj = dlquantizer(net,'ExecutionEnvironment','FPGA');
```

Use the `calibrate` function to exercise the network and collect range information for the learnable parameters in the network layers.

```
dlQuantObj.calibrate(imdsTrain);
```

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and FPGA board. In this example the target FPGA board is the Xilinx ZCU102 SOC board. The bitstream uses an `int8` data type.

```
wfObj_int8 = dlhdl.Workflow('Network', dlQuantObj, 'Bitstream', 'zcu102_int8', 'Target', hTarget
```

Compile and Deploy the Quantized Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment. Because the total number of frames exceeds the default value of 30, set the `InputFrameNumberLimit` to `64` to run predictions in chunks of 64 frames to prevent timeouts.

```
dn = compile(wfObj_int8,'InputFrameNumberLimit',64)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### The network includes the following layers:
     1   'imageinput'    Image Input                   32×32×1 images with 'zerocenter' normaliza
     2   'conv_1'        2-D Convolution               32 3×3×1 convolutions with stride [1  1] an
     3   'relu_1'        ReLU                          ReLU
     4   'conv_2'        2-D Convolution               32 3×3×32 convolutions with stride [1  1],
     5   'relu_2'        ReLU                          ReLU
     6   'conv_3'        2-D Convolution               32 3×3×32 convolutions with stride [1  1],
     7   'relu_3'        ReLU                          ReLU
     8   'conv_4'        2-D Convolution               2 1×1×32 convolutions with stride [1  1] an
     9   'softmax'       Softmax                       softmax
    10   'classoutput'   Pixel Classification Layer   Class weighted cross-entropy loss with clas

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.PixelClassificationLayer' is implem
### Compiling layer group: conv_1>>conv_4 ...
### Compiling layer group: conv_1>>conv_4 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "0.0 MB"
    "SystemBufferOffset"        "0x00800000"     "28.0 MB"
    "InstructionDataOffset"     "0x02400000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02800000"     "4.0 MB"
    "EndOffset"                 "0x02c00000"     "Total: 44.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
```

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` method of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board and download the network weights and biases. The `deploy` function programs the FPGA device and displays progress messages, and the required time to deploy the network.

```
deploy(wfObj_int8)
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
```

```
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_int8.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_int8.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 09-Nov-2022 12:06:24
```

Run Prediction

Segment the test image using `semanticseg_FPGA` and display the results using `labeloverlay`.

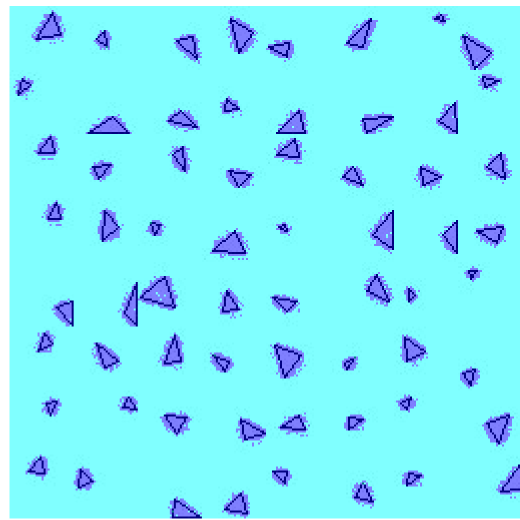
```
networkInputSize = net.Layers(1).InputSize(1:2);
imgTestSize = size(imgTest);


assert(all(mod(imgTestSize(1:2), networkInputSize)) == 0, 'The 2D image input size should be a mu

numberOfBlocks = imgTestSize./networkInputSize;
totalBlocks = prod(numberOfBlocks);


splitImage = mat2cell(imgTest, networkInputSize(1)*ones(1, numberOfBlocks(1)), networkInputSize(1
multiFrameInput = zeros([networkInputSize 1 totalBlocks]);
for i=1:totalBlocks
    multiFrameInput(:,:,:,i) = splitImage{i};
end


result_int8 = semanticseg_FPGA(multiFrameInput, wfObj_int8.Network, wfObj_int8);
```

```
### Finished writing input activations.
### Running in multi-frame mode with 64 inputs.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                     85510                  0.00034                       64           54
    conv_1                  13576                  0.00005
    conv_2                  29679                  0.00012
    conv_3                  29933                  0.00012
    conv_4                  12303                  0.00005
 * The clock frequency of the DL processor is: 250MHz
```

The quantized network has a performance of 2921.9 frames per second. Concatenate the images to highlight the triangles identified in the input image.

```
concatenatedResult_int8 = [];
for i=1:numberOfBlocks(2)
```

```
    subset = result_int8(:,:,numberOfBlocks(1)*(i-1)+1:i*numberOfBlocks(1));
    verticalConcatenation_int8 = [];
    for j=1:numberOfBlocks(1)
        verticalConcatenation_int8 = [verticalConcatenation_int8; subset(:,:,j)];
    end
    concatenatedResult_int8 = [concatenatedResult_int8 verticalConcatenation_int8];
end

croppedFinal_int8 = labeloverlay(imgTest, concatenatedResult_int8);
figure
imshow(croppedFinal_int8)
```



References

[1] Chen, Liang-Chieh, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. "Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation." arXiv, August 22, 2018. http://arxiv.org/abs/1802.02611.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `dlquantizer` | `calibrate`

Run Sequence Forecasting Using a GRU Layer on an FPGA

Reduce the time to train a sequence forecasting network by swapping out the LSTM later for a gated recurrent unit (GRU) layer. Use the deployed network to predict future values by using open-loop and closed-loop forecasting. Use MATLAB® to retrieve the prediction results from the target device.

Modified Waveform Data Network

The network attached to this example was trained using the "Time Series Forecasting Using Deep Learning". In this example the LSTM layer was swapped out for a GRU layer. This example uses the `WaveformData.mat` data set, which contains 2000 synthetically generated waveforms of varying lengths with three channels. This example uses a trained network with a GRU layer to forecast future values of the waveforms given the values from the previous time steps using both closed loop and open loop forecasting.

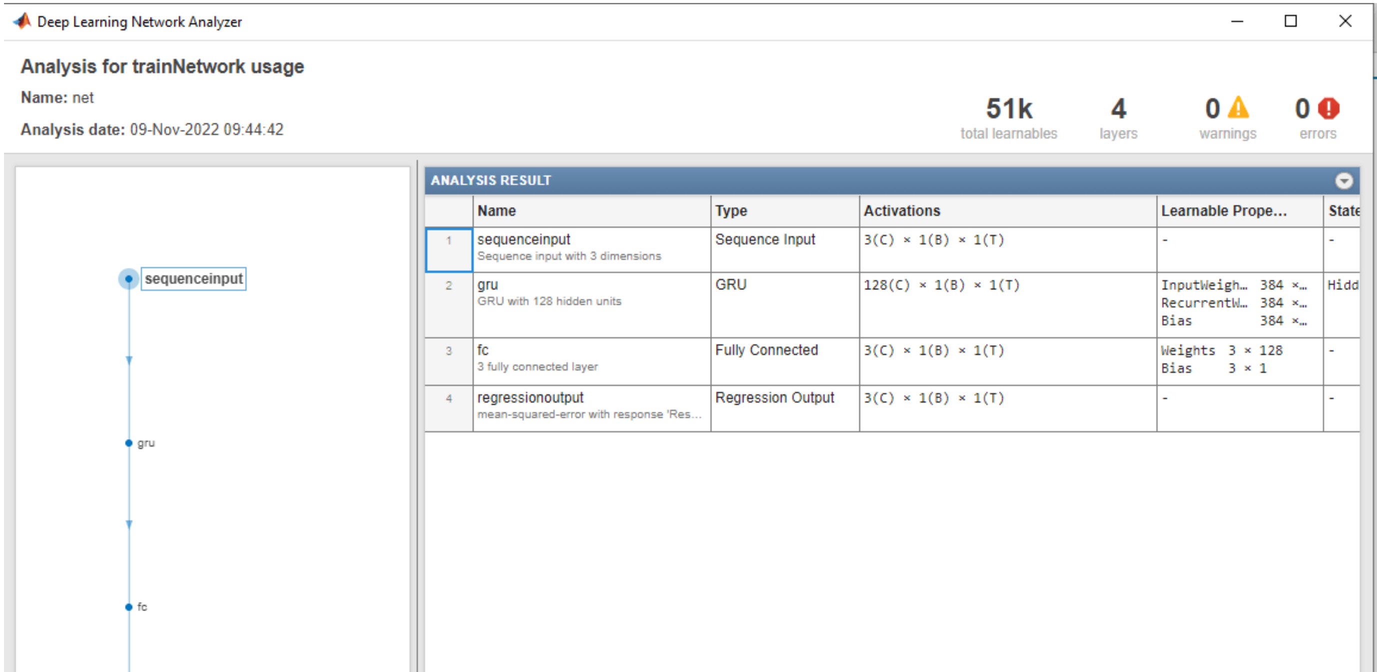
Load the Pretrained Network

To load the GRU layer network enter:

```
load grunet
```

Use the `analyzeNetwork` function to obtain information about the network layers. the function returns a graphical representation of the network that contains detailed parameter information for every layer in the network.

```
analyzeNetwork(net)
```



Define FPGA Board Interface

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Specify that the interface is for a Xilinx board with an Ethernet interface.

To create the target object, enter:

```
hTarget_gru = dlhdl.Target('Xilinx',Interface='Ethernet');
```

To use the JTAG interface, install Xilinx™ Vivado™ Design Suite 2020.2. To set the Xilinx Vivado toolpath, enter:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.2\bin\vivado.ba
hTarget = dlhdl.Target('Xilinx',Interface='JTAG');
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and the bitstream name. Ensure that the bitstream name matches the data type and the FPGA board. In this example the target FPGA board is the Xilinx ZCU102 SOC board. The bitstream uses a single data type.

```
hW_gru = dlhdl.Workflow(Network=net,Bitstream='zcu102_lstm_single',Target=hTarget_gru);
```

Tu run the example on the Xilinx ZC706 board, enter:

```
hW = dlhdl.Workflow(Network=net,Bitstream='zc706_lstm_single',Target=hTarget);
```

Compile the GRU Layer Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment. The total number of frames exceeds the default value of 30. Set the `InputFrameNumberLimit` name-value argument to `1000` to run predictions in chunks of 1000 frames to prevent timeouts.

```
dn = compile(hW_gru,'InputFrameNumberLimit',1000)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_lstm_single.
### The network includes the following layers:
     1   'sequenceinput'      Sequence Input      Sequence input with 3 dimensions             (
     2   'gru'                GRU                 GRU with 128 hidden units                    (
     3   'fc'                 Fully Connected     3 fully connected layer                      (
     4   'regressionoutput'   Regression Output   mean-squared-error with response 'Response'  (

### Notice: The layer 'sequenceinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented 
### Notice: The layer 'regressionoutput' with type 'nnet.cnn.layer.RegressionOutputLayer' is imp
### Compiling layer group: gru.wh ...
### Compiling layer group: gru.wh ... complete.
### Compiling layer group: gru.rh ...
### Compiling layer group: gru.rh ... complete.
### Compiling layer group: gru.w1 ...
### Compiling layer group: gru.w1 ... complete.
### Compiling layer group: gru.w2 ...
### Compiling layer group: gru.w2 ... complete.
### Compiling layer group: fc ...
### Compiling layer group: fc ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    ________________
```

```
    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "20.0 MB"
    "InstructionDataOffset"     "0x02000000"     "4.0 MB"
    "FCWeightDataOffset"        "0x02400000"     "4.0 MB"
    "EndOffset"                 "0x02800000"     "Total: 40.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {{1×2 cell}  [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
             ddrInfo: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` function of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The `deploy` function starts programming the FPGA device and displays progress messages, and the required time to deploy the network.

```
 deploy(hW_gru)
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Deep learning network programming has been skipped as the same network is already loaded on t
```

Test Network

Prepare the test data for prediction. Normalize the test data using the statistics calculated from the training data. Forecast the values using the GRU layer network. To forecast the values of future time steps of a sequence, specify the targets as the test sequences with values shifted by one time step. In other words, at each time step of the input sequence, the GRU layer network learns to predict the value of the next time step.

```
load Waveformdata
numChannels = size(data{1},1);
numObservations = numel(data);

idxTrain = 1:floor(0.9*numObservations);
idxTest = floor(0.9*numObservations)+1:numObservations;
dataTrain = data(idxTrain);
dataTest = data(idxTest);

for n = 1:numel(dataTrain)
    X = dataTrain{n};
    XTrain{n} = X(:,1:end-1);
    TTrain{n} = X(:,2:end);
end

muX = mean(cat(2,XTrain{:}),2);
sigmaX = std(cat(2,XTrain{:}),0,2);
muT = mean(cat(2,TTrain{:}),2);
```

```
sigmaT = std(cat(2,TTrain{:}),0,2);

for n = 1:size(dataTest,1)
    X = dataTest{n};
    XTest{n} = (X(:,1:end-1) - muX) ./ sigmaX;
    TTest{n} = (X(:,2:end) - muT) ./ sigmaT;
end
```

Make predictions using the test data.

```
YTest_gru = predict(hW_gru,XTest{1},Profile = 'on');
```

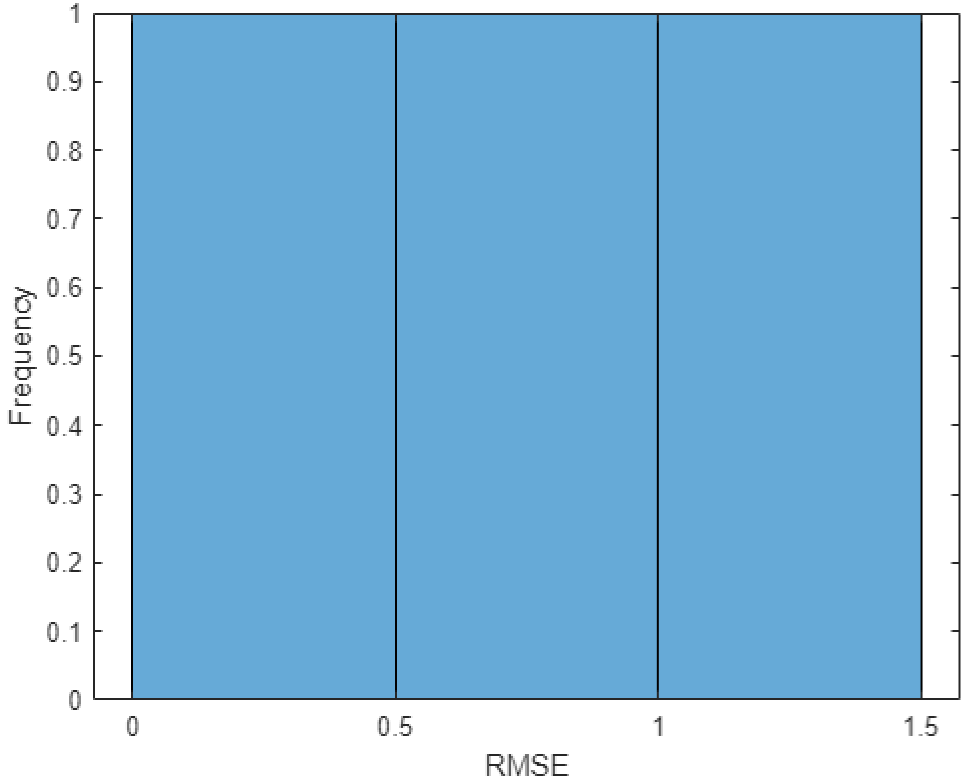
```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 115.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------            -------------               ---------        --
Network                     32322                  0.00015                       115           37
    gru.wh                    548                  0.00000
    gru.rh                   7538                  0.00003
    memSeparator_0             98                  0.00000
    gru.w1                   7469                  0.00003
    gru.w2                   7649                  0.00003
    gru.sigmoid_1             222                  0.00000
    gru.sigmoid_2             214                  0.00000
    gru.multiplication_2      288                  0.00000
    gru.multiplication_4      334                  0.00000
    gru.multiplication_1      344                  0.00000
    gru.addition_2            294                  0.00000
    gru.addition_1            294                  0.00000
    gru.tanh_1                198                  0.00000
    gru.multiplication_3      288                  0.00000
    gru.addition_3            298                  0.00000
    fc                       6246                  0.00003
 * The clock frequency of the DL processor is: 220MHz
```

To evaluate the accuracy, calculate the root mean squared error (RMSE) between the predictions and the target for each test sequence.

```
for i = 1:size(YTest_gru,1)
    rmse(i) = sqrt(mean((YTest_gru(i) - TTest{1}(i)).^2,"all"));
end
```

Visualize the errors in a histogram. Lower values indicate greater accuracy.

```
figure
histogram(rmse)
xlabel("RMSE")
ylabel("Frequency")
```



Calculate the mean RMSE over all test observations.

```
mean(rmse)
```

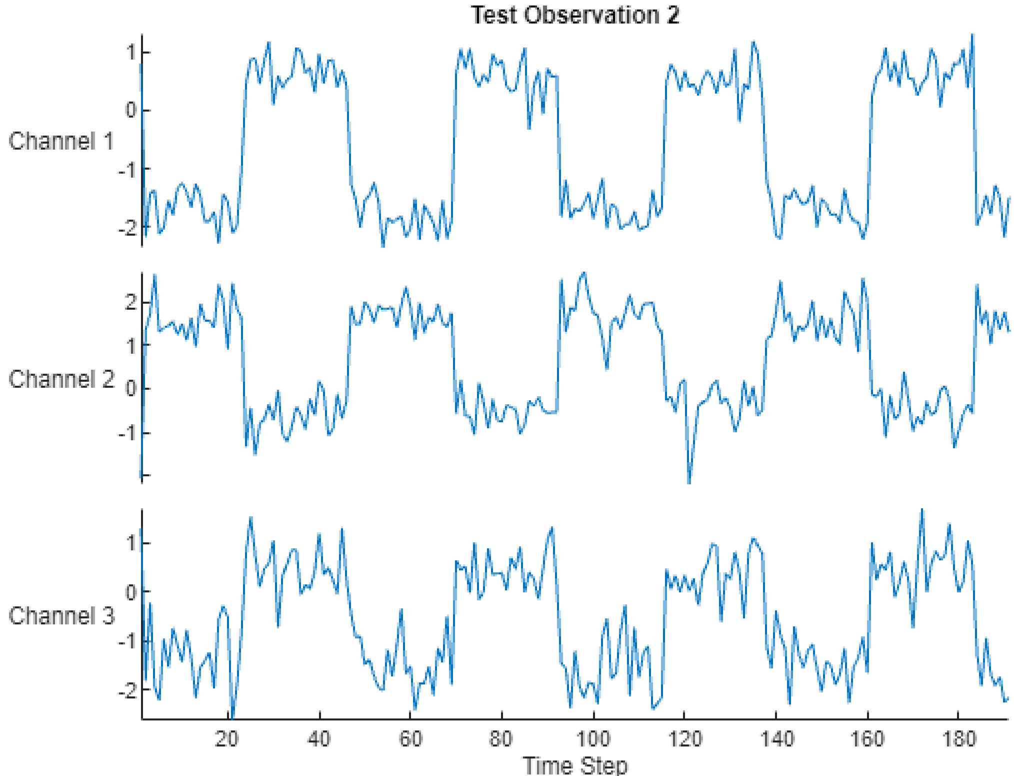
```
ans = single
    0.7688
```

Forecast Future Time Steps

To forecast the values of multiple future time steps, when given an input time series or sequence, use the `predictAndUpdateState` function. This function predicts time steps one at a time and updates the network state at each prediction. For each prediction, use the previous prediction as the input to the function.

Visualize one of the test sequences in a plot.

```
idx = 2;
X_gru = XTest{idx};
T_gru = TTest{idx};

figure
stackedplot(X_gru',DisplayLabels="Channel " + (1:numChannels))
xlabel("Time Step")
title("Test Observation " + idx)
```



Open-Loop Forecasting

Open-loop forecasting predicts the next time step in a sequence using only the input data. When making predictions for subsequent time steps, you collect the true values form your data source and use those as input. For example, suppose that you want to predict the value for time step t of a sequence by using data collected in time steps 1 through $t-1$. To make predictions for time step $t+1$, wait until you record the true value for time step t and use that value as input to make the next prediction. Use open-loop forecasting when you have true values to provide to the network before making the next prediction.

Initialize the network state by resetting the state using the `resetState` function, then make an initial prediction using the first few time steps of the input data. Update the network state by using the first 75 time steps of the input data.

```
resetState(hW_gru)
offset = 75;
[~,~] = predictAndUpdateState(hW_gru,X_gru(:,1:offset));
```

```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 75.
```

To forecast further predictions, loop over time steps and update the network state by using the `predictAndUpdateState` function. Forecast values for the remaining time steps of the test observation by looping over the time steps of the input data and using them as input to the network. The first prediction is the value that corresponds to the time step `offset + 1`.

```
numTimeSteps = size(X_gru,2);
numPredictionTimeSteps = numTimeSteps - offset;
Y_gru = zeros(numChannels,numPredictionTimeSteps);

for t = 1:numPredictionTimeSteps
    Xt_gru = X_gru(:,offset+t);
    Y_gru(:,t) = predictAndUpdateState(hW_gru,Xt_gru);
end

### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

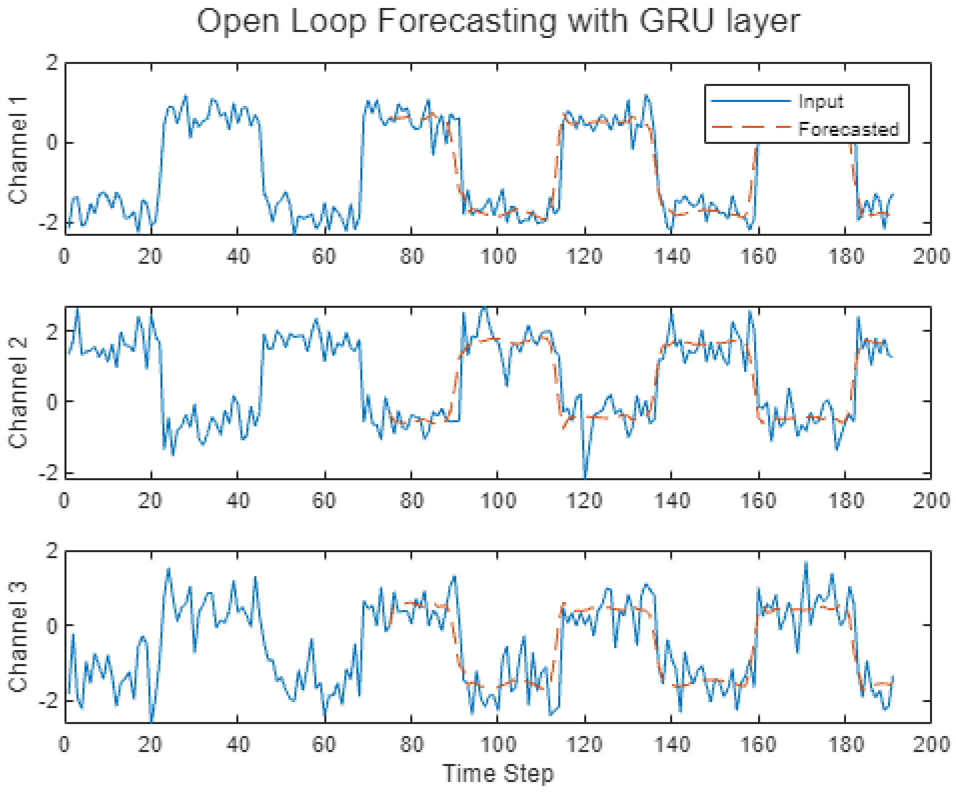
```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
```

Compare the predictions with the target values.

```
figure
t = tiledlayout(numChannels,1);
title(t,"Open Loop Forecasting with GRU layer")

for i = 1:numChannels
    nexttile
    plot(T_gru(i,:))
    hold on
    plot(offset:numTimeSteps,[T_gru(i,offset) Y_gru(i,:)],'--')
    ylabel("Channel " + i)
end

xlabel("Time Step")
nexttile(1)
legend(["Input" "Forecasted"])
```



Closed-Loop Forecasting

Closed-loop forecasting predicts subsequent time steps in a sequence by using the previous predictions as input. In this case, the model does not require the true values to make the prediction. For example, suppose that you want to predict the value for time steps t through $t + k$ of the sequence by using data collected in time steps 1 through $t - 1$. To make predictions for time step i, use the predicted value for time step $i - 1$ as input. Use closed-loop forecasting to forecast multiple subsequent time steps or when you do not have true values to provide to the network before making the next prediction.

Initialize the network state by resetting the state using the `resetState` function, then make an initial prediction, `Z,` using the first few time steps of the input data. Update the network state by using the first 75 time steps of the input data.

```
resetState(hW_gru)
offset = size(X_gru,2);
[Z, ~] = predictAndUpdateState(hW_gru,X_gru);
```

```
### Resetting network state.
### Finished writing input activations.
### Running a sequence of length 191.
```

To forecast further predictions, loop over time steps and update the network state by using the `predictAndUpdateState` function. Forecast the next 200 time steps by iteratively passing the previously predicted value to the network. Because the network does not require the input data to make any further predictions, you can specify any number of time steps to forecast.

```
numPredictionTimeSteps = 200;
Xt_gru = Z(:,end);
Y_gru = zeros(numChannels,numPredictionTimeSteps);

for t = 1:numPredictionTimeSteps
    [Y_gru(:,t),~] =  predictAndUpdateState(hW_gru,Xt_gru);
    Xt_gru = Y_gru(:,t);
end
```

```
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

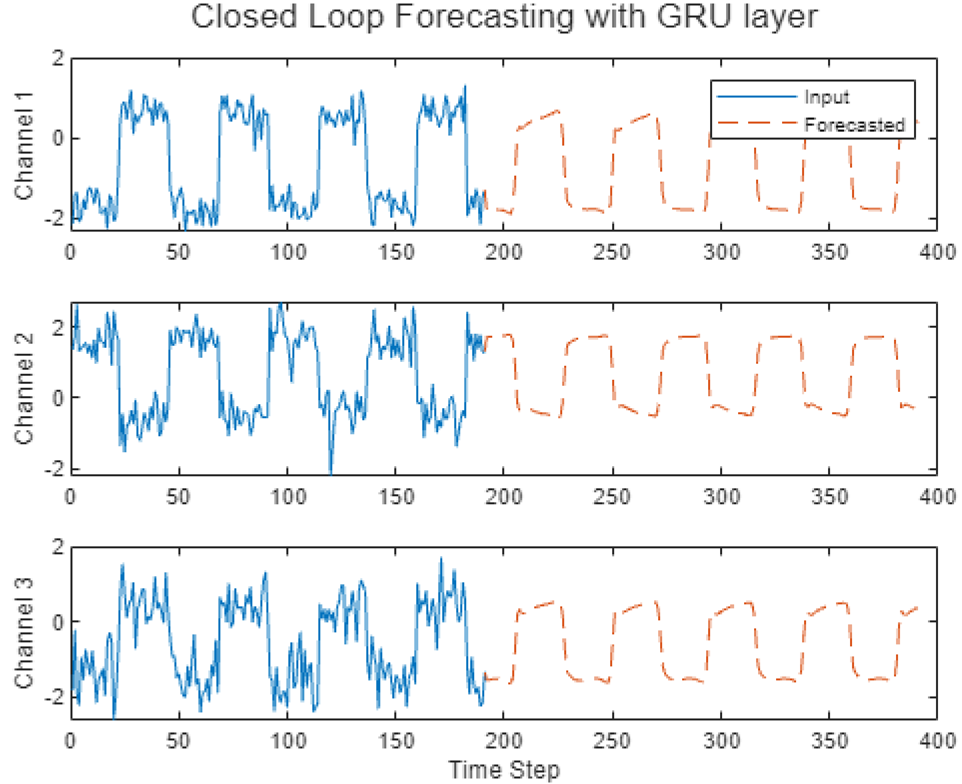
```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
```

```
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
### Finished writing input activations.
### Running a sequence of length 1.
```

Visualize the forecasted values in a plot.

