

Deep Learning HDL Toolbox™

User's Guide



MATLAB®

R2022a



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— 2022 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)

What is Deep Learning?

1

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

Deep Learning Processor

2

Deep Learning Processor Architecture	2-2
DDR External Memory	2-2
Generic Convolution Processor	2-2
Activation Normalization	2-3
Conv Controller (Scheduling)	2-3
Generic FC Processor	2-3
FC Controller (Scheduling)	2-3
Deep Learning Processor Applications	2-3

Applications and Examples

3

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

Deep Learning on FPGA Overview

4

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

Prototype Deep Learning Networks on FPGA and SoCs Workflow	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

Fast MATLAB to FPGA Connection Using LIBIIO/Ethernet

6

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

Networks and Layers

7

Supported Networks, Layers, Boards, and Tools	7-2
Supported Pretrained Networks	7-2
Supported Layers	7-10
Supported Boards	7-25
Third-Party Synthesis Tools and Version Support	7-25
Image Input Layer Normalization Hardware Implementation	7-26

Custom Processor Configuration Workflow

8

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	8-16
Generate Custom Bitstream to Meet Custom Deep Learning Network Requirements	8-18
Create Deep Learning Processor Configuration for Custom Layers	8-25
Deploy Custom Layer Networks	8-25
Create a Deep learning Processor Configuration	8-25
Create Custom Layer MATLAB Function	8-26
Create Custom Layer Simulink Function	8-27
Register Custom Layer and Model	8-27
Generate Verification Model for Custom Layer	8-28
Simulate and Validate Custom Layer Model	8-30
Generate Custom Bitstream	8-31
Deploy and Predict Custom Layer Network on Hardware	8-31
Custom Layer Registration File	8-31
Register, Validate, and Deploy Custom Hyperbolic Tangent Layer Network to FPGA	8-34

Custom Processor Code Generation Workflow

9

Generate Custom Bitstream	9-2
Intel Bitstream Resource Utilization	9-2
Xilinx Bitstream Resource Utilization	9-3
Generate Custom Processor IP	9-4

Featured Examples

10

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	10-17
Image Category Classification by Using Deep Learning	10-21
Defect Detection	10-30
Profile Network for Performance Improvement	10-47
Bicyclist and Pedestrian Classification by Using FPGA	10-51
Visualize Activations of a Deep Learning Network by Using LogoNet .	10-56
Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core	10-62
Run a Deep Learning Network on FPGA with Live Camera Input	10-67
Running Convolution-Only Networks by Using FPGA Deployment	10-77
Accelerate Prototyping Workflow for Large Networks by Using Ethernet	10-82
Create Series Network for Quantization	10-90
Custom Deep Learning Processor Generation to Meet Performance Requirements	10-94
Deploy Quantized Network Example	10-98
Quantize Network for FPGA Deployment	10-110
Evaluate Performance of Deep Learning Network on Custom Processor Configuration	10-117
Customize Bitstream Configuration to Meet Resource Use Requirements	10-123
Vehicle Detection Using DAG Network Based YOLO v2 Deployed to FPGA	10-128
Customize Bitstream Configuration to Meet Resource Use Requirements	10-137
Image Classification Using DAG Network Deployed to FPGA	10-142
Classify Images on an FPGA Using a Quantized DAG Network	10-150
Classify ECG Signals Using DAG Network Deployed To FPGA	10-163
Prototype and Verify Deep Learning Networks Without Target Hardware	10-177

Classify Images on FPGA by Using Quantized GoogLeNet Network . . .	10-184
Estimate Resource Utilization for Custom Board and Reference Design	10-197
Speech Command Recognition by Using FPGA	10-200
Modulation Classification by Using FPGA	10-211
Deploy Simple Adder Network by using MATLAB Deployment Script and Deployment Instructions File	10-223
Human Pose Estimation by Using Segmentation DAG Network Deployed to FPGA	10-231
Semantic Segmentation of Multispectral Images by Using Quantized U- Net on FPGA	10-237

Deep Learning Quantization

11

Quantization of Deep Neural Networks	11-2
Precision and Range	11-2
Histograms of Dynamic Ranges	11-2
Quantization Workflow Prerequisites	11-10
Calibration	11-13
Workflow	11-13
Validation	11-15
Workflow	11-15
Code Generation and Deployment	11-18

Deep Learning Processor IP Core User Guide

12

Deep Learning Processor IP Core	12-2
Use the Compiler Output for System Integration	12-3
External Memory Address Map	12-3
Compiler Optimizations	12-3
Leg Level Compilations	12-4
External Memory Data Format	12-6
Key Terminology	12-6
Convolution Module External Memory Data Format	12-6

Fully Connected Module External Memory Data Format	12-7
Deep Learning Processor Register Map	12-9
Interface with the Deep Learning Processor IP Core	12-11
Create Deep Learning Processor Configuration	12-11
Select Data Processing Mode	12-11
Design Processing Mode Interface Signals	12-12
Design Batch Processing Mode Interface	12-16
Design Streaming Mode Interface	12-19
Access Data from DDR	12-24
Define Custom Board and Reference Design for Deep Learning Processor	
IP Core Workflow	12-25

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

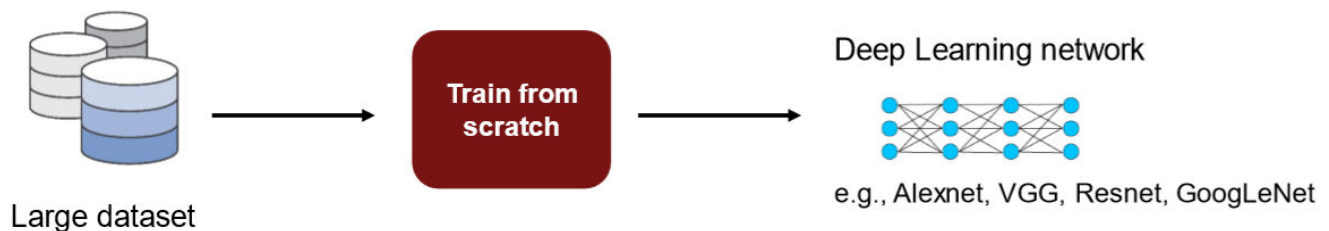
Deep learning is a branch of machine learning that teaches computers to do what comes naturally to humans: learn from experience. The learning algorithms use computational methods to “learn” information directly from data without relying on a predetermined equation as model. Deep learning uses neural 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

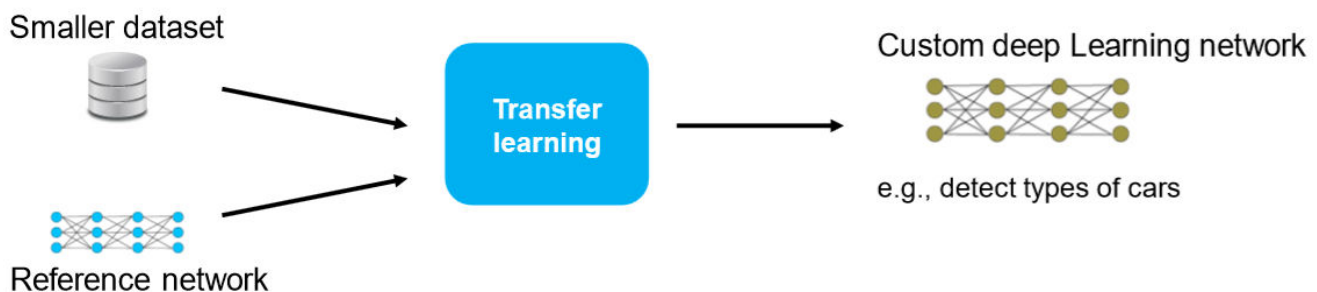
Training a deep learning neural network from scratch requires a large amount of labeled data. To create the network architecture by using Neural Network Toolbox™, you can use the built-in layers, define your own layers, or import layers from Caffe models. The neural network is then trained by using the 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

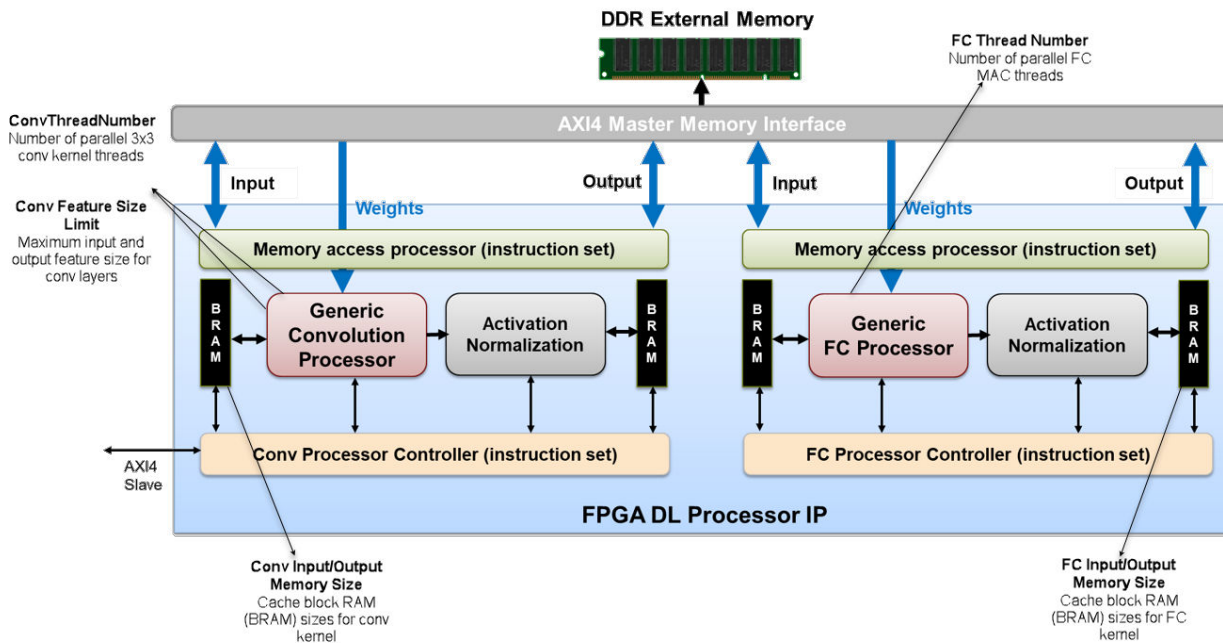
Layers in deep learning networks are trained for extracting features from the input data. This approach uses the network as a feature extractor. The features extracted after the training process can be put into various machine learning models such as Support Vector Machines (SVM).

Deep Learning Processor

Deep Learning Processor Architecture

The software provides a generic deep learning processor IP core that is target-independent and can be deployed to any custom platform that you specify. The processor can be reused and shared to accommodate deep neural networks that have various layer sizes and parameters. Use this processor to rapidly prototype deep neural networks from MATLAB, and then deploy the network to FPGAs.

This figure shows the deep learning processor architecture.



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

DDR External Memory

You can store the input images, the weights, and the output images in the external DDR memory. The processor consists of four AXI4 Master interfaces that communicate with the external memory. Using one of the AXI4 Master interfaces, you can load the input images onto the Block RAM (BRAM). The Block RAM provides the activations to the Generic Convolution Processor.

Generic Convolution Processor

The Generic Convolution Processor performs the equivalent operation of one convolution layer. Using another AXI4 Master interface, the weights for the convolution operation are provided to the Generic Convolution Processor. The Generic Convolution Processor then performs the convolution operation on the input image and provides the activations for the Activation Normalization. The processor is generic because it can support tensors and shapes of various sizes.

Activation Normalization

Based on the neural network that you provide, the **Activation Normalization** module serves the purpose of adding the ReLU nonlinearity, a maxpool layer, or performs Local Response Normalization (LRN). You see that the processor has two **Activation Normalization** units. One unit follows the **Generic Convolution Processor**. The other unit follows the **Generic FC Processor**.

Conv Controller (Scheduling)

Depending on the number of convolution layers that you have in your pretrained network, the **Conv Controller (Scheduling)** acts as ping-pong buffers. The **Generic Convolution Processor** and **Activation Normalization** can process one layer at a time. To process the next layer, the **Conv Controller (Scheduling)** moves back to the BRAM and then performs the convolution and activation normalization operations for all convolution layers in the network.

Generic FC Processor

The **Generic FC Processor** performs the equivalent operation of one fully-connected layer (FC). Using another AXI4 Master interface, the weights for the fully-connected layer are provided to the **Generic FC Processor**. The **Generic FC Processor** then performs the fully-connected layer operation on the input image and provides the activations for the **Activation Normalization** module. This processor is also generic because it can support tensors and shapes of various sizes.

FC Controller (Scheduling)

The **FC Controller (Scheduling)** works similar to the **Conv Controller (Scheduling)**. The **FC Controller (Scheduling)** coordinates with the FIFO to act as ping-pong buffers for performing the fully-connected layer operation and **Activation Normalization** depending on the number of FC layers, and ReLU, maxpool, or LRN features that you have in your neural network. After the **Generic FC Processor** and **Activation Normalization** modules process all the frames in the image, the predictions or scores are transmitted through the AXI4 Master interface and stored in the external DDR memory.

Deep Learning Processor Applications

One application of the custom deep learning processor IP core is the MATLAB controlled deep learning processor. To create this processor, integrate the deep learning processor IP with the HDL Verifier™ MATLAB as AXI Master IP by using the AXI4 slave interface. Through a JTAG or PCI express interface, you can import various pretrained neural networks from MATLAB, execute the operations specified by the network in the deep learning processor IP, and return the classification results to MATLAB.

For more information, see “MATLAB Controlled Deep Learning Processor” on page 3-2.

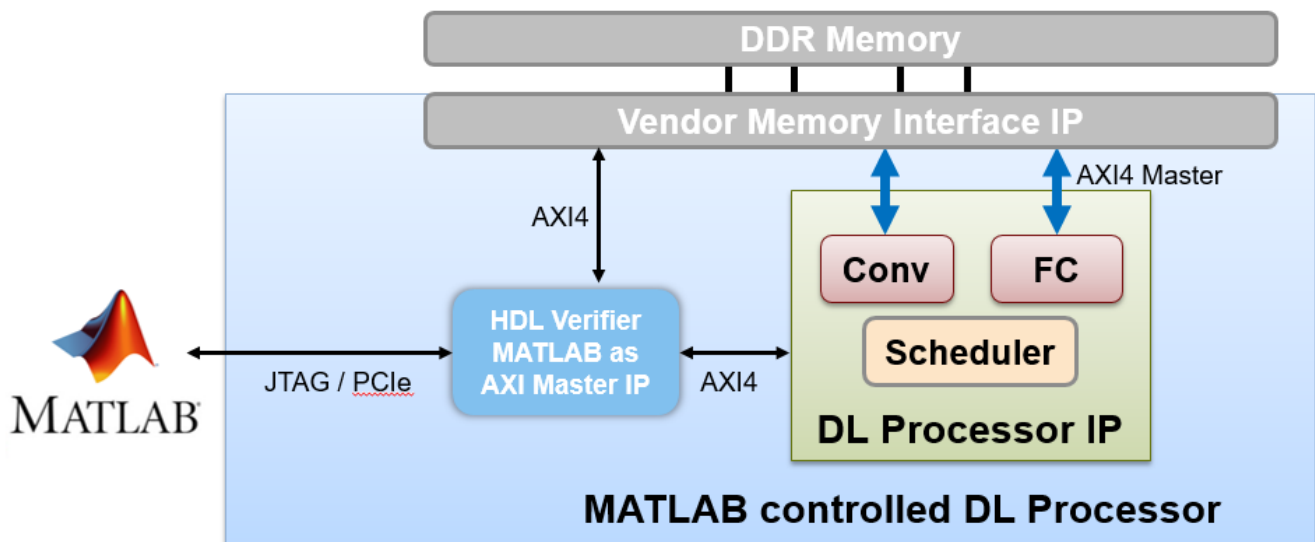
Applications and Examples

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 “Deep Learning Processor Applications” on page 2-3 .
- MATLAB as AXI Master IP, see “Set Up for 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.



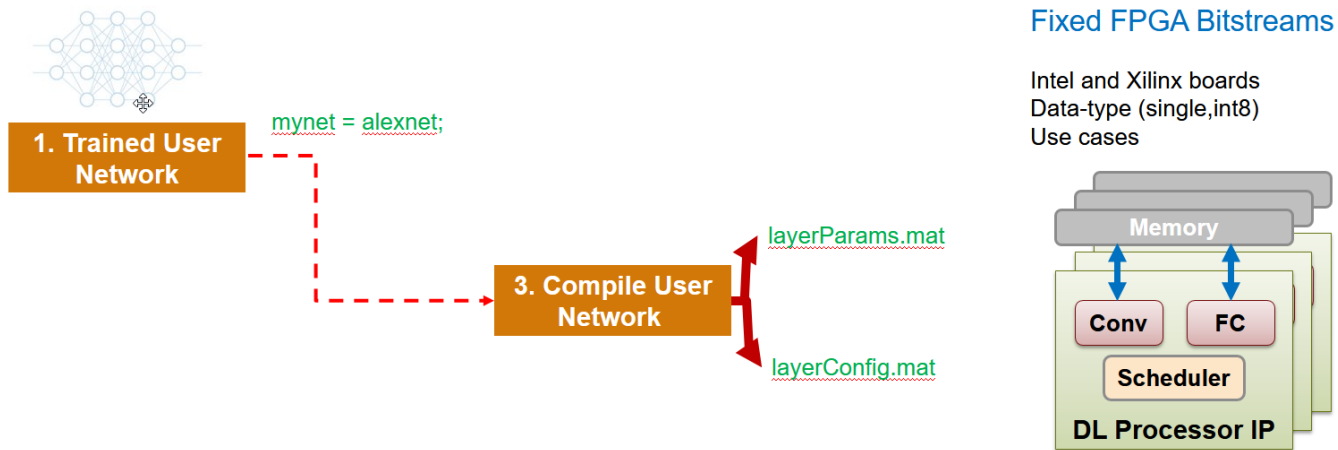
To integrate the generic deep learning processor IP with the MATLAB as AXI Master, use the AXI4 Slave interface of the deep learning processor IP core. By using a JTAG or PCI express interface, the IP responds to read or write commands from MATLAB. Therefore, you can use the MATLAB controlled deep learning 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. Following example illustrate how to deploy the pretrained series network, AlexNet, to an Intel® Arria® 10 SoC development kit.

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.

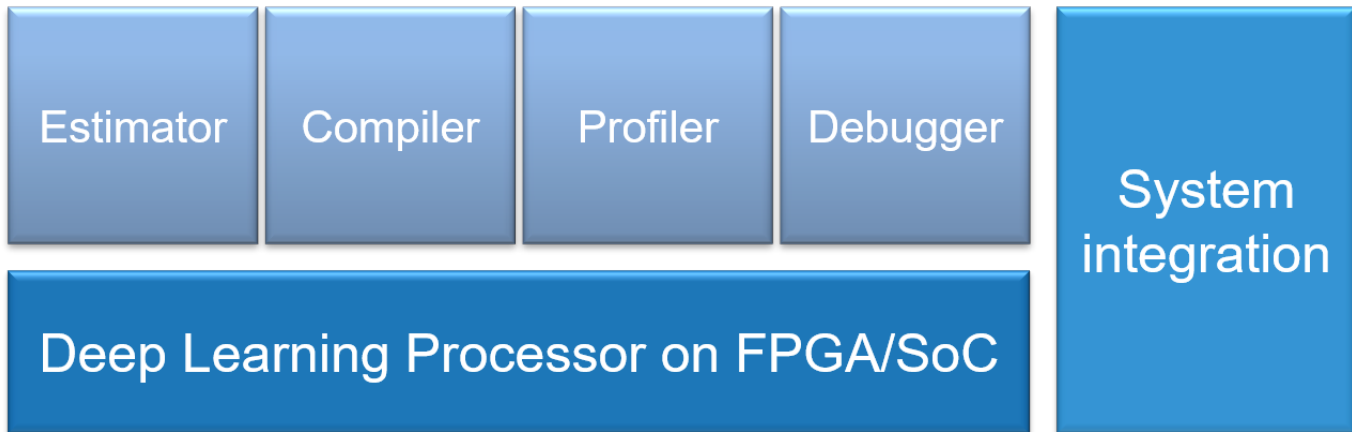
See Also

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

Deep Learning on FPGA Solution

The deep learning on field programmable 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 SoCs Workflow" 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" on page 11-2
Compare the accuracy of your quantized pretrained series networks against your single data type pretrained series network.	"Validation" on page 11-15
Run a quantized pretrained series network on your target FPGA board.	"Code Generation and Deployment" on page 11-18

Workflow and APIS

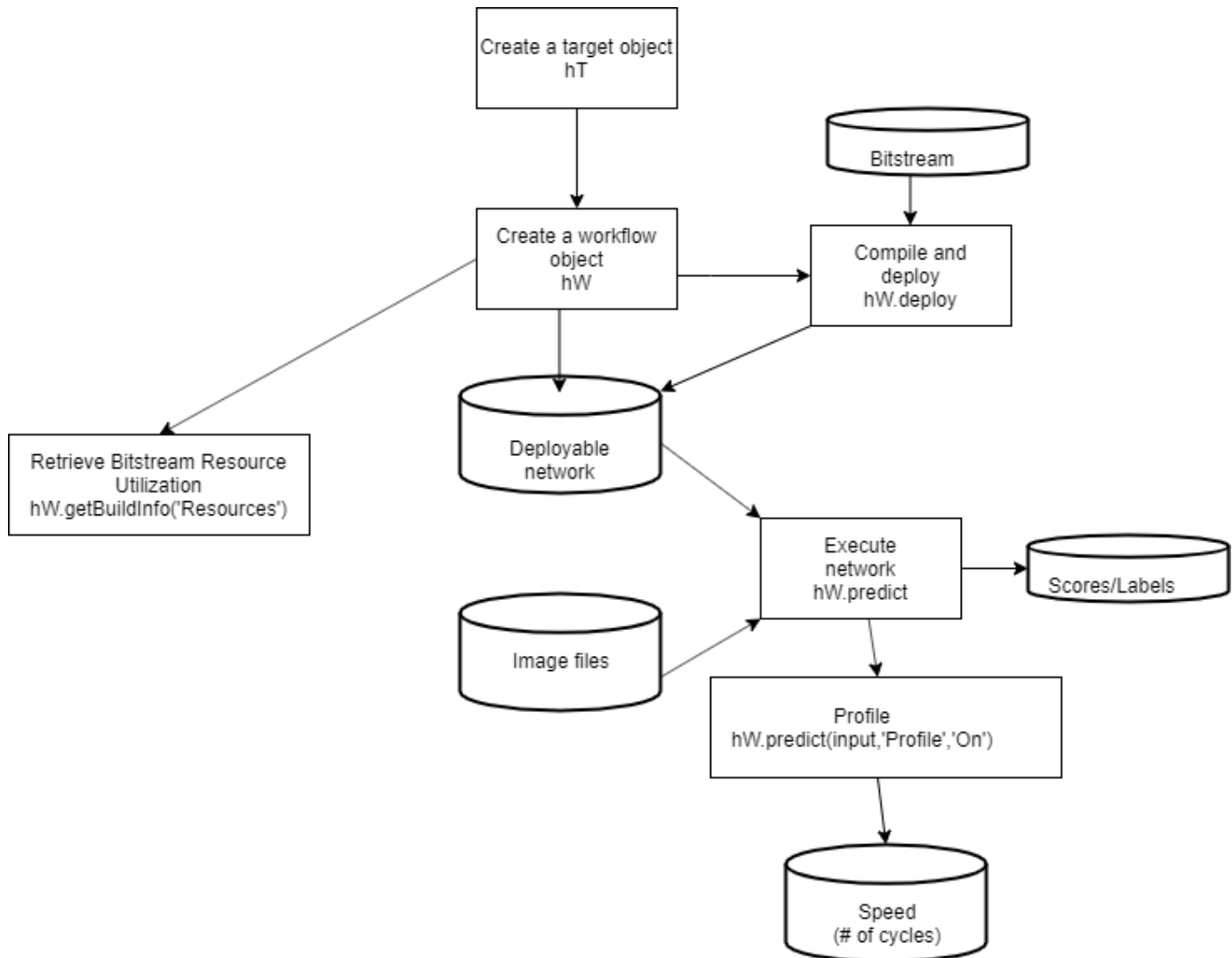
- “Prototype Deep Learning Networks on FPGA and SoCs Workflow” on page 5-2
- “Profile Inference Run” on page 5-4
- “Multiple Frame Support” on page 5-7
- “Initialize Deployed Deep Learning Processor Without Using a MATLAB Connection” on page 5-9

Prototype Deep Learning Networks on FPGA and SoCs Workflow

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, “Predict Outcome and Profile Results”.