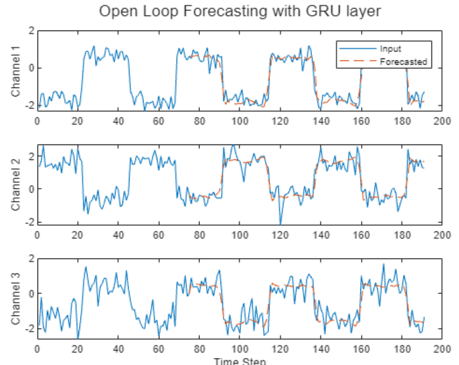
```
numTimeSteps = offset + numPredictionTimeSteps;

figure
t = tiledlayout(numChannels,1);
title(t,"Closed Loop Forecasting with GRU layer")

for i = 1:numChannels
    nexttile
    plot(T_gru(i,1:offset))
    hold on
    plot(offset:numTimeSteps,[T_gru(i,offset) Y_gru(i,:)],'--')
    ylabel("Channel " + i)
end

xlabel("Time Step")
nexttile(1)
legend(["Input" "Forecasted"])
```

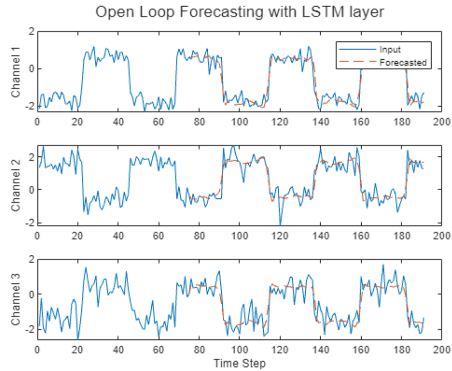


Closed-loop forecasting allows you to forecast an arbitrary number of time steps, but can be less accurate when compared to open-loop forecasting because the network does not have access to the true values during the forecasting process.

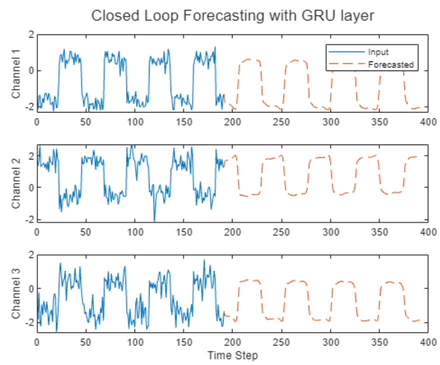
Compare Network Predictions

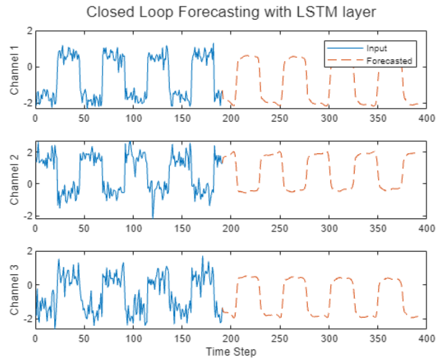
Compare the predictions of the LSTM layer network to the GRU layer network. This image shows the compariuson between the GRU layer network and LSTM layer network for open loop forecasting. The GRU layer network has a performance of 6734.9 frames per second and the LSTM layer network has a performance of 5632.3 frames per second. To learn how to deploy the LSTM layer network to an FPGA, see “Run Sequence Forecasting on FPGA by Using Deep Learning HDL Toolbox” on page 10-262.





This image shows the comparison between the GRU layer network and LSTM layer network for closed loop forecasting.





See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `predictAndUpdateState` | `resetState`

Deploy YAMNet Networks to FPGAs With and Without Cross-Layer Equalization

This example shows how to deploy a YAMNet network with and without cross-layer equalization to an FPGA. Cross -layer equalization can improve quantized network performance by reducing the variance of the network learnable parameters in the channels while maintaining the original network mapping. You can compare the accuracies of the network with and without cross-layer equalization.

Prerequisites

- Xilinx™ Zynq® Ultrascale+™ ZCU102 SoC development kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC
- Audio Toolbox™
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library Support Package
- GPU Coder™
- GPU Coder Interface for Deep Learning Libraries

Load Pretrained YAMNet Network and Download Data

The YAMNet sound classification network is trained on the AudioSet data set to predict audio events from the AudioSet ontology. This example classifies sounds from air compressor. The AudioSet data set includes recordings from air compressors[1].

The data set is classified into one healthy state and seven faulty states, for a total of eight classes.

To download and load the pretrained YAMNet network and a set of air compressor sounds, run these commands.

```
url = 'https://ssd.mathworks.com/supportfiles/audio/YAMNetTransferLearning.zip';
AirCompressorLocation = pwd;
dataFolder = fullfile(AirCompressorLocation,'YAMNetTransferLearning');

if ~exist(dataFolder,'dir')
    disp('Downloading pretrained network ...')
    unzip(url,AirCompressorLocation)
end
addpath(fullfile(AirCompressorLocation,'YAMNetTransferLearning'))
```

The final element of the `Layers` property is the classification output layer. The `Classes` property of this layer contains the names of the classes learned by the network.

```
load("airCompressorNet.mat");
net = airCompressorNet;
net.Layers(end).Classes
```

```
ans = 8×1 categorical
     Bearing
     Flywheel
     Healthy
     LIV
```

```
LOV
NRV
Piston
Riderbelt
```

Use the `analyzeNetwork` function to obtain information about the network layers. The function returns a graphical representation of the network that contains detailed parameter information for every layer in the network.

```
analyzeNetwork(net)
```

Create Calibration Data

The YAMNet network accepts inputs of size 96-by-64-by-1. Use the `getyamnet_CLEInput` helper function to obtain preprocessed mel spectograms of size 96-by-64-by-1. See Helper Functions on page 10-361. For best quantization results, the calibration data must be representative of actual inputs that re predicted by the YAMNet network. Expedite the calibration process by reducing the calibration data set to 20 mel spectograms.

```
numSpecs = 20;
[imOutCal, imdsCal] = getyamnet_CLEInput(numSpecs);
```

Calculate Accuracy of Quantized YAMNet Without Cross-Layer Equalization

Create a YAMNet network that does not use cross-layer equalization, quantize the network, and calculate its accuracy.

Calibrate and Quantize Network

Create a quantized network by using the `dlquantizer` object. Set the target execution environment to FPGA.

```
dlQuantObj = dlquantizer(net,ExecutionEnvironment='FPGA');
```

Use the `calibrate` function to exercise the network with sample inputs, collect the dynamic ranges of the weights and biases, and returns a table. Each row of the table contains range information for a learnable parameter of the quantized network.

```
calibrate(dlQuantObj,imdsCal);
```

Deploy the Quantized Network

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Specify that the interface is for a Xilinx board with an Ethernet interface.

```
hTarget = dlhdl.Target('Xilinx',Interface='Ethernet');
```

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and the bitstream name. Ensure that the bitstream matches the data type and the FPGA board. In this example, the target FPGA is the Xilinx ZCU102 SOC board. The bitstream uses an `int8` data type.

```
hW = dlhdl.Workflow(Network=dlQuantObj,Bitstream='zcu102_int8',Target=hTarget);
```

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment. Because the total number of frames exceeds the default value of 30, set the `InputFrameNumberLimit` to 100 to run predictions in chunks of 100 frames to prevent timeouts.

```
dn = compile(hW,InputFrameNumberLimit=100)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### The network includes the following layers:
     1   'input_1'                      Image Input                 96×64×1 images
     2   'conv2d'                       2-D Convolution             32 3×3×1 convolutions with str
     3   'activation'                   ReLU                        ReLU
     4   'depthwise_conv2d'             2-D Grouped Convolution     32 groups of 1 3×3×1 convoluti
     5   'activation_1'                 ReLU                        ReLU
     6   'conv2d_1'                     2-D Convolution             64 1×1×32 convolutions with st
     7   'activation_2'                 ReLU                        ReLU
     8   'depthwise_conv2d_1'           2-D Grouped Convolution     64 groups of 1 3×3×1 convoluti
     9   'activation_3'                 ReLU                        ReLU
    10   'conv2d_2'                     2-D Convolution             128 1×1×64 convolutions with s
    11   'activation_4'                 ReLU                        ReLU
    12   'depthwise_conv2d_2'           2-D Grouped Convolution     128 groups of 1 3×3×1 convolut
    13   'activation_5'                 ReLU                        ReLU
    14   'conv2d_3'                     2-D Convolution             128 1×1×128 convolutions with
    15   'activation_6'                 ReLU                        ReLU
    16   'depthwise_conv2d_3'           2-D Grouped Convolution     128 groups of 1 3×3×1 convolut
    17   'activation_7'                 ReLU                        ReLU
    18   'conv2d_4'                     2-D Convolution             256 1×1×128 convolutions with
    19   'activation_8'                 ReLU                        ReLU
    20   'depthwise_conv2d_4'           2-D Grouped Convolution     256 groups of 1 3×3×1 convolut
    21   'activation_9'                 ReLU                        ReLU
    22   'conv2d_5'                     2-D Convolution             256 1×1×256 convolutions with
    23   'activation_10'                ReLU                        ReLU
    24   'depthwise_conv2d_5'           2-D Grouped Convolution     256 groups of 1 3×3×1 convolut
    25   'activation_11'                ReLU                        ReLU
    26   'conv2d_6'                     2-D Convolution             512 1×1×256 convolutions with
    27   'activation_12'                ReLU                        ReLU
    28   'depthwise_conv2d_6'           2-D Grouped Convolution     512 groups of 1 3×3×1 convolut
    29   'activation_13'                ReLU                        ReLU
    30   'conv2d_7'                     2-D Convolution             512 1×1×512 convolutions with
    31   'activation_14'                ReLU                        ReLU
    32   'depthwise_conv2d_7'           2-D Grouped Convolution     512 groups of 1 3×3×1 convolut
    33   'activation_15'                ReLU                        ReLU
    34   'conv2d_8'                     2-D Convolution             512 1×1×512 convolutions with
    35   'activation_16'                ReLU                        ReLU
    36   'depthwise_conv2d_8'           2-D Grouped Convolution     512 groups of 1 3×3×1 convolut
    37   'activation_17'                ReLU                        ReLU
    38   'conv2d_9'                     2-D Convolution             512 1×1×512 convolutions with
    39   'activation_18'                ReLU                        ReLU
    40   'depthwise_conv2d_9'           2-D Grouped Convolution     512 groups of 1 3×3×1 convolut
    41   'activation_19'                ReLU                        ReLU
    42   'conv2d_10'                    2-D Convolution             512 1×1×512 convolutions with
    43   'activation_20'                ReLU                        ReLU
    44   'depthwise_conv2d_10'          2-D Grouped Convolution     512 groups of 1 3×3×1 convolut
    45   'activation_21'                ReLU                        ReLU
    46   'conv2d_11'                    2-D Convolution             512 1×1×512 convolutions with
    47   'activation_22'                ReLU                        ReLU
    48   'depthwise_conv2d_11'          2-D Grouped Convolution     512 groups of 1 3×3×1 convolut
    49   'activation_23'                ReLU                        ReLU
    50   'conv2d_12'                    2-D Convolution             1024 1×1×512 convolutions with
    51   'activation_24'                ReLU                        ReLU
    52   'depthwise_conv2d_12'          2-D Grouped Convolution     1024 groups of 1 3×3×1 convolu
```

```
    53   'activation_25'              ReLU                         ReLU
    54   'conv2d_13'                  2-D Convolution              1024 1×1×1024 convolutions wi
    55   'activation_26'              ReLU                         ReLU
    56   'global_average_pooling2d'   2-D Global Average Pooling   2-D global average pooling
    57   'dense'                      Fully Connected              8 fully connected layer
    58   'softmax'                    Softmax                      softmax
    59   'Sounds'                     Classification Output        crossentropyex with 'Bearing'

### Notice: The layer 'input_1' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in sof
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwa
### Notice: The layer 'Sounds' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemente
### Compiling layer group: conv2d>>activation_26 ...
### Compiling layer group: conv2d>>activation_26 ... complete.
### Compiling layer group: global_average_pooling2d ...
### Compiling layer group: global_average_pooling2d ... complete.
### Compiling layer group: dense ...
### Compiling layer group: dense ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "8.0 MB"
    "OutputResultOffset"        "0x00800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x01000000"     "28.0 MB"
    "InstructionDataOffset"     "0x02c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03000000"     "32.0 MB"
    "FCWeightDataOffset"        "0x05000000"     "4.0 MB"
    "EndOffset"                 "0x05400000"     "Total: 84.0 MB"

### Network compilation complete.

dn = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
             ddrInfo: [1×1 struct]
```

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` function of the `dlhdl.Workflow` object. This function uses the programming file to program the FPGA board and downloads the network weights and biases. The `deploy` function programs the FPGA device and displays progress messages, and the required time to deploy the network.

```
deploy(hW)
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_int8.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_int8.bit
```

```
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 13-Dec-2022 12:15:20
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 13-Dec-2022 12:15:20
```

Test Network

Prepare the test data for prediction. Use the entire data set consisting of preprocessed mel spectrograms. Compare the predictions of the quantized network to the predictions of Deep Learning Toolbox.

```
[imOutPred, imdPred] = getyamnet_CLEInput(88);
```

Calculate the accuracy of the predictions of the quantized network with respect to the predictions from Deep Learning Toolbox by using the `getNetworkAccuracy` helper function. See Helper Functions on page 10-361.

```
quantizedAccuracy = getNetworkAccuracy(hW,imOutPred,net)
```

```
### Finished writing input activations.
### Running in multi-frame mode with 88 inputs.
```

```
quantizedAccuracy = 0.6477
```

Calculate Accuracy of Quantized YAMNet With Cross Layer Equalization

Create a YAMNet network with cross-layer equalization, quantize the network, and calculate its accuracy.

Create, Calibrate and Quantize a Cross-Layer Equalized Network

Create a cross layer equalized network by using the `equalizeLayers` function.

```
netCLE = equalizeLayers(net);
```

Create a quantized YAMNet with cross layer equalization by using the `dlquantizer` object. Set the target execution environment to FPGA.

```
dlQuantObjCLE = dlquantizer(netCLE,ExecutionEnvironment='FPGA');
```

Use the `calibrate` function to exercise the network with sample inputs, collect the dynamic ranges of the weights and biases, and returns a table. Each row of the table contains range information for a learnable parameter of the quantized network.

```
calibrate(dlQuantObjCLE,imdsCal);
```

Deploy Quantized Network

Define the target FPGA board programming interface by using the `dlhdl.Target` object. Specify that the interface is for a Xilinx board with an Ethernet interface.

```
hTargetCLE = dlhdl.Target('Xilinx',Interface='Ethernet');
```

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and the bitstream name. Ensure that the bitstream matches the data type and the FPGA board. In this example, the target FPGA is the Xilinx ZCU102 SOC board. The bitstream uses an `int8` data type.

```
hWCLE = dlhdl.Workflow(Network=dlQuantObjCLE,Bitstream='zcu102_int8',Target=hTargetCLE);
```

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment. Because the total number of frames exceeds the default value of 30, set the `InputFrameNumberLimit` to 100 to run predictions in chunks of 100 frames to prevent timeouts.

```
dnCLE = compile(hWCLE,InputFrameNumberLimit=100)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### The network includes the following layers:
     1   'input_1'                    Image Input                  96×64×1 images
     2   'conv2d'                     2-D Convolution              32 3×3×1 convolutions with str
     3   'activation'                 ReLU                         ReLU
     4   'depthwise_conv2d'           2-D Grouped Convolution      32 groups of 1 3×3×1 convoluti
     5   'activation_1'               ReLU                         ReLU
     6   'conv2d_1'                   2-D Convolution              64 1×1×32 convolutions with st
     7   'activation_2'               ReLU                         ReLU
     8   'depthwise_conv2d_1'         2-D Grouped Convolution      64 groups of 1 3×3×1 convoluti
     9   'activation_3'               ReLU                         ReLU
    10   'conv2d_2'                   2-D Convolution              128 1×1×64 convolutions with s
    11   'activation_4'               ReLU                         ReLU
    12   'depthwise_conv2d_2'         2-D Grouped Convolution      128 groups of 1 3×3×1 convolut
    13   'activation_5'               ReLU                         ReLU
    14   'conv2d_3'                   2-D Convolution              128 1×1×128 convolutions with 
    15   'activation_6'               ReLU                         ReLU
    16   'depthwise_conv2d_3'         2-D Grouped Convolution      128 groups of 1 3×3×1 convolut
    17   'activation_7'               ReLU                         ReLU
    18   'conv2d_4'                   2-D Convolution              256 1×1×128 convolutions with 
    19   'activation_8'               ReLU                         ReLU
    20   'depthwise_conv2d_4'         2-D Grouped Convolution      256 groups of 1 3×3×1 convolut
    21   'activation_9'               ReLU                         ReLU
    22   'conv2d_5'                   2-D Convolution              256 1×1×256 convolutions with 
    23   'activation_10'              ReLU                         ReLU
    24   'depthwise_conv2d_5'         2-D Grouped Convolution      256 groups of 1 3×3×1 convolut
    25   'activation_11'              ReLU                         ReLU
    26   'conv2d_6'                   2-D Convolution              512 1×1×256 convolutions with 
    27   'activation_12'              ReLU                         ReLU
    28   'depthwise_conv2d_6'         2-D Grouped Convolution      512 groups of 1 3×3×1 convolut
    29   'activation_13'              ReLU                         ReLU
    30   'conv2d_7'                   2-D Convolution              512 1×1×512 convolutions with 
    31   'activation_14'              ReLU                         ReLU
    32   'depthwise_conv2d_7'         2-D Grouped Convolution      512 groups of 1 3×3×1 convolut
    33   'activation_15'              ReLU                         ReLU
    34   'conv2d_8'                   2-D Convolution              512 1×1×512 convolutions with 
    35   'activation_16'              ReLU                         ReLU
    36   'depthwise_conv2d_8'         2-D Grouped Convolution      512 groups of 1 3×3×1 convolut
    37   'activation_17'              ReLU                         ReLU
    38   'conv2d_9'                   2-D Convolution              512 1×1×512 convolutions with 
    39   'activation_18'              ReLU                         ReLU
    40   'depthwise_conv2d_9'         2-D Grouped Convolution      512 groups of 1 3×3×1 convolut
```

```
    41   'activation_19'              ReLU                         ReLU
    42   'conv2d_10'                  2-D Convolution              512 1×1×512 convolutions with
    43   'activation_20'              ReLU                         ReLU
    44   'depthwise_conv2d_10'        2-D Grouped Convolution      512 groups of 1 3×3×1 convolu
    45   'activation_21'              ReLU                         ReLU
    46   'conv2d_11'                  2-D Convolution              512 1×1×512 convolutions with
    47   'activation_22'              ReLU                         ReLU
    48   'depthwise_conv2d_11'        2-D Grouped Convolution      512 groups of 1 3×3×1 convolu
    49   'activation_23'              ReLU                         ReLU
    50   'conv2d_12'                  2-D Convolution              1024 1×1×512 convolutions with
    51   'activation_24'              ReLU                         ReLU
    52   'depthwise_conv2d_12'        2-D Grouped Convolution      1024 groups of 1 3×3×1 convolu
    53   'activation_25'              ReLU                         ReLU
    54   'conv2d_13'                  2-D Convolution              1024 1×1×1024 convolutions wi
    55   'activation_26'              ReLU                         ReLU
    56   'global_average_pooling2d'   2-D Global Average Pooling   2-D global average pooling
    57   'dense'                      Fully Connected              8 fully connected layer
    58   'softmax'                    Softmax                      softmax
    59   'Sounds'                     Classification Output        crossentropyex with 'Bearing'

### Notice: The layer 'input_1' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in sof
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwa
### Notice: The layer 'Sounds' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implement
### Compiling layer group: conv2d>>activation_26 ...
### Compiling layer group: conv2d>>activation_26 ... complete.
### Compiling layer group: global_average_pooling2d ...
### Compiling layer group: global_average_pooling2d ... complete.
### Compiling layer group: dense ...
### Compiling layer group: dense ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    ________________

    "InputDataOffset"           "0x00000000"     "8.0 MB"
    "OutputResultOffset"        "0x00800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x01000000"     "28.0 MB"
    "InstructionDataOffset"     "0x02c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03000000"     "32.0 MB"
    "FCWeightDataOffset"        "0x05000000"     "4.0 MB"
    "EndOffset"                 "0x05400000"     "Total: 84.0 MB"

### Network compilation complete.

dnCLE = struct with fields:
             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {}
             ddrInfo: [1×1 struct]
```

To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` function of the `dlhdl.Workflow` object. This function uses the programming file to program the FPGA board and

downloads the network weights and biases. The `deploy` function programs the FPGA device and displays progress messages, and the required time to deploy the network.

```
deploy(hWCLE)
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_int8.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_int8.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 13-Dec-2022 12:18:12
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 13-Dec-2022 12:18:12
```

Test Network

Prepare the test data for prediction. Use the entire data set consisting of preprocessed mel spectrograms. Compare the predictions of the quantized network to the predictions of Deep Learning Toolbox.

```
[imOutPredCLE, imdPredCLE] = getyamnet_CLEInput(88);
```

Compare the accuracy of the quantized network predictions against the accuracy of the predictions from Deep Learning Toolbox, by using the `getNetworkAccuracy` helper function. See Helper Functions on page 10-361.

```
quantizedAccuracyCLE = getNetworkAccuracy(hWCLE, imOutPredCLE, netCLE)
```

```
### Finished writing input activations.
### Running in multi-frame mode with 88 inputs.

quantizedAccuracyCLE = 0.8636
```

Using the cross-layer equalization improves the prediction accuracy of the network. The accuracy of the network with cross-layer equalization (`netCLE`) is 86.36% and the accuracy of the network without cross-layer equalization (`net`) is 64.77%.

Helper Functions

The `getyamnet_CLEInput` helper function obtains preprocessed mel spectograms of size 96-by-64-by-1.

```
function [imOut,imds] = getyamnet_CLEInput(NumOfImg)
    if NumOfImg > 88
        error('Provide an input lesser than or equal to the size of this dataset of 88 images.')
    end
```

```
    spectData = load('yamnetInput.mat');
    spectData = spectData.melSpectYam;
    imOut = spectData(:,:,:,1:NumOfImg);
    imds = augmentedImageDatastore([96 64],imOut);
end
```

The `getNetworkAccuracy` helper function retrieves predictions from the FPGA and compares them to the predictions from Deep Learning Toolbox™.

```
function accuracy = getNetworkAccuracy(workFlow, data, network) % Function gets accuracy of pred
% Predictions by workflow object
hwPred = workFlow.predict(data); % Matrix of probability for each class
[hwValue, hwIdx] = max(hwPred,[],2); % Get value and index of class with max probability
hwClasses = network.Layers(end).Classes(hwIdx); % Get class names from index

% predictions by DL toolbox object
dlPred = predict(network, data); % Matrix of probability for each class
[dlValue, dlIdx] = max(dlPred,[],2); % Get value and index of class with max probability
dlClasses = network.Layers(end).Classes(dlIdx); % Get class names from index

accuracy = nnz(dlClasses == hwClasses)/size(data,4); % Number of same predictions/total predicti

end
```

References

[1] Verma, Nishchal K., et al. “Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors.” *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291–309. *DOI.org (Crossref)*, doi:10.1109/TR.2015.2459684.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `dlquantizer` | `calibrate` | `equalizeLayers`

Increase Image Resolution Using VDSR Network Running on FPGA

This example shows how to create a high-resolution image from a low-resolution image using a very-deep super-resolution (VDSR) neural network running on an FPGA. Use Deep Learning HDL Toolbox™ to deploy the VDSR network and MATLAB® to retrieve the high-resolution image. To create the trained VDSR network used in this example, see “Increase Image Resolution Using Deep Learning”.

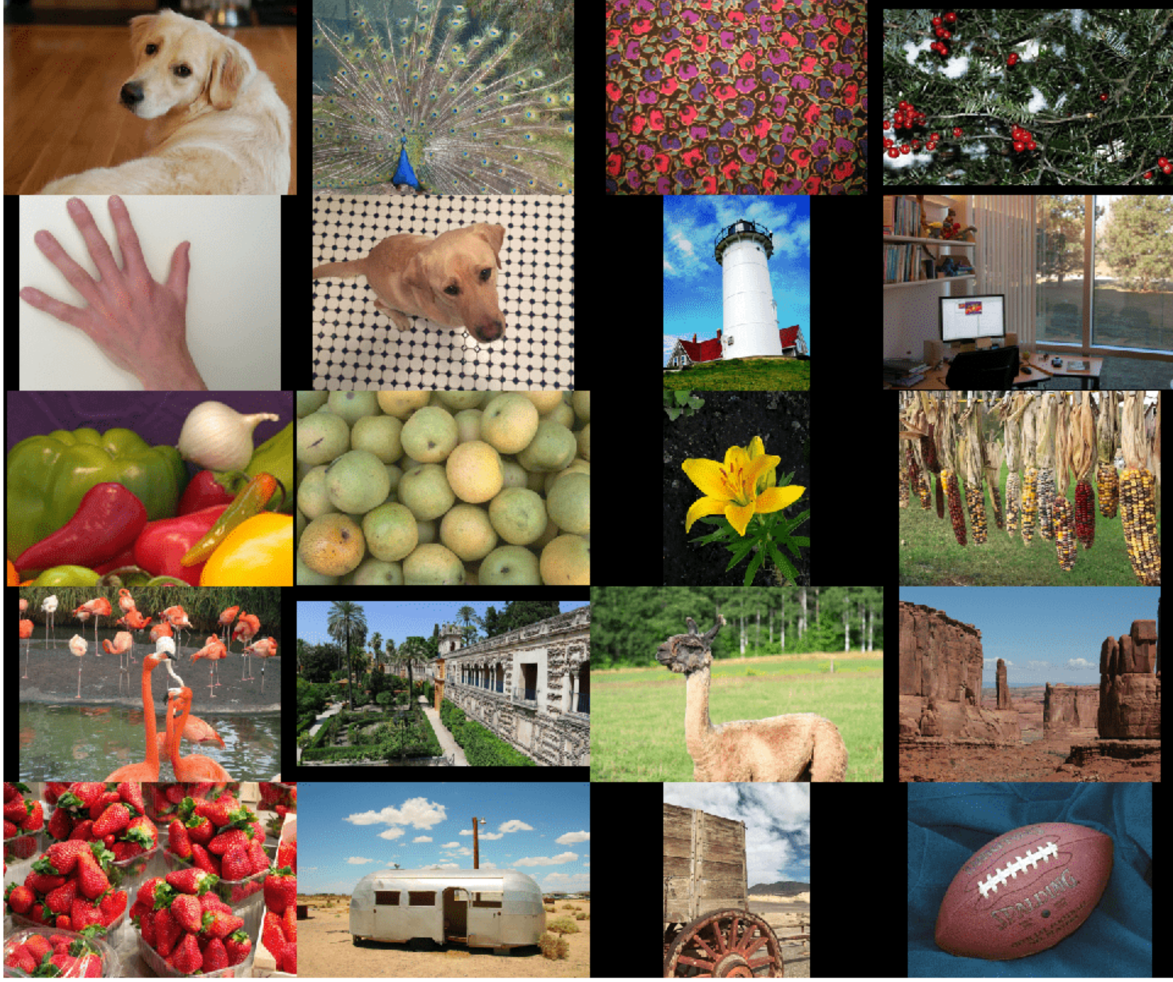
Create Sample Low-Resolution Image

The test data set, `testImages`, contains 20 undistorted images. Load the images into an `imageDatastore` object.

```
fileNames = ["sherlock.jpg","peacock.jpg","fabric.png","greens.jpg", ...
    "hands1.jpg","kobi.png","lighthouse.png","office_4.jpg", ...
    "onion.png","pears.png","yellowlily.jpg","indiancorn.jpg", ...
    "flamingos.jpg","sevilla.jpg","llama.jpg","parkavenue.jpg", ...
    "strawberries.jpg","trailer.jpg","wagon.jpg","football.jpg"];
filePath = fullfile(matlabroot,"toolbox","images","imdata")+filesep;
filePathNames = strcat(filePath,fileNames);
testImages = imageDatastore(filePathNames);
```

Display the test images as a montage.

```
montage(testImages)
```



Select one of the test images to use to test the super-resolution network.

```
testImage = sherlock.jpg ;
Ireference = imread(testImage);
Ireference = im2double(Ireference);
imshow(Ireference)
title("High-Resolution Reference Image")
```

Create a low-resolution version of the high-resolution reference image by using `imresize` with a scaling factor of 0.25. The high-frequency components of the image are lost during the down scaling.

```
scaleFactor = 0.25;
Ilowres = imresize(Ireference,scaleFactor,"bicubic");
imshow(Ilowres)
title("Low-Resolution Image")
```

Improve Image Resolution Using Bicubic Interpolation

A standard way to increase image resolution without deep learning is to use bicubic interpolation. Upscale the low-resolution image using bicubic interpolation so that the resulting high-resolution image is the same size as the reference image.

```
[nrows,ncols,np] = size(Ireference);
Ibicubic = imresize(Ilowres,[nrows ncols],"bicubic");
imshow(Ibicubic)
title("High-Resolution Image Obtained Using Bicubic Interpolation")
```

High-Resolution Image Obtained Using Bicubic Interpolation

Improve Image Resolution Using a Pretrained VDSR Network in Single Precision

VDSR is trained using only the luminance channel of an image because human perception is more sensitive to changes in brightness than to changes in color.