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 $\text{LastFrameLatency(cycles)} / \text{Clock Frequency}$. 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 $(\text{FramesNum} * \text{Clock Frequency}) / \text{Total Latency}$. 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 SoCs Workflow” on page 5-2

- “Profile Network for Performance Improvement” on page 10-47

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 12-3.

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.



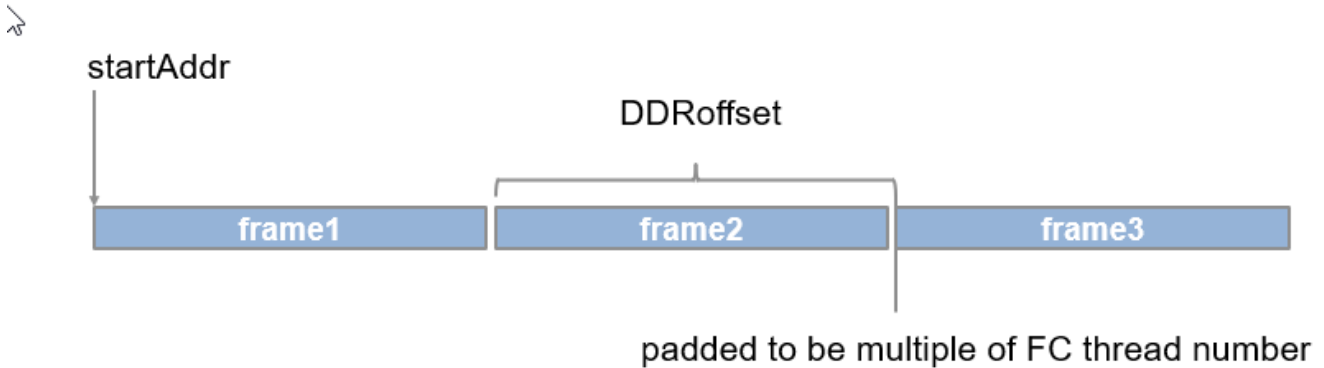
Output DDR Format

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 12-3.

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 Register Map” on page 12-9.

See Also

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

More About

- “Prototype Deep Learning Networks on FPGA and SoCs Workflow” 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:mprd0.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:

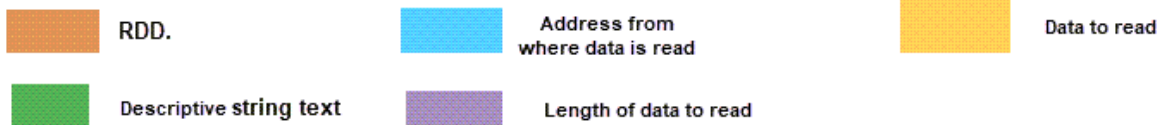
- 'RDD. ', a null terminated string indicating start of the read command.
- 'Text description. ', a null terminated string that indicated the address from where the data is read and length of the data to be read.
- Hexadecimal representation of the register address from where data is read. This data 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 ; .....ò.....
00003a80h: 08 20 00 01 00 8C 00 00 00 F2 00 00 00 00 00 08 ; .....ò.....
00003a90h: 08 20 00 01 00 8C 00 00 00 F2 00 00 00 00 00 08 ; .....ò.....
00003aa0h: 08 20 00 01 00 8C 00 00 00 F2 00 00 00 84 00 00 ; .....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 ; .....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: 36.0x81800000x00000024x00000084.....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 10-223

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

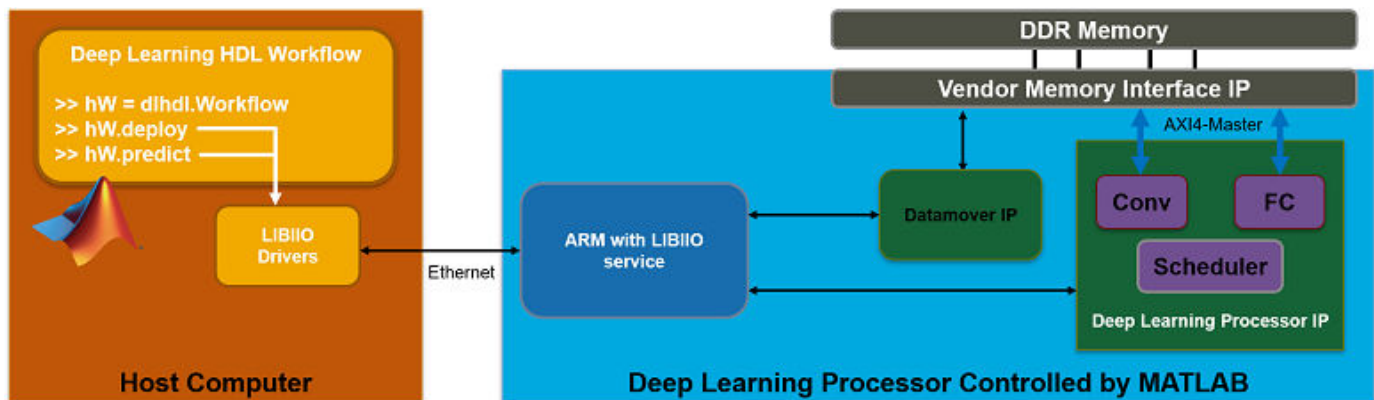
“LIBIIO/Ethernet Performance” on page 6-2

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

d\hd\l.Target

More About

- “Accelerate Prototyping Workflow for Large Networks by Using Ethernet” on page 10-82

Networks and Layers

Supported Networks, Layers, Boards, and Tools

In this section...
“Supported Pretrained Networks” on page 7-2
“Supported Layers” on page 7-10
“Supported Boards” on page 7-25
“Third-Party Synthesis Tools and Version Support” on page 7-25
“Image Input Layer Normalization Hardware Implementation” on page 7-26

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 7-10. 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	
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 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 10-17.	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 10-51.	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 10-30.	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.	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_bldetnet	snet_bldetnet is a custom convolutional neural network used to identify and classify defects. For more information, see “Defect Detection” on page 10-30.	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.	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 <code>darknet53</code> .	Directed acyclic graph (DAG) network based	No. Network exceeds PL DDR memory size.	No. Network fully connected layer exceeds memory size.	Yes	Yes	No. Network fully connected layer exceeds memory size.	Yes	Classification
ResNet-18	ResNet-18 convolutional neural network. For the pretrained ResNet-18 model, see <code>resnet18</code> .	Directed acyclic graph (DAG) network based	Yes		Yes	Yes	Yes	Yes	Classification
ResNet-50	ResNet-50 convolutional neural network. For the pretrained ResNet-50 model, see <code>resnet50</code> .	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 look only 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 10-128.	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 <code>mobilenetv2</code> .	Directed acyclic graph (DAG) network based	Yes	No. Fully Connected layer exceeds PL DDR memory size.	Yes	No	No. Fully Connected layer exceeds PL DDR memory size.	No	Classification
GoogLeNet	GoogLeNet convolutional neural network. For the pretrained GoogLeNet model, see <code>googlenet</code> .	Directed acyclic graph (DAG) network based	No. To use the bitstream, enable the <code>LRNBlockGeneration</code> property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the <code>LRNBlockGeneration</code> property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the <code>LRNBlockGeneration</code> property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the <code>LRNBlockGeneration</code> property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the <code>LRNBlockGeneration</code> property of the processor configuration for the bitstream and generate the bitstream again.	No. To use the bitstream, enable the <code>LRNBlockGeneration</code> 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
-------	--	--	-----------------------------	-----------------------------	-----------------------------	-----------------------------	-----------------------------	-----	--------------


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 7-26.	Yes. Runs as single datatype in SW.

Convolution and Fully Connected Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible
 convolution2dL ayer	HW	Convolution (Conv)	<p>A 2-D convolutional layer applies sliding convolutional filters to the input.</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none"> • Filter size must be 1-36. • Stride size must be 1-15 and square. • Padding size must be in the range 0-8. • Dilation factor must be [1 1]. • Padding value is not supported. 	Yes



 <p>groupedConvolution2dLayer</p>	HW	Convolution (Conv)	<p>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.</p> <p>Code generation is now supported for a 2-D grouped convolution layer that has the NumGroups property set as 'channel-wise'.</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none"> • Filter size must be 1-15 and square. For example [1 1] or [14 14]. When the NumGroups is set as 'channel-wise', filter size must be 3-14. • Stride size must be 1-15 and square. • Padding size must be in the range 0-8. 	Yes
--	----	--------------------	--	-----


			<ul style="list-style-type: none">• 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.	
--	--	--	--	--

 <p>transposedConv 2dLayer</p>	HW	Convolution (Conv)	<p>A transposed 2-D convolution layer upsamples feature maps.</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none"> • Filter size must be 1-8 and square. • Stride size must be 1-36 and square. • Padding size must be in the range 0-8. • Padding value is not supported. 	Yes
 <p>fullyConnected Layer</p>	HW	Fully Connected (FC)	<p>A fully connected layer multiplies the input by a weight matrix, and then adds a bias vector.</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none"> • 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.	<p>A ReLU layer performs a threshold operation to each element of the input where any value less than zero is set to zero.</p> <p>A ReLU layer is supported only when it is preceded by any of these layers:</p> <ul style="list-style-type: none"> • Convolution • Fully Connected • Adder 	Yes
 leakyReluLayer	HW	Layer is fused.	<p>A leaky ReLU layer performs a threshold operation where any input value less than zero is multiplied by a fixed scalar.</p> <p>A leaky ReLU layer is supported only when it is preceded by any of these layers:</p> <ul style="list-style-type: none"> • Convolution • Fully Connected • Adder 	Yes

 <p>clippedReluLayer</p>	<p>HW</p>	<p>Layer is fused.</p>	<p>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.</p> <p>A clipped ReLU layer is supported only when it is preceded by any of these layers:</p> <ul style="list-style-type: none"> • Convolution • Fully Connected • Adder 	<p>Yes</p>
---	-----------	------------------------	--	------------


Normalization, Dropout, and Cropping Layers



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


 crossChannelNormalizationLayer	HW	Convolution (Conv)	<p>A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.</p> <p>The WindowChannelsize must be in the range of 3-9 for code generation.</p>	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

Pooling and Unpooling Layers

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


 <p>maxPooling2dLayer</p>	HW	Convolution (Conv)	<p>A max pooling layer performs downsampling by dividing the layer input into rectangular pooling regions and computing the maximum of each region.</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none"> • Pool size must be 1-36. • Stride size must be 1-15 and square. • Padding size must be in the range 0-2. <p>HasUnpoolingOutputs is supported. When this parameter is enabled, these limitations apply for code generation for this layer:</p> <ul style="list-style-type: none"> • 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. • Filter size and stride size must be square. For example, [2 2]. 	<p>Yes</p> <p>No, when HasUnpoolingOutputs is enabled.</p>
--	----	--------------------	--	--


 maxUnpooling2d Layer	HW	Convolution (Conv)	A max unpooling layer unpools the output of a max pooling layer.	No
 averagePooling 2dLayer	HW	Convolution (Conv)	<p>An average pooling layer performs downsampling by dividing the layer input into rectangular pooling regions and computing the average values of each region.</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none"> • Pool size must be 1-36. • Stride size must be 1-15 and square. • Padding size must be in the range 0-2. 	Yes

 <p>globalAveragePooling2dLayer</p>	HW	Convolution (Conv)	<p>A global average pooling layer performs downsampling by computing the mean of the height and width dimensions of the input.</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none"> • When the layer is implemented in the Conv module, the pool size must be 1-36 and square. • Can accept inputs of sizes up to 15-by-15-by-N. 	Yes
--	----	--------------------	---	-----




Combination Layers


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

 additionLayer	HW	Inherit from input.	<p>An addition layer adds inputs from multiple neural network layers element-wise.</p> <p>You can now generated code for this layer with <code>int8</code> data type when the layer is combined with a Leaky ReLU or Clipped ReLU layer.</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none">• 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.	<p>A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension).</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none"> • The input activation feature number must be a multiple of the square root of the “ConvThreadNumber”. • No layer may drive more than one input to any depth concatenation layer. • 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

Output Layer

Layer	Layer Type Hardware (HW) or Software(SW)	Description and Limitations	INT8 Compatible
 softmaxLayer	SW and HW	<p>A softmax layer applies a softmax function to the input.</p> <p>If the softmax layer is implemented in hardware:</p> <ul style="list-style-type: none"> • 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	HW	<p>A sigmoid layer applies a sigmoid function to the input.</p> <p>When generating code for a network using this layer, these limitations apply:</p> <ul style="list-style-type: none"> • 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. 	Yes
--	----	--	-----

Keras and ONNX Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible
nnet.keras.layer.FlattenCSyleLayer	HW	Layer will be fused	<p>Flatten activations into 1-D layers assuming C-style (row-major) order.</p> <p>A nnet.keras.layer.FlattenCSyleLayer is supported only when it is followed by a fully connected layer.</p>	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.	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 8-25 .	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

Third-Party Synthesis Tools and Version Support

Deep Learning HDL Toolbox has been tested with:

- Xilinx Vivado Design Suite 2020.1
- Intel Quartus Prime 18.1

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

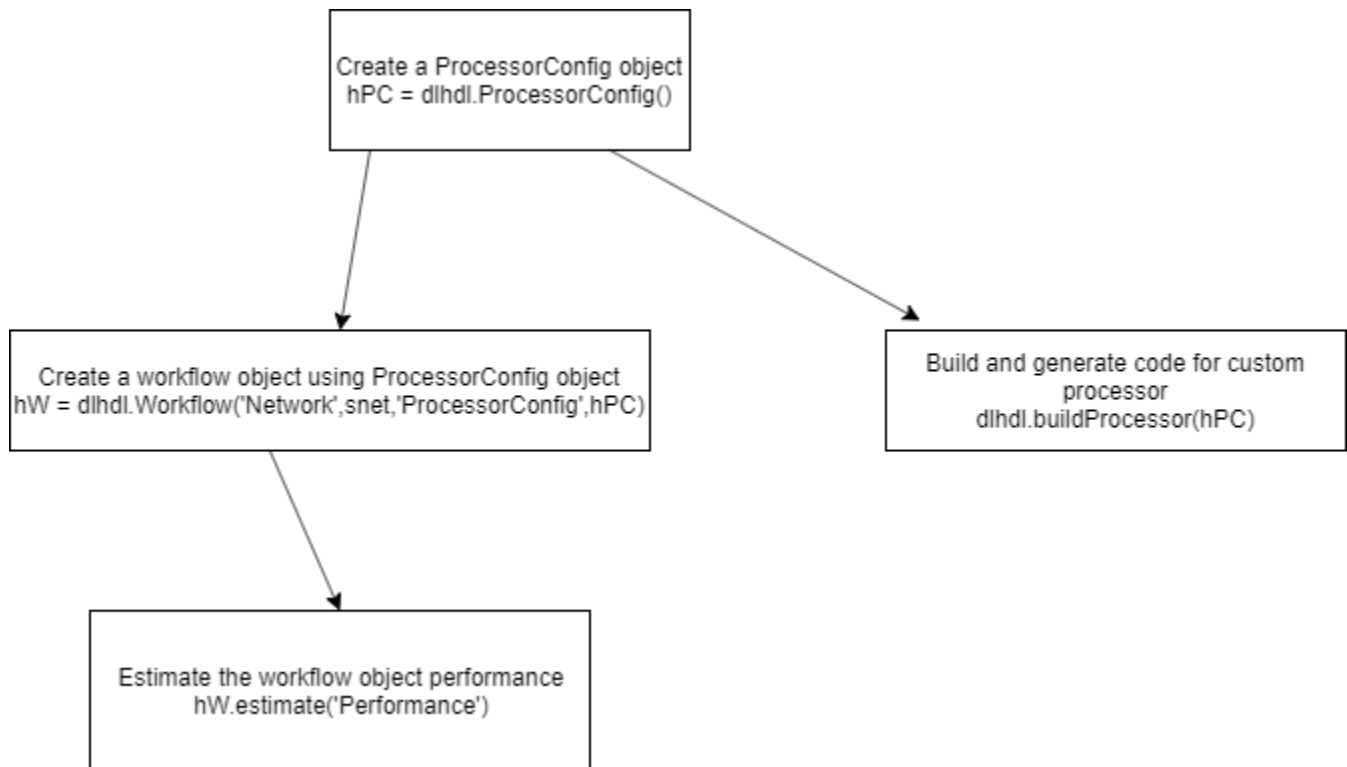
- “Custom Processor Configuration Workflow” on page 8-2
- “Estimate Performance of Deep Learning Network” on page 8-3
- “Estimate Resource Utilization for Custom Processor Configuration” on page 8-10
- “Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization” on page 8-16
- “Generate Custom Bitstream to Meet Custom Deep Learning Network Requirements” on page 8-18
- “Create Deep Learning Processor Configuration for Custom Layers” on page 8-25
- “Register, Validate, and Deploy Custom Hyperbolic Tangent Layer Network to FPGA” on page 8-34

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 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 12-2.

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 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/gpuocoder/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

Network	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total Latency	Frame
	-----	-----	-----	-----	-----
conv_1	39853460	0.19927	1	39853460	
maxpool_1	6825287	0.03413			
conv_2	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 K
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.lay
### 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

Network	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	23634966	0.10743	1	23634966
___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 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 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.layer.BatchNormalizationLayer'
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.ClassificationLayer' is implemented in software
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	TotalTime
	-----	-----	-----	-----
Network	21219873	0.10610	1	21219873
___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 K
        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.layer
### 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	Total
	-----	-----	-----	-----
Network	23634966	0.10743	1	23634966
___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 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 8-16

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'
  LRNBBlockGeneration: 'on'
  ConvThreadNumber: 16
  InputMemorySize: [227 227 3]
  OutputMemorySize: [227 227 3]
  FeatureSizeLimit: 2048

Processing Module "fc"
  ModuleGeneration: 'on'
  SoftmaxBlockGeneration: 'off'
  FThreadNumber: 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

The user wants 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 (shipping) 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 user resource budget and is on the higher end of the cost spectrum. You can achieve target performance and resource use budget by quantizing the target deep learning network and customizing the custom default bitstream configuration.

In this example create a custom bitstream configuration to match your resource budget and performance requirements.

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');
dlquantObj.calibrate(Image)
```

```
ans=21x5 table
```

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv_1_Weights' }	{'batchnorm_1'}	"Weights"	-0.017061
{'conv_1_Bias' }	{'batchnorm_1'}	"Bias"	-0.025344
{'conv_2_Weights' }	{'batchnorm_2'}	"Weights"	-0.54744
{'conv_2_Bias' }	{'batchnorm_2'}	"Bias"	-1.1787
{'conv_3_Weights' }	{'batchnorm_3'}	"Weights"	-0.39927
{'conv_3_Bias' }	{'batchnorm_3'}	"Bias"	-0.85118
{'fc_Weights' }	{'fc' }	"Weights"	-0.22558

```

{'fc_Bias'           }      {'fc'           }      "Bias"           -0.011837
{'imageinput'       }      {'imageinput'    }      "Activations"    0
{'imageinput_normalization'}  {'imageinput'    }      "Activations"    -22.566
{'conv_1'           }      {'batchnorm_1'   }      "Activations"    -7.9196
{'relu_1'           }      {'relu_1'        }      "Activations"    0
{'maxpool_1'        }      {'maxpool_1'     }      "Activations"    0
{'conv_2'           }      {'batchnorm_2'   }      "Activations"    -8.4641
{'relu_2'           }      {'relu_2'        }      "Activations"    0
{'maxpool_2'        }      {'maxpool_2'     }      "Activations"    0
:

```

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`.

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

```
hPC_reference =
```

```

    Processing Module "conv"
      ModuleGeneration: 'on'
      LRNBBlockGeneration: 'on'
      ConvThreadNumber: 64
      InputMemorySize: [227 227 3]
      OutputMemorySize: [227 227 3]
      FeatureSizeLimit: 2048

    Processing Module "fc"
      ModuleGeneration: 'on'
      SoftmaxBlockGeneration: 'off'
      FCThreadNumber: 16
      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: 'int8'

    System Level Properties
      TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation K
      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` function of the `dlhdl.ProcessorConfig` object. The function returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

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

```
hPC_reference.estimatePerformance(dlquantObj)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer'
### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in :
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is impl
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
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

```
hPC_reference.estimateResources
```

Deep Learning Processor Estimator Resource Results

	DSPs	Block RAM*	LUTs(CLB/ALUT)
	-----	-----	-----
Available	2520	912	274080
DL_Processor	768(31%)	386(43%)	193454(71%)

* 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 - 768
- 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 the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

To reduce the resource use for the custom bitstream, modify the `KernelDataType` for the `conv`, `fc`, and `adder` modules. Modify the `ConvThreadNumber` to reduce DSP slice count. Reduce the `InputMemorySize` and `OutputMemorySize` for the `conv` module to reduce BRAM count.

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

```
hPC_custom =
```

```
    Processing Module "conv"
      ModuleGeneration: 'on'
      LRNBlockGeneration: 'on'
      ConvThreadNumber: 4
      InputMemorySize: [30 30 1]
      OutputMemorySize: [30 30 1]
      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: '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

To estimate the performance of the digits 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).

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

```
hPC_custom.estimatePerformance(dlquantObj)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer'
### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in :
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is impl
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	267467	0.00134	1	2
___conv_1	25628	0.00013		
___maxpool_1	42337	0.00021		
___conv_2	30957	0.00015		
___maxpool_2	68153	0.00034		
___conv_3	56981	0.00028		
___fc	43411	0.00022		

* The clock frequency of the DL processor is: 200MHz

```
hPC_custom.estimateResources
```

Deep Learning Processor Estimator Resource Results

	DSPs	Block RAM*	LUTs(CLB/ALUT)
	-----	-----	-----
Available	2520	912	274080
DL_Processor	120(5%)	108(12%)	69517(26%)

* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices

The estimated performance is 750 FPS and the estimated resource use counts are:

- Digital signal processor (DSP) slice count - 129
- 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 8-16

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
- "Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization" on page 8-16

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: '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 K
  TargetFrequency: 200
  SynthesisTool: 'Xilinx Vivado'
  ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
  SynthesisToolChipFamily: 'Zynq UltraScale+'
  SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
  SynthesisToolPackageName: ''
  SynthesisToolSpeedValue: ''

```

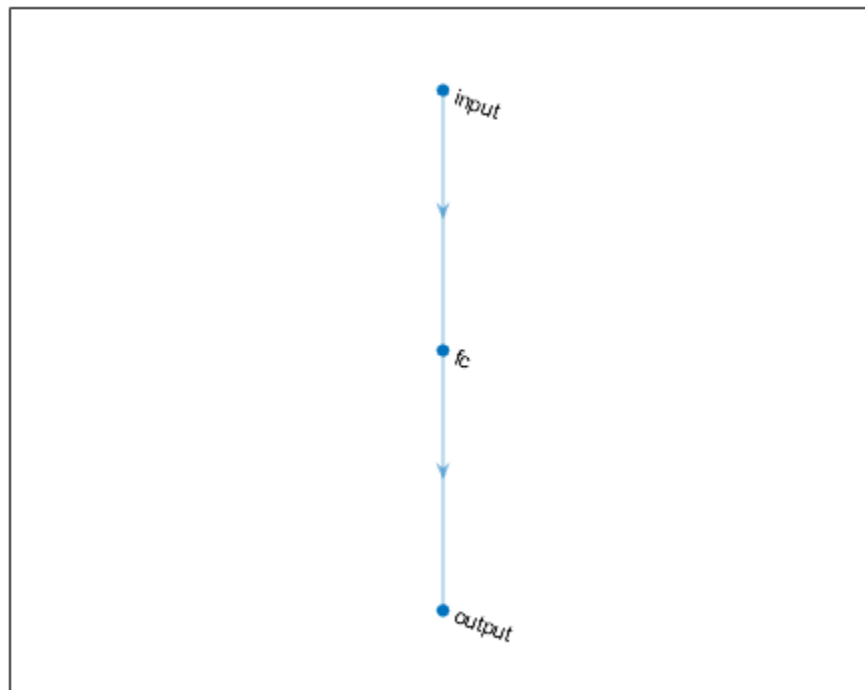
Optimize Processor Configuration for a Custom Fully Connected (FC) Layer only Network

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	368(15%)	508(56%)	224761(83%)

* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices

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

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

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	TotalTime
Network	137574	0.00069	1	0.00069
__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 — 368
- 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%)	40740(15%)

* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices

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

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

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	TotalTime
Network	137574	0.00069	1	0.00069
__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 "on" to "off" because
### 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 "adder" property "ModuleGeneration" changed from "on" to "off" because
```

```

    Processing Module "conv"
        ModuleGeneration: 'off'

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

    Processing Module "adder"
        ModuleGeneration: 'off'

    Processor Top Level Properties
        RunTimeControl: 'register'
        InputDataInterface: 'External Memory'
        OutputDataInterface: 'External Memory'
        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%)	40740(15%)

* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices

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

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

```
Deep Learning Processor Estimator Performance Results
```

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	137574	0.00069	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 8-16

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.

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

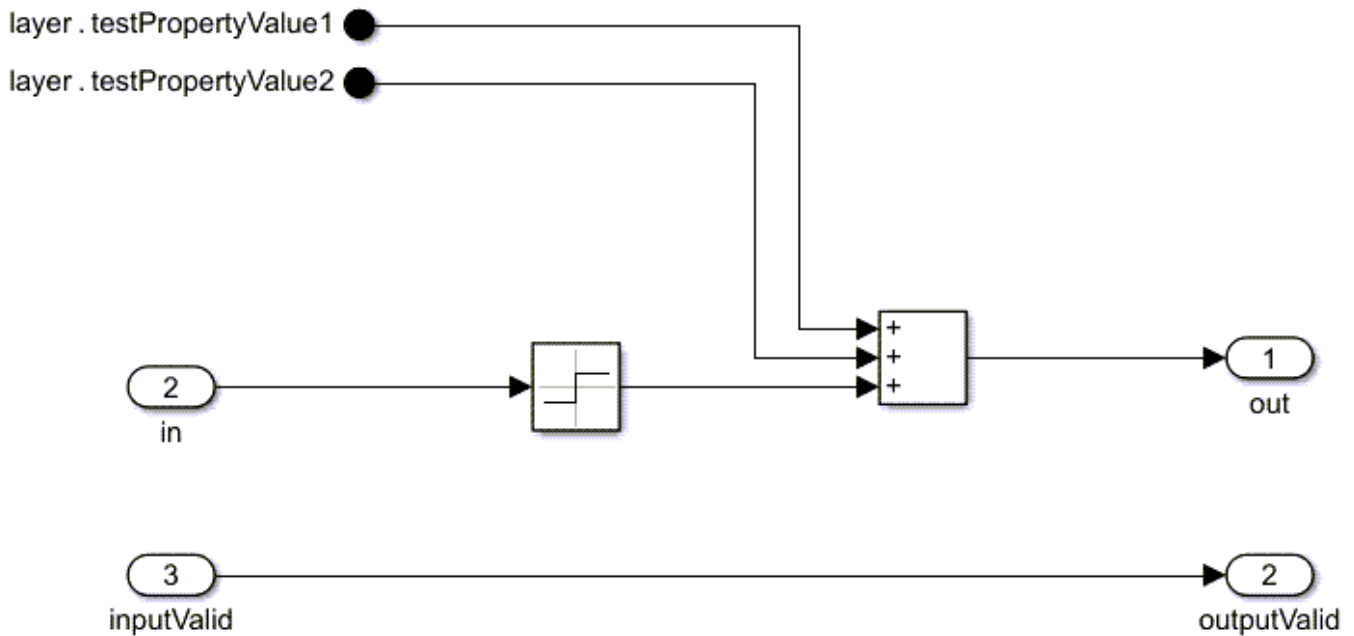
```

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: ''
```

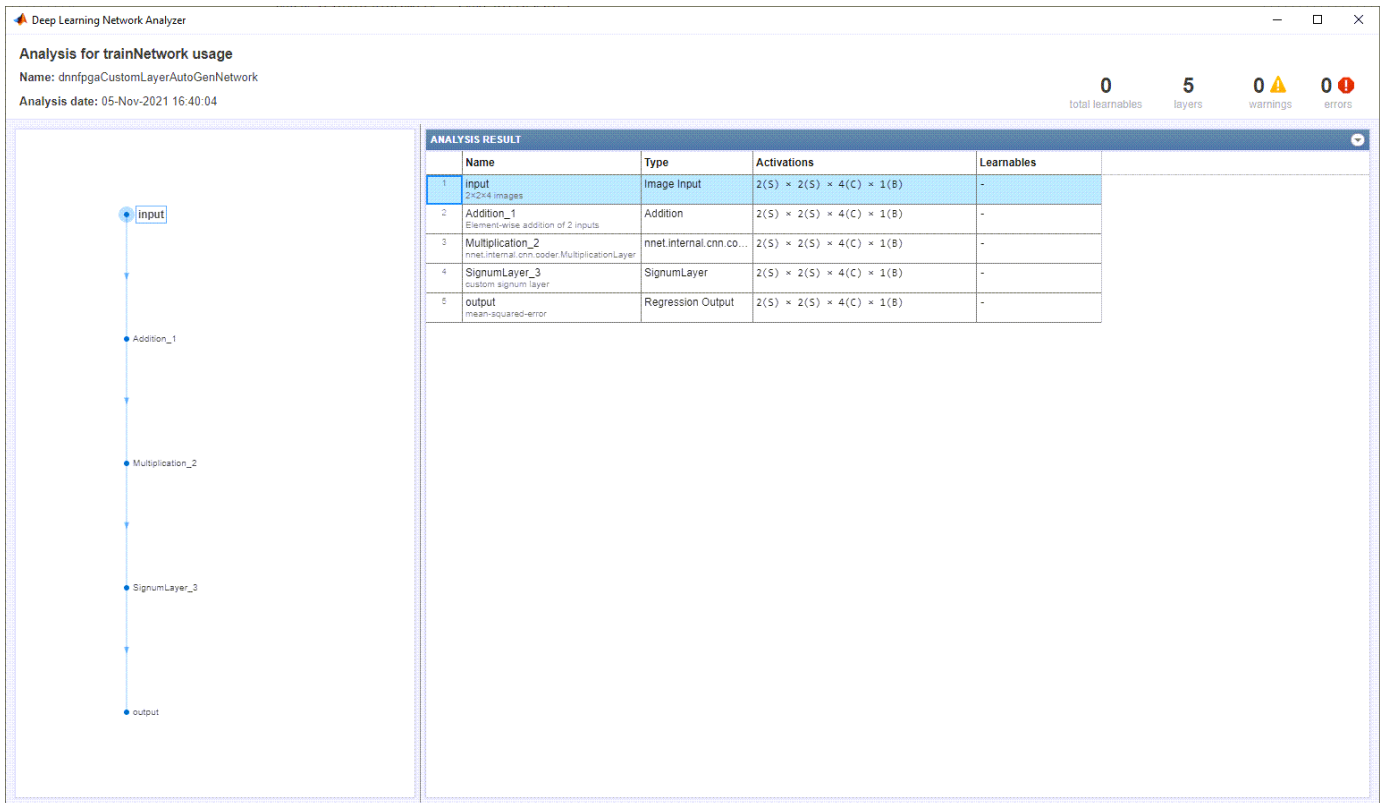
Generate Verification Model for Custom Layer

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.

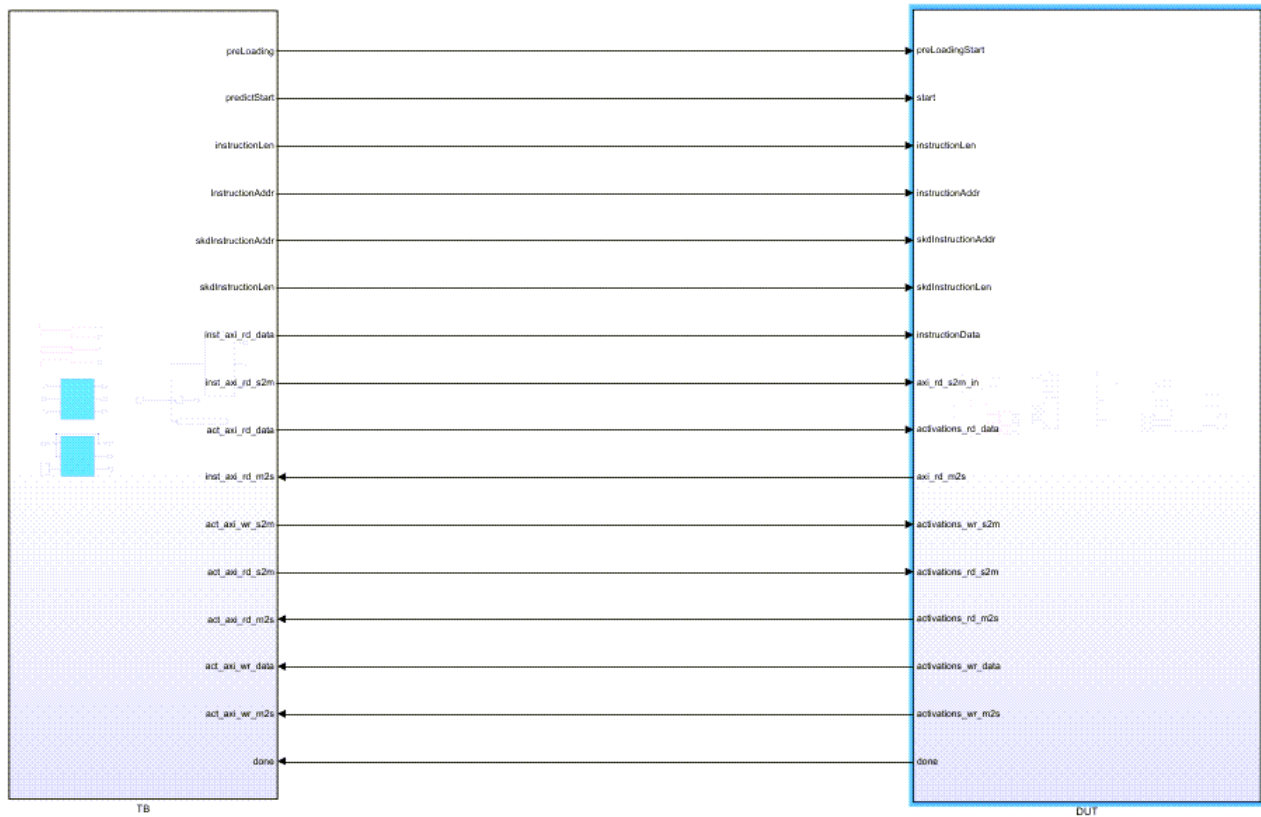
`openCustomLayerModel(hPC)`

```
### 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.



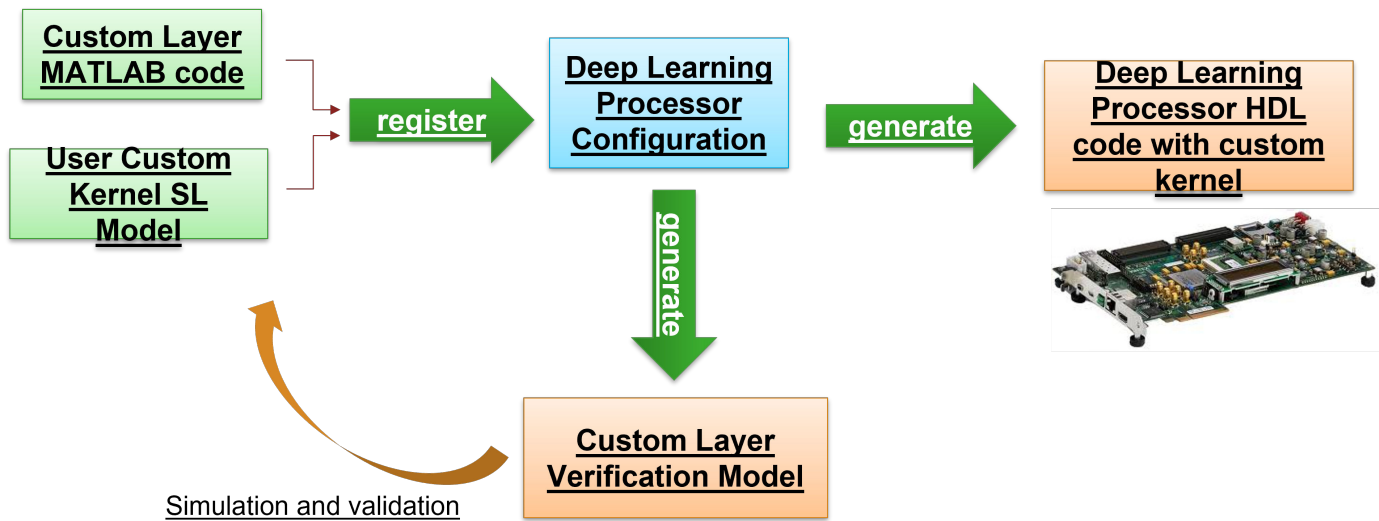
Simulate and Validate Custom Layer Model

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 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);
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 = dldhdl_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 dldhdl.CustomLayer
% object which are used to register custom layer information for Deep
% Learning HDL Toolbox workflow
% 3. Use dldhdl.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 = { ...
    dldhdl.CustomLayer('Name', 'Addition', 'Layer', additionLayer(2), 'Model', 'model/customLayers/dnnfpgaAdditionLayerModel.slx'),
    dldhdl.CustomLayer('Name', 'Multiplication', 'Layer', multiplicationLayer(2), 'Model', 'model/customLayers/dnnfpgaMultiplicationLayerModel.slx')
};

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 = dldhdl_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 dldhdl.CustomLayer
% object which are used to register custom layer information for Deep
% Learning HDL Toolbox workflow
% 3. Use dldhdl.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 = { ...
    dldhdl.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 = dldhdl.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 tanh layer, see “Register, Validate, and Deploy Custom Hyperbolic Tangent Layer Network to FPGA” on page 8-34 .

See Also

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

Register, Validate, and Deploy Custom Hyperbolic Tangent Layer Network to FPGA

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

- Use the custom hyperbolic tangent (tanh) layer from Deep Learning Toolbox™.
- Register the custom hyperbolic tangent (tanh) layer by using the `registerCustomLayer` method.
- Validate the custom hyperbolic tangent (tanh) layer by generating a custom layer verification model.
- Generate a custom bitstream.

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: 'on'
SegmentationBlockGeneration: 'off'
      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'
      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 K
  TargetFrequency: 200
  SynthesisTool: 'Xilinx Vivado'
  ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
  SynthesisToolChipFamily: 'Zynq UltraScale+'
  SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
  SynthesisToolPackageName: ''
  SynthesisToolSpeedValue: ''

```

Register Custom Layer and Model

To register an instance of the custom layer and custom layer Simulink® model use the `registerCustomLayer` method. This example leverages the built-in hyperbolic tangent (`tanh`) layer from Deep Learning Toolbox™. Deep Learning HDL Toolbox™ uses the Simulink® model to generate a verification model for the custom layer.

```
hTanhLayer = tanhLayer('Name', 'tanh')
```

```
hTanhLayer =
  TanhLayer with properties:
```

```
  Name: 'tanh'
```

```
  Learnable Parameters
  No properties.
```

```
  State Parameters
  No properties.
```

```
  Show all properties
```

```
registerCustomLayer(hPC, Layer = hTanhLayer, Model = 'dnnfpgaTanhLayerModel.slx')
```

The custom deep learning processor configuration has a `tanhLayer` under the custom processing module. The custom hyperbolic tangent (`tanh`) 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)
```

```

### The 'Network' property is empty for the given workflow object. An auto-generated network is p
### Notice: The layer 'input' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softw
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented i

```

```
### 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"
"EndOffset"	"0x02400000"	"Total: 36.0 MB"

The `openCustomLayerModel` method generates a verification model file called `dnnfpgaCustomLayerVerificationModel.slx` for your custom layer.

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.

```
verifyCustomLayerModel(hPC)
```

```
### Compiling Simulink model 'dnnfpgaCustomLayersTB' ...
```

```
Warning: '<a href="matlab:slprivate('open_and_hilite_port_hyperlink', 'hilite', ['dnnfpgaCustomL
```

```
Warning: '<a href="matlab:slprivate('open_and_hilite_port_hyperlink', 'hilite', ['dnnfpgaCustomL
```

```
Warning: '<a href="matlab:slprivate('open_and_hilite_port_hyperlink', 'hilite', ['dnnfpgaCustomL
```

```
### Complete Simulink model 'dnnfpgaCustomLayersTB' compilation.  
Verification passed.
```

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 network as the `Network` argument and the custom bitstream `myCustomLayer.bit` as the `Bitstream` argument. Compile and deploy the `dlhdl.Workflow` object. To retrieve the prediction results from the deployed network, use MATLAB® and the `predict` method.

```
layers = [imageInputLayer([2 2 4], 'Name', 'data', 'Normalization', 'none');  
tanhLayer('Name', 'tanh');  
regressionLayer('Name', 'output');  
]  
myCustomNet = assembleNetwork(layers);  
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');
```

See Also

[dlhdl.Workflow](#) | [dlhdl.ProcessorConfig](#) | [registerCustomLayer](#) | [openCustomLayerModel](#) | [verifyCustomLayerModel](#) | [dlhdl.buildProcessor](#)

More About

- “Create Deep Learning Processor Configuration for Custom Layers” on page 8-25

Custom Processor Code Generation Workflow

- “Generate Custom Bitstream” on page 9-2
- “Generate Custom Processor IP” on page 9-4

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:

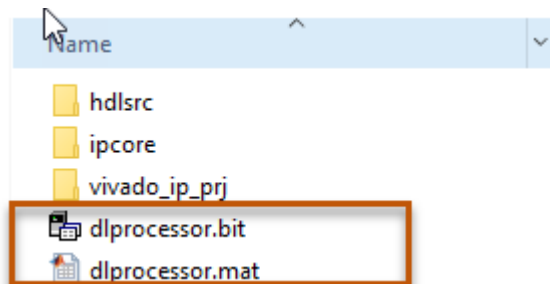
```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat');
```

- 3 Generate the custom bitstream.

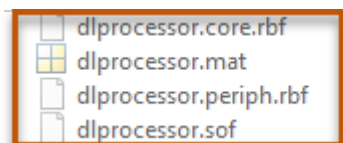
```
dlhdl.buildProcessor(hPC);
```

- 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;
```

Intel Bitstream Resource Utilization

To see the bitstream resource utilization for the reference (shipping) Intel bitstreams, see “Bitstream Resource Utilization” (Deep Learning HDL Toolbox Support Package for Intel FPGA and SoC Devices).

Xilinx Bitstream Resource Utilization

To see the bitstream resource utilization for the reference (shipping) Xilinx bitstreams, see “Bitstream Resource Utilization” (Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices).

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\2019.2\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 12-2

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 10-17
- “Image Category Classification by Using Deep Learning” on page 10-21
- “Defect Detection” on page 10-30
- “Profile Network for Performance Improvement” on page 10-47
- “Bicyclist and Pedestrian Classification by Using FPGA ” on page 10-51
- “Visualize Activations of a Deep Learning Network by Using LogoNet” on page 10-56
- “Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core” on page 10-62
- “Run a Deep Learning Network on FPGA with Live Camera Input” on page 10-67
- “Running Convolution-Only Networks by Using FPGA Deployment” on page 10-77
- “Accelerate Prototyping Workflow for Large Networks by Using Ethernet” on page 10-82
- “Create Series Network for Quantization” on page 10-90
- “Custom Deep Learning Processor Generation to Meet Performance Requirements” on page 10-94
- “Deploy Quantized Network Example” on page 10-98
- “Quantize Network for FPGA Deployment” on page 10-110
- “Evaluate Performance of Deep Learning Network on Custom Processor Configuration” on page 10-117
- “Customize Bitstream Configuration to Meet Resource Use Requirements” on page 10-123
- “Vehicle Detection Using DAG Network Based YOLO v2 Deployed to FPGA” on page 10-128
- “Customize Bitstream Configuration to Meet Resource Use Requirements” on page 10-137
- “Image Classification Using DAG Network Deployed to FPGA” on page 10-142
- “Classify Images on an FPGA Using a Quantized DAG Network” on page 10-150
- “Classify ECG Signals Using DAG Network Deployed To FPGA” on page 10-163
- “Prototype and Verify Deep Learning Networks Without Target Hardware” on page 10-177
- “Classify Images on FPGA by Using Quantized GoogLeNet Network” on page 10-184
- “Estimate Resource Utilization for Custom Board and Reference Design” on page 10-197
- “Speech Command Recognition by Using FPGA” on page 10-200
- “Modulation Classification by Using FPGA” on page 10-211
- “Deploy Simple Adder Network by using MATLAB Deployment Script and Deployment Instructions File” on page 10-223
- “Human Pose Estimation by Using Segmentation DAG Network Deployed to FPGA” on page 10-231

- “Semantic Segmentation of Multispectral Images by Using Quantized U-Net on FPGA”
on page 10-237

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, 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 18.1. 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\18.1\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)
```

```

hW =
  Workflow with properties:

      Network: [1x1 SeriesNetwork]
      Bitstream: 'arria10soc_single'
      ProcessorConfig: []
      Target: [1x1 dlhdl.Target]

```

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.layer'
      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 target
### 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');
imshow(inputImg);
```



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	Total
	-----	-----	-----	-----
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

See Also

More About

- “Create Simple Deep Learning 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, 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)
```

```
hW =  
  Workflow with properties:
```

```

Network: [1x1 SeriesNetwork]
Bitstream: 'zcu102_single'
ProcessorConfig: []
Target: [1x1 dlhdl.Target]

```

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	28x28x1 images with 'zerocenter' normalization
2	'conv_1'	Convolution	8 3x3x1 convolutions with stride [1 1] and padding
3	'batchnorm_1'	Batch Normalization	Batch normalization with 8 channels
4	'relu_1'	ReLU	ReLU
5	'maxpool_1'	Max Pooling	2x2 max pooling with stride [2 2] and padding
6	'conv_2'	Convolution	16 3x3x8 convolutions with stride [1 1] and padding
7	'batchnorm_2'	Batch Normalization	Batch normalization with 16 channels
8	'relu_2'	ReLU	ReLU
9	'maxpool_2'	Max Pooling	2x2 max pooling with stride [2 2] and padding
10	'conv_3'	Convolution	32 3x3x16 convolutions with stride [1 1] and padding
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.layer'
### 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 implemented
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 implemented
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

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');
imshow(inputImg);
```

5

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	Total
	-----	-----	-----	-----
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

See Also

More About

- “Create Simple Deep Learning Network for Classification”

Get Started with Deep Learning FPGA Deployment on Xilinx ZC706 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 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® Zynq® ZC706 Evaluation Kit
- Deep Learning HDL Toolbox™ Support Package for Xilinx® FPGA and SoC
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Load Pretrained Series Network

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

```
snet = getDigitsNetwork;
```

View the layers of the pretrained series network, by using the `analyzeNetwork` function.

```
analyzeNetwork(snet)
```

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 2019.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.l  
hTarget = dlhdl.Target('Xilinx');
```

Create WorkFlow Object

Create an object of the `dlhdl.Workflow` class. 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 Xilinx Zynq ZC706 board. The bitstream uses a single data type.

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

Compile MNIST Series Network

Run the `compile` function of the `dlhdl.Workflow` object, to compile the MNIST series network.

```
dn = hW.compile
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.lay  
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: [1x1 struct]
  LayerConfigs: [1x1 struct]
  NetConfigs: [1x1 struct]

```

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZC706 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 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

```

Load Example Image

Load the example image.

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



Run Prediction

Execute the `predict` function of the `dlhdl.Workflow` object and display the prediction result and network performance.