Convert the low-resolution image from the RGB color space to the luminance (`Iy`) and chrominance (`Icb` and `Icr`) channels by using the `rgb2ycbcr` (Image Processing Toolbox) function.

```
Iycbcr = rgb2ycbcr(Ilowres);
Iy = Iycbcr(:,:,1);
Icb = Iycbcr(:,:,2);
Icr = Iycbcr(:,:,3);
```

Upscale the luminance and two chrominance channels using bicubic interpolation. The upsampled chrominance channels, `Icb_bicubic` and `Icr_bicubic`, require no further processing.

```
Iy_bicubic = imresize(Iy,[nrows ncols],"bicubic");
Icb_bicubic = imresize(Icb,[nrows ncols],"bicubic");
Icr_bicubic = imresize(Icr,[nrows ncols],"bicubic");
```

Load Pretrained Network

Load the pretrained VDSR network.

```
load('trainedVDSRNet.mat')
```

View the network layers by using the Deep Network Designer app.

```
deepNetworkDesigner(net)
```

Define FPGA Board Interface

Define the target FPGA board programming interface by using a `dlhdl.Target` object. Create a programming interface with a custom name for your target device and an Ethernet interface to connect the target device to the host computer.

```
hT = dlhdl.Target('Xilinx',Interface="Ethernet");
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx® Zynq® UltraScale+™ MPSoC ZCU102 board and the bitstream uses the single data type.

```
hW = dlhdl.Workflow(Network=net,Bitstream='zcu102_single',Target=hT);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment.

```
dn = compile(hW,'InputFrameNumberLimit',600);
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### The network includes the following layers:
     1   'InputLayer'            Image Input         41×41×1 images
     2   'Conv1'                 2-D Convolution     64 3×3×1 convolutions with stride [1  1] ar
     3   'ReLU1'                 ReLU                ReLU
     4   'Conv2'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
     5   'ReLU2'                 ReLU                ReLU
     6   'Conv3'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
     7   'ReLU3'                 ReLU                ReLU
     8   'Conv4'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
     9   'ReLU4'                 ReLU                ReLU
    10   'Conv5'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    11   'ReLU5'                 ReLU                ReLU
    12   'Conv6'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    13   'ReLU6'                 ReLU                ReLU
    14   'Conv7'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    15   'ReLU7'                 ReLU                ReLU
    16   'Conv8'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    17   'ReLU8'                 ReLU                ReLU
```

```
    18   'Conv9'                   2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    19   'ReLU9'                   ReLU                ReLU
    20   'Conv10'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    21   'ReLU10'                  ReLU                ReLU
    22   'Conv11'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    23   'ReLU11'                  ReLU                ReLU
    24   'Conv12'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    25   'ReLU12'                  ReLU                ReLU
    26   'Conv13'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    27   'ReLU13'                  ReLU                ReLU
    28   'Conv14'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    29   'ReLU14'                  ReLU                ReLU
    30   'Conv15'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    31   'ReLU15'                  ReLU                ReLU
    32   'Conv16'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    33   'ReLU16'                  ReLU                ReLU
    34   'Conv17'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    35   'ReLU17'                  ReLU                ReLU
    36   'Conv18'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    37   'ReLU18'                  ReLU                ReLU
    38   'Conv19'                  2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    39   'ReLU19'                  ReLU                ReLU
    40   'Conv20'                  2-D Convolution     1 3×3×64 convolutions with stride [1  1] an
    41   'FinalRegressionLayer'    Regression Output   mean-squared-error with response 'Responsel

### Notice: The layer 'InputLayer' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'FinalRegressionLayer' with type 'nnet.cnn.layer.RegressionOutputLayer' is
### Compiling layer group: Conv1>>Conv20 ...
### Compiling layer group: Conv1>>Conv20 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "16.0 MB"
    "OutputResultOffset"        "0x01000000"     "16.0 MB"
    "SchedulerDataOffset"       "0x02000000"     "0.0 MB"
    "SystemBufferOffset"        "0x02000000"     "28.0 MB"
    "InstructionDataOffset"     "0x03c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04000000"     "4.0 MB"
    "EndOffset"                 "0x04400000"     "Total: 68.0 MB"

### Network compilation complete.
```

Program the Bitstream onto the FPGA and Download Network Weights

To deploy the network on the Xilinx® Zynq® UltraScale+ MPSoC ZCU102 hardware, run the `deploy` method of the `dlhdl.Workflow` object. This method programs the FPGA board using the output of the `compile` method and the programming file, downloads the network weights and biases, and displays the progress messages and the time it takes to deploy the network.

```
deploy(hW);
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
```

```
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_single.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_single.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 27-Dec-2022 13:38:36
```

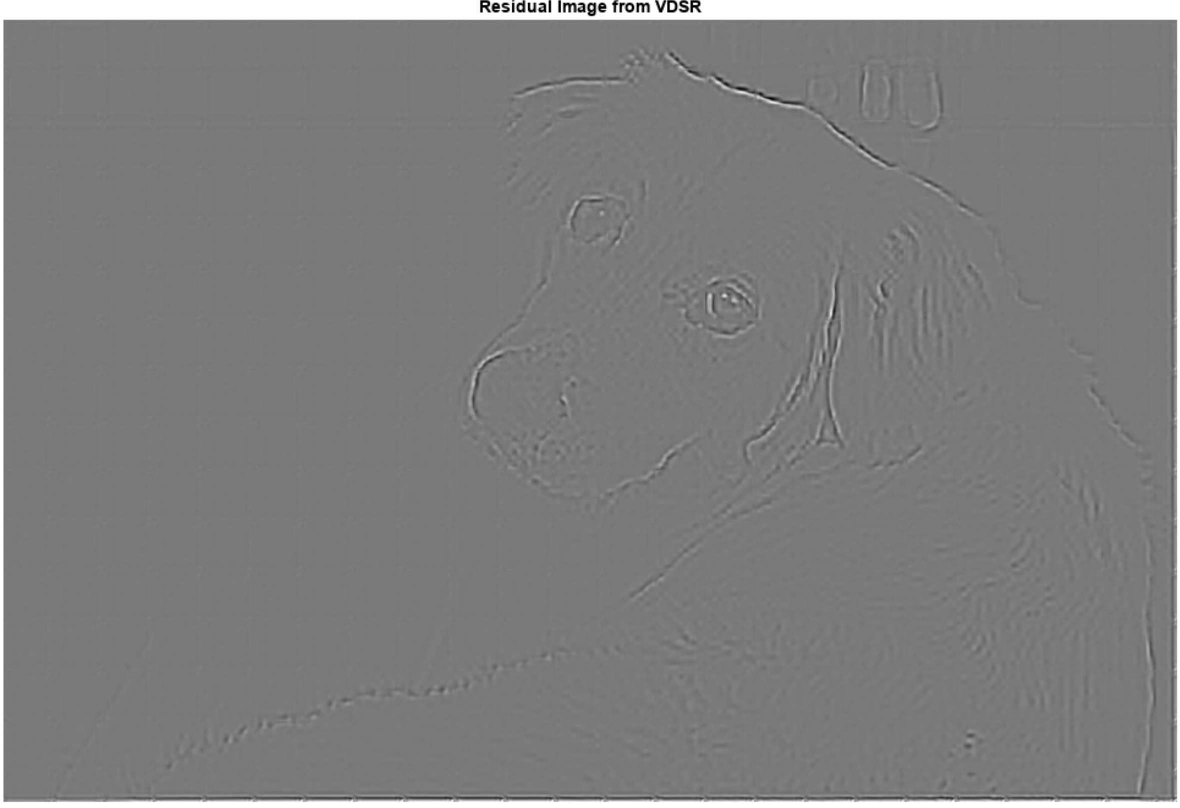
Test Network

Pass the upscaled luminance component, `Iy_bicubic`, through the trained VDSR network. To do this, use the helper function, `runNetworkOnHWVDSR` and observe the residual image obtained. To view the code of this function, see `Helper Functions` on page 10-377.

```
runNetworkOnHWVDSR;
```

```
### Finished writing input activations.
### Running in multi-frame mode with 384 inputs.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    9648715                  0.04386                     384          37048
    Conv1                    76769                  0.00035
    Conv2                   527210                  0.00240
    Conv3                   527660                  0.00240
    Conv4                   527358                  0.00240
    Conv5                   527328                  0.00240
    Conv6                   527678                  0.00240
    Conv7                   527244                  0.00240
    Conv8                   527190                  0.00240
    Conv9                   527233                  0.00240
    Conv10                  527560                  0.00240
    Conv11                  527263                  0.00240
    Conv12                  527219                  0.00240
    Conv13                  527536                  0.00240
    Conv14                  527414                  0.00240
    Conv15                  527228                  0.00240
    Conv16                  527596                  0.00240
    Conv17                  527401                  0.00240
    Conv18                  527211                  0.00240
    Conv19                  527451                  0.00240
    Conv20                   79115                  0.00036
 * The clock frequency of the DL processor is: 220MHz
```

```
I = double(Iresidual);
imshow(Iresidual,[])
title("Residual Image from VDSR")
```



Add the residual image to the upscaled luminance component to generate the high-resolution VDSR luminance component.

```
Isr = Iy_bicubic + Iresidual;
```

Concatenate the high-resolution VDSR luminance component with the upscaled color components. Convert the image to the RGB color space by using the `ycbcr2rgb` (Image Processing Toolbox) function. The result is the final high-resolution color image.

```
Ivdsr = ycbcr2rgb(cat(3,Isr,Icb_bicubic,Icr_bicubic));
imshow(Ivdsr)
title("High-Resolution Image Obtained Using VDSR")
```

Improve Image Resolution By Using a Quantized VDSR Network

The single data type VDSR network performs at 22.8 frames per second. To improve the network performance quantize the network.

Create `dlquantizer` Object

Create a quantized network object by using the `dlquantizer` object. Set the target execution environment to FPGA.

```
dlq = dlquantizer(net,ExecutionEnvironment="FPGA");
```

Calibrate Quantized Network

Use the `calibrate` method to exercise the network by using sample inputs to collect the range information. The `calibrate` method exercises the network and collects the dynamic ranges for the learnable parameters of the layers of the network.

For best quantization results, the calibration data must be a representative of actual inputs that are predicted by the network.

```
imds = augmentedImageDatastore([640,960],Iy_bicubic);
calibrate(dlq,imds);
```

Prepare Network for Deployment

Prepare the network for deployment by creating a `dlhdl.Workflow` object. Specify the network and bitstream name. Ensure that the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Xilinx® Zynq® UltraScale+™ MPSoC ZCU102 board and the bitstream uses the `int8` data type.

```
hW = dlhdl.Workflow(Network=dlq,Bitstream='zcu102_int8',Target=hT);
```

Compile Network

Run the `compile` method of the `dlhdl.Workflow` object to compile the network and generate the instructions, weights, and biases for deployment.

```
dn = compile(hW,'InputFrameNumberLimit', 600);
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### The network includes the following layers:
     1   'InputLayer'            Image Input         41×41×1 images
     2   'Conv1'                 2-D Convolution     64 3×3×1 convolutions with stride [1  1] ar
     3   'ReLU1'                 ReLU                ReLU
     4   'Conv2'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
     5   'ReLU2'                 ReLU                ReLU
     6   'Conv3'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
     7   'ReLU3'                 ReLU                ReLU
     8   'Conv4'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
     9   'ReLU4'                 ReLU                ReLU
    10   'Conv5'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    11   'ReLU5'                 ReLU                ReLU
    12   'Conv6'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    13   'ReLU6'                 ReLU                ReLU
    14   'Conv7'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    15   'ReLU7'                 ReLU                ReLU
    16   'Conv8'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    17   'ReLU8'                 ReLU                ReLU
    18   'Conv9'                 2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    19   'ReLU9'                 ReLU                ReLU
    20   'Conv10'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    21   'ReLU10'                ReLU                ReLU
    22   'Conv11'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    23   'ReLU11'                ReLU                ReLU
    24   'Conv12'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    25   'ReLU12'                ReLU                ReLU
    26   'Conv13'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    27   'ReLU13'                ReLU                ReLU
    28   'Conv14'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    29   'ReLU14'                ReLU                ReLU
    30   'Conv15'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    31   'ReLU15'                ReLU                ReLU
    32   'Conv16'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    33   'ReLU16'                ReLU                ReLU
    34   'Conv17'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    35   'ReLU17'                ReLU                ReLU
    36   'Conv18'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    37   'ReLU18'                ReLU                ReLU
    38   'Conv19'                2-D Convolution     64 3×3×64 convolutions with stride [1  1] a
    39   'ReLU19'                ReLU                ReLU
```

```
    40   'Conv20'                 2-D Convolution     1 3×3×64 convolutions with stride [1  1] ar
    41   'FinalRegressionLayer'   Regression Output   mean-squared-error with response 'Responsel

### Notice: The layer 'InputLayer' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'FinalRegressionLayer' with type 'nnet.cnn.layer.RegressionOutputLayer' is
### Compiling layer group: Conv1>>Conv20 ...
### Compiling layer group: Conv1>>Conv20 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "8.0 MB"
    "OutputResultOffset"        "0x00800000"     "8.0 MB"
    "SchedulerDataOffset"       "0x01000000"     "0.0 MB"
    "SystemBufferOffset"        "0x01000000"     "28.0 MB"
    "InstructionDataOffset"     "0x02c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03000000"     "4.0 MB"
    "EndOffset"                 "0x03400000"     "Total: 52.0 MB"

### Network compilation complete.
```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx® Zynq® UltraScale+ MPSoC ZCU102 hardware, run the `deploy` method of the `dlhdl.Workflow` object. This method programs the FPGA board using the output of the `compile` method and the programming file, downloads the network weights and biases, and displays the progress messages and the time it takes to deploy the network.

```
deploy(hW);
```

```
### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_int8.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_int8.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 27-Dec-2022 13:41:24
```

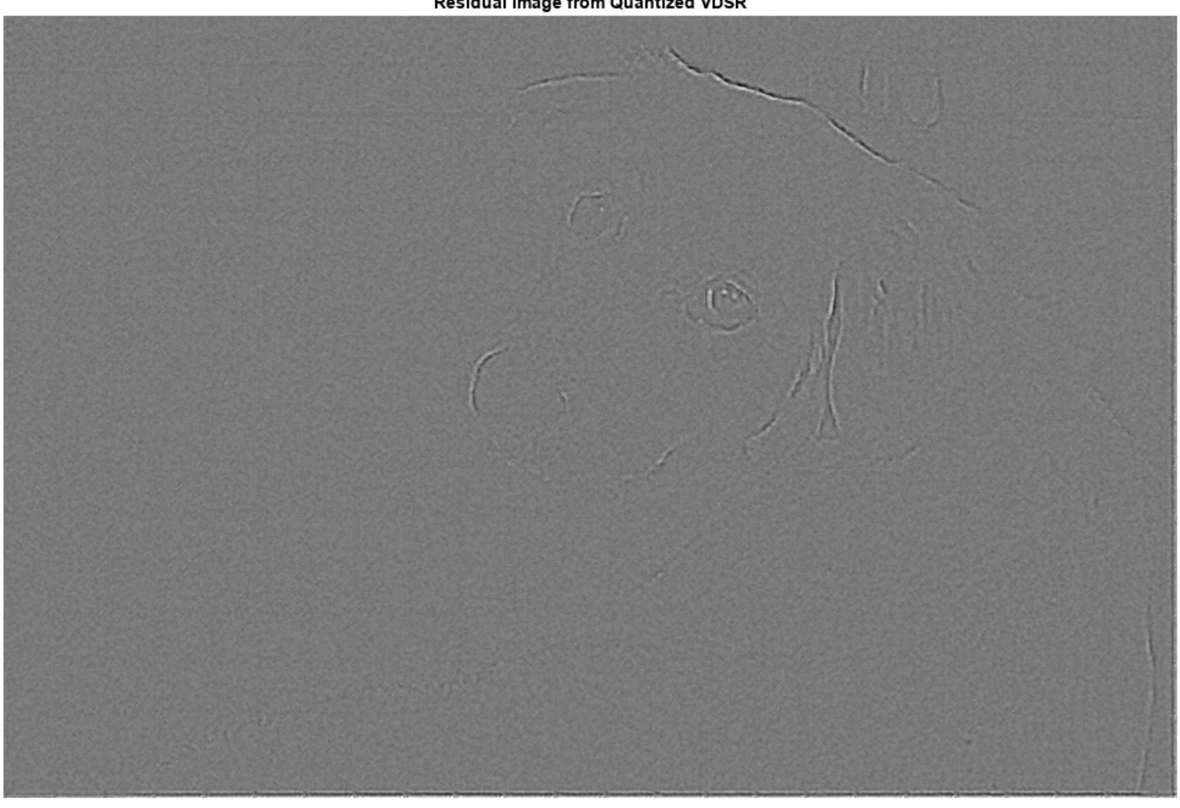
Test Network

Pass the upscaled luminance component, `Iy_bicubic`, through the trained VDSR network. To do this, use the helper function, `runNetworkOnHW.`

```
runNetworkOnHWVDSR;
```

```
### Finished writing input activations.
### Running in multi-frame mode with 384 inputs.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    2740874                  0.01096                     384         10525
    Conv1                    38433                  0.00015
    Conv2                   148211                  0.00059
    Conv3                   148181                  0.00059
    Conv4                   148406                  0.00059
    Conv5                   148126                  0.00059
    Conv6                   148047                  0.00059
    Conv7                   148156                  0.00059
    Conv8                   148122                  0.00059
    Conv9                   148311                  0.00059
    Conv10                  148289                  0.00059
    Conv11                  148534                  0.00059
    Conv12                  148056                  0.00059
    Conv13                  148181                  0.00059
    Conv14                  148113                  0.00059
    Conv15                  148195                  0.00059
    Conv16                  148209                  0.00059
    Conv17                  148183                  0.00059
    Conv18                  148456                  0.00059
    Conv19                  148257                  0.00059
    Conv20                   34357                  0.00014
 * The clock frequency of the DL processor is: 250MHz
```

```
Iresidual = double(Iresidual);
imshow(Iresidual,[])
title("Residual Image from Quantized VDSR")
```



Add the residual image to the upscaled luminance component to generate the high-resolution VDSR luminance component.

```
Isr = Iy_bicubic + Iresidual;
```

Concatenate the high-resolution VDSR luminance component with the upscaled color components. Convert the image to the RGB color space by using the `ycbcr2rgb` (Image Processing Toolbox) function. The result is the final high-resolution color image.

```
Ivdsrquantized = ycbcr2rgb(cat(3,Isr,Icb_bicubic,Icr_bicubic));
imshow(Ivdsrquantized)
title("High-Resolution Image Obtained Using Quantized VDSR")
```

Compare Visual and Performance Results

Examine a small region inside of each image. Specify a region of interest (ROI) using a vector `roi` in the format `[x y width height]`. The elements define the `x`- and `y`- coordinates of the top-left corner, and the width and height of the ROI.

```
roi = [360 50 400 350];
```

Crop the high-resolution images to this ROI, and display the results as a montage. The quantized VDSR image has clearer details and sharper resolutions than the high-resolution image created using single data type VDSR.

```
montage({imcrop(Ivdsr,roi);imcrop(Ivdsrquantized,roi)})
title("High Resolution Results Using VDSR (Left) vs. Quantized VDSR (Right)");
```

The quantized VDSR network has a performance of 91.2 frames per second compared to the 22.8 frames per second for the single-data-type network.

Helper Functions

The `runNetworkOnHWVDSR` function:

- Generates a bicubic interpolation of the input image.
- Splits the bicubic interpolation image into smaller blocks.
- Passes the smaller blocks as multi-frame inputs to the deployed network.
- Uses the `predict` method to retrieve the higher-resolution individual smaller block images.
- Combines the higher-resolution block images into the output image and returns the output image.

```
%   Copyright 2022 The MathWorks, Inc.

networkInputSize = net.Layers(1).InputSize(1:2);
imgTestSize = size(Iy_bicubic);
Iy_bicubic_new = zeros(imgTestSize + ([41 41]-mod(imgTestSize,41)));
Iy_bicubic_new(1:imgTestSize(1), 1:imgTestSize(2)) = Iy_bicubic ;
numberOfBlocks = ceil(imgTestSize./networkInputSize);
totalBlocks = prod(numberOfBlocks);
splitImage = mat2cell(Iy_bicubic_new, networkInputSize(1)*ones(1, numberOfBlocks(1)), networkInpu
multiFrameInput = zeros([networkInputSize 1 totalBlocks]);
for i=1:(totalBlocks)
 multiFrameInput(:,:,:,i) = splitImage{i};
end

residualImage = hW.predict(multiFrameInput, 'Profile','on');

concatenatedResult = [];
for i=1:numberOfBlocks(2)
 subset = residualImage(:,:,numberOfBlocks(1)*(i-1)+1:i*numberOfBlocks(1));
 verticalConcatenation = [];
```

```
 for j=1:numberOfBlocks(1)
 verticalConcatenation = [verticalConcatenation; subset(:,:,j)];
 end
 concatenatedResult = [concatenatedResult verticalConcatenation];
end
Iresidual = concatenatedResult(1:imgTestSize(1), 1:imgTestSize(2));
```

References

[1] Kim, J., J. K. Lee, and K. M. Lee. "Accurate Image Super-Resolution Using Very Deep Convolutional Networks." *Proceedings of the IEEE® Conference on Computer Vision and Pattern Recognition*. 2016, pp. 1646-1654.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `dlquantizer` | `calibrate`

Deploy Image Recognition Network on FPGA With and Without Pruning

This example shows you how to deploy an image recognition network with and without convolutional filter pruning. Filter pruning is a compression technique that uses some criterion to identify and remove the least important filters in a network, which reduces the overall memory footprint of the network without significantly reducing the network accuracy.

Load Unpruned Network

Load the unpruned trained network. For information on network training, see “Train Residual Network for Image Classification”.

```
load("trainedYOLONet.mat");
```

Test Network

Load a test image. The test image is a part of the CIFAR-10 data set[1]. To download the data set, see the Prepare Data section in “Train Residual Network for Image Classification”.

```
load("testImage.mat");
```

Use the `runonHW` function to:

- Prepare the network for deployment.
- Compile the network to generate weights, biases, and instructions.
- Deploy the network to the FPGA board.
- Retrieve the prediction results using MATLAB®.

To view the code for this function, see Helper Functions on page 10-389.

```
[~, speedInitial] = runOnHW(trainedNet,testImage,'zcu102_single');
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'input' of type 'ImageInputLayer' is split into an image input layer 'inpu
### The network includes the following layers:
     1   'input'         Image Input             32×32×3 images with 'zerocenter' normalization
     2   'convInp'       2-D Convolution         16 3×3×3 convolutions with stride [1  1] and pad
     3   'reluInp'       ReLU                    ReLU
     4   'S1U1_conv1'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
     5   'S1U1_relu1'    ReLU                    ReLU
     6   'S1U1_conv2'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
     7   'add11'         Addition                Element-wise addition of 2 inputs
     8   'relu11'        ReLU                    ReLU
     9   'S1U2_conv1'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
    10   'S1U2_relu1'    ReLU                    ReLU
    11   'S1U2_conv2'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
    12   'add12'         Addition                Element-wise addition of 2 inputs
    13   'relu12'        ReLU                    ReLU
    14   'S1U3_conv1'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
    15   'S1U3_relu1'    ReLU                    ReLU
    16   'S1U3_conv2'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
```

```
    17   'add13'         Addition                Element-wise addition of 2 inputs
    18   'relu13'        ReLU                    ReLU
    19   'S2U1_conv1'    2-D Convolution         32 3×3×16 convolutions with stride [2  2] and pa
    20   'S2U1_relu1'    ReLU                    ReLU
    21   'S2U1_conv2'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    22   'skipConv1'     2-D Convolution         32 1×1×16 convolutions with stride [2  2] and pa
    23   'add21'         Addition                Element-wise addition of 2 inputs
    24   'relu21'        ReLU                    ReLU
    25   'S2U2_conv1'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    26   'S2U2_relu1'    ReLU                    ReLU
    27   'S2U2_conv2'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    28   'add22'         Addition                Element-wise addition of 2 inputs
    29   'relu22'        ReLU                    ReLU
    30   'S2U3_conv1'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    31   'S2U3_relu1'    ReLU                    ReLU
    32   'S2U3_conv2'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    33   'add23'         Addition                Element-wise addition of 2 inputs
    34   'relu23'        ReLU                    ReLU
    35   'S3U1_conv1'    2-D Convolution         64 3×3×32 convolutions with stride [2  2] and pa
    36   'S3U1_relu1'    ReLU                    ReLU
    37   'S3U1_conv2'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    38   'skipConv2'     2-D Convolution         64 1×1×32 convolutions with stride [2  2] and pa
    39   'add31'         Addition                Element-wise addition of 2 inputs
    40   'relu31'        ReLU                    ReLU
    41   'S3U2_conv1'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    42   'S3U2_relu1'    ReLU                    ReLU
    43   'S3U2_conv2'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    44   'add32'         Addition                Element-wise addition of 2 inputs
    45   'relu32'        ReLU                    ReLU
    46   'S3U3_conv1'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    47   'S3U3_relu1'    ReLU                    ReLU
    48   'S3U3_conv2'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    49   'add33'         Addition                Element-wise addition of 2 inputs
    50   'relu33'        ReLU                    ReLU
    51   'globalPool'    2-D Average Pooling     8×8 average pooling with stride [1  1] and paddi
    52   'fcFinal'       Fully Connected         10 fully connected layer
    53   'softmax'       Softmax                 softmax
    54   'classoutput'   Classification Output   crossentropyex with 'airplane' and 9 other class

### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple
### Compiling layer group: convInp>>reluInp ...
### Compiling layer group: convInp>>reluInp ... complete.
### Compiling layer group: S1U1_conv1>>S1U1_conv2 ...
### Compiling layer group: S1U1_conv1>>S1U1_conv2 ... complete.
### Compiling layer group: S1U2_conv1>>S1U2_conv2 ...
### Compiling layer group: S1U2_conv1>>S1U2_conv2 ... complete.
### Compiling layer group: S1U3_conv1>>S1U3_conv2 ...
### Compiling layer group: S1U3_conv1>>S1U3_conv2 ... complete.
### Compiling layer group: skipConv1 ...
### Compiling layer group: skipConv1 ... complete.
### Compiling layer group: S2U1_conv1>>S2U1_conv2 ...
### Compiling layer group: S2U1_conv1>>S2U1_conv2 ... complete.
### Compiling layer group: S2U2_conv1>>S2U2_conv2 ...
### Compiling layer group: S2U2_conv1>>S2U2_conv2 ... complete.
### Compiling layer group: S2U3_conv1>>S2U3_conv2 ...
### Compiling layer group: S2U3_conv1>>S2U3_conv2 ... complete.
### Compiling layer group: skipConv2 ...
```

```
### Compiling layer group: skipConv2 ... complete.
### Compiling layer group: S3U1_conv1>>S3U1_conv2 ...
### Compiling layer group: S3U1_conv1>>S3U1_conv2 ... complete.
### Compiling layer group: S3U2_conv1>>S3U2_conv2 ...
### Compiling layer group: S3U2_conv1>>S3U2_conv2 ... complete.
### Compiling layer group: S3U3_conv1>>S3U3_conv2 ...
### Compiling layer group: S3U3_conv1>>S3U3_conv2 ... complete.
### Compiling layer group: globalPool ...
### Compiling layer group: globalPool ... complete.
### Compiling layer group: fcFinal ...
### Compiling layer group: fcFinal ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "28.0 MB"
    "InstructionDataOffset"     "0x02800000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02c00000"     "4.0 MB"
    "FCWeightDataOffset"        "0x03000000"     "4.0 MB"
    "EndOffset"                 "0x03400000"     "Total: 52.0 MB"

### Network compilation complete.

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 21-Dec-2022 20:01:08
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 21-Dec-2022 20:01:08
### Finished writing input activations.
### Running single input activation.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------             ---------        --
Network                     823750                  0.00374                       1           8
    input_norm                7211                  0.00003
    convInp                  14087                  0.00006
    S1U1_conv1               32161                  0.00015
    S1U1_conv2               32341                  0.00015
    add11                    30573                  0.00014
    S1U2_conv1               32548                  0.00015
    S1U2_conv2               32211                  0.00015
    add12                    30583                  0.00014
    S1U3_conv1               32098                  0.00015
    S1U3_conv2               32315                  0.00015
    add13                    30703                  0.00014
    skipConv1                20596                  0.00009
    S2U1_conv1               21157                  0.00010
    S2U1_conv2               26337                  0.00012
    add21                    15333                  0.00007
    S2U2_conv1               26827                  0.00012
```

```
    S2U2_conv2               26586                   0.00012
    add22                    15383                   0.00007
    S2U3_conv1               26530                   0.00012
    S2U3_conv2               26513                   0.00012
    add23                    15373                   0.00007
    skipConv2                25497                   0.00012
    S3U1_conv1               25094                   0.00011
    S3U1_conv2               41593                   0.00019
    add31                     7694                   0.00003
    S3U2_conv1               41819                   0.00019
    S3U2_conv2               41656                   0.00019
    add32                     7884                   0.00004
    S3U3_conv1               41414                   0.00019
    S3U3_conv2               41882                   0.00019
    add33                     7684                   0.00003
    globalPool               10203                   0.00005
    fcFinal                   3677                   0.00002
 * The clock frequency of the DL processor is: 220MHz
```

Load Pruned Network

Load the trained, pruned network. For more information on network training, see “Prune Image Classification Network Using Taylor Scores”.

```
load("prunedDAGNet.mat");
```

Test Network

Load a test image. The test image is a part of the CIFAR-10 data set[1]. To download the data set, see the Prepare Data section in “Train Residual Network for Image Classification”.

```
load("testImage.mat");
```

Use the `runonHW` function to:

- Prepare the network for deployment.
- Compile the network to generate weights, biases, and instructions.
- Deploy the network to the FPGA board.
- Retrieve the prediction results using MATLAB®.