```
[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	Total
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

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™

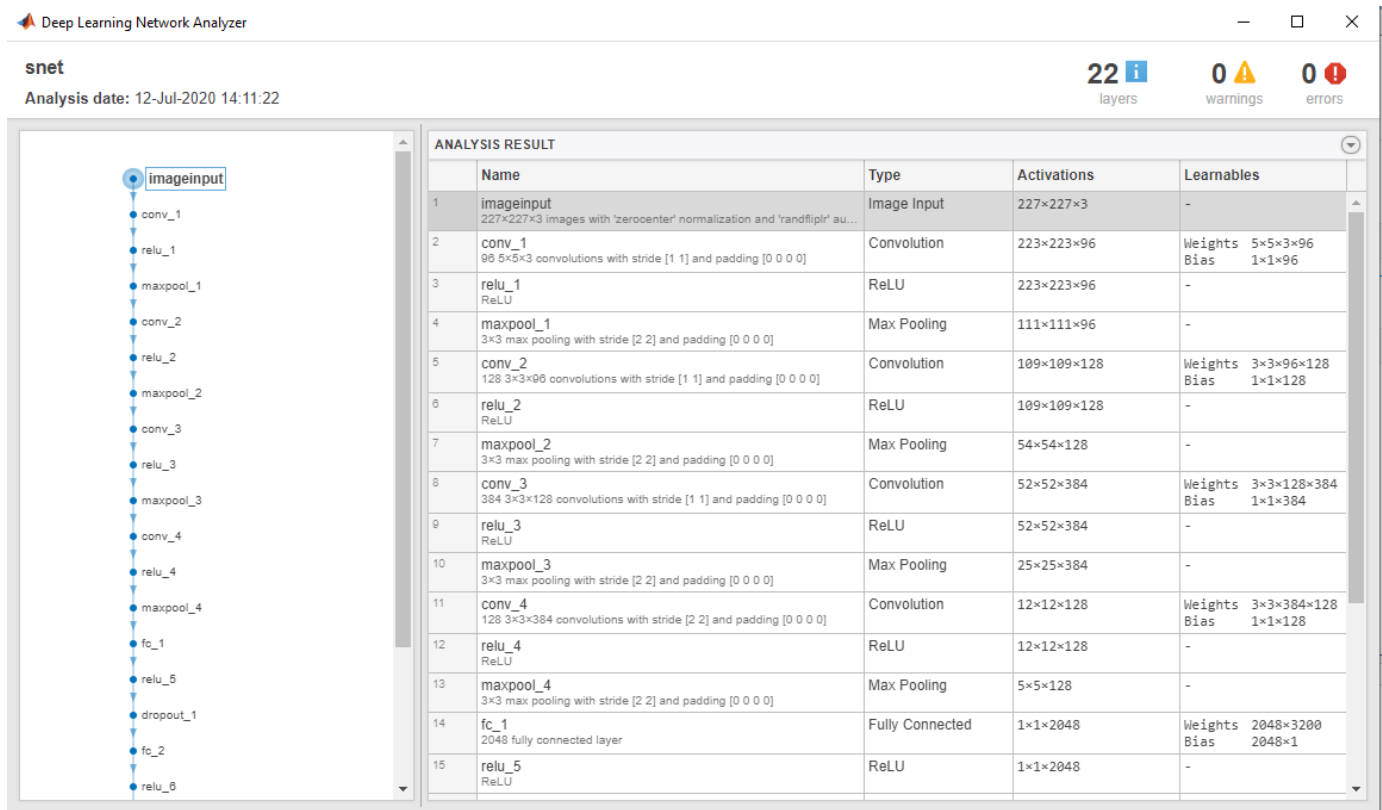
Load the Pretrained Series Network

To load the pretrained series network logonet, enter:

```
snet = getLogoNetwork();
```

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

```
analyzeNetwork(snet)
```



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 2019.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.f
```

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: [1x1 struct]
    LayerConfigs: [1x1 struct]
    NetConfigs: [1x1 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 `dlnet.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.

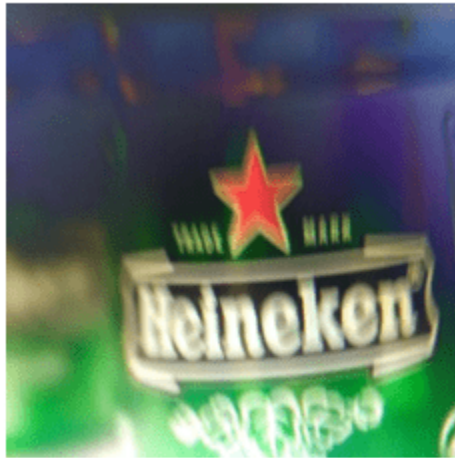
```
hw.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target
### 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	Total
	-----	-----	-----	-----
Network	38865102	0.17666	1	38865102
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'
```

Deploy Transfer Learning Network for Lane Detection

This example shows how to create, compile, and deploy a `dlhdl.Workflow` object that has a lane detection convolutional neural network as the network object, by using the Deep learning HDL Toolbox™. The network can detect and output lane marker boundaries 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.

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

To load the pretrained series network `lanenet`, enter:

```
snet = getLaneDetectionNetwork;
```

Normalize the Input Layer

To normalize the input layer by modifying its type, enter:

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

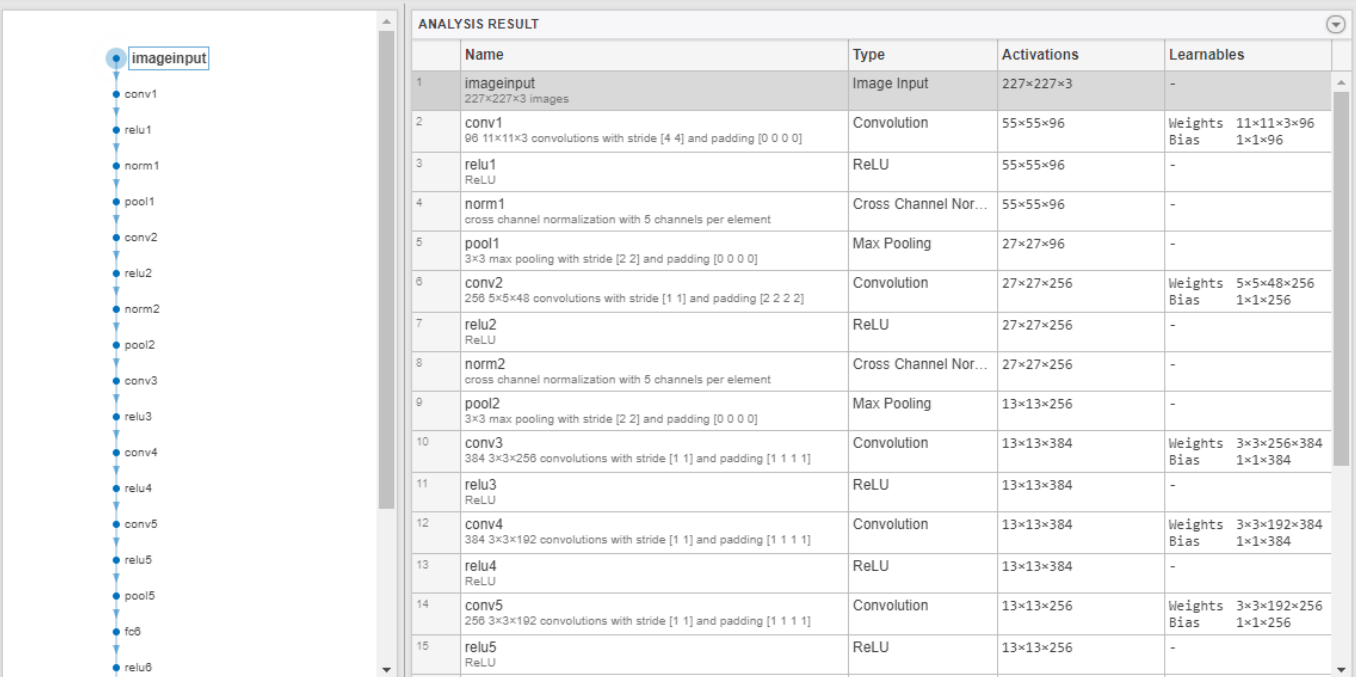
To view the layers of the pretrained series network, enter:

```
analyzeNetwork(snet)  
% The saved network contains 23 layers including input, convolution, ReLU, cross channel normaliz  
% max pool, fully connected, and the regression output layers.
```

Deep Learning Network Analyzer

snet
Analysis date: 12-Jul-2020 14:21:19

23 layers 0 warnings 0 errors



	Name	Type	Activations	Learnables
1	imageinput 227×227×3 images	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 256 5×5×48 convolutions with stride [1 1] and padding [2 2 2 2]	Convolution	27×27×256	Weights 5×5×48×256 Bias 1×1×256
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 384 3×3×192 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	13×13×384	Weights 3×3×192×384 Bias 1×1×384
13	relu4 ReLU	ReLU	13×13×384	-
14	conv5 256 3×3×192 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	13×13×256	Weights 3×3×192×256 Bias 1×1×256
15	relu5 ReLU	ReLU	13×13×256	-

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');
```

Generate Bitstream to Run Network

The lane 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 data 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)
```

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

To learn how to use the generated bitstream file, see [Generate Custom Bitstream](#).

Create Workflow Object

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 lanenet neural network, `snet`, as the network.

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

Compile the Lanenet series Network

To compile the lanenet series 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"	"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 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 target
### 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
```

Run Prediction for Example Video

Run the `demoOnVideo` function for the `dlhdl.Workflow` class object. This function loads the example video, executes the `predict` function of the `dlhdl.Workflow` object, and then plots the result.

```
demoOnVideo(hw,1);
```

```
### Finished writing input activations.
```

```
### Running single input activations.
```

Deep Learning Processor Profiler Performance Results

	LastLayerLatency(cycles)	LastLayerLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	24904175	0.11320	1	24904175
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

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:

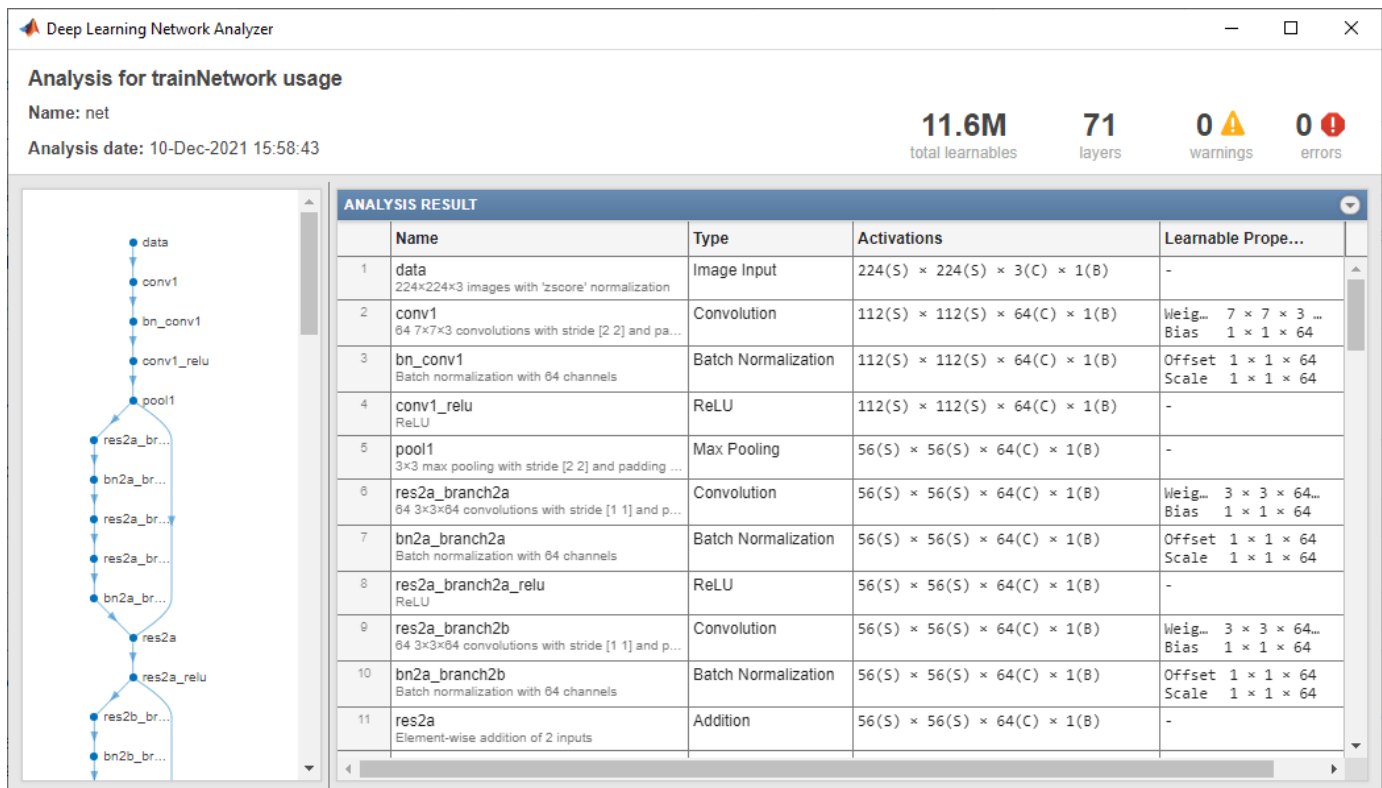
```
% net = vgg19;
```

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 2019.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.l
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
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 wi
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 wi
15	'res2b_branch2a_relu'	ReLU	ReLU
16	'res2b_branch2b'	Convolution	64 3×3×64 convolutions
17	'bn2b_branch2b'	Batch Normalization	Batch normalization wi
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 wi
22	'res3a_branch2a_relu'	ReLU	ReLU
23	'res3a_branch2b'	Convolution	128 3×3×128 convolution
24	'bn3a_branch2b'	Batch Normalization	Batch normalization wi
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 wi
29	'res3b_branch2a'	Convolution	128 3×3×128 convolution
30	'bn3b_branch2a'	Batch Normalization	Batch normalization wi
31	'res3b_branch2a_relu'	ReLU	ReLU
32	'res3b_branch2b'	Convolution	128 3×3×128 convolution
33	'bn3b_branch2b'	Batch Normalization	Batch normalization wi
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 wi
38	'res4a_branch2a_relu'	ReLU	ReLU
39	'res4a_branch2b'	Convolution	256 3×3×256 convolution
40	'bn4a_branch2b'	Batch Normalization	Batch normalization wi
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 wi
45	'res4b_branch2a'	Convolution	256 3×3×256 convolution
46	'bn4b_branch2a'	Batch Normalization	Batch normalization wi
47	'res4b_branch2a_relu'	ReLU	ReLU
48	'res4b_branch2b'	Convolution	256 3×3×256 convolution
49	'bn4b_branch2b'	Batch Normalization	Batch normalization wi
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 wi
54	'res5a_branch2a_relu'	ReLU	ReLU
55	'res5a_branch2b'	Convolution	512 3×3×512 convolution

56	'bn5a_branch2b'	Batch Normalization	Batch normalization with
57	'res5a'	Addition	Element-wise addition o
58	'res5a_relu'	ReLU	ReLU
59	'res5a_branch1'	Convolution	512 1x1x256 convolution
60	'bn5a_branch1'	Batch Normalization	Batch normalization with
61	'res5b_branch2a'	Convolution	512 3x3x512 convolution
62	'bn5b_branch2a'	Batch Normalization	Batch normalization with
63	'res5b_branch2a_relu'	ReLU	ReLU
64	'res5b_branch2b'	Convolution	512 3x3x512 convolution
65	'bn5b_branch2b'	Batch Normalization	Batch normalization with
66	'res5b'	Addition	Element-wise addition o
67	'res5b_relu'	ReLU	ReLU
68	'pool5'	2-D Global Average Pooling	2-D global average pool
69	'fc1000'	Fully Connected	1000 fully connected la
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: [1x1 struct]
    instructions: [1x1 struct]
    registers: [1x1 struct]
    syncInstructions: [1x1 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
```

```

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

```

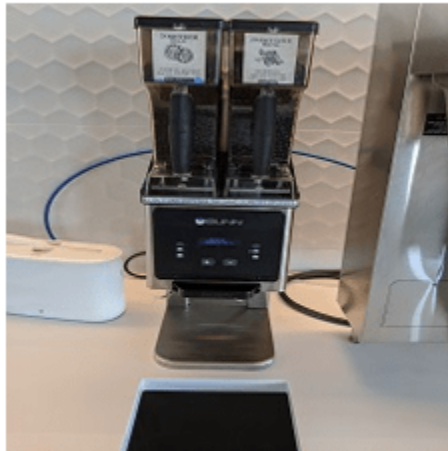
Load Image for Prediction

Load the example image.

```

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

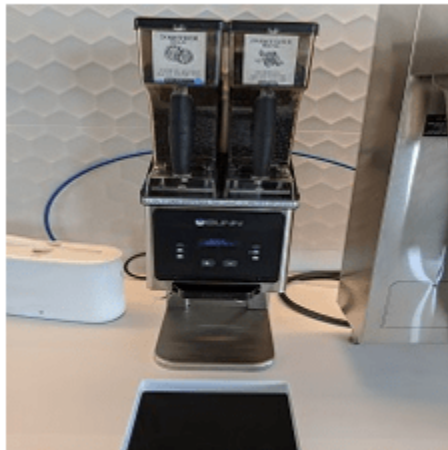
```



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	Total
	-----	-----	-----	-----
Network	24100982	0.10955	1	24
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}
```

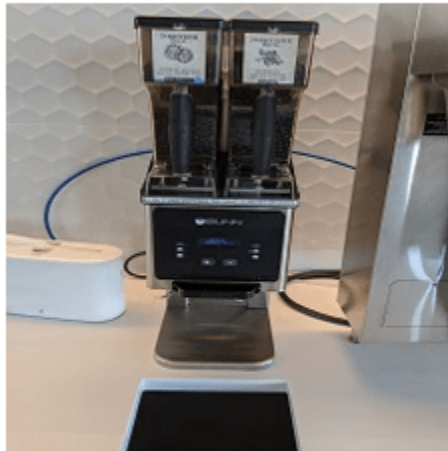
```
ans =
'Polaroid camera'
```

Run Prediction for Multiple Images

Load multiple images and retrieve their prediction results by using the multiple 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
```




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_blemnet`.

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

```
snet_blemnet =
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 2020.1.

Set the Xilinx Vivado toolpath.

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.1\bin\vivado.'
    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 data 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](#).

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' normalization
2	'conv1'	Convolution	96 11×11×1 convolutions with stride [4 4] and padding
3	'relu1'	ReLU	ReLU
4	'norm1'	Cross Channel Normalization	cross channel normalization with 5 channels per group
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 [2 2] and padding
7	'relu2'	ReLU	ReLU
8	'norm2'	Cross Channel Normalization	cross channel normalization with 5 channels per group
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 padding
11	'relu3'	ReLU	ReLU
12	'conv4'	Grouped Convolution	2 groups of 192 3×3×192 convolutions with stride [1 1] and padding
13	'relu4'	ReLU	ReLU
14	'conv5'	Grouped Convolution	2 groups of 128 3×3×192 convolutions with stride [1 1] and padding
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: [1x1 struct]
    instructions: [1x1 struct]
    registers: [1x1 struct]
    syncInstructions: [1x1 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	Total
	-----	-----	-----	-----
Network	12231156	0.05560	1	12
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

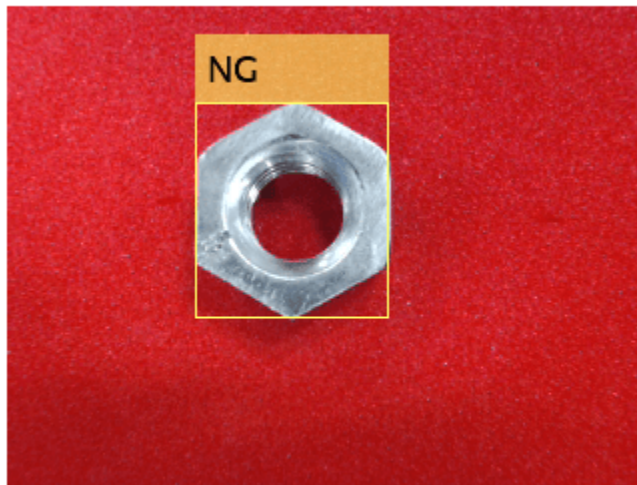
```

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_blemnetnet,'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' normal.
2	'conv_1'	Convolution	20 5×5×1 convolutions with stride [1 1] a

```

3  'relu_1'      ReLU      ReLU
4  'maxpool_1'  Max Pooling 2x2 max pooling with stride [2 2] and padding
5  'crossnorm'  Cross Channel Normalization cross channel normalization with 5 channels
6  'conv_2'     Convolution 20 5x5x20 convolutions with stride [1 1]
7  'relu_2'     ReLU      ReLU
8  'maxpool_2'  Max Pooling 2x2 max pooling with stride [2 2] and padding
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: [1x1 struct]
    instructions: [1x1 struct]
    registers: [1x1 struct]
    syncInstructions: [1x1 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 target
### 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	Total
--------------------------	---------------------------	-----------	-------

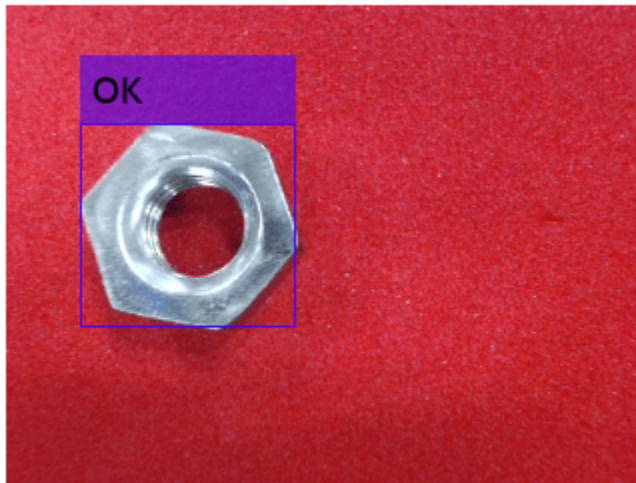
	-----	-----	-----
Network	4892622	0.02224	1
conv_1	467921	0.00213	
maxpool_1	188086	0.00085	
crossnorm	159500	0.00072	
conv_2	397561	0.00181	
maxpool_2	41455	0.00019	
fc_1	3614625	0.01643	
fc_2	23355	0.00011	

* 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: [1x1 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=21x5 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 data 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](#).

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	128x128x1 images with 'zerocenter' normal...
2	'conv_1'	Convolution	20 5x5x1 convolutions with stride [1 1] a
3	'relu_1'	ReLU	ReLU
4	'maxpool_1'	Max Pooling	2x2 max pooling with stride [2 2] and pad
5	'crossnorm'	Cross Channel Normalization	cross channel normalization with 5 channe
6	'conv_2'	Convolution	20 5x5x20 convolutions with stride [1 1]
7	'relu_2'	ReLU	ReLU
8	'maxpool_2'	Max Pooling	2x2 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: [1x1 struct]
    instructions: [1x1 struct]
    registers: [1x1 struct]
    syncInstructions: [1x1 struct]
```

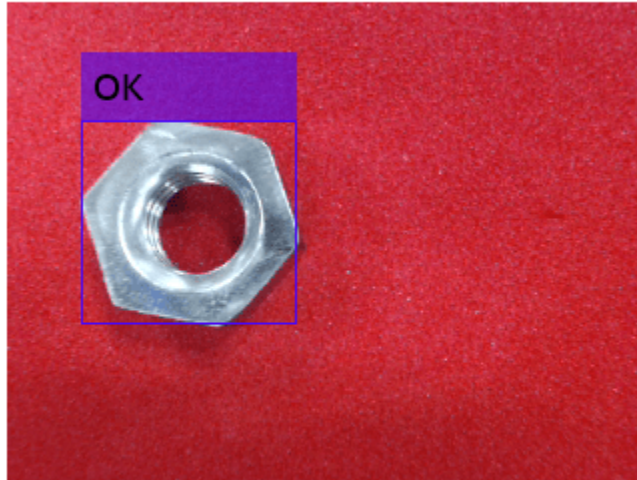
To deploy the network on the Xilinx ZCU102 SoC hardware, run the `deploy` function of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The `deploy` function starts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

```

```

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	Tot
	-----	-----	-----	---
Network	1754969	0.00798	1	1
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

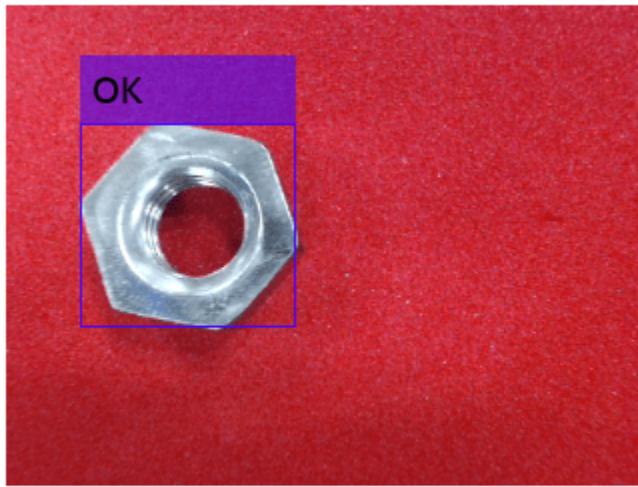
```

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	1
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

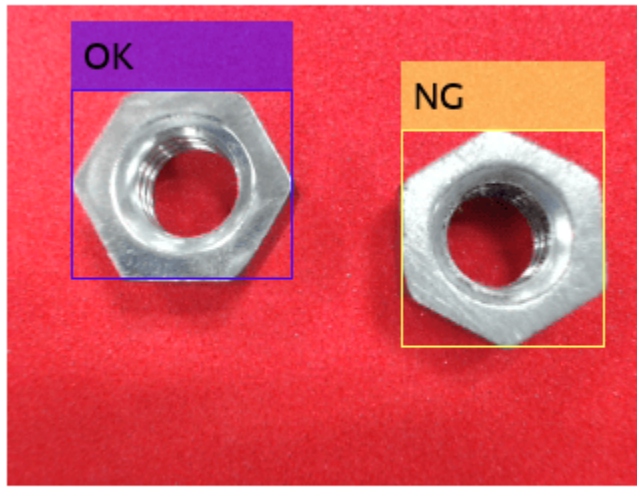
	LastFrameLatency(cycles) -----	LastFrameLatency(seconds) -----	FramesNum -----	Tota --
Network	1754486	0.00797	1	1
conv_1	271014	0.00123		
maxpool_1	87662	0.00040		
crossnorm	125835	0.00057		
conv_2	149789	0.00068		
maxpool_2	19661	0.00009		
fc_1	1085505	0.00493		
fc_2	14930	0.00007		

* 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.

Profile Network for Performance Improvement

This example shows how to improve the performance of the deployed deep learning network, by identifying bottle neck layers from the profiler 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

To load the pretrained digits series network, enter:

```
snet = getDigitsNetwork();
```

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

```
snet.Layers
```

```
ans =
```

```
15x1 Layer array with layers:
```

1	'imageinput'	Image Input	28x28x1 images with 'zerocenter' normalization
2	'conv_1'	Convolution	8 3x3x1 convolutions with stride [1 1] and padding
3	'batchnorm_1'	Batch Normalization	Batch normalization with 8 channels
4	'relu_1'	ReLU	ReLU
5	'maxpool_1'	Max Pooling	2x2 max pooling with stride [2 2] and padding
6	'conv_2'	Convolution	16 3x3x8 convolutions with stride [1 1] and padding
7	'batchnorm_2'	Batch Normalization	Batch normalization with 16 channels
8	'relu_2'	ReLU	ReLU
9	'maxpool_2'	Max Pooling	2x2 max pooling with stride [2 2] and padding
10	'conv_3'	Convolution	32 3x3x16 convolutions with stride [1 1] and padding
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

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. For Ethernet interface, enter:

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

To use the JTAG interface, install Xilinx™ Vivado™ Design Suite 2019.2. Set up the path to your installed Xilinx Vivado executable if it is not already set up. For example, to set the toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.exe');
```

For JTAG interface, enter:

```
% hTarget = dlhdl.Target('Xilinx','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 digits 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 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.layer'
      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` 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.

```
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 FC Processor.
### FC Weights loaded. Current time is 28-Jun-2020 12:24:21
```

Load Example Image

Load the example image.

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

Run the Prediction

Execute the predict function of the `dlhdl.Workflow` object that has profile option set to 'on' to display the latency and throughput results.

```
[~, 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	Total
	-----	-----	-----	-----
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 Bottle Neck Layer

Remove the NumFrames, Total latency, and Frames/s from the profiler's results table. This includes removing the module level and network level profiler results. Retain only the network layer profiler results. Once the bottle neck layer has been identified display the bottle neck layer index, running time, and information.

```
speed('Network', :) = [];
speed('___conv_module', :) = [];
speed('___fc_module', :) = [];
speed = removevars(speed, {'NumFrames', 'Total Latency(cycles)', 'Frame/s'});
```

```
% then sort the profiler's results in descending ordering
speed = sortrows(speed, 'Latency(cycles)', 'descend');
```

```
% the first row in the profile table is the bottleneck layer. Thus the
% following
```

```
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
end
```

```
% displly the bottle neck layer index
dnnfpga.disp(['Bottleneck layer index is ', num2str(idx), '.']);

### Bottleneck layer index is 13.

% displly the bottle neck layer running time percentage
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.

% displly the bottle neck layer information
dnnfpga.disp('Bottleneck layer information: ');

### Bottleneck layer information:

disp(currLayer);

FullyConnectedLayer with properties:

    Name: 'fc'

Hyperparameters
    InputSize: 1568
    OutputSize: 10

Learnable Parameters
    Weights: [10x1568 single]
    Bias: [10x1 single]

Show all properties
```

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.

Prerequisites

- Xilinx™ Vivado™ Design Suite 2019.2
- 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`.

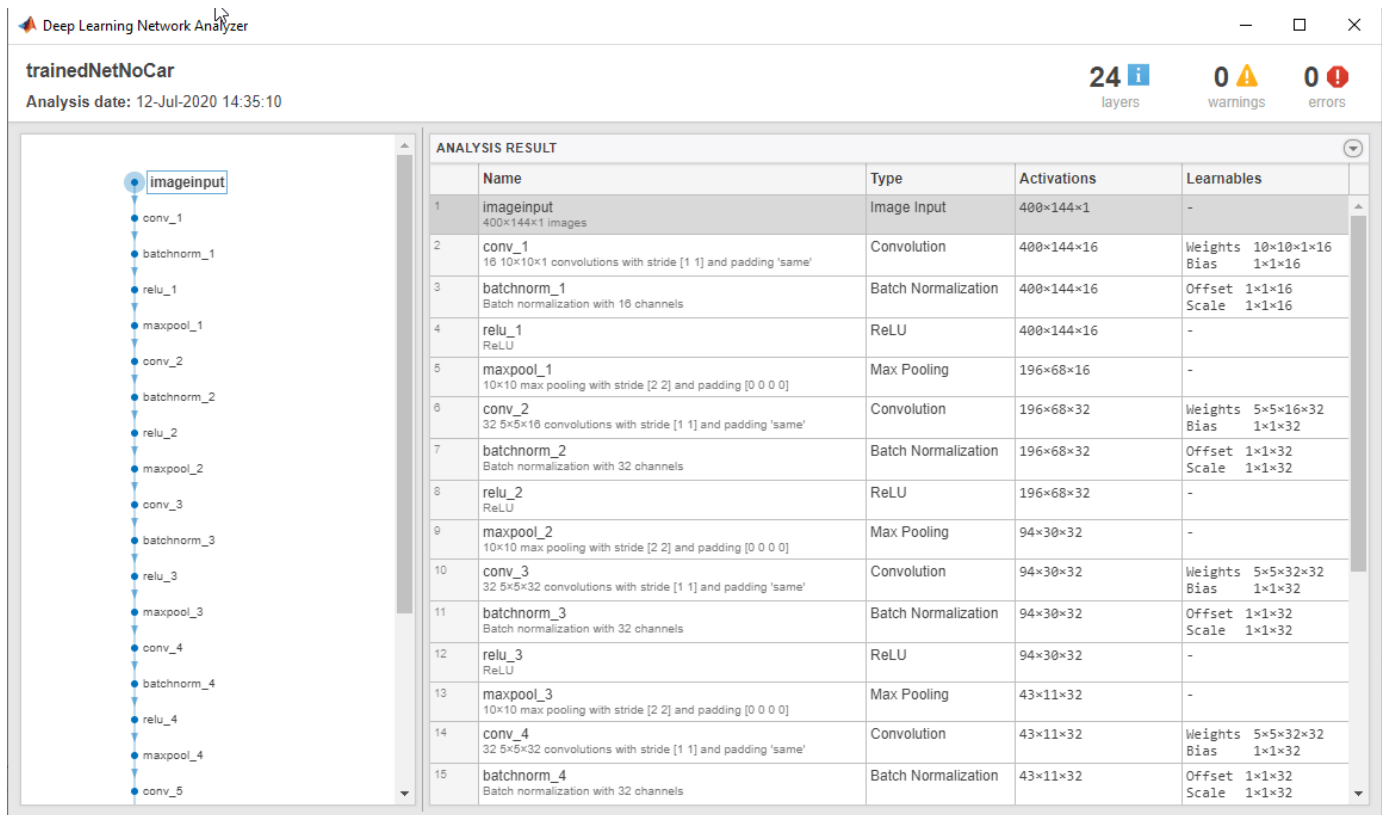
Load Data and Network

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);
```



Set up HDL Toolpath

Set up the path to your installed Xilinx™ Vivado™ Design Suite 2019.2 executable if it is not already set up. For example, to set the toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Vivado\2019.2\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.layer'
      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 target
### Deep learning network programming has been skipped as the same network is already loaded on the target
```

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	TotalTime
Network	9430692	0.04287	1	9.000000
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 = 1x5 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	Total
	-----	-----	-----	-----
Network	9446929	0.04294	5	47
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		

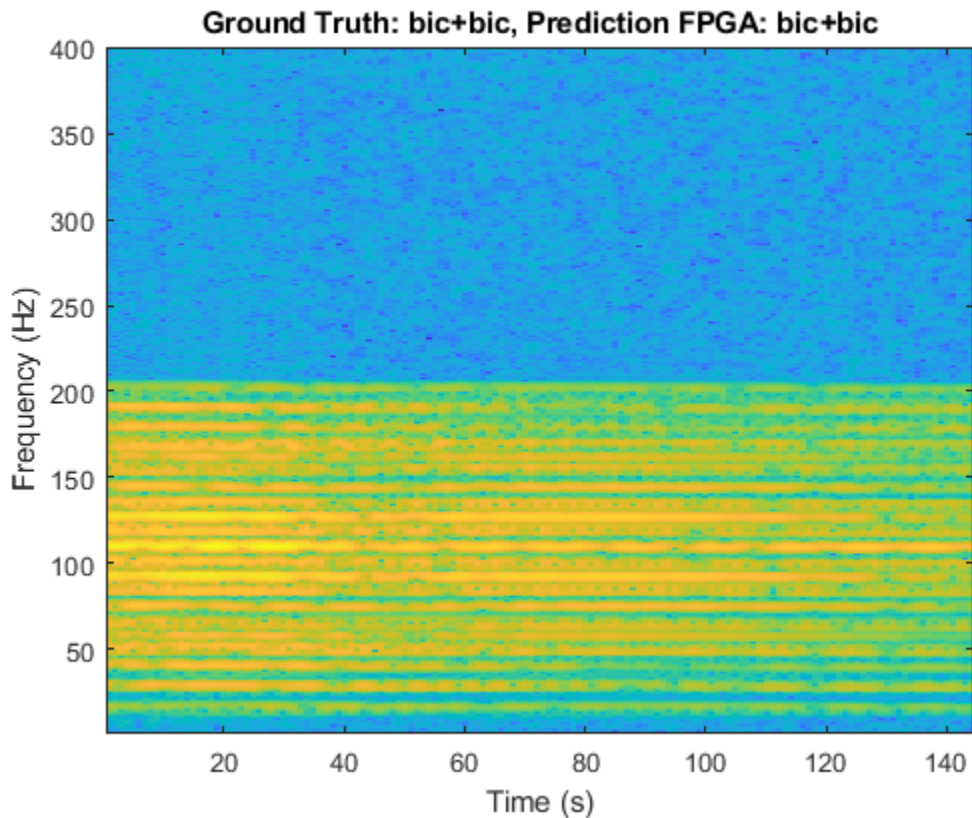
```

        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(predTestLabelBatch(k)))
    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.

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

- 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 18.1. 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\18.1\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	Total
	-----	-----	-----	-----
Network	10182024	0.06788	1	10
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.

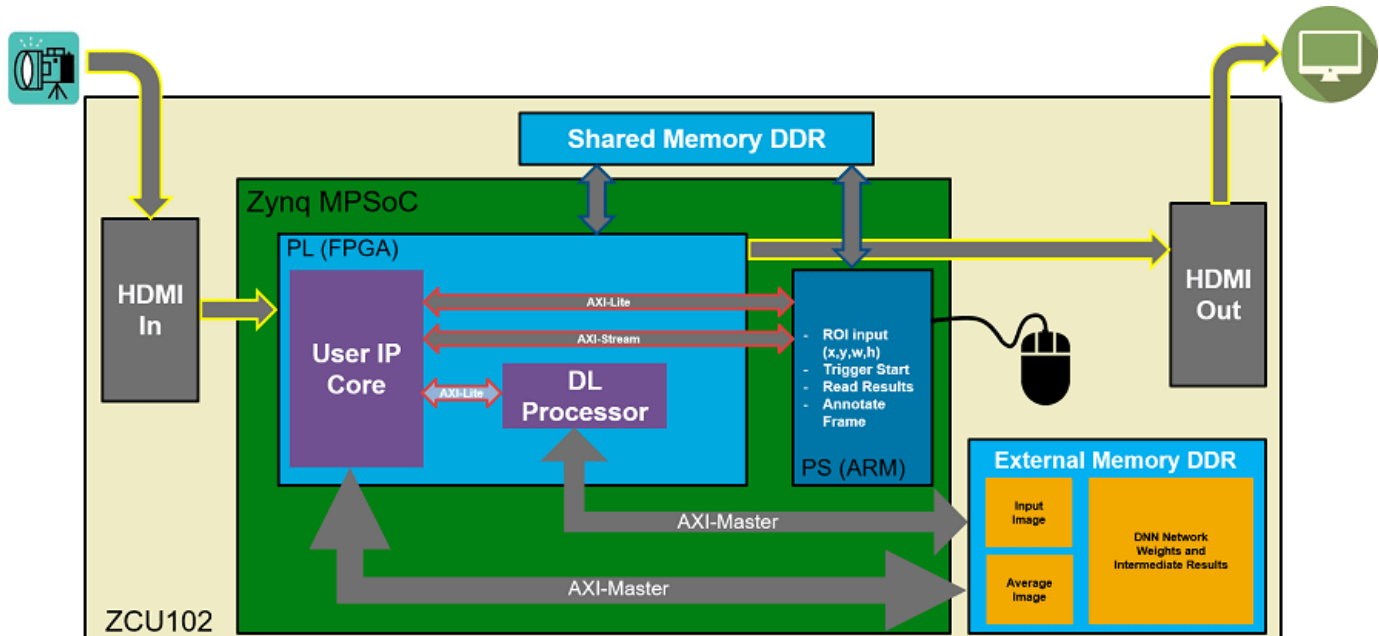
See Also

More About

- activations

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 10-67.

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 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 9-4. 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_grj_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_I_CtotalLength_IP0	inport	uint32	AXI4	x"22C"	
nc_I_Coffset_IP0	inport	uint32	AXI4	x"230"	
nc_I_CtotalLength_Conv	inport	uint32	AXI4	x"234"	
nc_I_Coffset_Conv	inport	uint32	AXI4	x"238"	

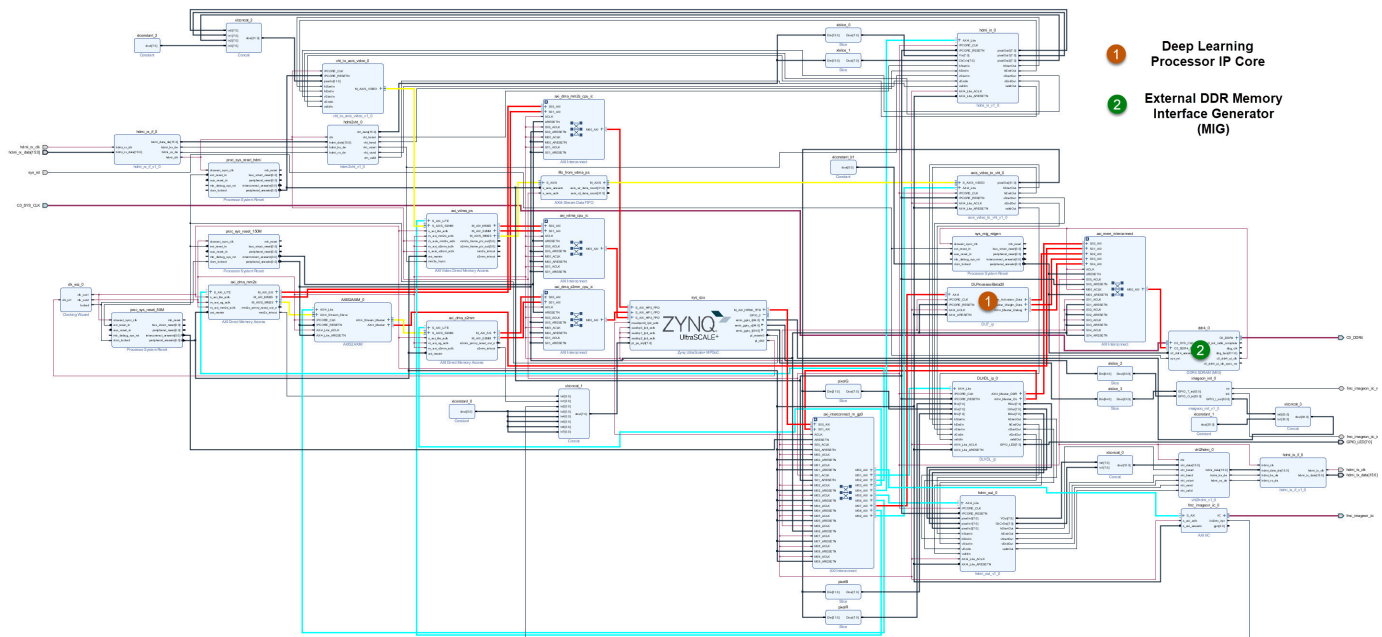
The HTML report contains a description of the deep learning processor IP core, instructions for using the core and integrating the core into your Vivado™ reference design, and a list of AXI4 registers. You will need the AXI4 register list to enter addresses into the Vivado™ Address Mapping tool. For more information about the AXI4 registers, see “Deep Learning Processor Register Map” on page 12-9.

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:

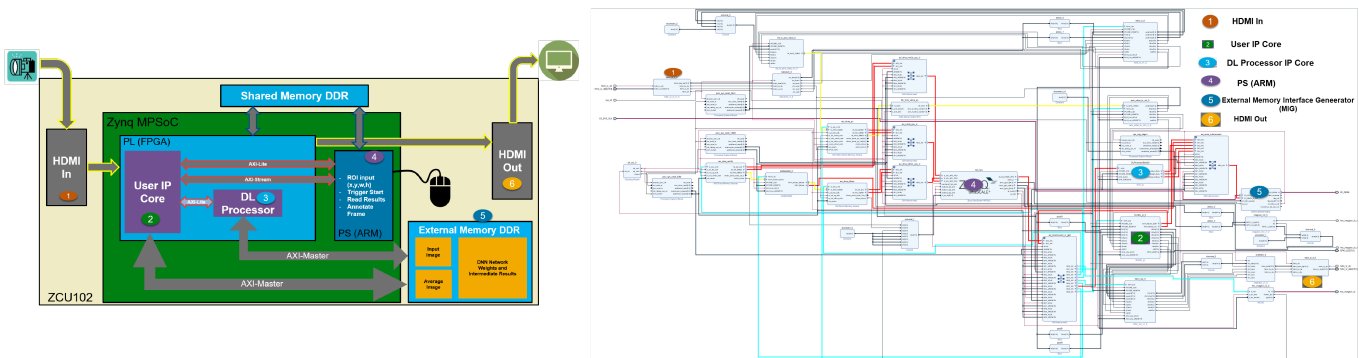
- Connect the generated deep learning processor IP core AXI4 slave interface to an AXI4 master device such as a JTAG AXI master IP core or a Zynq™ processing system (PS). Use the AXI4 master device to communicate with the deep learning processor IP core.
- Connect the vendor provided external memory 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 `d\hdl.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 10-67.

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:

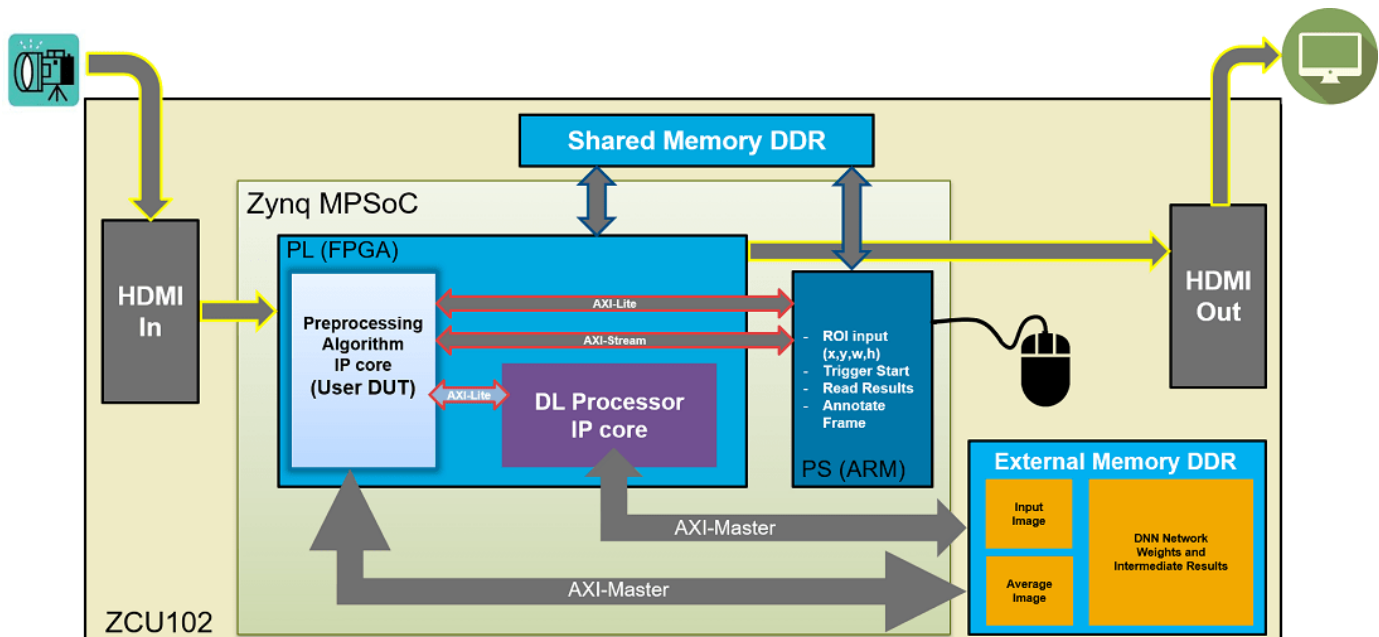
- Deep Learning HDL Toolbox™
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- Deep Learning Toolbox™
- HDL Coder™
- Simulink™

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.

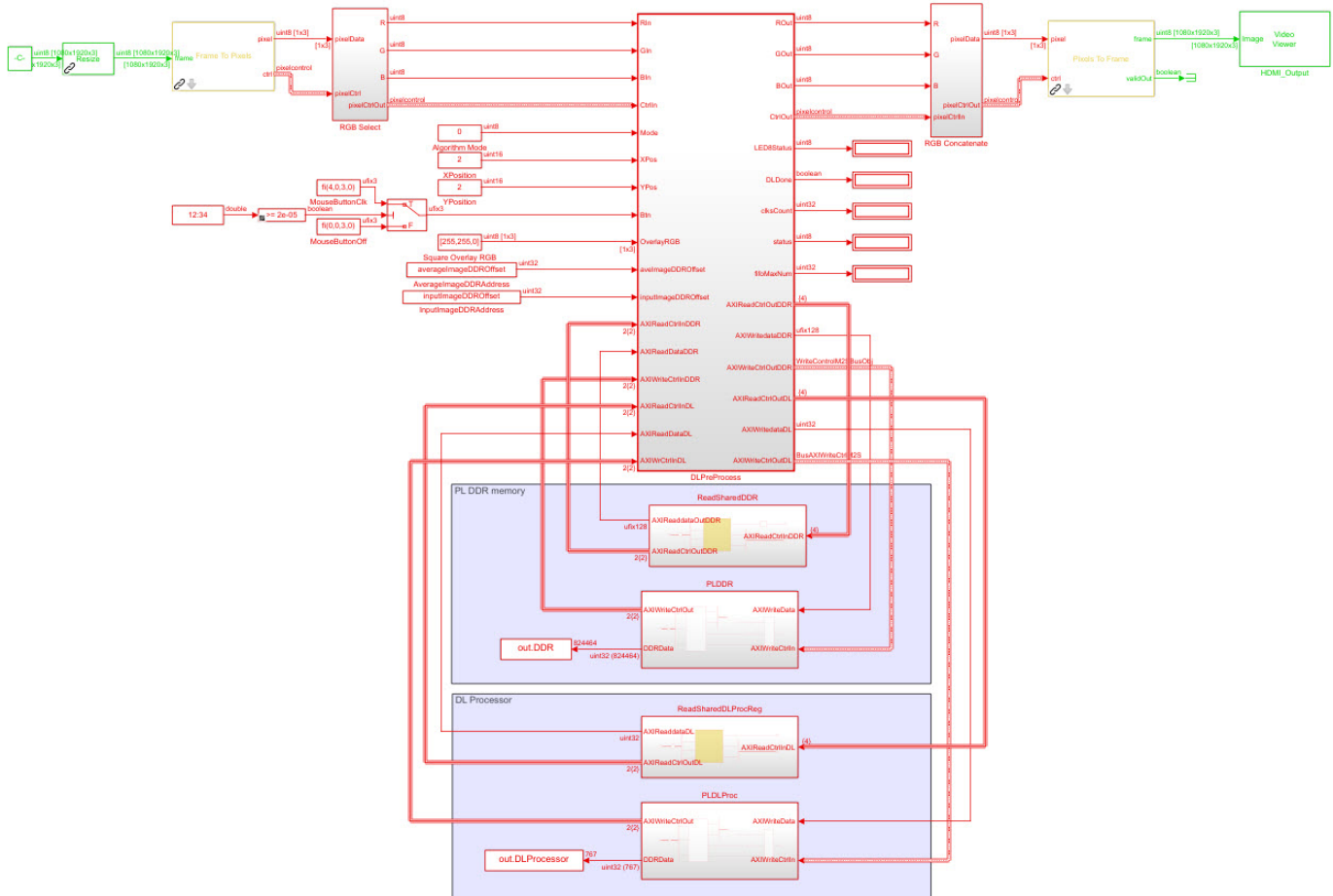
Select and Resize the Region of Interest

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 modelled 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.