To view the code for this function, see Helper Functions on page 10-389.

```
[~, speedPruned] = runOnHW(trainedNet,testImage,'zcu102_single');
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### Notice: The layer 'input' of type 'ImageInputLayer' is split into an image input layer 'input
### The network includes the following layers:
     1   'input'         Image Input           32×32×3 images with 'zerocenter' normalization
     2   'convInp'       2-D Convolution       16 3×3×3 convolutions with stride [1  1] and pad
     3   'reluInp'       ReLU                  ReLU
     4   'S1U1_conv1'    2-D Convolution       16 3×3×16 convolutions with stride [1  1] and pa
     5   'S1U1_relu1'    ReLU                  ReLU
     6   'S1U1_conv2'    2-D Convolution       16 3×3×16 convolutions with stride [1  1] and pa
     7   'add11'         Addition              Element-wise addition of 2 inputs
     8   'relu11'        ReLU                  ReLU
```

```
     9   'S1U2_conv1'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
    10   'S1U2_relu1'    ReLU                    ReLU
    11   'S1U2_conv2'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
    12   'add12'         Addition                Element-wise addition of 2 inputs
    13   'relu12'        ReLU                    ReLU
    14   'S1U3_conv1'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
    15   'S1U3_relu1'    ReLU                    ReLU
    16   'S1U3_conv2'    2-D Convolution         16 3×3×16 convolutions with stride [1  1] and pa
    17   'add13'         Addition                Element-wise addition of 2 inputs
    18   'relu13'        ReLU                    ReLU
    19   'S2U1_conv1'    2-D Convolution         32 3×3×16 convolutions with stride [2  2] and pa
    20   'S2U1_relu1'    ReLU                    ReLU
    21   'S2U1_conv2'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    22   'skipConv1'     2-D Convolution         32 1×1×16 convolutions with stride [2  2] and pa
    23   'add21'         Addition                Element-wise addition of 2 inputs
    24   'relu21'        ReLU                    ReLU
    25   'S2U2_conv1'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    26   'S2U2_relu1'    ReLU                    ReLU
    27   'S2U2_conv2'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    28   'add22'         Addition                Element-wise addition of 2 inputs
    29   'relu22'        ReLU                    ReLU
    30   'S2U3_conv1'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    31   'S2U3_relu1'    ReLU                    ReLU
    32   'S2U3_conv2'    2-D Convolution         32 3×3×32 convolutions with stride [1  1] and pa
    33   'add23'         Addition                Element-wise addition of 2 inputs
    34   'relu23'        ReLU                    ReLU
    35   'S3U1_conv1'    2-D Convolution         64 3×3×32 convolutions with stride [2  2] and pa
    36   'S3U1_relu1'    ReLU                    ReLU
    37   'S3U1_conv2'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    38   'skipConv2'     2-D Convolution         64 1×1×32 convolutions with stride [2  2] and pa
    39   'add31'         Addition                Element-wise addition of 2 inputs
    40   'relu31'        ReLU                    ReLU
    41   'S3U2_conv1'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    42   'S3U2_relu1'    ReLU                    ReLU
    43   'S3U2_conv2'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    44   'add32'         Addition                Element-wise addition of 2 inputs
    45   'relu32'        ReLU                    ReLU
    46   'S3U3_conv1'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    47   'S3U3_relu1'    ReLU                    ReLU
    48   'S3U3_conv2'    2-D Convolution         64 3×3×64 convolutions with stride [1  1] and pa
    49   'add33'         Addition                Element-wise addition of 2 inputs
    50   'relu33'        ReLU                    ReLU
    51   'globalPool'    2-D Average Pooling     8×8 average pooling with stride [1  1] and paddi
    52   'fcFinal'       Fully Connected         10 fully connected layer
    53   'softmax'       Softmax                 softmax
    54   'classoutput'   Classification Output   crossentropyex with 'airplane' and 9 other class

### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple
### Compiling layer group: convInp>>reluInp ...
### Compiling layer group: convInp>>reluInp ... complete.
### Compiling layer group: S1U1_conv1>>S1U1_conv2 ...
### Compiling layer group: S1U1_conv1>>S1U1_conv2 ... complete.
### Compiling layer group: S1U2_conv1>>S1U2_conv2 ...
### Compiling layer group: S1U2_conv1>>S1U2_conv2 ... complete.
### Compiling layer group: S1U3_conv1>>S1U3_conv2 ...
### Compiling layer group: S1U3_conv1>>S1U3_conv2 ... complete.
### Compiling layer group: skipConv1 ...
```

```
### Compiling layer group: skipConv1 ... complete.
### Compiling layer group: S2U1_conv1>>S2U1_conv2 ...
### Compiling layer group: S2U1_conv1>>S2U1_conv2 ... complete.
### Compiling layer group: S2U2_conv1>>S2U2_conv2 ...
### Compiling layer group: S2U2_conv1>>S2U2_conv2 ... complete.
### Compiling layer group: S2U3_conv1>>S2U3_conv2 ...
### Compiling layer group: S2U3_conv1>>S2U3_conv2 ... complete.
### Compiling layer group: skipConv2 ...
### Compiling layer group: skipConv2 ... complete.
### Compiling layer group: S3U1_conv1>>S3U1_conv2 ...
### Compiling layer group: S3U1_conv1>>S3U1_conv2 ... complete.
### Compiling layer group: S3U2_conv1>>S3U2_conv2 ...
### Compiling layer group: S3U2_conv1>>S3U2_conv2 ... complete.
### Compiling layer group: S3U3_conv1>>S3U3_conv2 ...
### Compiling layer group: S3U3_conv1>>S3U3_conv2 ... complete.
### Compiling layer group: globalPool ...
### Compiling layer group: globalPool ... complete.
### Compiling layer group: fcFinal ...
### Compiling layer group: fcFinal ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "28.0 MB"
    "InstructionDataOffset"     "0x02800000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02c00000"     "4.0 MB"
    "FCWeightDataOffset"        "0x03000000"     "4.0 MB"
    "EndOffset"                 "0x03400000"     "Total: 52.0 MB"

### Network compilation complete.

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Deep learning network programming has been skipped as the same network is already loaded on t
### Finished writing input activations.
### Running single input activation.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                     824245                  0.00375                       1            8
    input_norm                7276                  0.00003
    convInp                  14176                  0.00006
    S1U1_conv1               32251                  0.00015
    S1U1_conv2               32406                  0.00015
    add11                    30623                  0.00014
    S1U2_conv1               32502                  0.00015
    S1U2_conv2               32280                  0.00015
    add12                    30633                  0.00014
    S1U3_conv1               32099                  0.00015
    S1U3_conv2               32325                  0.00015
    add13                    30603                  0.00014
```

```
    skipConv1                20652                  0.00009
    S2U1_conv1               21183                  0.00010
    S2U1_conv2               26313                  0.00012
    add21                    15363                  0.00007
    S2U2_conv1               26797                  0.00012
    S2U2_conv2               26626                  0.00012
    add22                    15363                  0.00007
    S2U3_conv1               26537                  0.00012
    S2U3_conv2               26496                  0.00012
    add23                    15393                  0.00007
    skipConv2                25491                  0.00012
    S3U1_conv1               25104                  0.00011
    S3U1_conv2               41594                  0.00019
    add31                     7794                  0.00004
    S3U2_conv1               41720                  0.00019
    S3U2_conv2               41642                  0.00019
    add32                     7864                  0.00004
    S3U3_conv1               41455                  0.00019
    S3U3_conv2               42002                  0.00019
    add33                     7784                  0.00004
    globalPool               10010                  0.00005
    fcFinal                   3701                  0.00002
 * The clock frequency of the DL processor is: 220MHz
```

Quantize Pruned Network

You can quantize the pruned network to obtain an improved performance.

Create an `augmentedImageDataStore` object to store the training images.

```
imds = augmentedImageDatastore([32,32],testImage);
```

Create a `dlquantizer` object.

```
dlqObj = dlquantizer(prunedDAGNet, ExecutionEnvironment="FPGA");
```

Calibrate the `dlquantizer` object using the training images.

```
calibrate(dlqObj,imds)
```

```
ans=100×5 table
     Optimized Layer Name      Network Layer Name    Learnables / Activations     MinValue     Ma
    ______________________    __________________    ________________________    __________    __

    {'convInp_Weights'   }      {'convInp'   }             "Weights"           -0.0060522    0.
    {'convInp_Bias'      }      {'convInp'   }             "Bias"                -0.23065     
    {'S1U1_conv1_Weights'}      {'S1U1_conv1'}             "Weights"             -0.36637     
    {'S1U1_conv1_Bias'   }      {'S1U1_conv1'}             "Bias"                0.076761     
    {'S1U1_conv2_Weights'}      {'S1U1_conv2'}             "Weights"              -0.8197     
    {'S1U1_conv2_Bias'   }      {'S1U1_conv2'}             "Bias"                -0.27783     
    {'S1U2_conv1_Weights'}      {'S1U2_conv1'}             "Weights"             -0.29579     
    {'S1U2_conv1_Bias'   }      {'S1U2_conv1'}             "Bias"                -0.55448     
    {'S1U2_conv2_Weights'}      {'S1U2_conv2'}             "Weights"             -0.78735     
    {'S1U2_conv2_Bias'   }      {'S1U2_conv2'}             "Bias"                -0.50762     
    {'S1U3_conv1_Weights'}      {'S1U3_conv1'}             "Weights"             -0.18651     
    {'S1U3_conv1_Bias'   }      {'S1U3_conv1'}             "Bias"                -0.33809     
    {'S1U3_conv2_Weights'}      {'S1U3_conv2'}             "Weights"             -0.49925     
    {'S1U3_conv2_Bias'   }      {'S1U3_conv2'}             "Bias"                -0.42145     
    {'S2U1_conv1_Weights'}      {'S2U1_conv1'}             "Weights"              -0.1328
```

```
    {'S2U1_conv1_Bias'   }      {'S2U1_conv1'}             "Bias"               -0.097249
      ⋮
```

Use the `runonHW` function to:

- Prepare the network for deployment.
- Compile the network to generate weights, biases, and instructions.
- Deploy the network to the FPGA board.
- Retrieve the prediction results using MATLAB®.

To view the code for this function, see Helper Functions on page 10-389.

```
[~, speedQuantized] = runOnHW(dlqObj,testImage,'zcu102_int8');
```

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv
### The network includes the following layers:
     1   'input'          Image Input            32×32×3 images with 'zerocenter' normalization
     2   'convInp'        2-D Convolution        16 3×3×3 convolutions with stride [1  1] and pad
     3   'reluInp'        ReLU                   ReLU
     4   'S1U1_conv1'     2-D Convolution        5 3×3×16 convolutions with stride [1  1] and pad
     5   'S1U1_relu1'     ReLU                   ReLU
     6   'S1U1_conv2'     2-D Convolution        16 3×3×5 convolutions with stride [1  1] and pad
     7   'add11'          Addition               Element-wise addition of 2 inputs
     8   'relu11'         ReLU                   ReLU
     9   'S1U2_conv1'     2-D Convolution        8 3×3×16 convolutions with stride [1  1] and pad
    10   'S1U2_relu1'     ReLU                   ReLU
    11   'S1U2_conv2'     2-D Convolution        16 3×3×8 convolutions with stride [1  1] and pad
    12   'add12'          Addition               Element-wise addition of 2 inputs
    13   'relu12'         ReLU                   ReLU
    14   'S1U3_conv1'     2-D Convolution        14 3×3×16 convolutions with stride [1  1] and pa
    15   'S1U3_relu1'     ReLU                   ReLU
    16   'S1U3_conv2'     2-D Convolution        16 3×3×14 convolutions with stride [1  1] and pa
    17   'add13'          Addition               Element-wise addition of 2 inputs
    18   'relu13'         ReLU                   ReLU
    19   'S2U1_conv1'     2-D Convolution        22 3×3×16 convolutions with stride [2  2] and pa
    20   'S2U1_relu1'     ReLU                   ReLU
    21   'S2U1_conv2'     2-D Convolution        27 3×3×22 convolutions with stride [1  1] and pa
    22   'skipConv1'      2-D Convolution        27 1×1×16 convolutions with stride [2  2] and pa
    23   'add21'          Addition               Element-wise addition of 2 inputs
    24   'relu21'         ReLU                   ReLU
    25   'S2U2_conv1'     2-D Convolution        30 3×3×27 convolutions with stride [1  1] and pa
    26   'S2U2_relu1'     ReLU                   ReLU
    27   'S2U2_conv2'     2-D Convolution        27 3×3×30 convolutions with stride [1  1] and pa
    28   'add22'          Addition               Element-wise addition of 2 inputs
    29   'relu22'         ReLU                   ReLU
    30   'S2U3_conv1'     2-D Convolution        26 3×3×27 convolutions with stride [1  1] and pa
    31   'S2U3_relu1'     ReLU                   ReLU
    32   'S2U3_conv2'     2-D Convolution        27 3×3×26 convolutions with stride [1  1] and pa
    33   'add23'          Addition               Element-wise addition of 2 inputs
    34   'relu23'         ReLU                   ReLU
    35   'S3U1_conv1'     2-D Convolution        37 3×3×27 convolutions with stride [2  2] and pa
    36   'S3U1_relu1'     ReLU                   ReLU
    37   'S3U1_conv2'     2-D Convolution        39 3×3×37 convolutions with stride [1  1] and pa
    38   'skipConv2'      2-D Convolution        39 1×1×27 convolutions with stride [2  2] and pa
```

```
    39   'add31'         Addition                 Element-wise addition of 2 inputs
    40   'relu31'        ReLU                     ReLU
    41   'S3U2_conv1'    2-D Convolution          38 3×3×39 convolutions with stride [1  1] and pa
    42   'S3U2_relu1'    ReLU                     ReLU
    43   'S3U2_conv2'    2-D Convolution          39 3×3×38 convolutions with stride [1  1] and pa
    44   'add32'         Addition                 Element-wise addition of 2 inputs
    45   'relu32'        ReLU                     ReLU
    46   'S3U3_conv1'    2-D Convolution          36 3×3×39 convolutions with stride [1  1] and pa
    47   'S3U3_relu1'    ReLU                     ReLU
    48   'S3U3_conv2'    2-D Convolution          39 3×3×36 convolutions with stride [1  1] and pa
    49   'add33'         Addition                 Element-wise addition of 2 inputs
    50   'relu33'        ReLU                     ReLU
    51   'globalPool'    2-D Average Pooling      8×8 average pooling with stride [1  1] and paddi
    52   'fcFinal'       Fully Connected          10 fully connected layer
    53   'softmax'       Softmax                  softmax
    54   'classoutput'   Classification Output    crossentropyex with 'airplane' and 9 other class

### Notice: The layer 'input' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwar
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is imple
### Compiling layer group: convInp>>reluInp ...
### Compiling layer group: convInp>>reluInp ... complete.
### Compiling layer group: S1U1_conv1>>S1U1_conv2 ...
### Compiling layer group: S1U1_conv1>>S1U1_conv2 ... complete.
### Compiling layer group: S1U2_conv1>>S1U2_conv2 ...
### Compiling layer group: S1U2_conv1>>S1U2_conv2 ... complete.
### Compiling layer group: S1U3_conv1>>S1U3_conv2 ...
### Compiling layer group: S1U3_conv1>>S1U3_conv2 ... complete.
### Compiling layer group: skipConv1 ...
### Compiling layer group: skipConv1 ... complete.
### Compiling layer group: S2U1_conv1>>S2U1_conv2 ...
### Compiling layer group: S2U1_conv1>>S2U1_conv2 ... complete.
### Compiling layer group: S2U2_conv1>>S2U2_conv2 ...
### Compiling layer group: S2U2_conv1>>S2U2_conv2 ... complete.
### Compiling layer group: S2U3_conv1>>S2U3_conv2 ...
### Compiling layer group: S2U3_conv1>>S2U3_conv2 ... complete.
### Compiling layer group: skipConv2 ...
### Compiling layer group: skipConv2 ... complete.
### Compiling layer group: S3U1_conv1>>S3U1_conv2 ...
### Compiling layer group: S3U1_conv1>>S3U1_conv2 ... complete.
### Compiling layer group: S3U2_conv1>>S3U2_conv2 ...
### Compiling layer group: S3U2_conv1>>S3U2_conv2 ... complete.
### Compiling layer group: S3U3_conv1>>S3U3_conv2 ...
### Compiling layer group: S3U3_conv1>>S3U3_conv2 ... complete.
### Compiling layer group: globalPool ...
### Compiling layer group: globalPool ... complete.
### Compiling layer group: fcFinal ...
### Compiling layer group: fcFinal ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SchedulerDataOffset"       "0x00800000"     "4.0 MB"
    "SystemBufferOffset"        "0x00c00000"     "28.0 MB"
```

```
    "InstructionDataOffset"     "0x02800000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02c00000"     "4.0 MB"
    "FCWeightDataOffset"        "0x03000000"     "4.0 MB"
    "EndOffset"                 "0x03400000"     "Total: 52.0 MB"

### Network compilation complete.

### Programming FPGA Bitstream using Ethernet...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming FPGA device on Xilinx SoC hardware board at 192.168.1.101...
### Copying FPGA programming files to SD card...
### Setting FPGA bitstream and devicetree for boot...
# Copying Bitstream zcu102_int8.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/zcu102_int8.bit
# Copying Devicetree devicetree_dlhdl.dtb to /mnt/hdlcoder_rd
# Set Devicetree to hdlcoder_rd/devicetree_dlhdl.dtb
# Set up boot for Reference Design: 'AXI-Stream DDR Memory Access : 3-AXIM'
### Rebooting Xilinx SoC at 192.168.1.101...
### Reboot may take several seconds...
### Attempting to connect to the hardware board at 192.168.1.101...
### Connection successful
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 21-Dec-2022 20:05:47
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 21-Dec-2022 20:05:47
### Finished writing input activations.
### Running single input activation.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------             ---------        --
Network                    211741                  0.00085                       1           2
    convInp                  7580                  0.00003
    S1U1_conv1               7045                  0.00003
    S1U1_conv2               7366                  0.00003
    add11                    9215                  0.00004
    S1U2_conv1               7570                  0.00003
    S1U2_conv2               7329                  0.00003
    add12                    9195                  0.00004
    S1U3_conv1              11003                  0.00004
    S1U3_conv2              11193                  0.00004
    add13                    8535                  0.00003
    skipConv1                7126                  0.00003
    S2U1_conv1               6258                  0.00003
    S2U1_conv2               7329                  0.00003
    add21                    4354                  0.00002
    S2U2_conv1               8931                  0.00004
    S2U2_conv2               9074                  0.00004
    add22                    4294                  0.00002
    S2U3_conv1               9013                  0.00004
    S2U3_conv2               9042                  0.00004
    add23                    4614                  0.00002
    skipConv2                6693                  0.00003
    S3U1_conv1               6360                  0.00003
```

```
    S3U1_conv2                 6488                    0.00003
    add31                      1500                    0.00001
    S3U2_conv1                 6313                    0.00003
    S3U2_conv2                 6570                    0.00003
    add32                      1450                    0.00001
    S3U3_conv1                 6123                    0.00002
    S3U3_conv2                 6908                    0.00003
    add33                      1450                    0.00001
    globalPool                 3525                    0.00001
    fcFinal                    2108                    0.00001
 * The clock frequency of the DL processor is: 250MHz
```

Compare the Original, Pruned, and Pruned and Quantized Network Performance

Determine the impact of pruning and quantizing on the network. Pruning does not have an impact on the network performance. However, pruning and quantizing the network improves the performance from 266 frames per second to 1166 frames per second.

```
fprintf('The performance achieved for the original network is %s frames per second. \n', speedIn
```

```
The performance achieved for the original network is 266.2188 frames per second.
```

```
fprintf('The performance achieved after pruning is %s frames per second. \n', speedPruned.("Frame
```

```
The performance achieved after pruning is 266.0607 frames per second.
```

```
fprintf('The performance achieved after pruning and quantizing the network to int8 fixed point is
```

```
The performance achieved after pruning and quantizing the network to int8 fixed point is 1166.00
```

Helper Functions

The `runOnHW` function prepares the network for deployment, compiles the network, deploys the network to the FPGA board, and retrieves the prediction results.