Deep Learning Pre-Process Hardware Algorithm Target Model



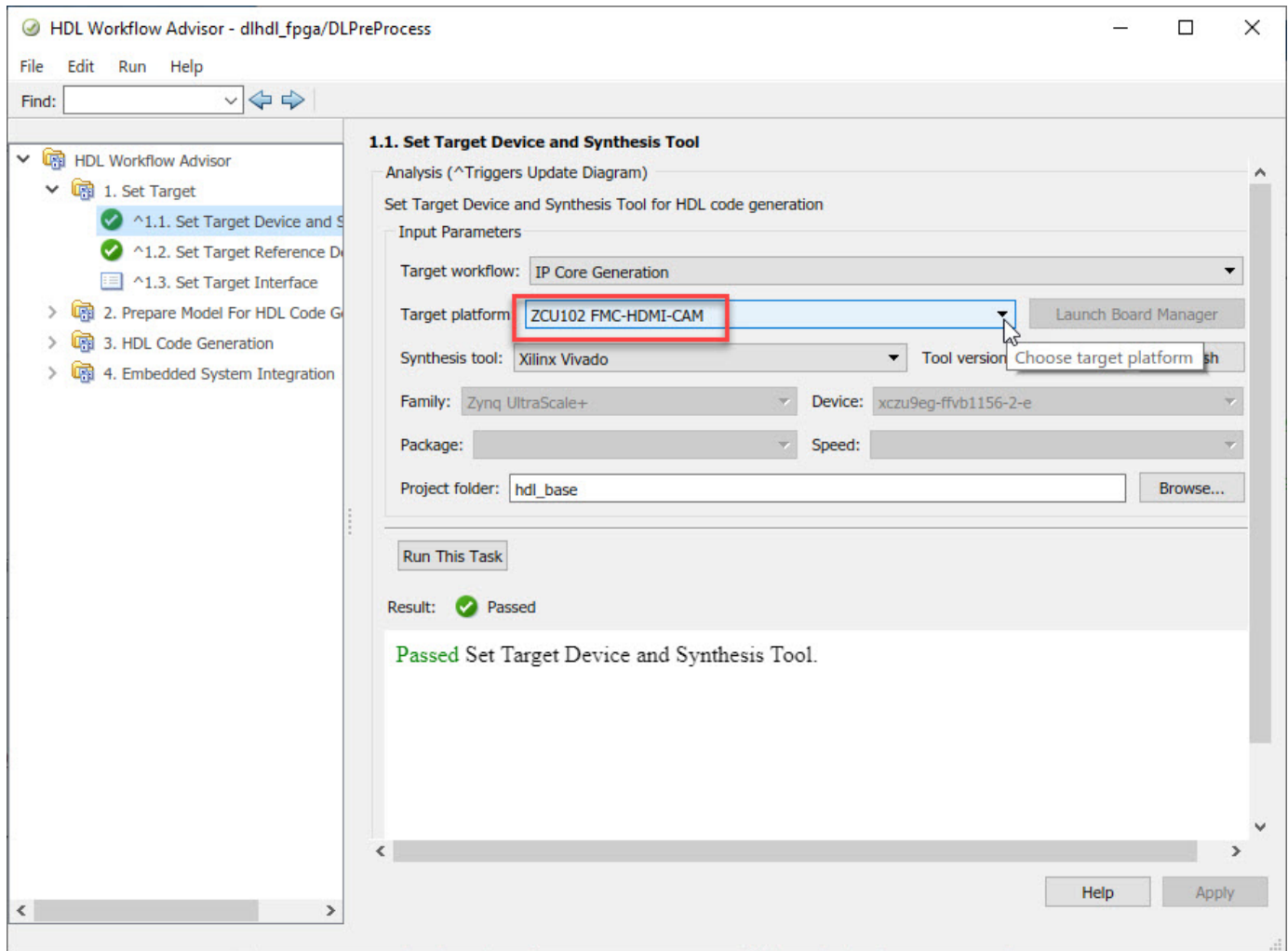
Generate Preprocessing Logic HDL IP Core

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 10-62.

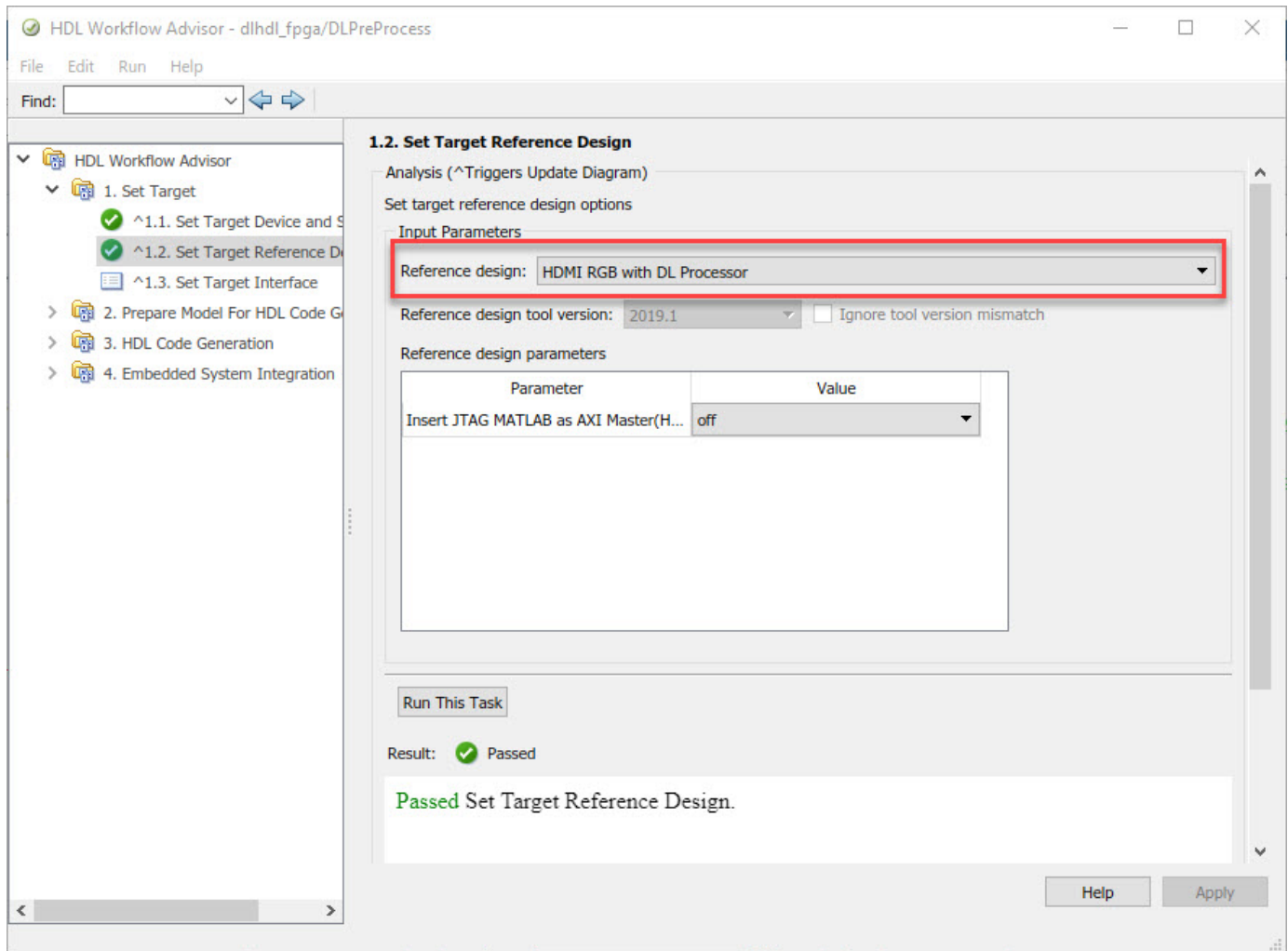
Start the HDL Coder HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy the preprocessing logic model on Zynq hardware. This workflow is the standard HDL Coder workflow. In this example the only difference is that this reference design contains the generated deep learning 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**.

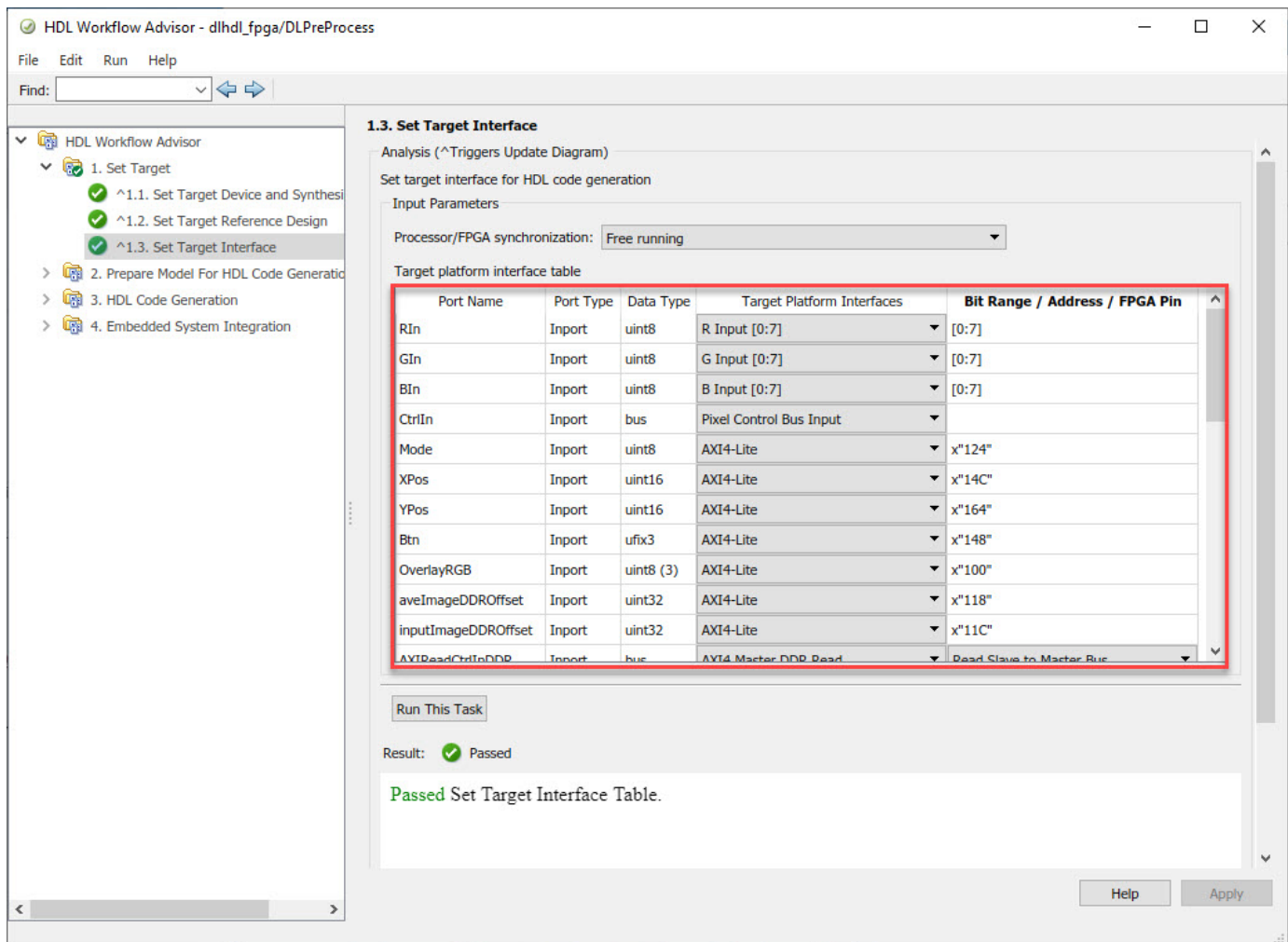
In Task 1.1, **IP Core Generation** is selected for **Target workflow** and **ZCU102-FMC-HDMI-CAM** is selected for **Target platform**.



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.



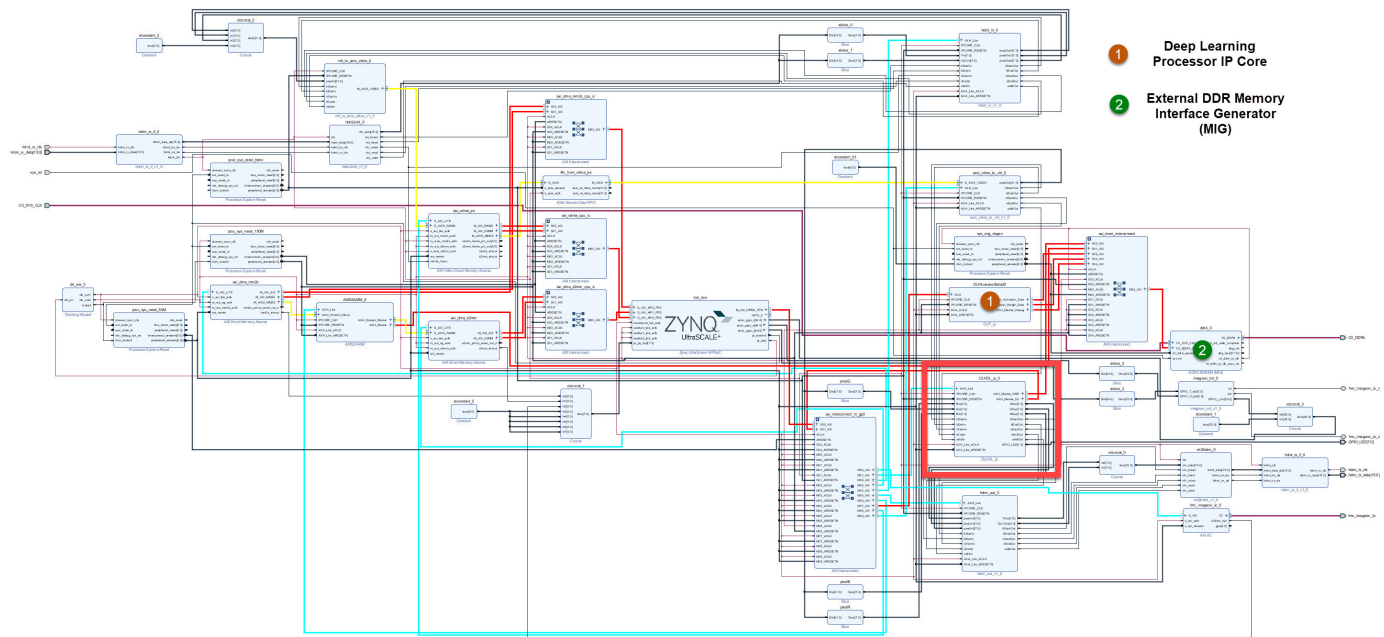
2. Right-click Task 3.2, **Generate RTL Code and IP Core**, and then select **Run to Selected Task**. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

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 10-62.

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.

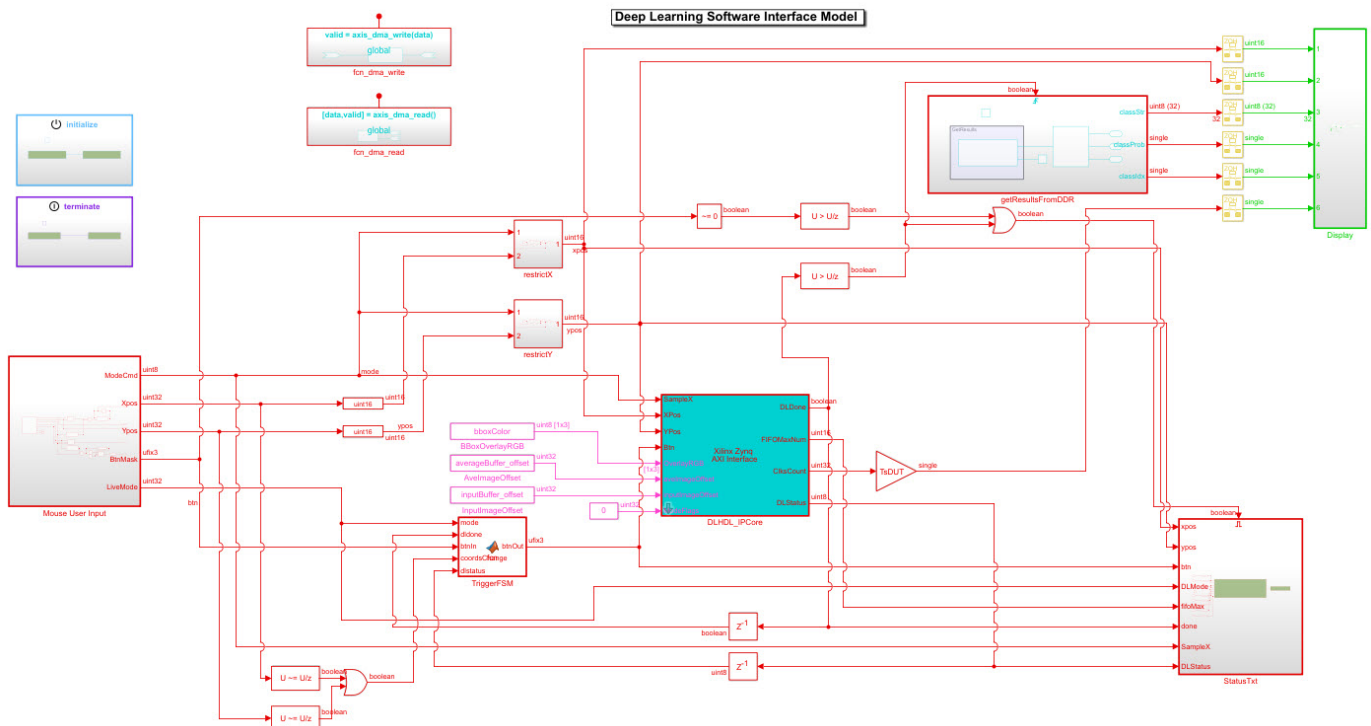
Deploy and Validate the Integrated Reference Design

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 `d1hdl.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.

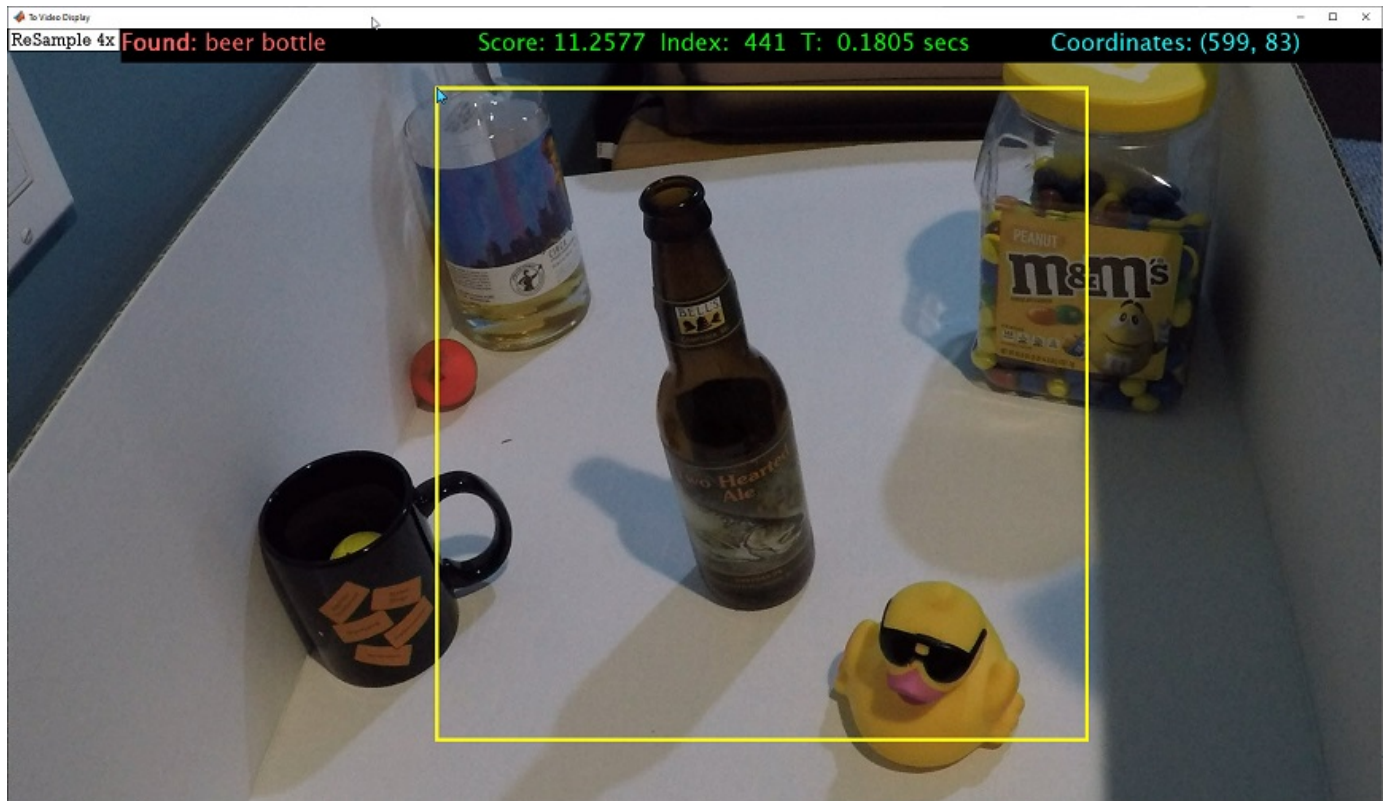
The deep learning processor IP core sends the result back to the external DDR memory. The software model running on the ARM processor retrieves the prediction result and annotates the prediction on the output video stream. This screenshot shows that you can read the ARM processor prediction result by 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.

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 2019.2. To set the Xilinx Vivado toolpath, enter:

```
%hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'D:/share/apps/HDLTools/Vivado/2019.2');
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.layer'
      offset_name      offset_address      allocated_space
```

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: [1x1 struct]
    LayerConfigs: [1x1 struct]
    NetConfigs: [1x1 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 target
### Deep learning network programming has been skipped as the same network is already loaded on the target
```

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	Total
	-----	-----	-----	-----
Network	2813005	0.01279	1	2813005
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 = 1x4
```

```
    56    56     1    64
```

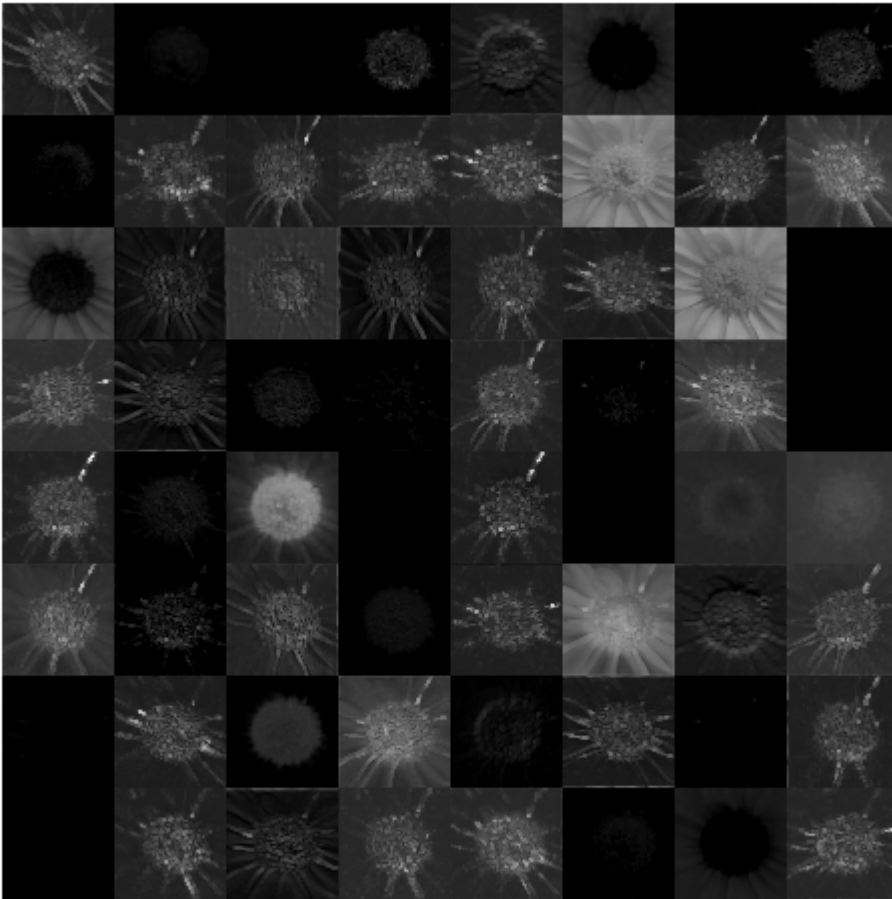
The third dimension in the input to `imtile` function represents the image color. Set the third dimension to size 1 because the activation signals in this example are scalars and do not include color. The fourth dimension indexes the channel.

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.

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

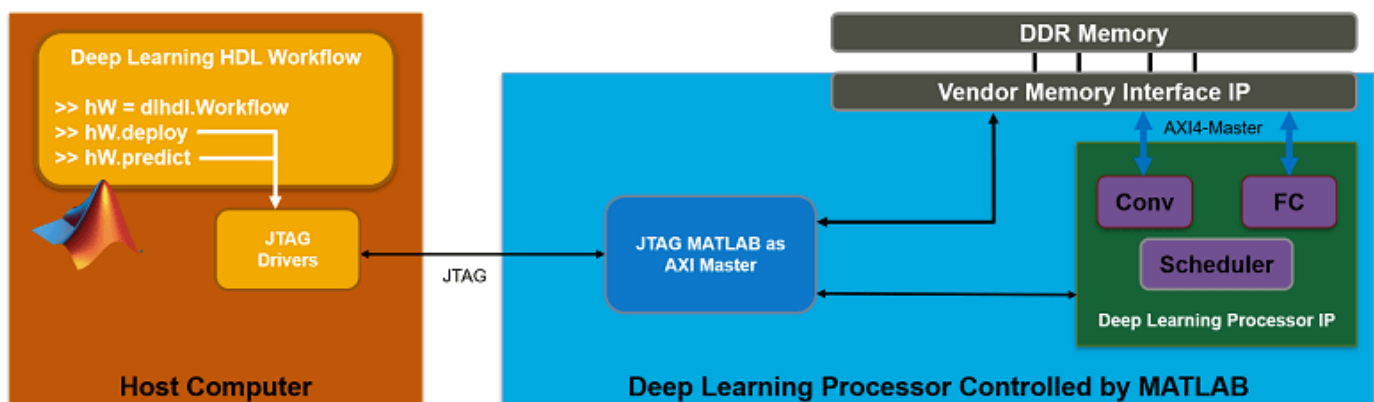
There are two hardware interfaces for establishing a connection between the host computer and FPGA board: JTAG and Ethernet.

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.

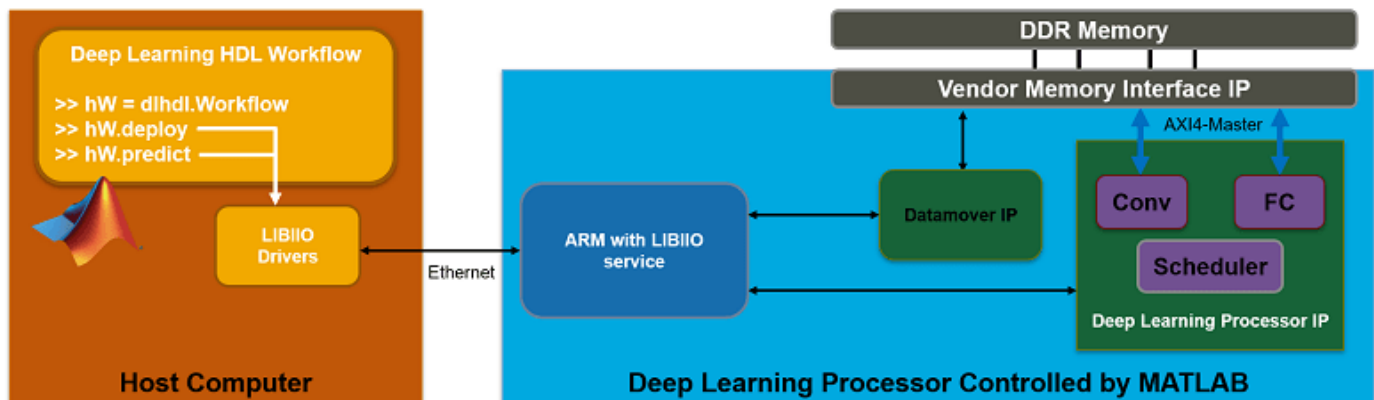


Ethernet Interface

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.



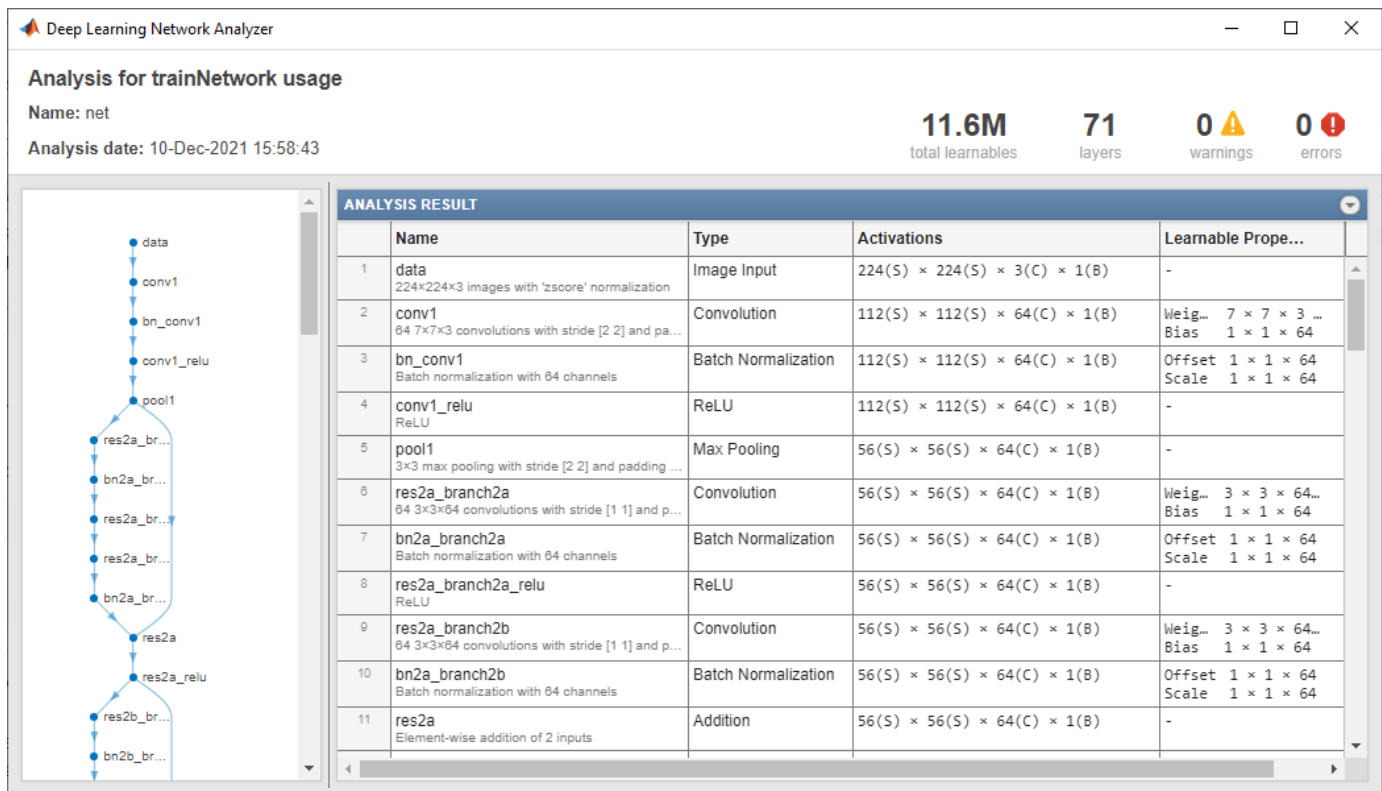
Load and Compile Deep Learning Network

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 v
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 wi
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 wi
15	'res2b_branch2a_relu'	ReLU	ReLU

16	'res2b_branch2b'	Convolution	64 3×3×64 convolutions
17	'bn2b_branch2b'	Batch Normalization	Batch normalization with
18	'res2b'	Addition	Element-wise addition of
19	'res2b_relu'	ReLU	ReLU
20	'res3a_branch2a'	Convolution	128 3×3×64 convolutions
21	'bn3a_branch2a'	Batch Normalization	Batch normalization with
22	'res3a_branch2a_relu'	ReLU	ReLU
23	'res3a_branch2b'	Convolution	128 3×3×128 convolutions
24	'bn3a_branch2b'	Batch Normalization	Batch normalization with
25	'res3a'	Addition	Element-wise addition of
26	'res3a_relu'	ReLU	ReLU
27	'res3a_branch1'	Convolution	128 1×1×64 convolutions
28	'bn3a_branch1'	Batch Normalization	Batch normalization with
29	'res3b_branch2a'	Convolution	128 3×3×128 convolutions
30	'bn3b_branch2a'	Batch Normalization	Batch normalization with
31	'res3b_branch2a_relu'	ReLU	ReLU
32	'res3b_branch2b'	Convolution	128 3×3×128 convolutions
33	'bn3b_branch2b'	Batch Normalization	Batch normalization with
34	'res3b'	Addition	Element-wise addition of
35	'res3b_relu'	ReLU	ReLU
36	'res4a_branch2a'	Convolution	256 3×3×128 convolutions
37	'bn4a_branch2a'	Batch Normalization	Batch normalization with
38	'res4a_branch2a_relu'	ReLU	ReLU
39	'res4a_branch2b'	Convolution	256 3×3×256 convolutions
40	'bn4a_branch2b'	Batch Normalization	Batch normalization with
41	'res4a'	Addition	Element-wise addition of
42	'res4a_relu'	ReLU	ReLU
43	'res4a_branch1'	Convolution	256 1×1×128 convolutions
44	'bn4a_branch1'	Batch Normalization	Batch normalization with
45	'res4b_branch2a'	Convolution	256 3×3×256 convolutions
46	'bn4b_branch2a'	Batch Normalization	Batch normalization with
47	'res4b_branch2a_relu'	ReLU	ReLU
48	'res4b_branch2b'	Convolution	256 3×3×256 convolutions
49	'bn4b_branch2b'	Batch Normalization	Batch normalization with
50	'res4b'	Addition	Element-wise addition of
51	'res4b_relu'	ReLU	ReLU
52	'res5a_branch2a'	Convolution	512 3×3×256 convolutions
53	'bn5a_branch2a'	Batch Normalization	Batch normalization with
54	'res5a_branch2a_relu'	ReLU	ReLU
55	'res5a_branch2b'	Convolution	512 3×3×512 convolutions
56	'bn5a_branch2b'	Batch Normalization	Batch normalization with
57	'res5a'	Addition	Element-wise addition of
58	'res5a_relu'	ReLU	ReLU
59	'res5a_branch1'	Convolution	512 1×1×256 convolutions
60	'bn5a_branch1'	Batch Normalization	Batch normalization with
61	'res5b_branch2a'	Convolution	512 3×3×512 convolutions
62	'bn5b_branch2a'	Batch Normalization	Batch normalization with
63	'res5b_branch2a_relu'	ReLU	ReLU
64	'res5b_branch2b'	Convolution	512 3×3×512 convolutions
65	'bn5b_branch2b'	Batch Normalization	Batch normalization with
66	'res5b'	Addition	Element-wise addition of
67	'res5b_relu'	ReLU	ReLU
68	'pool5'	2-D Global Average Pooling	2-D global average pooling
69	'fc1000'	Fully Connected	1000 fully connected layers
70	'prob'	Softmax	softmax
71	'ClassificationLayer_predictions'	Classification Output	crossentropy with 't

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', and
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software.
### 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 2019.2 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\2019.2\bin\vivado.)
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)
```

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');
```

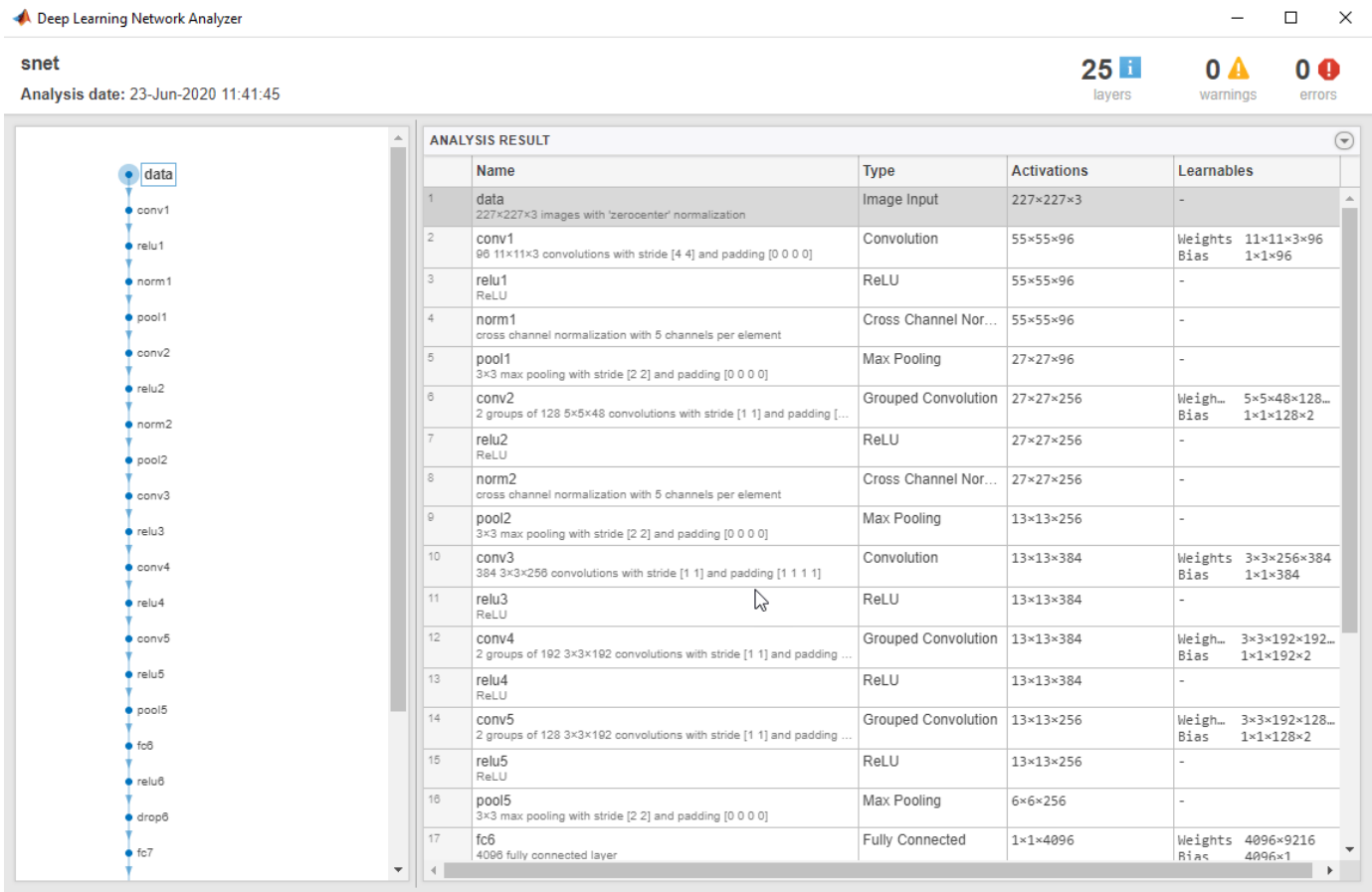
Load Pretrained Network

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)
```



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');

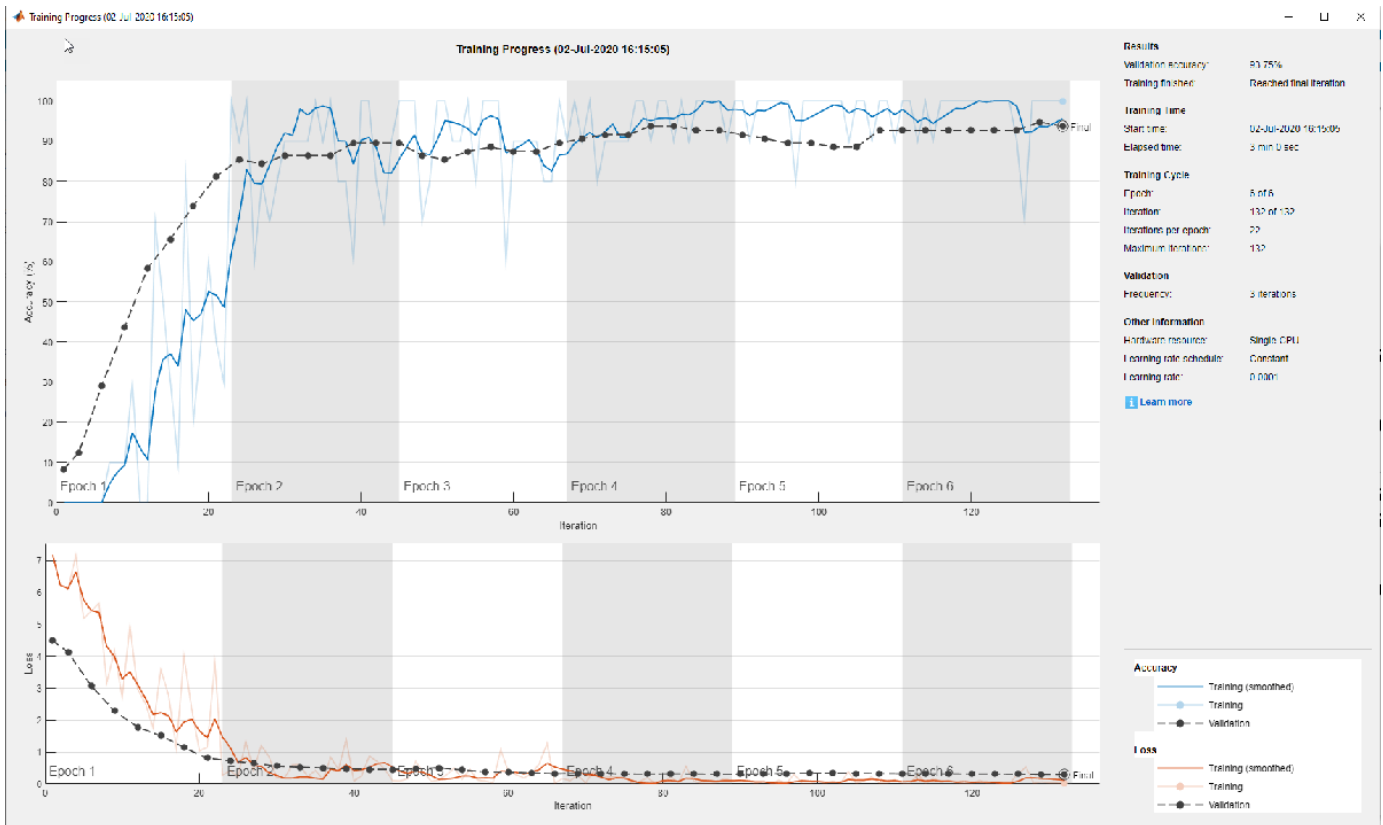
```

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);

```

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 Libraries

Load Pretrained Series Network

To load the pretrained series network LogoNet, enter:

```
snet = getLogoNetwork;
```

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: '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 K
  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' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is impl

```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
Network	39926006	0.18148	1	39926006
__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

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: 'on'
      ConvThreadNumber: 64
      InputMemorySize: [227 227 3]
      OutputMemorySize: [227 227 3]
      FeatureSizeLimit: 2048

    Processing Module "fc"
      ModuleGeneration: 'on'
      SoftmaxBlockGeneration: 'off'
      FCThreadNumber: 16
      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: 'int8'

    System Level Properties
      TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation K
      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:

```

dlquantObj = dlquantizer(snet, 'ExecutionEnvironment', 'FPGA');
Image = imageDatastore('heineken.png', 'Labels', 'Heineken');
dlquantObj.calibrate(Image);

```

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)

```

Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
 ### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is impl

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	14306694	0.04769	1	14306694
conv_1	3477191	0.01159		
maxpool_1	1876680	0.00626		
conv_2	2932291	0.00977		
maxpool_2	723536	0.00241		
conv_3	2611391	0.00870		
maxpool_3	882544	0.00294		
conv_4	641788	0.00214		
maxpool_4	14025	0.00005		
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 21.2 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\2019.2\bin\vivado.l
% 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 “Obtain Prediction Results for Quantized LogoNet Network”.

Deploy Quantized Network Example

This example shows how to train, compile, and deploy a `dlhdl.Workflow` object that has quantized ResNet-18 as the network object by using the Deep Learning HDL Toolbox™ Support Package for Xilinx FPGA and SoC. 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 from the target device.

Required Products

For this example, you need:

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model Quantization Library
- Deep Learning Toolbox Model for ResNet-18 Network
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- MATLAB Coder Interface for Deep Learning Libraries.

Load Pretrained DAG Network

To load the pretrained DAG network ResNet-18, enter:

```
net = resnet18;
```

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

```
analyzeNetwork(net);
```

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 = net.Layers(1).InputSize;
```

```
inputSize = 1×3
```

```
227 227 3
```

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;  
newDir = fullfile(matlabroot, 'examples', 'deeplearning_shared', 'data', 'logos_dataset.zip');  
copyfile(newDir, curDir, 'f');  
  
unzip('logos_dataset.zip');  
  
imds = imageDatastore('logos_dataset', ...  
    'IncludeSubfolders', true, ...  
    'LabelSource', 'foldernames');  
  
[imdsTrain, imdsValidation] = splitEachLabel(imds, 0.7, 'randomized');
```

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 the layers, except the last three layers, from the pretrained network.

Extract the layer graph from the trained network.

```
lgraph = layerGraph(net)

lgraph =
  LayerGraph with properties:
    Layers: [71x1 nnet.cnn.layer.Layer]
    Connections: [78x2 table]
    InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}
```

Remove 'fc1000', 'prob' and 'ClassificationLayer_predictions' layers from the `lgraph`.

```
layers = net.SortedLayers;
for i = 0:2
    lgraph = removeLayers(lgraph, layers(end-i).Name);
end
```

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. Set the fully connected layer to have the same size as the number of classes in the new data.