```
function [result, speed] = runOnHW(network, image, bitstream)
   wfObj = dlhdl.Workflow(Network=network,Bitstream=bitstream);
    wfObj.Target = dlhdl.Target("xilinx", Interface="Ethernet");
    compile(wfObj);
    deploy(wfObj);
    [result,speed] = predict(wfObj,image, Profiler='on');
end
```

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `compile` | `deploy` | `predict` | `dlquantizer` | `calibrate`

Deep Learning Quantization

- “Quantization Workflow Prerequisites” on page 11-2
- “Calibration” on page 11-5
- “Validation” on page 11-7
- “Code Generation and Deployment” on page 11-10

Quantization Workflow Prerequisites

This page describes the products required to quantize, simulate, and deploy deep learning networks using Deep Learning Toolbox Model Quantization Library. The prerequisites required depend on your selections at each stage of the quantization workflow.

Prerequisites for All Quantization Workflows

The following requirements apply to all stages of the quantization workflow.

- Deep Learning Toolbox
- Deep Learning Toolbox Model Quantization Library

Supported Networks and Layers

The following links describe the networks and layers supported for each execution environment.

- **GPU** — "Supported Networks, Layers, and Classes" (GPU Coder)
- **FPGA** — "Supported Networks, Layers, Boards, and Tools" on page 7-2
- **CPU** — "Networks and Layers Supported for Code Generation" (MATLAB Coder)
- **MATLAB** — "Networks and Layers Supported for Code Generation" (MATLAB Coder)

Note When the Execution Environment is set to MATLAB, only the layers for the Intel MKL-DNN deep learning library are supported.

Prerequisites for Calibration

The prerequisites for calibration depend on your selection of calibration environment.

- **Calibrate on host GPU (default) —**
  - Parallel Computing Toolbox™
  - GPU Coder™ Interface for Deep Learning
  - CUDA® enabled NVIDIA® GPU with compute capability 3.2 or higher.
- **Calibrate on host CPU —**
  - MATLAB Coder™ Interface for Deep Learning

On Windows®, the `MinGW C/C++` compiler is not supported. Use `Microsoft Visual C++ 2019`, `Microsoft Visual C++ 2017`, or `Microsoft Visual C++ 2015`.

On Linux®, use a GCC C/C++ compiler.

For a list of supported compilers, see Supported and Compatible Compilers.

Prerequisites for Quantization

To quantize your network for simulation in MATLAB using the `quantize` function or the **Export > Export Quantized Network** option in the **Deep Network Quantize** app, no additional prerequisites are required.

Prerequisites for Validation

The following are required to validate your quantized network for deployment using the `validate` function or the **Quantize and Validate** button in the **Deep Network Quantizer** app.

Execution Environment	Prerequisites for Validation
GPU	• Parallel Computing Toolbox • GPU Coder Interface for Deep Learning • CUDA enabled NVIDIA GPU with compute capability 6.1, 6.3 or higher. • “Setting Up the Prerequisite Products” (GPU Coder)
FPGA	• MATLAB Coder Interface for Deep Learning • Deep Learning HDL Toolbox • Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices • Deep Learning HDL Toolbox Support Package for Intel FPGA and SoC Devices • `hdlsetuptoolpath` (HDL Coder)
CPU	• MATLAB Coder Interface for Deep Learning • MATLAB Coder • Embedded Coder® • ARM Compute Library. For more information, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder)
MATLAB	• N/A

For the **FPGA** execution environment, you can choose to validate your quantized network using simulation when you set the `Simulate` property of `dlquantizer` to `'on'`. This option requires only Deep Learning HDL Toolbox.

For CPU and GPU deployment, the software generates code for a convolutional deep neural network by quantizing the weights, biases, and activations of the convolution layers to 8-bit scaled integer data types. The quantization is performed by providing the calibration result file produced by the `calibrate` function to the `codegen` command.

Code generation does not support quantized deep neural networks produced by the `quantize` function.

See Also

Related Examples

- “Quantization of Deep Neural Networks”
- “Quantize Residual Network Trained for Image Classification and Generate CUDA Code”
- “Quantize Network for FPGA Deployment” on page 10-104
- “Generate int8 Code for Deep Learning Networks” (MATLAB Coder)

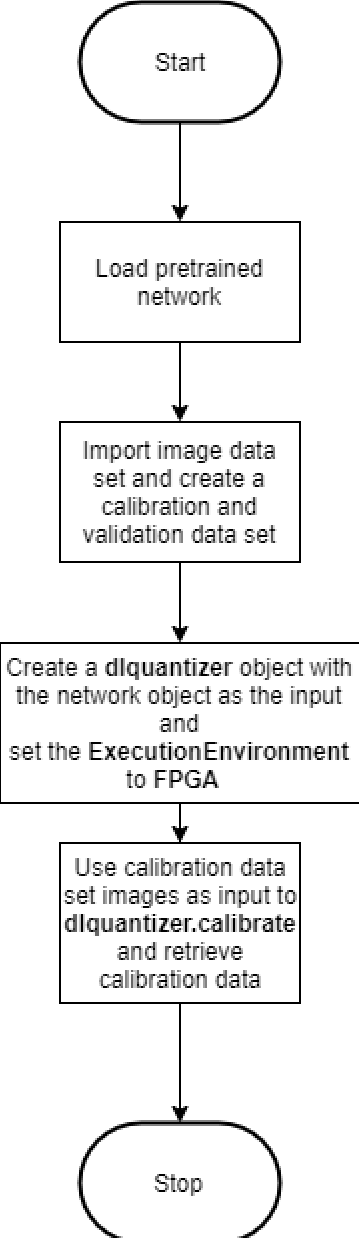
Calibration

Workflow

Collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the quantized network and the dynamic ranges of the activations in all layers.

The `calibrate` method uses the collected dynamic ranges to generate an exponents file. The `dlhdl.Workflow` class `compile` method uses the exponents file to generate a configuration file that contains the weights and biases of the quantized network.

This workflow is the workflow to calibrate your quantized series deep learning network.



See Also

`calibrate` | `dlquantizationOptions` | `dlquantizer` | `validate`

More About

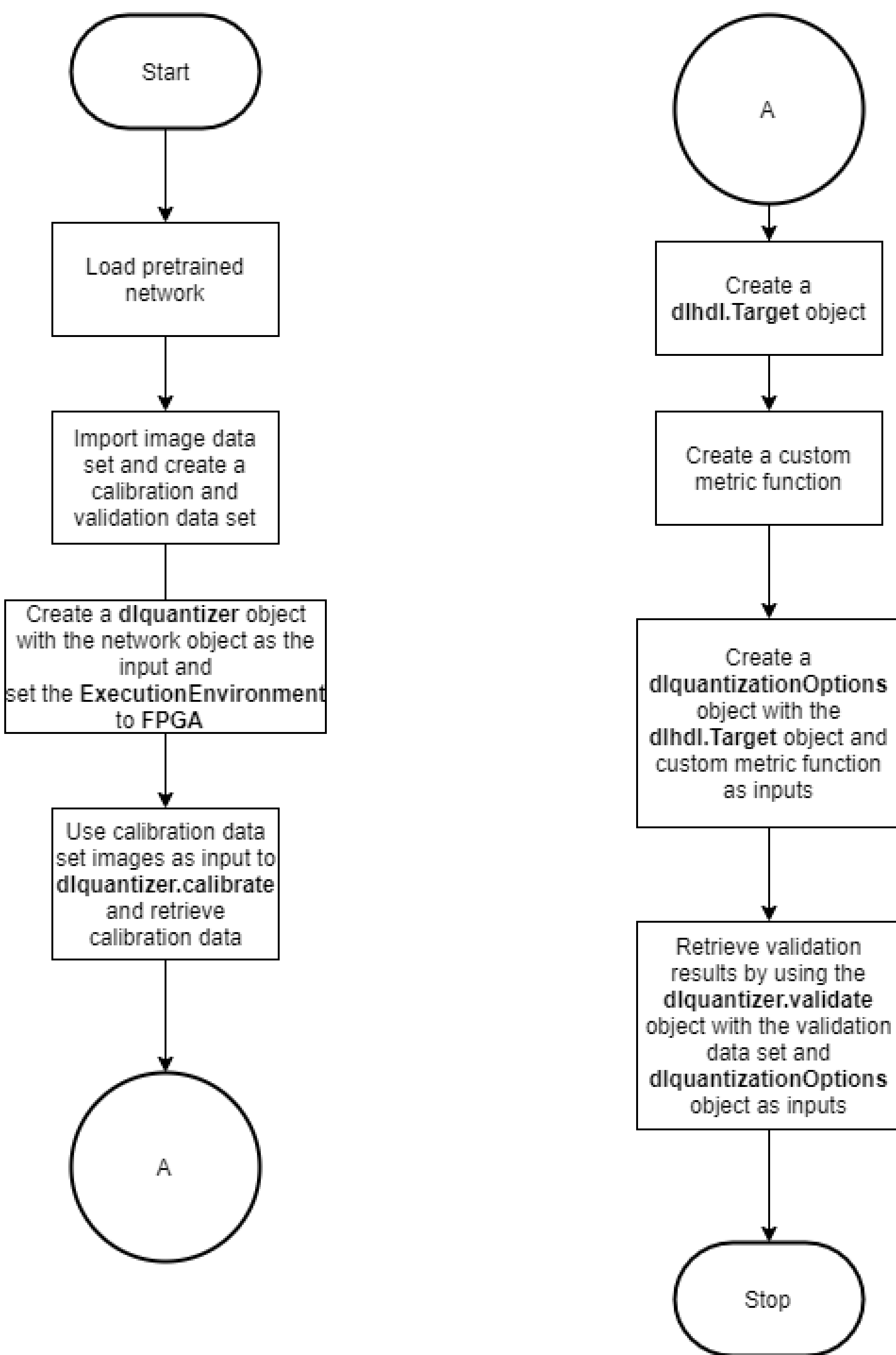
- “Quantization of Deep Neural Networks”
- “Validation” on page 11-7
- “Code Generation and Deployment” on page 11-10

Validation

Workflow

Before deploying the quantized network to your target FPGA or SoC board, to verify the accuracy of your quantized network, use the validation workflow.

This workflow is the workflow to validate your quantized series deep learning network.



See Also

`validate` | `dlquantizationOptions` | `dlquantizer`

More About

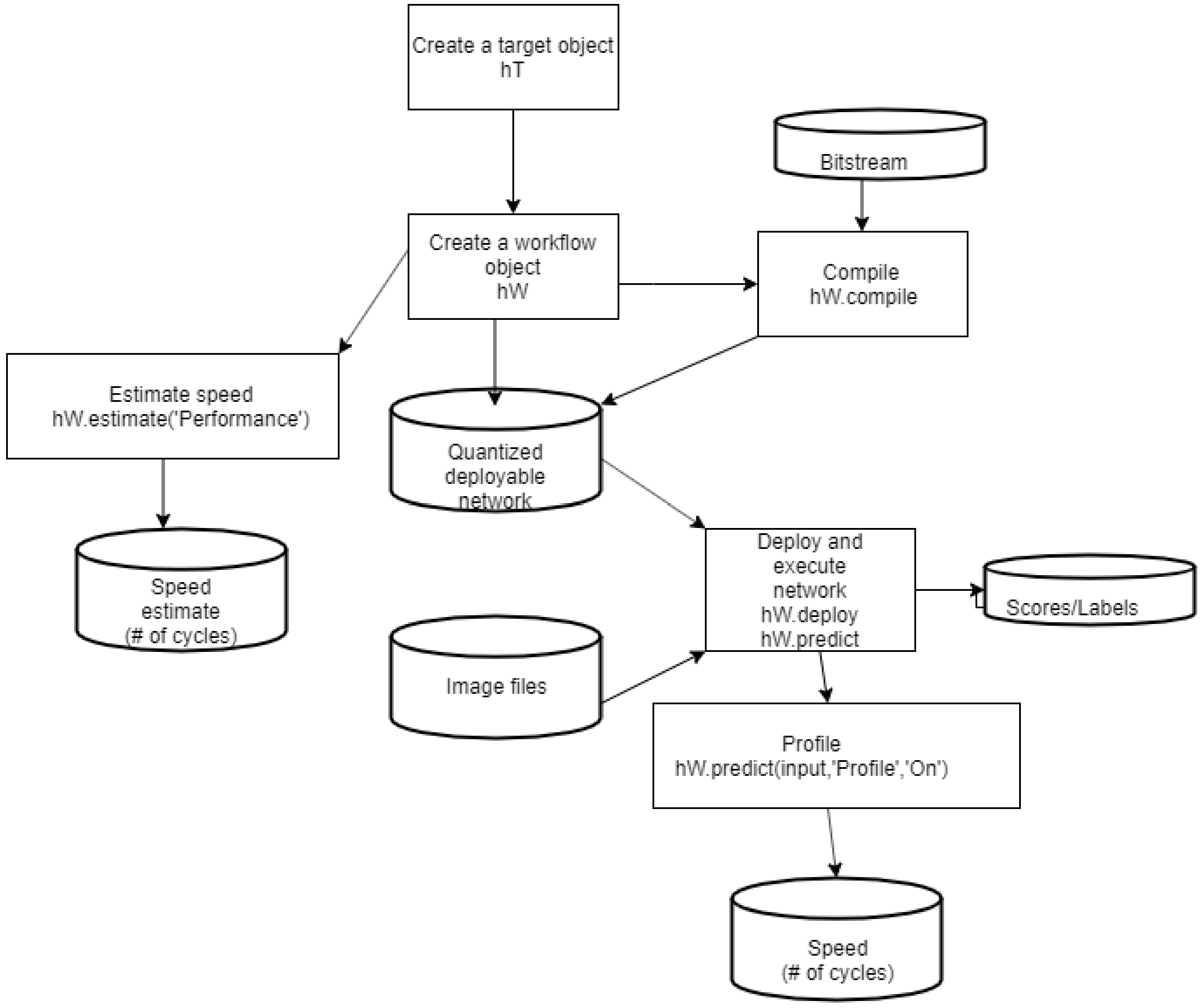
- “Quantization of Deep Neural Networks”
- “Calibration” on page 11-5
- “Code Generation and Deployment” on page 11-10

Code Generation and Deployment

To generated code for and deploy your quantized deep learning network, create an object of class `dlhdl.Workflow`. Use this object to accomplish tasks such as:

- Compile and deploy the quantized deep learning network on a target FPGA or SoC board by using the `deploy` function.
- Estimate the speed of the quantized deep learning network in terms of number of frames per second by using the `estimate` function.
- Execute the deployed quantized deep learning network and predict the classification of input images by using the `predict` function.
- Calculate the speed and profile of the deployed quantized deep learning network by using the `predict` function. Set the `Profile` parameter to `on`.

This figure illustrates the workflow to deploy your quantized deep learning network to the FPGA boards.



See Also

`dlhdl.Workflow` | `dlhdl.Target` | `dlquantizer`

More About

- "Quantization of Deep Neural Networks"
- "Calibration" on page 11-5
- "Validation" on page 11-7

Deep Learning Processor IP Core User Guide

- “Generate Custom Generic Deep Learning Processor IP Core” on page 12-2
- “Deep Learning Processor IP Core” on page 12-5
- “Use the Compiler Output for System Integration” on page 12-6
- “External Memory Data Format” on page 12-9
- “Deep Learning Processor IP Core Report” on page 12-14
- “Interface with the Deep Learning Processor IP Core” on page 12-17
- “Deep Learning Processor IP Core Generation for Custom Board” on page 12-33

Generate Custom Generic Deep Learning Processor IP Core

This example shows how to generate a custom generic deep learning processor IP core. Integrate the generated generic deep learning processor IP core into your larger reference design. To learn how to integrate the generic deep learning processor IP core into your reference design, see “Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core” on page 10-65.

Create Generic Deep Learning Processor Configuration

Create a custom deep learning processor configuration, by using the `dlhdl.ProcessorConfig` object. Set the `TargetPlatform` of the deep learning processor configuration to `'Generic Deep Learning Processor'`.

```
hPC = dlhdl.ProcessorConfig;
hPC.TargetPlatform = 'Generic Deep Learning Processor';
```

Display the modified deep learning processor configuration.

```
hPC
```

```
hPC = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                 SegmentationBlockGeneration: 'on'
                             ConvThreadNumber: 16
                              InputMemorySize: [227 227 3]
                             OutputMemorySize: [227 227 3]
                             FeatureSizeLimit: 2048

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                      SigmoidBlockGeneration: 'off'
                               FCThreadNumber: 4
                              InputMemorySize: 25088
                             OutputMemorySize: 4096

                  Processing Module "custom"
                            ModuleGeneration: 'on'
                                    Addition: 'on'
                              Multiplication: 'on'
                                    Resize2D: 'off'
                                     Sigmoid: 'off'
                                   TanhLayer: 'off'
                              InputMemorySize: 40
                             OutputMemorySize: 40

              Processor Top Level Properties
                               RunTimeControl: 'register'
                                RunTimeStatus: 'register'
                           InputStreamControl: 'register'
                          OutputStreamControl: 'register'
                                 SetupControl: 'register'
                            ProcessorDataType: 'single'
```

```
           System Level Properties
                  TargetPlatform: 'Generic Deep Learning Processor'
                 TargetFrequency: 200
                   SynthesisTool: 'Xilinx Vivado'
                 ReferenceDesign: ''
         SynthesisToolChipFamily: 'Zynq UltraScale+'
         SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
        SynthesisToolPackageName: ''
         SynthesisToolSpeedValue: ''
```

The generic deep learning processor configuration generates a generic Xilinx® IP core. To generate a generic Intel® core, enter:

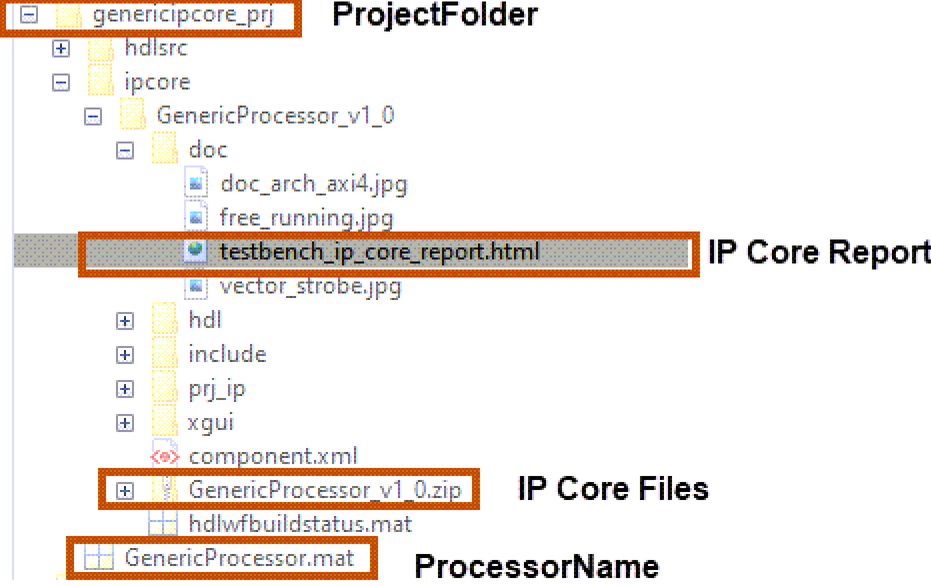
```
hPC.SynthesisTool = 'Altera QUARTUS II'
```

Generate Generic Deep Learning Processor IP Core

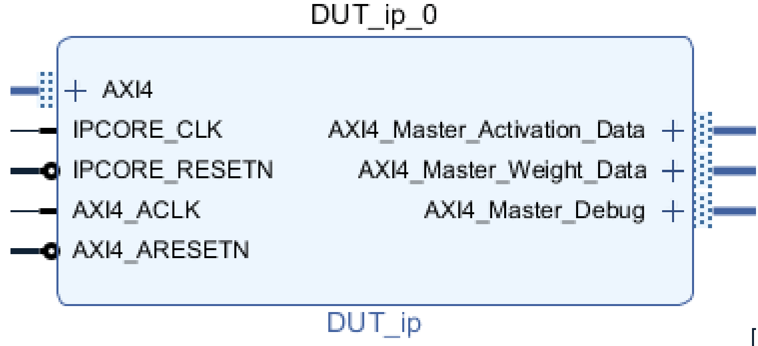
Generate a generic deep learning processor IP core by using the `dlhdl.buildProcessor` function. Set the `ProjectFolder`, `ProcessorName`, and `TargetLanguage` properties of the generic deep learning processor.

```
dlhdl.buildProcessor(hPC,'ProjectFolder','genericipcore_prj','ProcessorName','GenericProcessor',
```

This image shows the files generated for the generic deep learning processor IP core.



This image shows the generated generic deep learning processor IP core:



The generic IP core consists of:

- An AXI4 slave interface called `AXI4`.
- Three AXI4 master interfaces for activation, weights, and utility or debug data.

The `dlhdl.buildProcessor` function also generates an IP core generation report that contains:

- Register address mapping table
- IP core user guide
- IP core file list

For more information, see “Deep Learning Processor IP Core Report” on page 12-14.

See Also

`dlhdl.ProcessorConfig` | `dlhdl.buildProcessor`

More About

- “Custom IP Core Generation” (HDL Coder)
- “Use the Compiler Output for System Integration” on page 12-6
- “External Memory Data Format” on page 12-9
- “Deep Learning Processor IP Core Report” on page 12-14
- “Interface with the Deep Learning Processor IP Core” on page 12-17

Deep Learning Processor IP Core

The generated deep learning (DL) processor IP core is a standard AXI interface IP core that contains:

- AXI slave interface to program the DL processor IP core.
- AXI master interfaces to access the external memory of the target board.

To learn more about the deep learning processor IP core architecture, see “Deep Learning Processor IP Core Architecture” on page 2-2.

The DL processor IP core is generated using the HDL Coder™ IP core generation workflow. The generated IP core contains a standard set of registers and the generated IP core report. For more information, see “Deep Learning Processor IP Core Report” on page 12-14.

The DL processor IP core reads inputs from the external memory and sends outputs to the external memory. The external memory buffer allocation is calculated by the compiler based on the network size and your hardware design. For more information, see “Use the Compiler Output for System Integration” on page 12-6.

The input and output data stored in the external memory in a predefined format. For more information, see “External Memory Data Format” on page 12-9.

The deep learning processor is implemented as a standalone processor on the programmable logic (PL) portion of the FPGA and does not require the processing subsystem (PS) portion of the FPGA to operate. When you compile and deploy a deep learning network most of the network layers are implemented on the PL portion of the FPGA, except for the input and output layers. The layers in this table , “Supported Layers” on page 7-16 with the output format marked as HW are implemented in the PL layer and the layers marked as SW could be implemented on the PS component of an SoC or on the soft core processor on an FPGA when you integrate the deep learning processor into a larger system. In this case the communication between the PS and PL components occurs through DDR memory and Deep Learning HDL Toolbox does not automate the PS or soft core processor implementation.

When you use the `dlhdl.Workflow` object to deploy the network, Deep Learning HDL Toolbox implements the layers with SW output format in MATLAB. The communication between MATLAB and the PL component is through an Ethernet or JTAG interface with the layer activation data being written and read from the DDR memory.

See Also

More About

- “Custom IP Core Generation” (HDL Coder)
- “Use the Compiler Output for System Integration” on page 12-6
- “External Memory Data Format” on page 12-9
- “Deep Learning Processor IP Core Report” on page 12-14
- “Interface with the Deep Learning Processor IP Core” on page 12-17

Use the Compiler Output for System Integration

To integrate the generated deep learning processor IP core into your system reference design, use the `compile` method outputs. The `compile` method:

- Generates the external memory address map.
- Optimizes the network layers for deployment.
- Splits networks into smaller series networks called legs for deployment.

External Memory Address Map

Reduce the time to integrate the generated deep learning processor IP core into your reference design by using the `compile` method external memory address map. Use the address map to:

- Load the inputs to the deep learning processor IP core.
- Load the deep learning processor IP core instructions.
- Load the network weights and biases.
- Retrieve the prediction results.

The external memory address map consists of these address offsets:

- `InputDataOffset` — Address offset where the input images are loaded.
- `OutputResultOffset` — Output results are written starting at this address offset.
- `SchedulerDataOffset` — Address offset where the scheduler runtime activation data is written. The runtime activation data includes information such as hand off between the different deep learning processor kernels, instructions for the different deep learning processor kernels, and so on.
- `SystemBufferOffset` — Do not use the memory address starting at this offset and ending at the start of the `InstructionDataOffset`.
- `InstructionDataOffset` — All layer configuration (LC) instructions are written starting at this address offset.
- `ConvWeightDataOffset` — All conv processing module weights are written starting at this address offset.
- `FCWeightDataOffset` — All fully connected (FC) processing module weights are written starting at this address offset.
- `EndOffset` — DDR memory end offset for generated deep learning processor IP.

For an example of the generated external memory address map, see the "Compile dagnet network object". The example displays the external memory map generated for the ResNet-18 image recognition network and the Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA development board.

Compiler Optimizations

Optimize your custom deep learning network deployment by identifying layers that you can execute in a single operation on hardware by fusing these layers together. The `compile` method performs these layer fusions and optimizations:

- Batch normalization layer (`batchNormalizationLayer`) and 2-D convolution layer (`convolution2dLayer`).

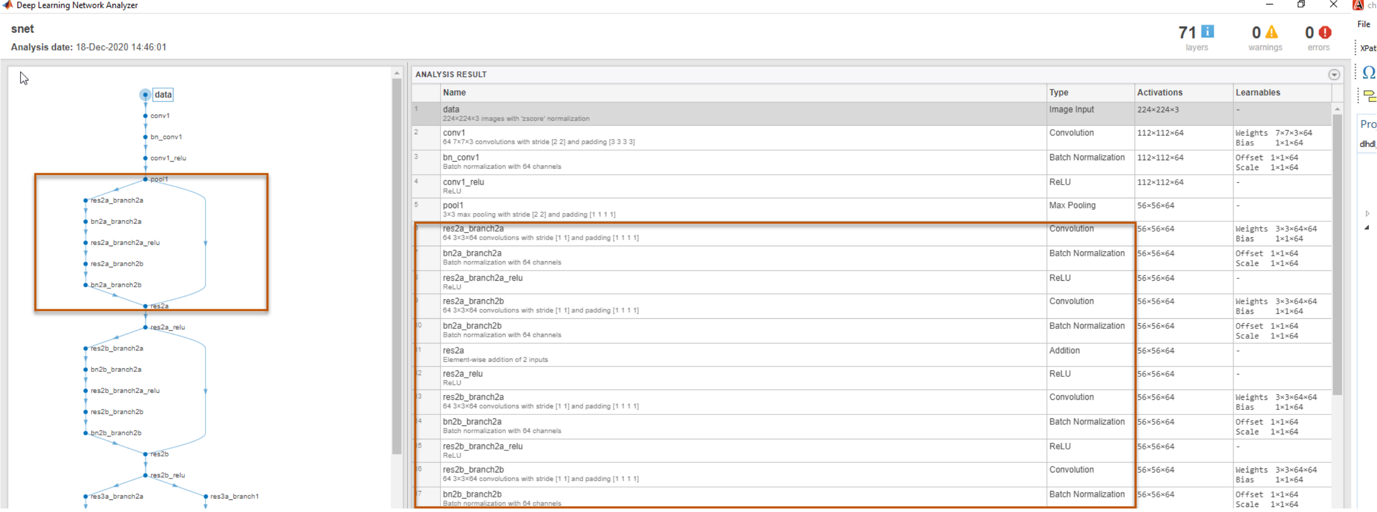
- 2-D zero padding layer (`nnet.keras.layer.ZeroPadding2dLayer`) and 2-D convolution layer (`convolution2dLayer`).
- 2-D zero padding layer (`nnet.keras.layer.ZeroPadding2dLayer`) and 2-D max polling layer (`maxPooling2dLayer`).

This code output is an example compiler optimization in the compiler log.

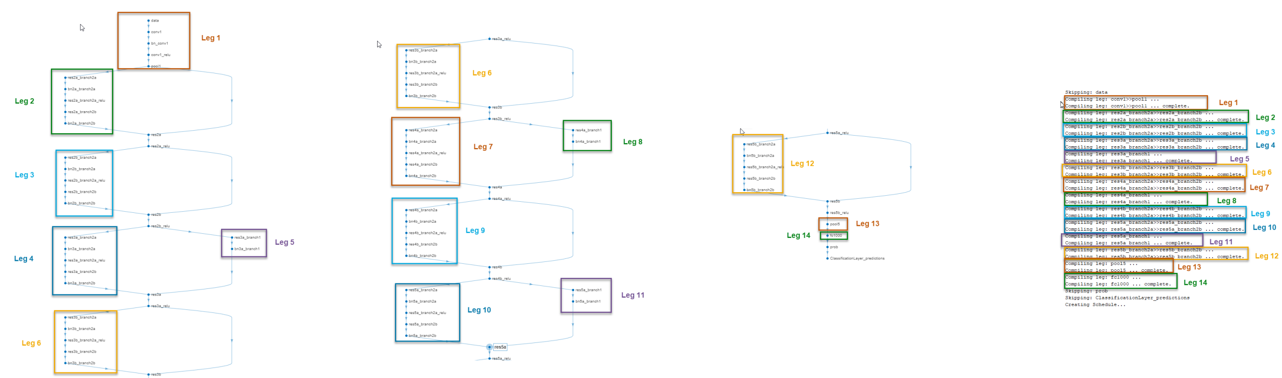
```
Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convolution2DLayer'
```

Leg Level Compilations

Identify subsets of your deep learning networks that could be split into smaller series networks, by using the `compile` method generated legs. A leg is a subset of the DAG network that you can convert into a series network. The `compile` function groups the legs based on the output format of the layers. The layer output format is defined as the data format of the deep learning processor module that processes that layer. The layer output format is conv, fc, or adder. For example, in this image, the `compile` function groups all the layers in `Leg 2` together because they have a conv output format. To learn about the layer output formats, see “Supported Layers” on page 7-16.



This image shows the legs of the ResNet-18 network created by the `compile` function and those legs highlighted on the ResNet-18 layer architecture.



See Also

More About

- "Deep Learning Processor IP Core" on page 12-5
- "External Memory Data Format" on page 12-9
- "Deep Learning Processor IP Core Report" on page 12-14
- "Interface with the Deep Learning Processor IP Core" on page 12-17

External Memory Data Format

To load the input image to the deployed deep learning processor IP core and retrieve the output results, you can read data from the external memory and write data to the external memory by using the `dlhdl.Workflow` workflow. This workflow formats your data. Or, you can manually format your input data. Process the formatted output data by using the external memory data format.

Key Terminology

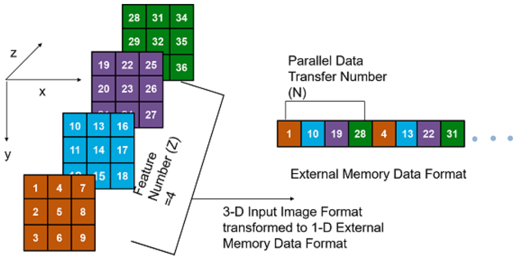
- `Parallel Data Transfer Number` refers to the number of pixels that are transferred every clock cycle through the AXI master interface. Use the letter `N` in place of the `Parallel Data Transfer Number`. Mathematically N is calculated as `power(2,nextpow2(sqrt(ConvThreadNumber)))`. For example, if the convolution thread number is nine, the calculated value of N is four. See “ConvThreadNumber”.
- `Feature Number` refers to the value of the z dimension of an x-by-y-by-z matrix. For example, most input images are of dimension x-by-y-by-three, with three referring to the red, green, and blue channels of an image. Use the letter `Z` in place of the `Feature Number`.
- `Thread Number` refers to the number of channels of the input that are operated upon simultaneously in a convolution style layer. Use the letter `C` in place of the `Thread Number`. Mathematically `C` is calculated as `sqrt(ConvThreadNumber)`. For example, if the convolution thread number is nine, the calculated value of `C` is three. See “ConvThreadNumber”.

Convolution Module External Memory Data Format

The inputs and outputs of the deep learning processor convolution module are typically three-dimensional (3-D).The external memory stores the data in a one-dimensional (1-D) vector. Converting the 3-D input image into 1-D to store in the external memory :

1. Sends `N` number of data in the `z` dimension of the matrix.
2. Sends the image information along the x dimension of the input image.
3. Sends the image information along the y dimension of the input image.
4. After the first `NXY` block is completed, we then send the next `NXY` block along the `z` dimension of the matrix.

The image demonstrates how the data stored in a 3-by-3-by-4 matrix is translated into a 1-by-36 matrix that is then stored in the external memory.

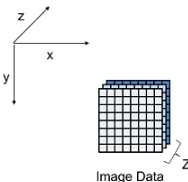


Data Padding for Power of Two Thread Numbers

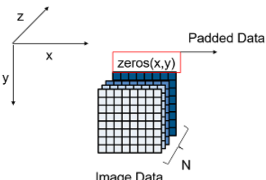
When the image `Feature Number` (Z) is not a multiple of the `Parallel Data Transfer Number` (N), then we must pad a zeroes matrix of size x-by-y along the z dimension of the matrix to make the image `Z` value a multiple of `N`.

For example, if your input image is an x-by-y matrix with a `Z` value of three and the value of `N` is four, pad the image with a zeros matrix of size x-by-y to make the input to the external memory an x-by-y-by-4 matrix.

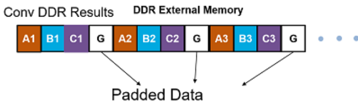
This image is the input image format before padding.



This image is the input image format after zero padding.



The image shows the example output external memory data format for the input matrix after the zero padding. In the image, A, B, and C are the three features of the input image and G is the zero- padded data to make the input image `Z` value four, which is a multiple of `N`.



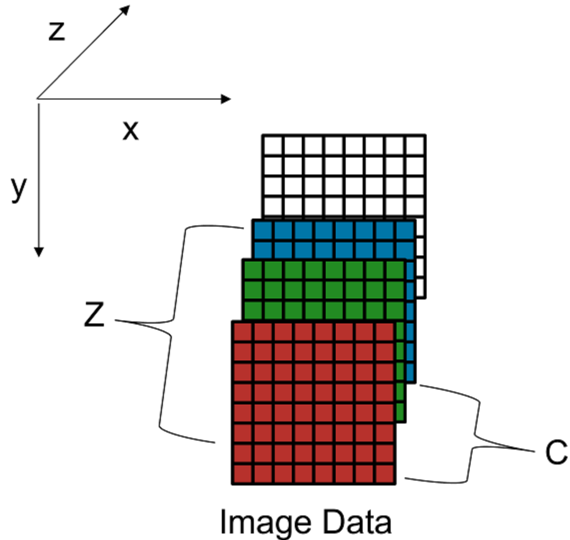
If your deep learning processor consists of only a convolution (conv) processing module, the output external data is using the conv module external data format, which means it possibly contains padded data if your output `Z` value is not a multiple of the `N` value. The padded data is removed when you use the `dlhdl.Workflow` workflow. If you do not use the `dlhdl.Workflow` workflow and directly read the output from the external memory, remove the padded data.

Data Padding for Non-Power of Two Thread Numbers

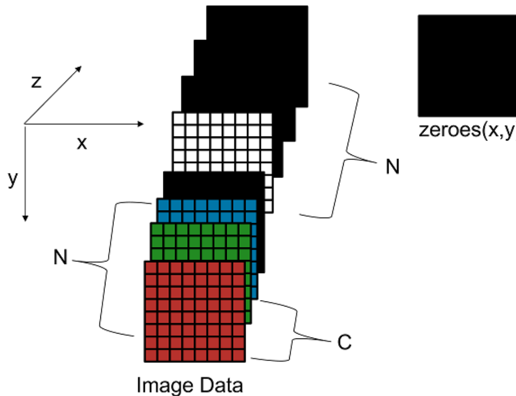
When the `Thread Number` `C` is not a power of two and lower than N, then we must pad a zeroes matrix of size x-by-y along the z dimension of the matrix. The zeroes matrix is inserted after every C number of elements along the z dimension of the matrix to make the Z value a multiple of N.

For example, if your input image is an x-by-y matrix with a `C` value of three and `N` and `Z` values of four, pad the image with a zeroes matrix of size x-by-y after the third channel and three zeroes matrices of x-by-y after the fourth channel to make the input to the external memory an x-by-y-by-eight matrix.

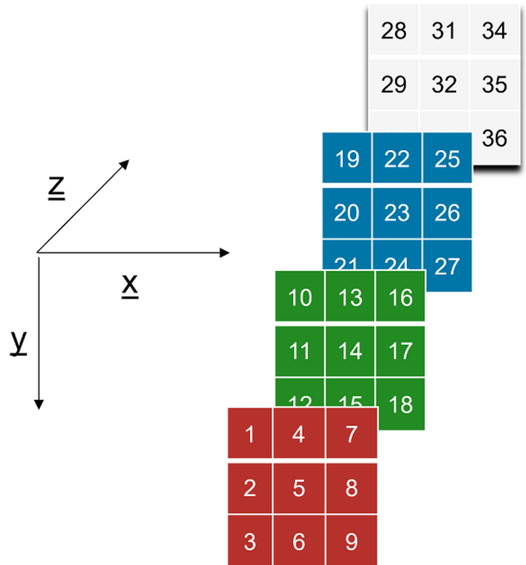
This image is the input image format before padding.



This image is the input image format after zero padding.



This image shows a sample three-by-three-by-four matrix passed as an input to a deep learning processor configuration with a `C` value of three and `N` value of four.



The image shows the example output external memory data format for the input matrix after the zero padding.

When the values of C and N are equal padding is required only when Z is not a multiple of C.

Calculation of Output Memory Size

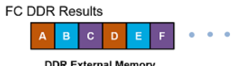
The size of the output for a deep learning processor IP core depends on the `Feature Number(Z)`, `Thread Number (C)`, and the `Parallel Data Thread Number (N)`. The formula to calculate the output memory size is `dimension1 * dimension2 * ceil(Z/C) * N`. For example, for an input matrix of size three-by-three-by-four the output memory size for a C and N value of four is `3 *3 *ceil(4/4) *4 = 36`. In this example the output is written four values at a time because the value of N is four.

For a three-by-three-by-four matrix with a C value of three and N value of four, the output size is `3 *3 *ceil(4/3) *4 =72`. In this example even when the output is written four values at a time only the first three values are valid as the fourth value is a zero padded value.

Fully Connected Module External Memory Data Format

If your deep learning network consists of both the convolution (conv) and fully connected (fc) layers, the output of the deep learning (DL) processor follows the fc module external memory data format.

The image shows the example external memory output data format for a fully connected output feature size of six. In the image, A, B, C, D, E, and F are the output features of the image.



See Also

More About

- “Deep Learning Processor IP Core” on page 12-5
- “Use the Compiler Output for System Integration” on page 12-6
- “Deep Learning Processor IP Core Report” on page 12-14
- “Interface with the Deep Learning Processor IP Core” on page 12-17

Deep Learning Processor IP Core Report

When you generate a deep learning processor IP core, Deep Learning HDL Toolbox generates an HTML custom IP core report. The report describes the behavior and content of the generated custom IP core. During processor generation, AXI4 slave registers are created to enable MATLAB or other master devices to control and program the deep learning (DL) processor IP core.

Summary

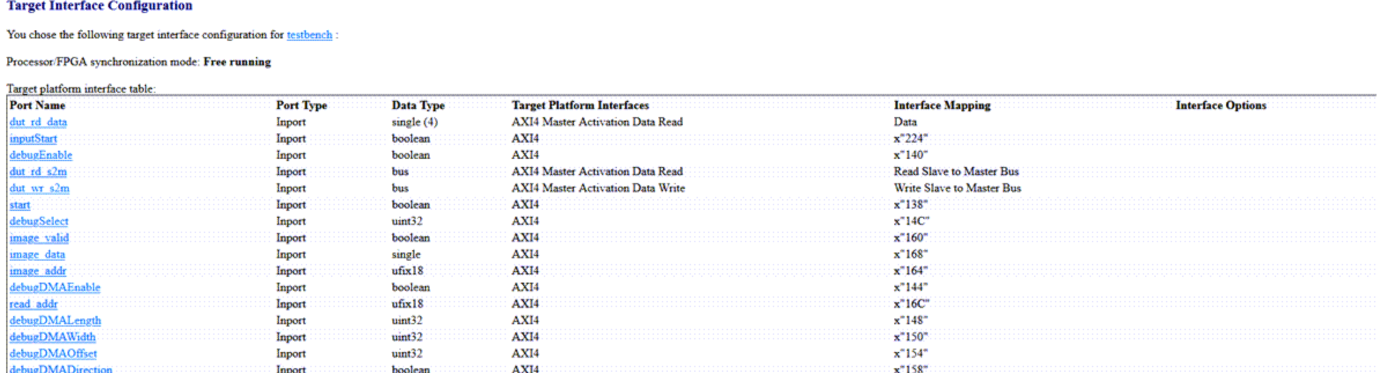
This section shows the Deep Learning HDL Toolbox settings when you generated the custom IP core.

Summary

IP core name	GenericProcessor
IP core version	1.0
IP core folder	genericipcore_prj\ipcore\GenericProcessor_v1_0
IP core zip file name	GenericProcessor_v1_0.zip
Target platform	Generic Deep Learning Processor Xilinx
Target tool	Xilinx Vivado
Target language	VHDL
Model	testbench
Model version	4.161
HDL Coder version	4.0
IP core generated on	22-Jun-2022 17:07:19
IP core generated for	DUT

Target Interface Configuration

This section shows how your model design under test (DUT) ports map to the target hardware interface and the processor/FPGA synchronization mode.



Target Interface Configuration

You chose the following target interface configuration for testbench :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
dut_rd_data	Inport	single (4)	AXI4 Master Activation Data Read	Data	
inputStart	Inport	boolean	AXI4	x"224"	
debugEnable	Inport	boolean	AXI4	x"140"	
dut_rd_s2m	Inport	bus	AXI4 Master Activation Data Read	Read Slave to Master Bus	
dut_wr_s2m	Inport	bus	AXI4 Master Activation Data Write	Write Slave to Master Bus	
start	Inport	boolean	AXI4	x"138"	
debugSelect	Inport	uint32	AXI4	x"14C"	
image_valid	Inport	boolean	AXI4	x"160"	
image_data	Inport	single	AXI4	x"168"	
image_addr	Inport	ufix18	AXI4	x"164"	
debugDMAEnable	Inport	boolean	AXI4	x"144"	
read_addr	Inport	ufix18	AXI4	x"16C"	
debugDMALength	Inport	uint32	AXI4	x"148"	
debugDMAWidth	Inport	uint32	AXI4	x"150"	
debugDMAOffset	Inport	uint32	AXI4	x"154"	
debugDMADirection	Inport	boolean	AXI4	x"158"	

Register Address Mapping

During custom processor generation, AXI4 slave registers are created to enable MATLAB or other master devices to control and program the DL processor IP core.

The DL processor IP core is generated by using the HDL Coder IP core generation workflow. The generated IP core contains a standard set of registers. For more information, see “Custom IP Core Generation” (HDL Coder).

For the full list of register offsets, see the `Register Address Mapping` table in the generated deep learning (DL) processor IP core report.

Register Address Mapping

The following AXI4 bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
AXI4_Master_Activation_Data_Rd_BaseAddr	0x8	Base Address offset for AXI4 Master Activation Data Read (Default Base Address: hex2dec(0))
AXI4_Master_Activation_Data_Wr_BaseAddr	0xC	Base Address offset for AXI4 Master Activation Data Write (Default Base Address: hex2dec(0))
AXI4_Master_Weight_Data_Rd_BaseAddr	0x10	Base Address offset for AXI4 Master Weight Data Read (Default Base Address: hex2dec(0))
AXI4_Master_Debug_Rd_BaseAddr	0x14	Base Address offset for AXI4 Master Debug Read (Default Base Address: hex2dec(0))
AXI4_Master_Debug_Wr_BaseAddr	0x18	Base Address offset for AXI4 Master Debug Write (Default Base Address: hex2dec(0))
IPCore_Timestamp	0x1C	contains unique IP timestamp (yymmddHHMM): 2206221707
AXIStreamInData_Data	0x100	data register for Inport AXIStreamInData, vector with 4 elements, address ends at 0x10C
AXIStreamInData_Strobe	0x110	strobe register for port AXIStreamInData
AXIStreamInValid_Data	0x114	data register for Inport AXIStreamInValid
AXIStreamOutReady_Data	0x118	data register for Inport AXIStreamOutReady

IP Core User Guide

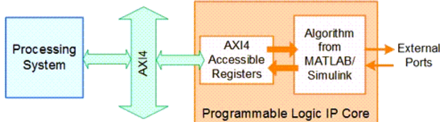
This section gives a high-level overview of the generated IP core architecture and instructions to integrate the IP core into your reference design.

IP Core User Guide

Theory of Operation

This IP core is designed to be connected to an embedded processor with an **AXI4 interface.** The processor acts as master, and the IP core acts as slave. By accessing the generated registers via the AXI4 interface, the processor can control the IP core, and read and write data from and to the IP core.

For example, to reset the IP core, write 0x1 to the bit 0 of IPCore_Reset register. To enable or disable the IP core, write 0x1 or 0x0 to the IPCore_Enable register. To access the data ports of the MATLAB/Simulink algorithm, read or write to the associated data registers.



IP Core File List

This section lists the files and folders that are a part of the generated deep learning processor IP core.

IP Core File List

The IP core folder is located at:
genericipcore_prj\ipcore\GenericProcessor_v1_0
Following files are generated under this folder:

IP core zip file
GenericProcessor_v1_0.zip

IP core report
doc\testbench_ip_core_report.html

IP core HDL source files
hdl\GenericProcessor_src_DUT_pkg.vhd
hdl\GenericProcessor_src_Subsystem1.vhd
hdl\GenericProcessor_src_AXI_Stream_top.vhd
hdl\GenericProcessor_src_EdgeDetection1.vhd
hdl\GenericProcessor_src_EdgeDetection3.vhd
hdl\GenericProcessor_src_Goto11.vhd
hdl\GenericProcessor_src_Goto12.vhd
hdl\GenericProcessor_src_Goto13.vhd
hdl\GenericProcessor_src_GotoTag.vhd
hdl\GenericProcessor_src_nfp_wire_single.vhd
hdl\GenericProcessor_src_DataTypeConv1.vhd
hdl\GenericProcessor_src_DataTypeConv2.vhd
hdl\GenericProcessor_src_EdgeDetection1_block.vhd
hdl\GenericProcessor_src_Scalar_Replicator1.vhd
hdl\GenericProcessor_src_Debug_Log_Tap.vhd
hdl\GenericProcessor_src_EdgeDetection1_block1.vhd
hdl\GenericProcessor_src_adder_output_burst_done.vhd
hdl\GenericProcessor_src_dvalid_conversion.vhd
hdl\GenericProcessor_src_dvalid_conversion1.vhd

See Also

More About

- “Deep Learning Processor IP Core” on page 12-5
- “Use the Compiler Output for System Integration” on page 12-6
- “External Memory Data Format” on page 12-9
- “Interface with the Deep Learning Processor IP Core” on page 12-17

Interface with the Deep Learning Processor IP Core

Retrieve predictions for a batch of images or for a data stream from a live camera input by using the generated deep learning processor IP core. Select between batch processing mode and streaming mode depending on available board resources, availability of input data, and application requirements. Use MATLAB to run your deep learning network on the generated deep learning processor IP core and retrieve the network prediction from the generated deep learning processor IP core.

Create Deep Learning Processor Configuration

To generate a deep learning processor IP core that has the required interfaces for processing multiple data frames, create a deep learning processor configuration by using the `dlhdl.ProcessorConfig` class. In the deep learning processor configuration:

- Set `InputRunTimeControl` and `OutputRunTimeControl` to either port or register.
- You must set `InputDataInterface` and `OutputDataInterface` to `ExternalMemory`.

Use the `dlhdl.buildProcessor` function with the deep learning processor configuration object as the input argument to generate the deep learning processor IP core. For example, this code generates a deep learning processor IP core with the interfaces to process multiple data frames.

```
hPC = dlhdl.ProcessorConfig;
hPC.InputRunTimeControl = 'port';
hPC.OutputRunTimeControl = 'port'
hPC.InputDataInterface = 'External Memory';
hPC.OutputDataInterface = 'External Memory';
dlhdl.buildProcessor(hPC);
```

Select Data Processing Mode

Choose between batch processing mode and streaming mode based on your resource requirements, availability of inputs, and interface complexity. This table lists the different selection criteria and which mode to select based on the selection criteria.

Selection Criteria	Batch Processing Mode	Streaming Mode
Availability of input data	All input data must be available before you trigger the deep learning processor IP core to start processing data.	Stream input data as and when data is available.
Memory requirements	Can require large memory resources to store all the input data and processed output data as the deep learning processor IP core processes all the data together.	Requires minimal memory resources. The smallest memory required is twice the size of one input data frame.
Interface Complexity	Simple protocol. No handshaking protocol required.	Complex protocol. You must implement a handshaking protocol.

Design Processing Mode Interface Signals

You can group the interface signals into run-time signals, handshaking signals, and setup control signals. Handshaking signals are used only when the data processing mode is set to streaming mode.

Run-Time Control Signals

This table lists the run-time control signals, data types, interface types, and description. The interface type depends on the `RunTimeControl` settings. For example, if `RunTimeControl` is set to `port`, the interface type is `port`.

Signal Name	Data Type	Configuration Control Parameter or	Interface Type (Port or Register)	Description	Interface Direction (Input or Output)
`InputStart`	logical	`RunTimeControl`	`port/register`	Signal from the user to the deep learning processor IP core to start processing the data.	Input
`FrameCount`	integer	`RunTimeControl`	`port/register`	Signal from the user to the deep learning processor IP core specifying the number of input data frames.	Input
`InputStop`	logical	`RunTimeControl`	`port/register`	Signal to stop the continuous streaming mode. To stop the continuous streaming mode, set this signal to `true`.	Input

Run-Time Status Signals

This table lists the run-time control signals, data types, interface types, and description. The interface type depends on the `RunTimeStatus` settings. For example, if `RunTimeStatus` is set to `port`, the interface type is `port`.

Signal Name	Data Type	Configuration Control Parameter or	Interface Type (Port or Register)	Description	Interface Direction (Input or Output)

Done	logical	RunTimeStatus	port/register	Signal indicating that the deep learning processor IP core has processed all input data and written the last output to memory.	Output
StreamingDone	logical	RunTimeStatus	port/register	Signal to test streaming mode. During testing, the signal becomes true when you retrieve the last output.	Output

Handshaking signals

This table lists the handshaking signals, data types, interface types, and description. These signals are used for streaming mode. The interface type depends on the InputRunTimeControl and OutputRunTimeControl settings. For example, if InputRunTimeControl is set to port, the interface type is port. To ensure proper functionality of the generated deep learning processor, you must specify the values for the signals that are ports or registers.

Signal Name	Data Type	Configuration Control Parameter or	Interface Type (Port or Register)	Description	Interface Direction (Input or Output)
InputAddr	uint32	InputStreamControl	port/register	Signal indicating the address location in memory for loading the input data. Use this signal when the InputValid signal is high.	Output

InputNext	logical	InputStreamControl	port/register	Signal to the deep learning processor IP core to indicate that the next data frame is available for processing. Use this signal when the InputValid signal is high.	Input
InputSize	uint32	InputStreamControl	port/register	Signal indicating the size in bytes of the next input data frame. Use this signal when the InputValid signal is high. The InputSize data includes the zero padding applied to the input data.	Output
InputValid	logical	InputStreamControl	port/register	Signal from the deep learning processor IP core indicating that the input data is valid.	Output
OutputAddr	uint32	OutputStreamControl	port/register	Signal indicating the address location in memory from where to retrieve the output data. Use this signal when the OutputValid signal is high.	Output

OutputNext	logical	OutputStreamControl	port/register	Signal to the deep learning processor IP core to indicate that you have read the current output data frame. Use this signal when the OutputValid signal is high.	Input
OutputSize	uint32	OutputStreamControl	port/register	Signal indicating the size of the next output data frame in bytes. Use this signal when the OutputValid signal is high.	Output
OutputValid	logical	OutputStreamControl	port/register	Signal from the deep learning processor IP core indicating that the output data is valid.	Output

Setup Control Signals

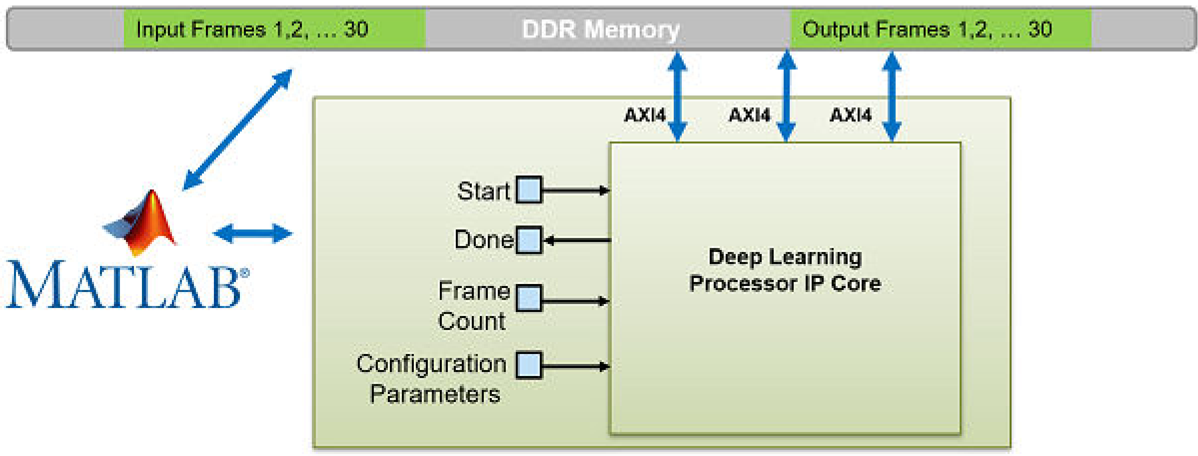
This table lists the setup control signals, data types, interface types, and description. The interface type depends on the SetupControl settings. For example, if SetupControl is set to port, the interface type is port.

Signal Name	Data Type	Configuration Control Parameter	Interface Type (Port or Register)	Description	Interface Direction (Input or Output)
StreamingMode	logical	SetupControl	port/ register	Signal from the user to the deep learning processor IP core specifying the data processing mode. `false` selects buffer mode and `true` selects streaming mode.	Input
UseCustomBaseAddr	logical	SetupControl	port/ register	Signal from the user to the deep learning processor IP core to use the customer specified input and output base addresses.`true` selects user addresses and `false` selects compiler generated addresses.	Input
InputBaseAddr	uint32	SetupControl	port/ register	User provided input base address. Specify the address before you toggle the `InputStart` signal.	Input
OutputBaseAddr	uint32	SetupControl	port/ register	User provided output base address. Specify the address before you toggle the `InputStart` signal.	Input

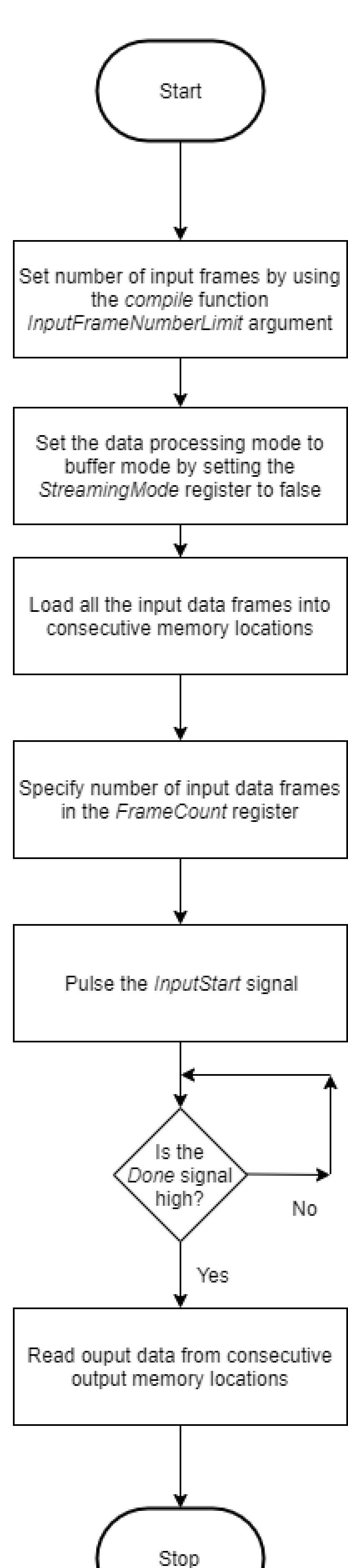
Design Batch Processing Mode Interface

When you have all your input data available and access to large double data rate (DDR) memory space, process multiple frames by using the batch processing mode. The figure shows the generated deep learning processor IP core with interface signals for the batch processing mode of operation. You use MATLAB and a `dlhdl.Workflow` object to run your deep learning network on the deep learning processor IP core. Retrieve the network prediction results from the deep learning processor IP core. To use batch mode, set the `FrameCount` register to a value greater than or equal to one.

To process a single data frame set the `FrameCount` register value to one. If the `FrameCount` is set to zero the deep learning processor runs intermittently and the `Done` signal does not become `true`.

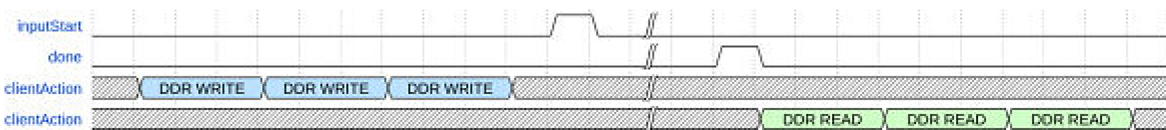


This flowchart shows the operation of the batch processing mode.



Start
Set number of input frames by using the *compile* function *InputFrameNumberLimit* argument
Set the data processing mode to buffer mode by setting the *StreamingMode* register to false
Load all the input data frames into consecutive memory locations
Specify number of input data frames in the *FrameCount* register
Pulse the *InputStart* signal
Is the *Done* signal high?
No
Yes
Read ouput data from consecutive output memory locations
Stop

This timing diagram shows the operation of the batch processing mode.



Load all the data frames into consecutive input DDR memory locations, toggle the `inputStart` signal, wait for the `done` signal to go high, and then read the output data from the consecutive output DDR memory locations. The `clientAction` signals represent your actions of loading input data and reading output data into the DDR memory.

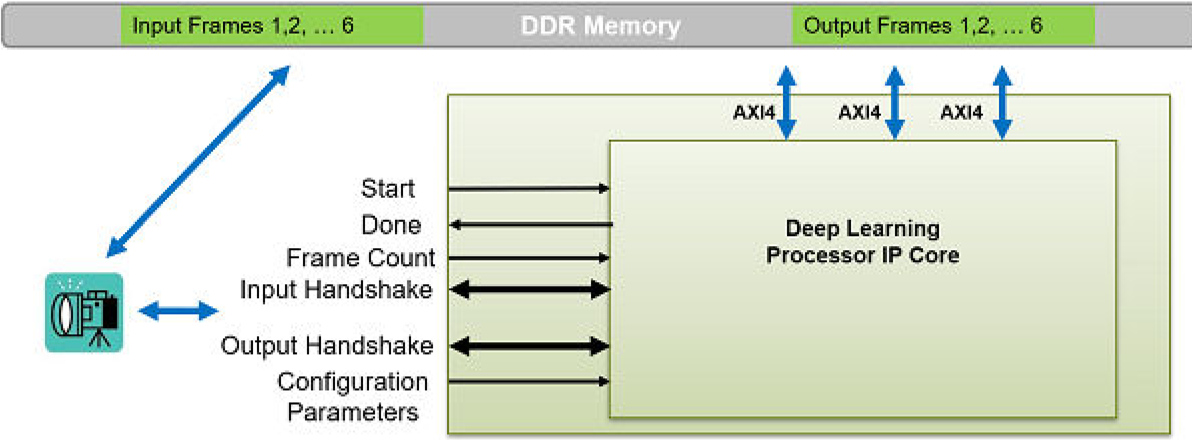
Design Streaming Mode Interface

When your input data is streaming in, when you have access to limited DDR memory space, and when your application requires handshaking protocols, process multiple frames by using the streaming mode. The figure shows the generated deep learning processor IP core with interface signals for the streaming mode of operation. In this figure, the live camera streams data to an image preprocessing design under test (DUT) that implements the streaming mode handshaking protocol to interact with the generated deep learning processor IP core.

Date can be streamed to the deep learning processor IP core in two modes:

- Stream Data up to a frame count value— In this mode the deep learning processor processes data frames up to the value specified in `FrameCount`. After processing all the frames the deep learning processor IP core sets the `Done` signal to `true`. To use this mode the `FrameCount` must be set to a value greater than or equal to one.

  To process a single data frame set the `FrameCount` register value to one.
- Continuous streaming mode— In this mode the deep learning processor IP core processes data frames until you set the `InputStop` value to `true`. To use this mode the `FrameCount` must be set to zero.



Streaming Mode up to a Frame Count

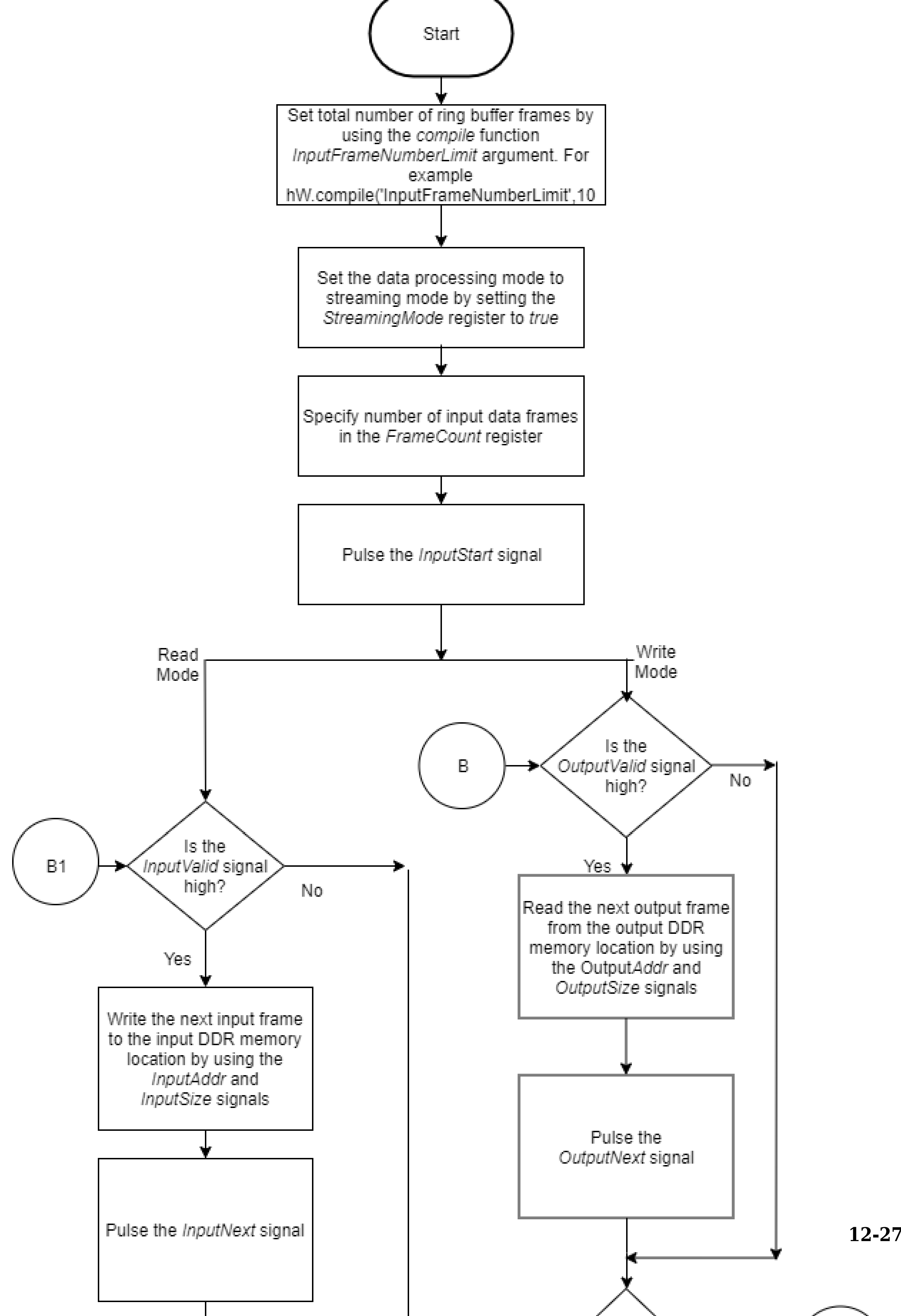
This flowchart shows the operation of the streaming mode data processing mode. The read and write operations occur in parallel.

The value set in the `InputFrameNumberLimit` specifies in terms of input and output frames the space that is allocated in the DDR for the input and output ring buffers. In streaming mode, this value must be at least two. When backpressure is applied, for values larger than two the deep learning processor IP core:

- Continues to accept input data until the input ring buffer is full.
- Continues to produce output data until the output ring buffer is full.

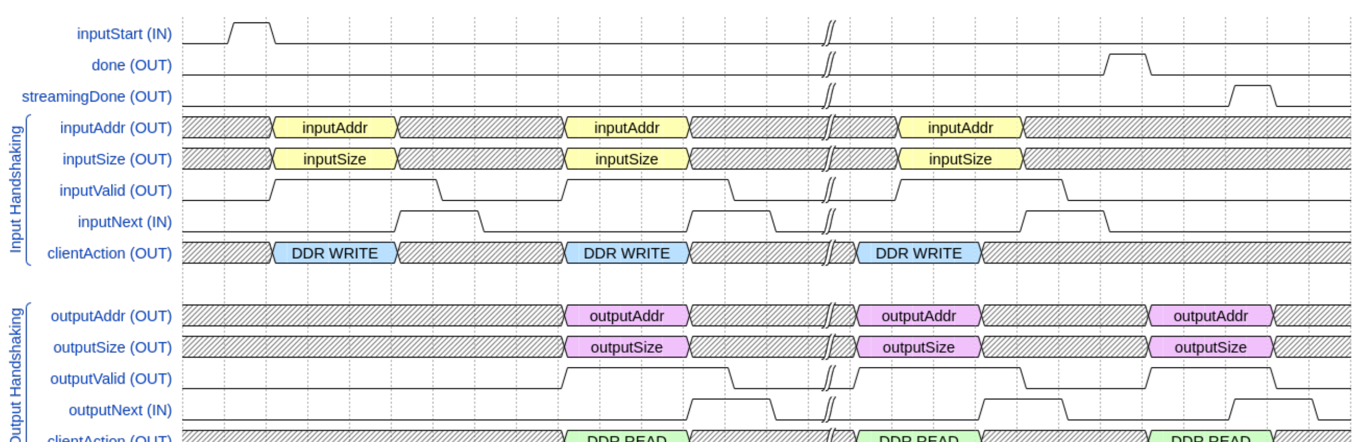
As streaming continues, the input and output buffers fill and drain based on output backpressure and input data availability.

This flowchart shows the operation of the streaming mode up to a frame count. The read and write operations occur in parallel.



Start
Set total number of ring buffer frames by using the compile function InputFrameNumberLimit argument. For example hW.compile('InputFrameNumberLimit',10
Set the data processing mode to streaming mode by setting the StreamingMode register to true
Specify number of input data frames in the FrameCount register
Pulse the InputStart signal
Read Mode
Write Mode
B
Is the OutputValid signal high?
No
Yes
Read the next output frame from the output DDR memory location by using the OutputAddr and OutputSize signals
Pulse the OutputNext signal
B1
Is the InputValid signal high?
No
Yes
Write the next input frame to the input DDR memory location by using the InputAddr and InputSize signals
Pulse the InputNext signal

This timing diagram shows the operation of the streaming mode up to a frame count.



1. Set the `InputFrameNumberLimit` argument of the `compile` method to a value greater than two.
2. Set the `StreamingMode` signal to `true`.
3. Set the number of data frames to process in the `FrameCount` register.
4. Pulse the `inputStart` signal. These next actions can be performed in parallel:

   a. Wait for the `inputValid` signal to become `true` and then:

      - Use the `inputAddr` and `inputSize` signals to write the next input data frame to DDR memory.
      - Pulse the `inputNext` signal.

   b. Wait for the `outputValid` signal to become `true` and then:

      - Use the `outputAddr` and `outputSize` signals to read the processed output data frame.
      - Pulse the `outputNext` signal.
5. Once the deep learning processor IP core has processed all the frames it sets the `done` signal to `true`.

The `clientAction` signals represent your actions of loading input data and reading output data into the DDR memory.

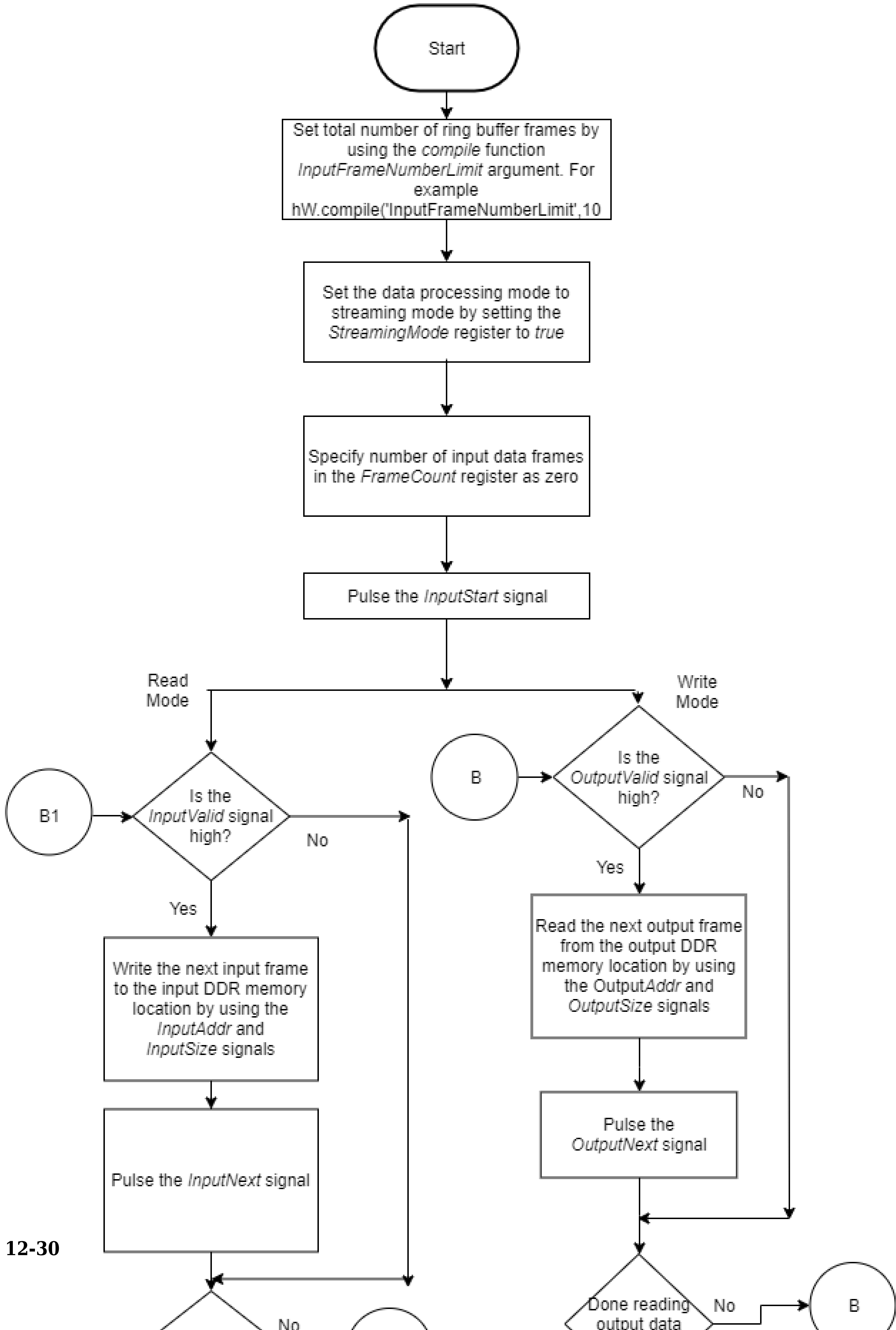
Continuous Streaming Mode

You can continuously stream data to the deep learning processor in continuous streaming mode. To use the continuous streaming mode, set the `FrameCount` to zero. To stop the data processing set the `InputStop` signal to `true`. The value set in the `InputFrameNumberLimit` specifies in terms of input and output frames the space that is allocated in the DDR for the input and output ring buffers. When backpressure is applied, for values larger than the value in `InputFrameNumberLimit` the deep learning processor IP core:

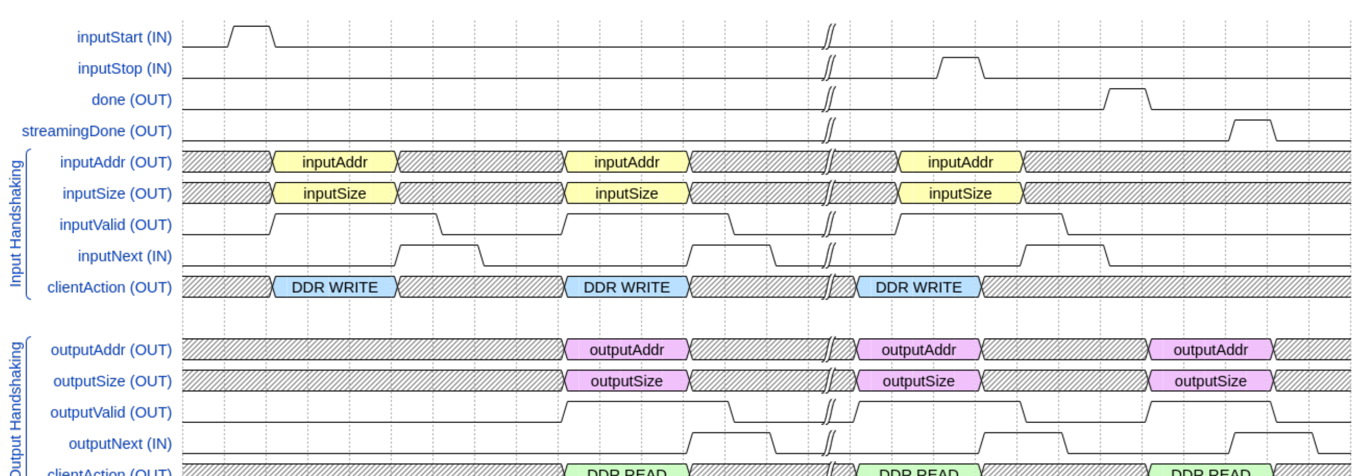
- Continues to accept input data until the input ring buffer is full.
- Continues to produce output data until the output ring buffer is full.

As streaming continues, the input and output buffers fill and drain based on output backpressure and input data availability.

This flowchart shows the operation of the continuous streaming mode. The read and write operations occur in parallel.


Start
Set total number of ring buffer frames by using the compile function InputFrameNumberLimit argument. For example hW.compile('InputFrameNumberLimit',10
Set the data processing mode to streaming mode by setting the StreamingMode register to true
Specify number of input data frames in the FrameCount register as zero
Pulse the InputStart signal
Read Mode
Write Mode
B1
Is the InputValid signal high?
No
Yes
Write the next input frame to the input DDR memory location by using the InputAddr and InputSize signals
Pulse the InputNext signal
No
B
Is the OutputValid signal high?
No
Yes
Read the next output frame from the output DDR memory location by using the OutputAddr and OutputSize signals
Pulse the OutputNext signal
Done reading output data
No
B

This timing diagram shows the operation of the continuous streaming mode.



1. Set the `InputFrameNumberLimit` argument of the `compile` method to a value greater than two.
2. Set the `StreamingMode` signal to `true`.
3. Set the number of data frames to process in the `FrameCount` register to zero.
4. Pulse the `inputStart` signal. These next actions can be performed in parallel:

   a. Wait for the `inputValid` signal to become `true` and then:

      - Use the `inputAddr` and `inputSize` signals to write the next input data frame to DDR memory.
      - Pulse the `inputNext` signal.

   b. Wait for the `outputValid` signal to become `true` and then:

      - Use the `outputAddr` and `outputSize` signals to read the processed output data frame.
      - Pulse the `outputNext` signal.
5. Once you have written all the input data and read all the output data pulse the `InputStop` signal.

Access Data from DDR

The deep learning IP core uses the three AXI4 Master interfaces to store and process:

- Activation data
- Weight data
- Debug data

The deep learning processor reads and writes data from the DDR based on the data processing mode of operation by using these AXI4 Master interfaces.

See Also

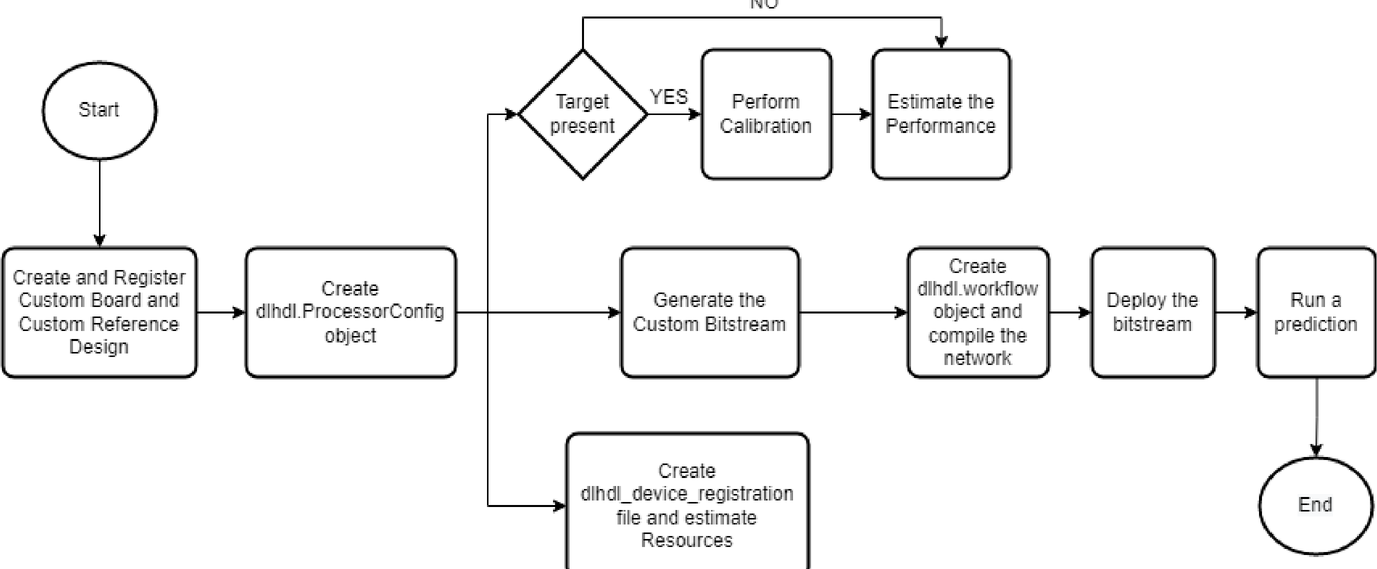
“Deep Learning Processor IP Core” on page 12-5 | “Use the Compiler Output for System Integration” on page 12-6 | “External Memory Data Format” on page 12-9

Deep Learning Processor IP Core Generation for Custom Board

This example shows how to create custom board and generate a deep learning processor IP core for the custom board. In this example you:

- Create a custom board and reference design
- Estimate the network performance and board resource utilization
- Generate a custom processor and bitstream
- Deploy the network by using the custom bitstream

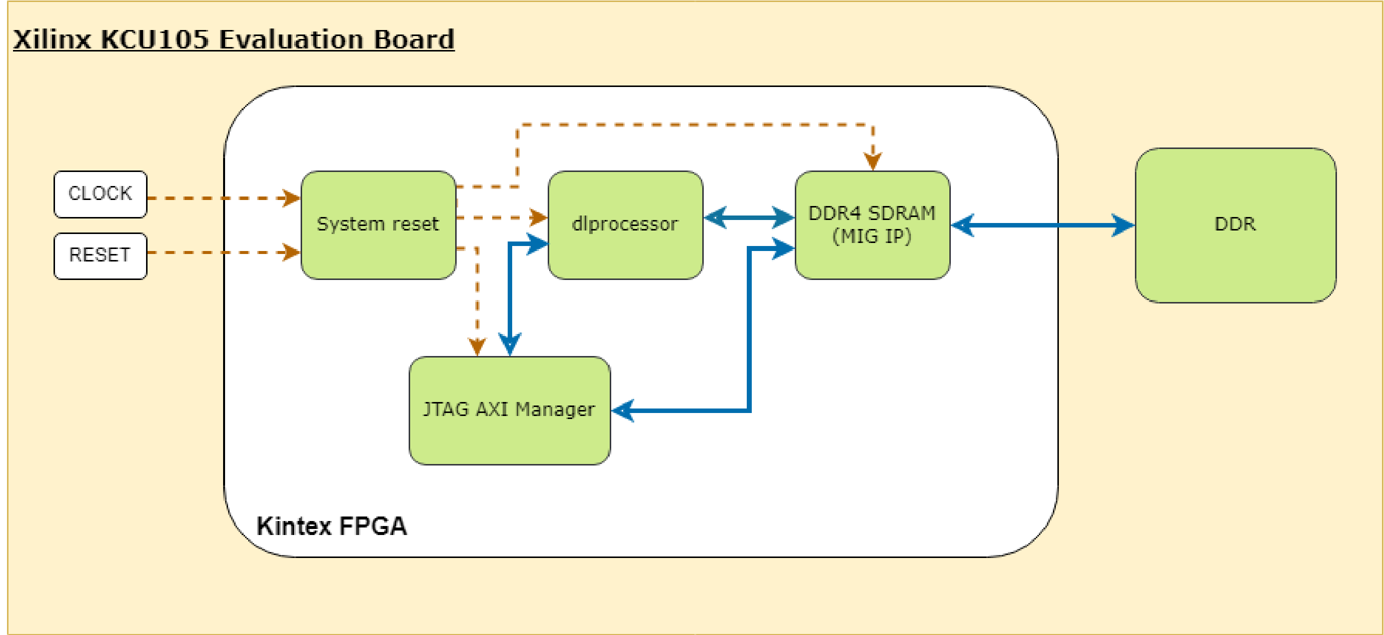
The image shows the process of deploying a network to a custom board and retrieving a prediction from the deployed network.



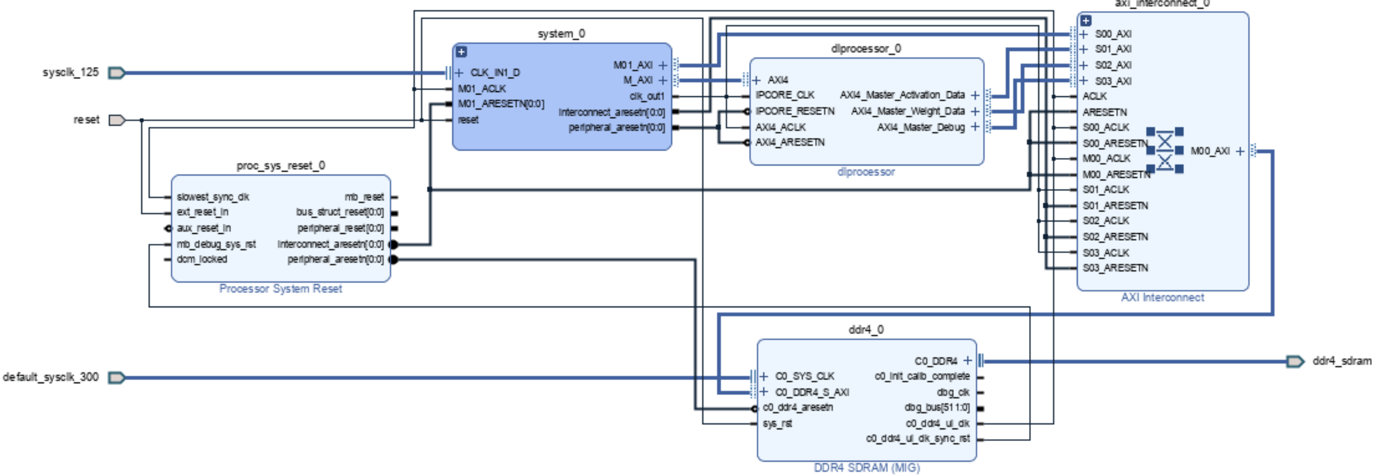
This example uses the Xilinx® Kintex® UltraScale™ KCU105 board. The board contains these blocks:

- System reset block — Used to feed the clock and reset signals to the design.
- Memory Interface Generator (MIG) IP block — Used to generate memory controllers and interfaces for Xilinx FPGAs.
- MATLAB JTAG AXI Manager block — Used by MATLAB to access onboard memory location. For more details, see “Set Up AXI Manager” (HDL Verifier).

Integrate the generated deep learning processor IP core into your reference design. For more details, see “Board and Reference Design Registration System” (HDL Coder).



This image shows the generated deep learning processor IP core `dl_processor0` integrated into the reference design.



Register Custom Board

Define the interface and attributes of a custom SoC board. To register the Xilinx® Kintex® UltraScale™ KCU105 board:

1. Create a board registration file with the name `hdlcoder_board_customization.m` and add it to the MATLAB path. The `hdlcoder_board_customization.m` function must return a second output. For more information, see “Register a Custom Board” (HDL Coder).

Set the target workflow to `DeepLearningProcessor`. For information on other target workflows supported by HDL Coder™ , see “Workflows in HDL Workflow Advisor” (HDL Coder).

```
function [boardList, workflow] = hdlcoder_board_customization
  % Board plugin registration file
  % 1. Any registration file with this name on MATLAB path will be picked up
  % 2. Registration file returns a cell array pointing to the location of
  %    the board plugins
  % 3. Board plugin must be a package folder accessible from MATLAB path,
  %    and contains a board definition file
  %
  % Copyright 2022 The MathWorks, Inc.

boardList = { ...
     'DLKCU105.plugin_board', ...
      };
workflow = hdlcoder.Workflow.DeepLearningProcessor;
end
```

2. Create the board definition file. To generate a deep learning processor, you must define the `ExternalMemorySize`. This property defines the memory size of the DDR on the target board.

```
% Copyright 2022 The MathWorks, Inc.

% Board definition of KCU105
function hB = plugin_board()

% Construct board object
hB = hdlcoder.Board;

hB.BoardName    = 'Xilinx Kintex-Ultrascale KCU105 evaluation board';

% FPGA device information
hB.FPGAVendor   = 'Xilinx';
hB.FPGAFamily   = 'KintexU';
hB.FPGADevice   = 'xcku040-ffva1156-2-e';
hB.FPGAPackage  = '';
hB.FPGASpeed    = '';

% Tool information
hB.SupportedTool = {'Xilinx Vivado'};

% FPGA JTAG chain position
hB.JTAGChainPosition = 1;

% Size of external DDR memory in bytes
hB.ExternalMemorySize = 0x80000000; % 2 GB

% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS18'});

% Custom board external I/O interface
hB.addExternalIOInterface( ...
     'InterfaceID',    'LEDs General Purpose', ...
     'InterfaceType',  'OUT', ...
     'PortName',       'GPLEDs', ...
     'PortWidth',      8, ...
     'FPGAPin',        {'AP8', 'H23', 'P20', 'P21', 'N22', 'M22', 'R23','P23'}, ...
     'IOPadConstraint', {'IOSTANDARD = LVCMOS18'});

% Custom board external I/O interface
hB.addExternalIOInterface( ...
```

```
    'InterfaceID',    'User Push Buttons', ...
    'InterfaceType',  'IN', ...
    'PortName',       'PB', ...
    'PortWidth',      1, ...
    'FPGAPin',        {'AE10'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS18'});
```

Register Custom Reference Design

Define the interface and attributes of a custom SoC reference design. To create a custom reference design:

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` that contains the list of reference design plugins associated with the board. For more information, see "Register a Custom Reference Design" (HDL Coder).

```
function [rd, boardName] = hdlcoder_ref_design_customization
% Reference design plugin registration file
% 1. The registration file with this name inside of a board plugin folder
%    will be picked up
% 2. Any registration file with this name on MATLAB path will also be picked up
% 3. The registration file returns a cell array pointing to the location of
%    the reference design plugins
% 4. The registration file also returns its associated board name
% 5. Reference design plugin must be a package folder accessible from
%    MATLAB path, and contains a reference design definition file
%
%   Copyright 2022 The MathWorks, Inc.

rd = {...
        'DLKCU105.matlab_3axi4_master_2020_1.plugin_rd', ...
  };

boardName = 'Xilinx Kintex-Ultrascale KCU105 evaluation board';

end
```

2. Create the reference design definition file. To generate a deep learning processor IP core, you must define these three AXI4 Master Interfaces:

- `AXI4 Master Activation Data`
- `AXI4 Master Weight Data`
- `AXI4 Master Debug`

```
function hRD = plugin_rd()
% Reference design definition

% Copyright 2022 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'AXI-Stream DDR Memory Access : 3-AXIM';
hRD.BoardName = 'Xilinx Kintex-Ultrascale KCU105 evaluation board';

% Tool information
hRD.SupportedToolVersion = {'2020.1','2020.2'};

% Add custom design files
```

```
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
  'CustomBlockDesignTcl', 'system_top.tcl',...
  'VivadoBoardPart',      'xilinx.com:kcu105:part0:1.0');

% Add HDL Verifier JTAG as AXI Master IP from support package
hRD.addIPRepository( ...
  'IPListFunction','hdlverifier.fpga.vivado.iplist', ...
  'NotExistMessage', 'IP Repository not found.');

% Add interfaces

% add clock interface
hRD.addClockInterface( ...
     'ClockConnection',      'system_0/clk_out1', ...
     'ResetConnection',      'system_0/peripheral_aresetn',...
     'DefaultFrequencyMHz',  125,...
     'MinFrequencyMHz',      10,...
     'MaxFrequencyMHz',      250,...
     'ClockNumber',          1,...
     'ClockModuleInstance',  'system_0/clk_wiz_0');

% add AXI4 and AXI4-Lite slave interfaces
% This slave interface is used for intracting between DDR4 and Deep Learning IP
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'system_0/M_AXI', ...
    'BaseAddress',         '0x44A00000',...
    'MasterAddressSpace',  'system_0/hdlverifier_axi_manager_0/axi4m',...
    'InterfaceType',       'AXI4');

% AXI4 Master Interface for the layer activation data with max data bit-width of 512
hRD.addAXI4MasterInterface(...
        'InterfaceID',         'AXI4 Master Activation Data', ...
        'ReadSupport',          true, ...
        'WriteSupport',         true, ...
        'MaxDataWidth',         512, ...
        'AddrWidth',            32, ...
        'InterfaceConnection', 'axi_interconnect_0/S01_AXI',...
        'TargetAddressSegments', {{'ddr4_0/C0_DDR4_MEMORY_MAP/C0_DDR4_ADDRESS_BLOCK',hex2dec('80

% AXI4 Master Interface for the layer weight data with max data bit-width of 512
hRD.addAXI4MasterInterface(...
        'InterfaceID',         'AXI4 Master Weight Data', ...
        'ReadSupport',          true, ...
        'WriteSupport',         true, ...
        'MaxDataWidth',         512, ...
        'AddrWidth',            32, ...
        'InterfaceConnection', 'axi_interconnect_0/S02_AXI',...
        'TargetAddressSegments', {{'ddr4_0/C0_DDR4_MEMORY_MAP/C0_DDR4_ADDRESS_BLOCK',hex2dec('80

% AXI4 Master Interface for the debugger with max data bit-width of 512
hRD.addAXI4MasterInterface(...
        'InterfaceID',         'AXI4 Master Debug', ...
        'ReadSupport',          true, ...
        'WriteSupport',         true, ...
        'MaxDataWidth',         512, ...
        'AddrWidth',            32, ...
        'InterfaceConnection', 'axi_interconnect_0/S03_AXI',...
        'TargetAddressSegments', {{'ddr4_0/C0_DDR4_MEMORY_MAP/C0_DDR4_ADDRESS_BLOCK',hex2dec('80
```

3. The reference design plugin file must contain information about the target interface and the deep learning processor IP core, the memory address space for the deep learning processor IP core, and a command to validate the reference design. The file also requires information on the resources consumed by the reference design. This information is used during resource estimation. Add the deep learning processor information to the reference design file:

```
% Deep learning specific properties
hRD.registerDeepLearningTargetInterface("JTAG");
hRD.registerDeepLearningMemoryAddressSpace(0x80000000, 0x80000000); % 2GB

% Resource utilization information
hRD.ResourcesUsed.LogicElements = 30500;
hRD.ResourcesUsed.DSP = 3;
hRD.ResourcesUsed.RAM = 26.5;
```

Performance Estimation

Reduce the time required to design and deploy a custom deep learning network that meets performance requirements by analyzing the layer-level latencies before deploying the network.

Estimate the performance of network for your custom board by collecting calibration data from the custom board, by:

1. Generating a calibration bitstream
2. Deploying the calibration bitstream to the target custom board
3. Retrieving the external to internal memory transaction latencies

Create a Processor Configuration object.

```
hPC = dlhdl.ProcessorConfig;
```

Specify the `TargetPlatform`. This automatically sets the `SynthesisToolChipFamily`, `SynthesisToolDeviceName`, and `ReferenceDesign` properties.

```
hPC.TargetPlatform = 'Xilinx Kintex-Ultrascale KCU105 evaluation board';
```

Set the target frequency.

```
hPC.TargetFrequency = 100;
```

This example uses a ResNet-18 pretrained network. For more details, see `resnet18`. Set the deep learning network:

```
net = resnet18;
```

To fit this design onto the target, reduce the number of parallel convolution processor kernel threads for the `conv` module to 9.

```
setModuleProperty(hPC, 'conv', 'ConvThreadNumber', 9);
```

Set the Xilinx Vivado toolpath to your design tool using the `hdlsetuptoolpath` function, then build the calibration bitstream.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
bitstreamPath = buildCalibrationBitstream(hPC);
```

Deploy the bitstream to the hardware and obtain the external- to-internal memory transaction latencies. You can use these values to get better estimates for the layer-level latencies.

```
deployCalibrationBitstream(hPC, bitstreamPath);
```

The `deployCalibrationBitstream` saves the calibration data from the hardware as a structure in the `CalibrationData` property of the `dlhdl.ProcessorConfig` object. The function also saves the calibration data as a MAT-file with the name `calibrationData.mat`. You can load this data into a new `dlhdl.ProcessorConfig` object by entering:

```
load('calibrationData.mat');
hPC.CalibrationData = calData;
```

Estimate the performance of the network for the custom processor configuration.

```
estimatePerformance(hPC, net);
```

```
              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                    34817713                  0.34818                       1           34817713              2.9
    ____data_norm_add        117907                  0.00118 
    ____data_norm            117907                  0.00118 
    ____conv1               3356175                  0.03356 
    ____pool1                635032                  0.00635 
    ____res2a_branch2a      1850730                  0.01851 
    ____res2a_branch2b      1850730                  0.01851 
    ____res2a                162115                  0.00162 
    ____res2b_branch2a      1850730                  0.01851 
    ____res2b_branch2b      1850730                  0.01851 
    ____res2b                162115                  0.00162 
    ____res3a_branch1       1012642                  0.01013 
    ____res3a_branch2a      1011307                  0.01011 
    ____res3a_branch2b      1599289                  0.01599 
    ____res3a                 79207                  0.00079 
    ____res3b_branch2a      1599289                  0.01599 
    ____res3b_branch2b      1599289                  0.01599 
    ____res3b                 79207                  0.00079 
    ____res4a_branch1        823109                  0.00823 
    ____res4a_branch2a       825037                  0.00825 
    ____res4a_branch2b      1552695                  0.01553 
    ____res4a                 39605                  0.00040 
    ____res4b_branch2a      1552695                  0.01553 
    ____res4b_branch2b      1552695                  0.01553 
    ____res4b                 39605                  0.00040 
    ____res5a_branch1       1182129                  0.01182 
    ____res5a_branch2a      1186237                  0.01186 
    ____res5a_branch2b      2284751                  0.02285 
    ____res5a                 19703                  0.00020 
    ____res5b_branch2a      2284751                  0.02285 
    ____res5b_branch2b      2284751                  0.02285 
    ____res5b                 19703                  0.00020 
    ____pool5                 47376                  0.00047 
    ____fc1000               188470                  0.00188 
 * The clock frequency of the DL processor is: 100MHz
```

Resource Estimation

Verify that the generated bistream and network fit on your target custom board, by using `estimateResources` to estimate the resource utilization. To learn how to estimate the resource

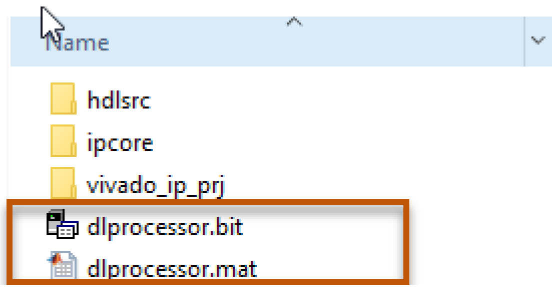
utilization for your custom boards, see “Estimate Resource Utilization for Custom Board and Reference Design” on page 10-191.

Generate Custom Bitstream for Custom Processor Configuration

Generate a bitstream for the custom processor configuration `hPC`.

```
dlhdl.buildProcessor(hPC);
```

Locate the bitstream file and associated MAT file at `cwd\dlhdl_prj\`, where `cwd` is your current working folder. The name of the bitstream file is `dlprocessor.bit`. The name of the MAT file is `dlprocessor.mat`. To use the generated bitstream for the supported Xilinx boards, copy the `dlprocessor.bit` and `dlprocessor.mat` files to the current working folder.



Deploy the Custom Bitstream and Run Predictions on the Network

After you generate the bitstream, deploy the network and run the predictions on the network. For more information, refer to the “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2 page. For an example on prototyping, see “Bicyclist and Pedestrian Classification by Using FPGA” on page 10-53.

Create Target Object

Create a target object with the vendor name of the target device. Specify the interface to connect the target device to the host using the `Interface` name-value pair. This example connects to the target using the JTAG interface.

```
hT = dlhdl.Target('Xilinx', 'Interface', 'JTAG')
```

Create Workflow Object for ResNet-18 Network

Create an object of the `dlhdl.Workflow` class. Specify the network, the bitstream name, and the target object.

```
hW = dlhdl.Workflow('Network', net, 'Bitstream', 'dlprocessor.bit', 'Target', hT);
```

Compile the Network

Run the `compile` function of the `dlhdl.Workflow` object.

```
compile(hW)
```

Deploy the Bitstream to the FPGA

To deploy the network on the Xilinx KCU105 Kintex hardware, run the `deploy` function of the `dlhdl.Workflow` object.

```
deploy(hW)
```

```
### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 07-Jun-2022 17:44:19
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 07-Jun-2022 17:44:27
```

Run Prediction for the Network

Load the sample image.

```
img = imread('sampleImage1.png');
imshow(img);
```

Run a prediction on the image. The `result` output argument contains the output of the layer preceding the `ClassificationOutputLayer` and `speed` contains the profiler table.

```
[result, speed] = predict(hW, img, 'Profile', 'on');
```

```
              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                    34197670                  0.34198                     1            34200121            2.9
    data_norm_add            157975                  0.00158
    data_norm                157887                  0.00158
    conv1                   2984413                  0.02984
    pool1                    516698                  0.00517
    res2a_branch2a          1734527                  0.01735
    res2a_branch2b          1734368                  0.01734
    res2a                    224302                  0.00224
    res2b_branch2a          1734604                  0.01735
    res2b_branch2b          1734348                  0.01734
    res2b                    224312                  0.00224
    res3a_branch1            979411                  0.00979
    res3a_branch2a           974770                  0.00975
    res3a_branch2b          1561164                  0.01561
    res3a                    109624                  0.00110
    res3b_branch2a          1561119                  0.01561
    res3b_branch2b          1561136                  0.01561
    res3b                    109624                  0.00110
    res4a_branch1            817239                  0.00817
    res4a_branch2a           819776                  0.00820
    res4a_branch2b          1555224                  0.01555
    res4a                     54841                  0.00055
    res4b_branch2a          1555469                  0.01555
    res4b_branch2b          1555284                  0.01555
    res4b                     54831                  0.00055
    res5a_branch1           1197847                  0.01198
    res5a_branch2a          1199724                  0.01200
    res5a_branch2b          2331422                  0.02331
    res5a                     27324                  0.00027
    res5b_branch2a          2331507                  0.02332
    res5b_branch2b          2331802                  0.02332
    res5b                     27274                  0.00027
    pool5                     81101                  0.00081
    fc1000                   196536                  0.00197
 * The clock frequency of the DL processor is: 100MHz
```

Get the output class from the prediction.

```
[value,idx] = max(result);
classNames = net.Layers(end).Classes;
classNames(idx)
```

```
ans =

  categorical

     monitor
```

See Also

hdlcoder.ReferenceDesign | registerDeepLearningMemoryAddressSpace | registerDeepLearningTargetInterface | validateReferenceDesignForDeepLearning

More About

- “Deep Learning Processor IP Core” on page 12-5
- “Use the Compiler Output for System Integration” on page 12-6
- “External Memory Data Format” on page 12-9
- “Interface with the Deep Learning Processor IP Core” on page 12-17
- “Register a Custom Board” (HDL Coder)
- “Register a Custom Reference Design” (HDL Coder)

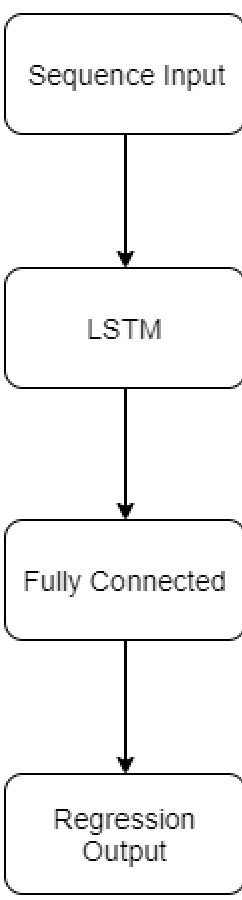
Deep Learning HDL Toolbox Support for LSTM Networks

- “Support for Long Short-Term Memory Networks” on page 13-2
- “How Deep Learning HDL Toolbox Compiles the LSTM Layer” on page 13-5
- “How Deep Learning HDL Toolbox Compiles the GRU Layer” on page 13-9

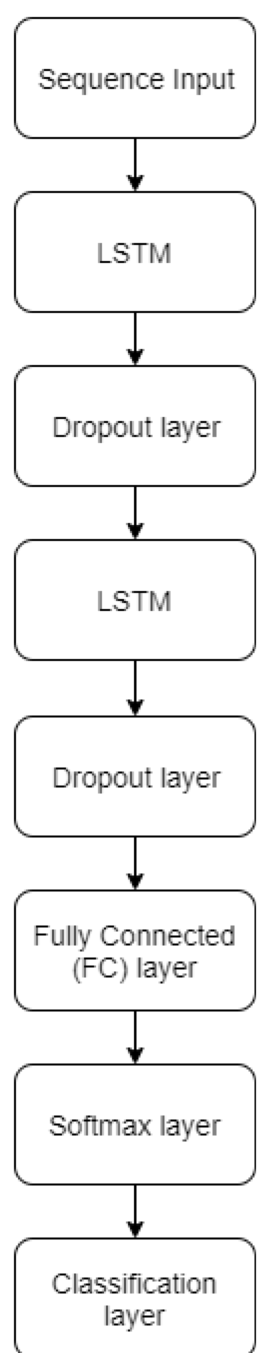
Support for Long Short-Term Memory Networks

A long short-term memory (LSTM) network is a type of recurrent neural network (RNN) that can learn long-term dependencies between time steps of sequence data. Deep Learning HDL Toolbox supports these LSTM network architectures:

- Single LSTM layer networks — A single LSTM layer network consists of only one LSTM layer. This diagram illustrates the architecture of a single LSTM layer network for sequence regression. The network starts with a sequence input layer followed by an LSTM layer. The network ends with a fully connected layer and a regression output layer.



- Stacked LSTM layer networks — A stacked LSTM layer network consists of multiple LSTM layers. In a stacked LSTM layer network, the preceding LSTM layer provides a sequence output to the following LSTM layer. This diagram illustrates the architecture of a stacked LSTM layer network used for classification. The network starts with a sequence input layer followed by an LSTM layer, dropout layer, second LSTM layer, and a second dropout layer. To predict class labels, the network ends with a fully connected layer, a softmax layer, and a classification output layer.



Deep Learning HDL Toolbox does not support bidirectional LSTM layers. For a list of supported layers, see “Supported Layers” on page 7-16.

Prediction and Forecasting

To make predictions on new data in an LSTM network, use `predict`. See “Run Sequence-to-Sequence Classification on FPGAs by Using Deep Learning HDL Toolbox” on page 10-246.

LSTM networks can remember the state of the network between predictions. The network state is useful when you do not have the complete time series in advance, or if you want to make multiple predictions on a long time series. To predict parts of a time series and update the network state, use `predictAndUpdateState`. To reset the network state between predictions, use `resetState`. To learn about forecasting future time steps of a sequence, see “Run Sequence Forecasting on FPGA by Using Deep Learning HDL Toolbox” on page 10-262.

See Also

More About

- “Long Short-Term Memory Neural Networks”
- “Prototype Deep Learning Networks on FPGA and SoC Devices” on page 5-2
- “How Deep Learning HDL Toolbox Compiles the LSTM Layer” on page 13-5

How Deep Learning HDL Toolbox Compiles the LSTM Layer

An LSTM is a type of recurrent neural network (RNN) that can learn long-term dependencies between time steps of sequence data. When you compile LSTM layers, Deep Learning HDL Toolbox splits the LSTM layer into components, generates instructions and memory offsets for those components. Integrate a deep learning processor IP core with LSTM layers into your reference design by:

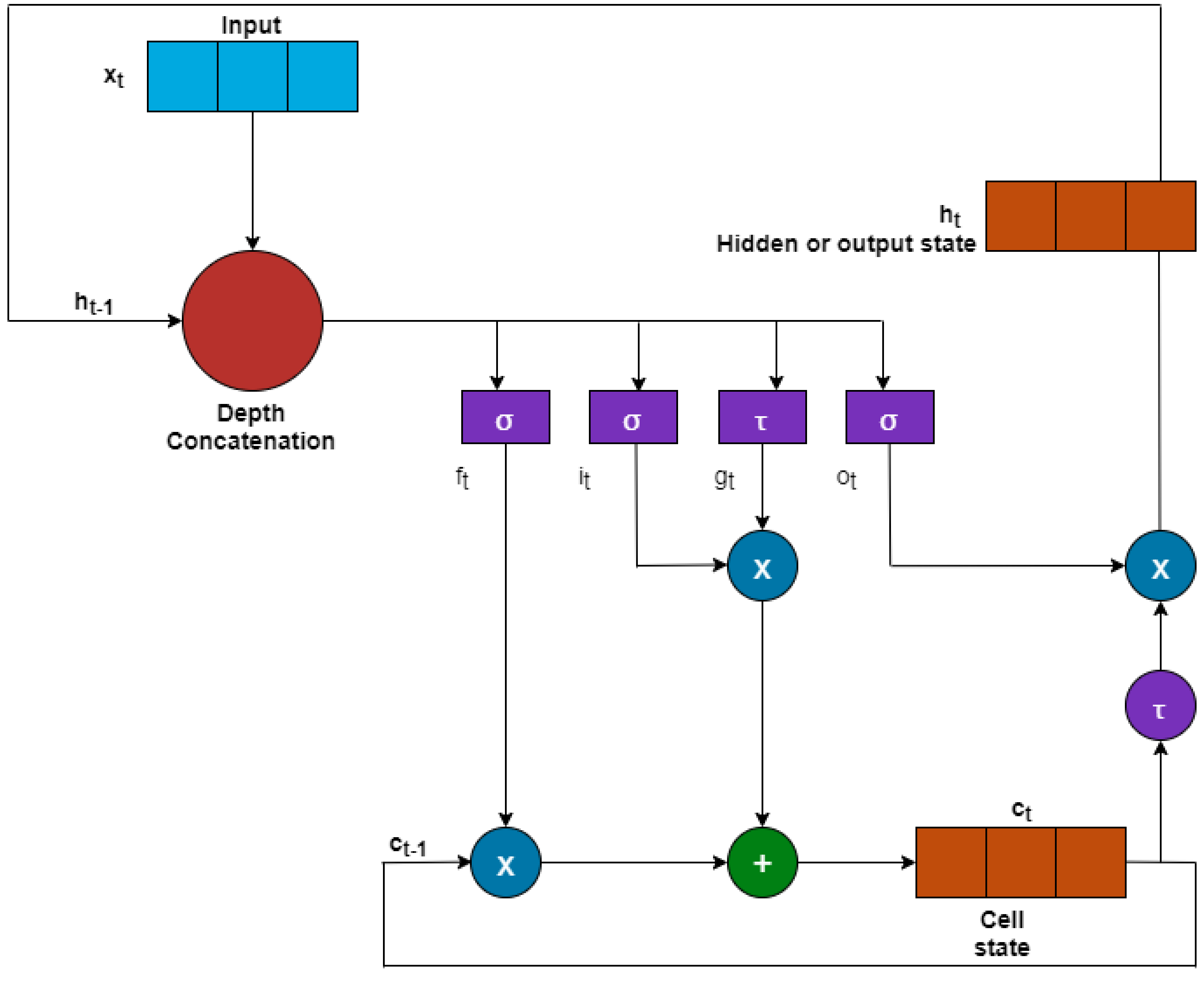
- Learning about the `compile` function generated LSTM layer components and how those components are optimized.
- Identifying the external memory addresses that store the generated LSTM layer components weights, biases, and instructions.

LSTM Layer Architecture

The LSTM layer uses a gating mechanism that controls the memorizing process. You can store, write, or read information in LSTMs by using gates that open and close. An LSTM layer consists of these components:

- Forget gate — The forget gate,`f` decides which information to remember and which information to forget.
- Input gate — The input gate, `i` updates the cell state using information from the input current state `x` and the previous hidden state `h`.
- Cell state — The cell state stores information from the new layer state based on the previous cell state, `c`. The current cell state is, `g`.
- Output gate — The output gate, `o` determines the value of the next hidden state, `h`.

This image shows the components of an LSTM layer:



Compiler Interpretation

The `compile` method of the `dlhdl.Workflow` object translates the:

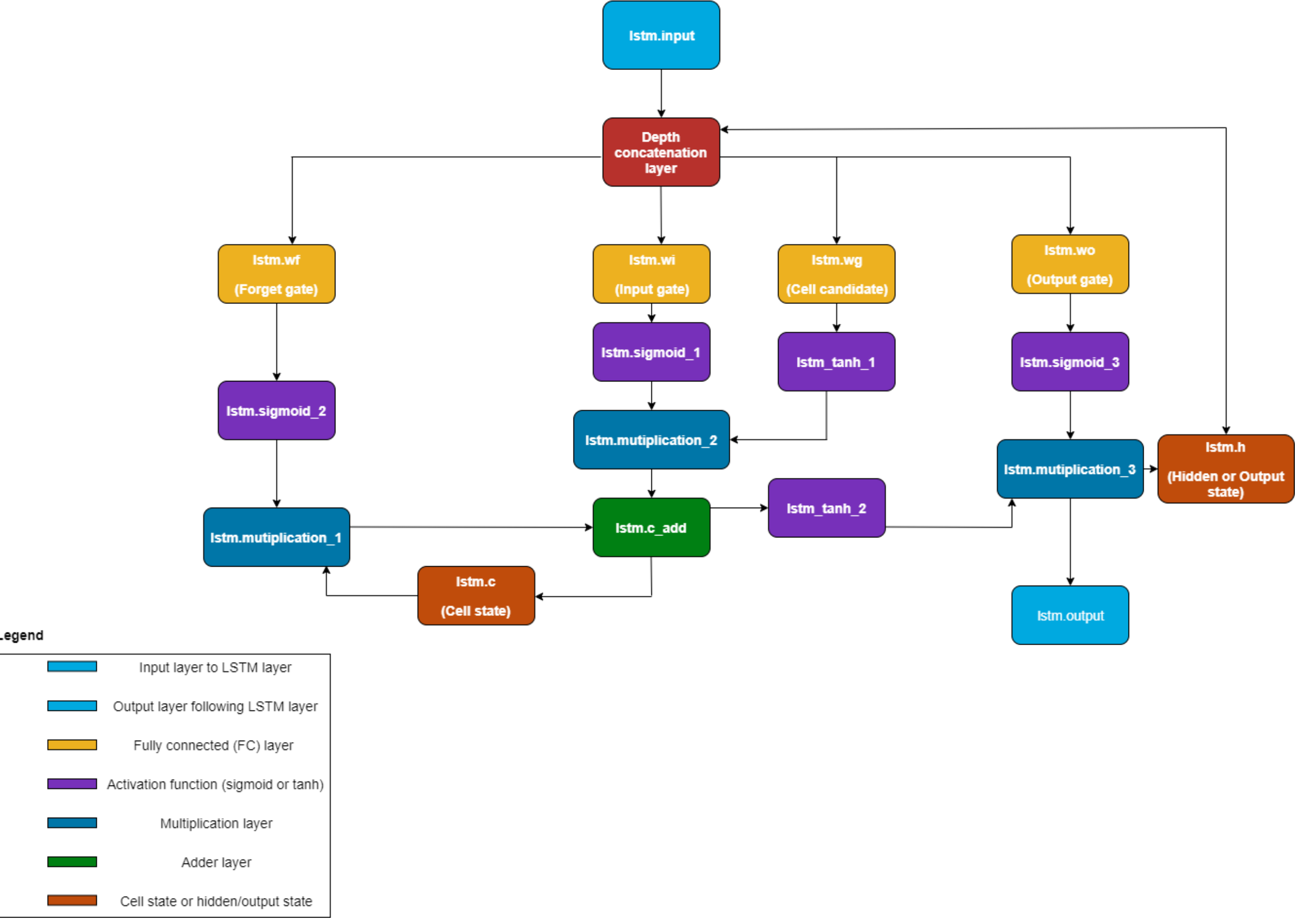
- Forget gate into `lstm.wf`
- Input gate into `lstm.wi`
- Cell candidate into `lstm.wg`
- Output gate into `lstm.wo`

The `compile` method

- Inserts a depth concatenation layer between the layer preceding the LSTM layer and the gates of the LSTM layer.
- Generates sigmoid, hyperbolic tangent, multiplication, and addition layers to replace the mathematical operations of the LSTM layer.

When the network has multiple stacked LSTM layers, the `compile` method uses the LSTM layer name when generating the translated instructions. For example, if the network has three stacked LSTM layers named `lstm_1`, `lstm_2`, and `lstm_3`, the `compile` method output is `lstm_1.wi`, `lstm_1.wo`, `lstm_1.wg`, `lstm_1.wf`, `lstm_2.wi`, and so on. The compiler schedules the different components of the LSTM layer such as fully connected layers, sigmoid blocks, tanh blocks, and so on, into different kernels within the deep learning processor architecture.

This image shows the graphical view of the `compile` method transformation of the LSTM layer:



To see the output of the `compile` method for an LSTM network, see “Run Sequence-to-Sequence Classification on FPGAs by Using Deep Learning HDL Toolbox”.

See Also

`dlhdl.Workflow` | `compile`

More About

- “Use the Compiler Output for System Integration” on page 12-6
- “Support for Long Short-Term Memory Networks” on page 13-2

- “Interface with the Deep Learning Processor IP Core” on page 12-17

How Deep Learning HDL Toolbox Compiles the GRU Layer

To manually deploy a gated recurrent unit (GRU) layer network to your target board, learn how the `compile` method of the `dlhdl.Workflow` object interprets the GRU layer in a network. When you compile GRU layers, Deep Learning HDL Toolbox splits the GRU layer into components, generates instructions and memory offsets for those components.

The `compile` method of the `dlhdl.Workflow` translates the:

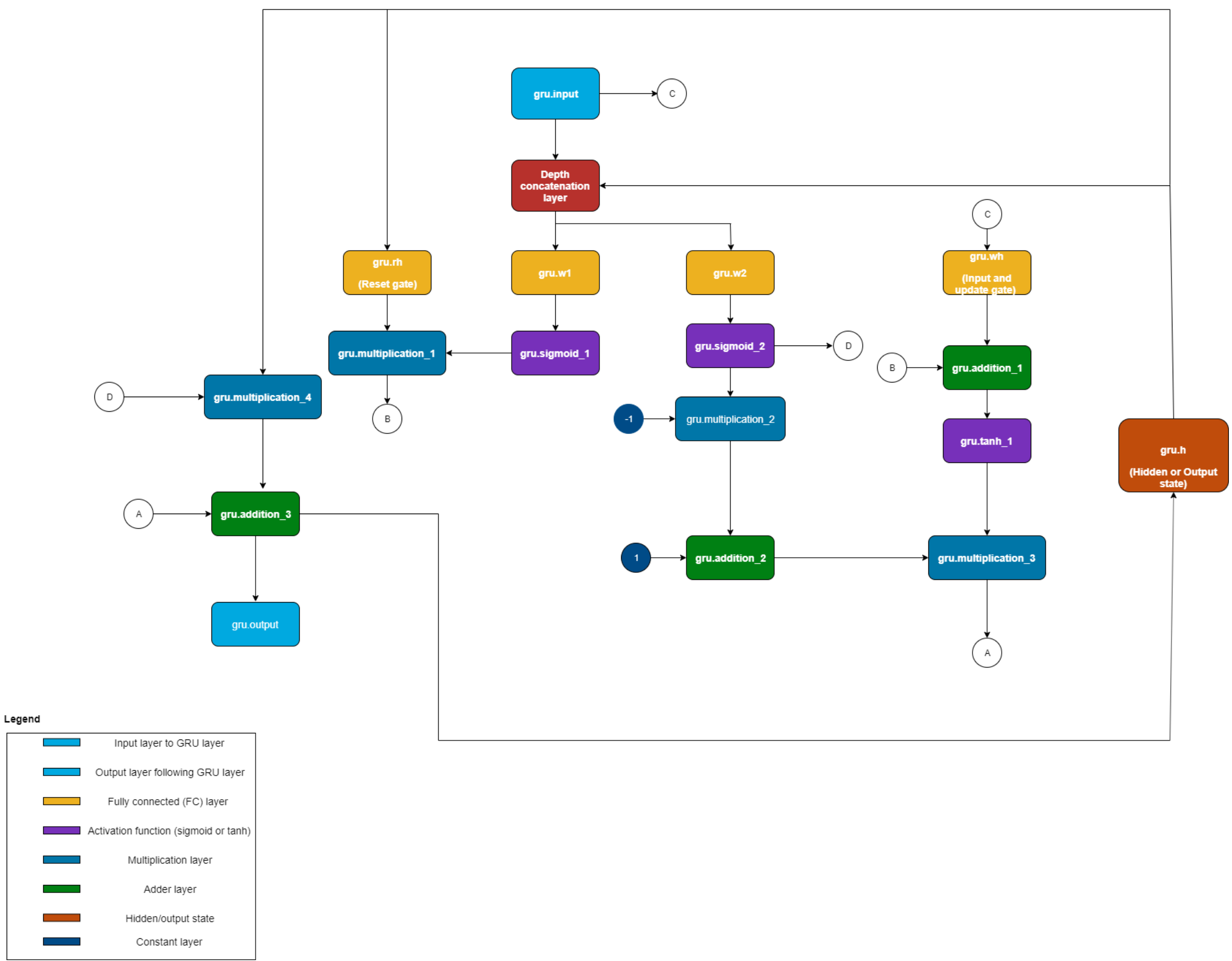
- Reset gate into `gru.rh`
- Input and update gate into `gru.wh`

Then, the `compile` method :

- Inserts a depth concatenation layer between the layer preceding the GRU layer and the gates of the GRU layer.
- Generates sigmoid, hyperbolic tangent, multiplication, and addition layers to replace the mathematical operations of the GRU layer.

When the network has multiple stacked GRU layers, the `compile` method uses the GRU layer name when generating the translated instructions. For example, if the network has three stacked GRU layers named `gru_1`, `gru_2`, and `gru_3`, the output of the `compile` method is `gru_1.wh`, `gru_1.rh`, `gru_2.wh`, `gru_2.rh`, and so on. The compiler schedules the different components of the GRU layer, such as fully connected layers, sigmoid blocks, tanh blocks, and so on, into different kernels in the deep learning processor architecture.

This image shows how the `compile` method translates the GRU layer:



To see the output of the `compile` method for a GRU layer network, see "Run Sequence Forecasting Using a GRU Layer on an FPGA".

See Also

`dlhdl.Workflow` | `compile`

More About

- "Use the Compiler Output for System Integration" on page 12-6
- "Support for Long Short-Term Memory Networks" on page 13-2
- "Interface with the Deep Learning Processor IP Core" on page 12-17