```
numClasses = numel(categories(imdsTrain.Labels));
numClasses = 32
```

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', 'newFC'),
    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 227-by-227-by-3, but the images in the image datastore 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);  
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.

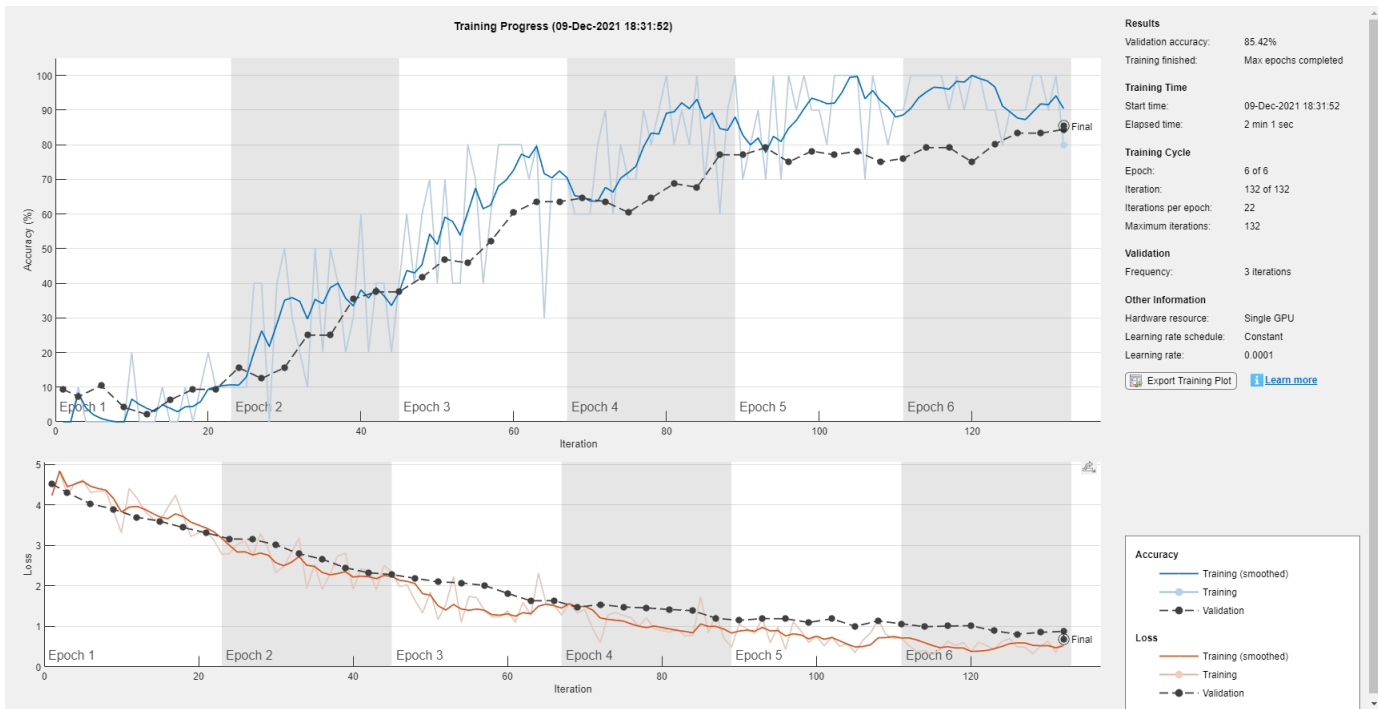
```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

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. 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 supported GPU device. For more information, see “GPU Support by Release” (Parallel Computing Toolbox)). 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 `dlquantizer` object and specify the network to quantize. Specify the execution environment as FPGA.

```
dlQuantObj = dlquantizer(netTransfer, 'ExecutionEnvironment', 'FPGA');
```

Calibrate Quantized Network

The `dlquantizer` object uses calibration data to collect 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 predicted by the LogoNet network. Expedite the calibration process by reducing the calibration data set to 20 images.

```
imageData = imageDatastore(fullfile(curDir, 'logos_dataset'), ...
    'IncludeSubfolders', true, 'FileExtensions', '.JPG', 'LabelSource', 'foldernames');
imageData_reduced = imageData.subset(1:20);
dlQuantObj.calibrate(imageData_reduced)
```

ans=95x5 table

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv1_Weights' }	{'conv1' }	"Weights"	-0.52595
{'conv1_Bias' }	{'conv1' }	"Bias"	-0.66142
{'res2a_branch2a_Weights' }	{'res2a_branch2a' }	"Weights"	-0.36239
{'res2a_branch2a_Bias' }	{'res2a_branch2a' }	"Bias"	-0.83058
{'res2a_branch2b_Weights' }	{'res2a_branch2b' }	"Weights"	-0.80143
{'res2a_branch2b_Bias' }	{'res2a_branch2b' }	"Bias"	-1.2691
{'res2b_branch2a_Weights' }	{'res2b_branch2a' }	"Weights"	-0.26073

```

{'res2b_branch2a_Bias' } {'res2b_branch2a'} "Bias" -1.0012
{'res2b_branch2b_Weights' } {'res2b_branch2b'} "Weights" -1.1361
{'res2b_branch2b_Bias' } {'res2b_branch2b'} "Bias" -1.1981
{'res3a_branch2a_Weights' } {'res3a_branch2a'} "Weights" -0.13934
{'res3a_branch2a_Bias' } {'res3a_branch2a'} "Bias" -0.54418
{'res3a_branch2b_Weights' } {'res3a_branch2b'} "Weights" -0.49925
{'res3a_branch2b_Bias' } {'res3a_branch2b'} "Bias" -0.66837
{'res3a_branch1_Weights' } {'res3a_branch1' } "Weights" -0.63797
{'res3a_branch1_Bias' } {'res3a_branch1' } "Bias" -1.0594
:

```

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 2020.1. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.1\bin\vivado.
```

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 Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the class, an instance of the `dlquantizer` object, the bitstream name, and the target information are specified. Specify `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 Xilinx ZCU102 SOC board and the bitstream uses the `int8` data type.

```
hW = dlhdl.Workflow('Network', dlQuantObj, 'Bitstream', 'zcu102_int8', 'Target', hTarget);
```

Compile the Quantized DAGNetwork

To compile the quantized ResNet-18 DAG network, run the `compile` function of the `dlhdl.Workflow` object.

```
dn = hW.compile
```

```

### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_int8.
### The network includes the following layers:
 1 'data' Image Input 224x224x3 images with 'zscore' norm
 2 'conv1' Convolution 64 7x7x3 convolutions with stride
 3 'bn_conv1' Batch Normalization Batch normalization with 64 channel
 4 'conv1_relu' ReLU ReLU
 5 'pool1' Max Pooling 3x3 max pooling with stride [2 2]
 6 'res2a_branch2a' Convolution 64 3x3x64 convolutions with stride
 7 'bn2a_branch2a' Batch Normalization Batch normalization with 64 channel
 8 'res2a_branch2a_relu' ReLU ReLU
 9 'res2a_branch2b' Convolution 64 3x3x64 convolutions with stride
10 'bn2a_branch2b' Batch Normalization Batch normalization with 64 channel

```

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
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
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	'res5a_branch2a'	Convolution	512 3×3×256 convolutions with stride
53	'bn5a_branch2a'	Batch Normalization	Batch normalization with 512 channels
54	'res5a_branch2a_relu'	ReLU	ReLU
55	'res5a_branch2b'	Convolution	512 3×3×512 convolutions with stride
56	'bn5a_branch2b'	Batch Normalization	Batch normalization with 512 channels
57	'res5a'	Addition	Element-wise addition of 2 inputs
58	'res5a_relu'	ReLU	ReLU
59	'res5a_branch1'	Convolution	512 1×1×256 convolutions with stride
60	'bn5a_branch1'	Batch Normalization	Batch normalization with 512 channels
61	'res5b_branch2a'	Convolution	512 3×3×512 convolutions with stride
62	'bn5b_branch2a'	Batch Normalization	Batch normalization with 512 channels
63	'res5b_branch2a_relu'	ReLU	ReLU
64	'res5b_branch2b'	Convolution	512 3×3×512 convolutions with stride
65	'bn5b_branch2b'	Batch Normalization	Batch normalization with 512 channels
66	'res5b'	Addition	Element-wise addition of 2 inputs
67	'res5b_relu'	ReLU	ReLU
68	'pool5'	2-D Global Average Pooling	2-D global average pooling

```

69  'newFC'           Fully Connected           32 fully connected layer
70  'newProb'         Softmax                   softmax
71  'newClassOutput' Classification Output      crossentropyex with 'adidas' and 3

### 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 'newProb' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in softwa
### Notice: The layer 'newClassOutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is in
### 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: 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"           "4.0 MB"
"ConvWeightDataOffset"    "0x03400000"           "16.0 MB"
"FCWeightDataOffset"      "0x04400000"           "4.0 MB"
"EndOffset"               "0x04800000"           "Total: 72.0 MB"

### Network compilation complete.

dn = struct with fields:
      weights: [1x1 struct]
      instructions: [1x1 struct]
      registers: [1x1 struct]
      syncInstructions: [1x1 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, 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 target
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 09-Dec-2021 18:36:39
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 09-Dec-2021 18:36:39
```

Load Example Images and Run the Prediction

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

```
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    I = imresize(I,[224 224]);
    imshow(I)
    [prediction, speed] = hw.predict(single(I),'Profile','on');
    [val, index] = max(prediction);
    netTransfer.Layers(end).ClassNames{index}
    label = netTransfer.Layers(end).ClassNames{index}
    title(string(label));
end

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

Deep Learning Processor Profiler Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	TotalTime
	-----	-----	-----	-----
Network	7335952	0.02934	1	7335952
conv1	1115480	0.00446		
pool1	238029	0.00095		
res2a_branch2a	269834	0.00108		
res2a_branch2b	269956	0.00108		
res2a	88905	0.00036		
res2b_branch2a	269794	0.00108		
res2b_branch2b	269965	0.00108		
res2b	88783	0.00036		

```

res3a_branch1      156139      0.00062
res3a_branch2a    227324      0.00091
res3a_branch2b    245055      0.00098
res3a              44462       0.00018
res3b_branch2a    244852      0.00098
res3b_branch2b    245048      0.00098
res3b              44426       0.00018
res4a_branch1     135525      0.00054
res4a_branch2a    136187      0.00054
res4a_branch2b    237212      0.00095
res4a              22312       0.00009
res4b_branch2a    236600      0.00095
res4b_branch2b    237466      0.00095
res4b              22542       0.00009
res5a_branch1     311891      0.00125
res5a_branch2a    311873      0.00125
res5a_branch2b    596194      0.00238
res5a              11201       0.00004
res5b_branch2a    595857      0.00238
res5b_branch2b    596713      0.00239
res5b              11431       0.00005
pool5              36976       0.00015
newFC              17733       0.00007

```

* The clock frequency of the DL processor is: 250MHz

```
ans =
'becks'
```

```
label =
'becks'
```

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

Deep Learning Processor Profiler Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	7336472	0.02935	1	7336472
conv1	1115736	0.00446		
pool1	237938	0.00095		
res2a_branch2a	269787	0.00108		
res2a_branch2b	270062	0.00108		
res2a	88865	0.00036		
res2b_branch2a	269710	0.00108		
res2b_branch2b	269870	0.00108		
res2b	88975	0.00036		
res3a_branch1	156178	0.00062		
res3a_branch2a	227565	0.00091		
res3a_branch2b	245130	0.00098		
res3a	44486	0.00018		
res3b_branch2a	244733	0.00098		
res3b_branch2b	244875	0.00098		
res3b	44486	0.00018		
res4a_branch1	135725	0.00054		
res4a_branch2a	136247	0.00054		
res4a_branch2b	236972	0.00095		
res4a	22382	0.00009		

```

res4b_branch2a      236891      0.00095
res4b_branch2b      237046      0.00095
res4b                22402       0.00009
res5a_branch1       312061      0.00125
res5a_branch2a      311738      0.00125
res5a_branch2b      596238      0.00238
res5a               11261       0.00005
res5b_branch2a      595867      0.00238
res5b_branch2b      596768      0.00239
res5b               11351       0.00005
pool5               36999       0.00015
newFC               17941       0.00007

```

* The clock frequency of the DL processor is: 250MHz

```
ans =
'nvidia'
```

```
label =
'nvidia'
```

```
### Finished writing input activations.
```

```
### Running single input activation.
```

Deep Learning Processor Profiler Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	7335649	0.02934	1	7335649
conv1	1115615	0.00446		
pool1	237521	0.00095		
res2a_branch2a	269761	0.00108		
res2a_branch2b	270039	0.00108		
res2a	88844	0.00036		
res2b_branch2a	269603	0.00108		
res2b_branch2b	269901	0.00108		
res2b	88855	0.00036		
res3a_branch1	156360	0.00063		
res3a_branch2a	227646	0.00091		
res3a_branch2b	245061	0.00098		
res3a	44526	0.00018		
res3b_branch2a	244836	0.00098		
res3b_branch2b	244944	0.00098		
res3b	44566	0.00018		
res4a_branch1	135820	0.00054		
res4a_branch2a	136251	0.00055		
res4a_branch2b	236828	0.00095		
res4a	22352	0.00009		
res4b_branch2a	237023	0.00095		
res4b_branch2b	236932	0.00095		
res4b	22392	0.00009		
res5a_branch1	311901	0.00125		
res5a_branch2a	311751	0.00125		
res5a_branch2b	596252	0.00239		
res5a	11281	0.00005		
res5b_branch2a	595829	0.00238		
res5b_branch2b	596743	0.00239		
res5b	11249	0.00004		
pool5	36913	0.00015		

```

    newFC                17867                0.00007
* The clock frequency of the DL processor is: 250MHz

ans =
'google'

label =
'google'

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

```

Deep Learning Processor Profiler Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	7336269	0.02935	1	7336269
conv1	1115457	0.00446		
pool1	238045	0.00095		
res2a_branch2a	269786	0.00108		
res2a_branch2b	269928	0.00108		
res2a	88895	0.00036		
res2b_branch2a	269765	0.00108		
res2b_branch2b	269928	0.00108		
res2b	88825	0.00036		
res3a_branch1	156232	0.00062		
res3a_branch2a	227320	0.00091		
res3a_branch2b	245058	0.00098		
res3a	44493	0.00018		
res3b_branch2a	244799	0.00098		
res3b_branch2b	245040	0.00098		
res3b	44456	0.00018		
res4a_branch1	135640	0.00054		
res4a_branch2a	136131	0.00054		
res4a_branch2b	237165	0.00095		
res4a	22312	0.00009		
res4b_branch2a	236583	0.00095		
res4b_branch2b	237521	0.00095		
res4b	22512	0.00009		
res5a_branch1	311839	0.00125		
res5a_branch2a	311902	0.00125		
res5a_branch2b	596212	0.00238		
res5a	11211	0.00004		
res5b_branch2a	595898	0.00238		
res5b_branch2b	596808	0.00239		
res5b	11381	0.00005		
pool5	36989	0.00015		
newFC	17951	0.00007		

* The clock frequency of the DL processor is: 250MHz

```

ans =
'shell'

label =
'shell'

```


becks



nvidia



google



shell



See Also

- `dlquantizer`
- `calibrate`
- `dlhdl.Workflow`
- "Quantization of Deep Neural Networks" on page 11-2
- "Prototype Deep Learning Networks on FPGA and SoCs Workflow" on page 5-2

Quantize Network for FPGA Deployment

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and validate the quantized network. Rapidly prototype the quantized network by using MATLAB simulation or an FPGA to validate the quantized network. In this example, you quantize the LogoNet neural network.

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 Libraries.

Load Pretrained Network

To load the pretrained LogoNet network and analyze the network architecture, enter:

```
snet = getLogoNetwork;
analyzeNetwork(snet);
```

The screenshot displays the Deep Learning Network Analyzer interface. On the left, a vertical flowchart shows the network architecture starting with 'Imageinput' and followed by layers: conv_1, relu_1, maxpool_1, conv_2, maxpool_2, conv_3, maxpool_3, conv_4, maxpool_4, fc_1, relu_5, dropout_1, fc_2, and relu_6. On the right, the 'ANALYSIS RESULT' table provides detailed information for each layer.

	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	-

Define Calibration and Validation Data Sets

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 to use 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;
newDir = fullfile(matlabroot, 'examples', 'deeplearning_shared', 'data', 'logos_dataset.zip');
copyfile(newDir, curDir, 'f');
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);
```

Create Quantized Network Object

Create a `dlquantizer` object and specify the network to quantize. Specify the execution environment as FPGA for one object. Run the MATLAB simulation on for the second `dlquantizer` object.

```
dlQuantObj_simulation = dlquantizer(snet, 'ExecutionEnvironment', "FPGA", 'Simulation', 'on');
dlQuantObj_FPGA = dlquantizer(snet, 'ExecutionEnvironment', "FPGA");
```

Calibrate Quantized Network

Use the `calibrate` function to exercise the network with sample inputs and collect the range information. The `calibrate` function exercises the network and 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.

```
dlQuantObj_simulation.calibrate(calibrationData_reduced)
```

ans=35x5 table

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv_1_Weights' }	{'conv_1' }	"Weights"	-0.048978
{'conv_1_Bias' }	{'conv_1' }	"Bias"	0.9999
{'conv_2_Weights' }	{'conv_2' }	"Weights"	-0.055511
{'conv_2_Bias' }	{'conv_2' }	"Bias"	-0.00061177
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.045947
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.001399
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.045967
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.0016
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.051394
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.00052311
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.0501
{'fc_2_Bias' }	{'fc_2' }	"Bias"	-0.001756
{'fc_3_Weights' }	{'fc_3' }	"Weights"	-0.05070
{'fc_3_Bias' }	{'fc_3' }	"Bias"	-0.0295
{'imageinput' }	{'imageinput' }	"Activations"	(
{'imageinput_normalization' }	{'imageinput' }	"Activations"	-139.3
:			

```
dlQuantObj_FPGA.calibrate(calibrationData_reduced)
```

```
ans=35x5 table
```

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv_1_Weights' }	{'conv_1' }	"Weights"	-0.04897
{'conv_1_Bias' }	{'conv_1' }	"Bias"	0.9999
{'conv_2_Weights' }	{'conv_2' }	"Weights"	-0.05551
{'conv_2_Bias' }	{'conv_2' }	"Bias"	-0.0006117
{'conv_3_Weights' }	{'conv_3' }	"Weights"	-0.04594
{'conv_3_Bias' }	{'conv_3' }	"Bias"	-0.001399
{'conv_4_Weights' }	{'conv_4' }	"Weights"	-0.04596
{'conv_4_Bias' }	{'conv_4' }	"Bias"	-0.0016
{'fc_1_Weights' }	{'fc_1' }	"Weights"	-0.05139
{'fc_1_Bias' }	{'fc_1' }	"Bias"	-0.0005231
{'fc_2_Weights' }	{'fc_2' }	"Weights"	-0.0501
{'fc_2_Bias' }	{'fc_2' }	"Bias"	-0.001756
{'fc_3_Weights' }	{'fc_3' }	"Weights"	-0.05070
{'fc_3_Bias' }	{'fc_3' }	"Bias"	-0.0295
{'imageinput' }	{'imageinput' }	"Activations"	(
{'imageinput_normalization' }	{'imageinput' }	"Activations"	-139.3
:			

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 2020.1. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.1\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)});
% options_simulation = dlquantizationOptions('MetricFcn',{@(x)hComputeAccuracy(x, snet, validationData)});
```

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_simulation);
```

```
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: [1x1 struct]
    Statistics: []
```

For an FPGA based validation, The `validate` 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 `validate` function uses the metric function defined in the `dlquantizationOptions` object to compare the results 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          227x227x3 images with 'zerocenter' normalization
 2  'conv_1'        Convolution          96 5x5x3 convolutions with stride [1 1] and padding
 3  'relu_1'        ReLU                 ReLU
 4  'maxpool_1'     Max Pooling          3x3 max pooling with stride [2 2] and padding
 5  'conv_2'        Convolution          128 3x3x96 convolutions with stride [1 1] and padding
 6  'relu_2'        ReLU                 ReLU
 7  'maxpool_2'     Max Pooling          3x3 max pooling with stride [2 2] and padding
 8  'conv_3'        Convolution          384 3x3x128 convolutions with stride [1 1] and padding
 9  'relu_3'        ReLU                 ReLU
10  'maxpool_3'     Max Pooling          3x3 max pooling with stride [2 2] and padding
11  'conv_4'        Convolution          128 3x3x384 convolutions with stride [2 2] and padding
12  'relu_4'        ReLU                 ReLU
13  'maxpool_4'     Max Pooling          3x3 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 classes

### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software.
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software.
### 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 target.
### Deep learning network programming has been skipped as the same network is already loaded on the target.
### 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 Logic Modules(ALM) in Intel devices.

```
### Notice: The layer 'imageinput' of type 'ImageInputLayer' is split into an image input layer and a normalization layer.
### Notice: The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software.
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	TotalTime
Network	40142478	0.18247	1	40142478
__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 I

```
### Finished writing input activations.
```

```
### Running single input activation.
```

```
prediction_FPGA = struct with fields:
```

```
  NumSamples: 1
```

```
  MetricResults: [1x1 struct]
```

```
  Statistics: [2x7 table]
```

View Performance of Quantized Neural Network

Examine the `MetricResults.Result` field of the validation output to see the performance of the quantized network.

```
prediction_simulation.MetricResults.Result
```

```
ans=2x2 table
```

NetworkImplementation	MetricOutput
{'Floating-Point'}	1
{'Quantized' }	1

```
prediction_FPGA.MetricResults.Result
```

```
ans=2x2 table
```

NetworkImplementation	MetricOutput
{'Floating-Point'}	1
{'Quantized' }	1

Examine the `QuantizedNetworkFPS` field of the validation output to see the frames per second performance of the quantized network.

```
prediction_FPGA.Statistics.FramesPerSecond(2)
```

```
ans = 19.0828
```

See Also

- `dlquantizer`
- `calibrate`
- `validate`

- `dlquantizationOptions`
- “Quantization of Deep Neural Networks” on page 11-2

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 K
  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.lay
### 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

Network	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	23634966	0.10743	1	23634966
___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'
      LRNBBlockGeneration: '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 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 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.layer'
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.ClassificationLayer' is implemented in software
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	TotalTime
	-----	-----	-----	-----
Network	21219873	0.10610	1	21.219873
___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 K
        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.layer
### 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	Total
Network	23634966	0.10743	1	23634966
___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

Customize Bitstream Configuration to Meet Resource Use Requirements

The user wants 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 (shipping) 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 user resource budget and is on the higher end of the cost spectrum. You can achieve target performance and resource use budget by quantizing the target deep learning network and customizing the custom default bitstream configuration.

In this example create a custom bitstream configuration to match your resource budget and performance requirements.

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');
dlquantObj.calibrate(Image)
```

```
ans=21x5 table
```

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv_1_Weights' }	{'batchnorm_1'}	"Weights"	-0.017061
{'conv_1_Bias' }	{'batchnorm_1'}	"Bias"	-0.025344
{'conv_2_Weights' }	{'batchnorm_2'}	"Weights"	-0.54744
{'conv_2_Bias' }	{'batchnorm_2'}	"Bias"	-1.1787
{'conv_3_Weights' }	{'batchnorm_3'}	"Weights"	-0.39927
{'conv_3_Bias' }	{'batchnorm_3'}	"Bias"	-0.85118

```

{'fc_Weights'          }      {'fc'          }      "Weights"          -0.22558
{'fc_Bias'             }      {'fc'          }      "Bias"             -0.011837
{'imageinput'         }      {'imageinput'  }      "Activations"      0
{'imageinput_normalization'}  {'imageinput'  }      "Activations"      -22.566
{'conv_1'              }      {'batchnorm_1'}  "Activations"      -7.9196
{'relu_1'              }      {'relu_1'       }      "Activations"      0
{'maxpool_1'          }      {'maxpool_1'   }      "Activations"      0
{'conv_2'              }      {'batchnorm_2'}  "Activations"      -8.4641
{'relu_2'              }      {'relu_2'       }      "Activations"      0
{'maxpool_2'          }      {'maxpool_2'   }      "Activations"      0
:

```

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`.

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

```
hPC_reference =
```

```

Processing Module "conv"
  ModuleGeneration: 'on'
  LRNBlockGeneration: 'on'
  ConvThreadNumber: 64
  InputMemorySize: [227 227 3]
  OutputMemorySize: [227 227 3]
  FeatureSizeLimit: 2048

```

```

Processing Module "fc"
  ModuleGeneration: 'on'
  SoftmaxBlockGeneration: 'off'
  FCThreadNumber: 16
  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: 'int8'

```

```

System Level Properties
  TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation K
  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` function of the `dlhdl.ProcessorConfig` object. The function returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

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

```
hPC_reference.estimatePerformance(dlquantObj)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer'
### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in :
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is impl
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
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

```
hPC_reference.estimateResources
```

Deep Learning Processor Estimator Resource Results

	DSPs	Block RAM*	LUTs(CLB/ALUT)
	-----	-----	-----
Available	2520	912	274080
DL_Processor	768(31%)	386(43%)	193454(71%)

* 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 - 768
- 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 the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

To reduce the resource use for the custom bitstream, modify the `KernelDataType` for the `conv`, `fc`, and `adder` modules. Modify the `ConvThreadNumber` to reduce DSP slice count. Reduce the `InputMemorySize` and `OutputMemorySize` for the `conv` module to reduce BRAM count.

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

```
hPC_custom =
```

```
    Processing Module "conv"
      ModuleGeneration: 'on'
      LRNBlockGeneration: 'on'
      ConvThreadNumber: 4
      InputMemorySize: [30 30 1]
      OutputMemorySize: [30 30 1]
      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: '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

To estimate the performance of the digits 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).

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

hPC_custom.estimatePerformance(dlquantObj)

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.BatchNormalizationLayer'
### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in 'nnet.cnn.layer.ImageInputLayer'
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in 'nnet.cnn.layer.ClassificationOutputLayer'
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
Network	267467	0.00134	1	2
conv_1	25628	0.00013		
maxpool_1	42337	0.00021		
conv_2	30957	0.00015		
maxpool_2	68153	0.00034		
conv_3	56981	0.00028		
fc	43411	0.00022		

* The clock frequency of the DL processor is: 200MHz

hPC_custom.estimateResources

Deep Learning Processor Estimator Resource Results

	DSPs	Block RAM*	LUTs(CLB/ALUT)
Available	2520	912	274080
DL_Processor	120(5%)	108(12%)	69517(26%)

* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices

The estimated performance is 750 FPS and the estimated resource use counts are:

- Digital signal processor (DSP) slice count - 129
- 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.

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` 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` 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™) and [Anchor Boxes for Object Detection \(Computer Vision Toolbox\)](#).

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).

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,inputSize));
```

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` 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 (hh:mm:ss)	Mini-batch RMSE	Mini-batch Loss	Base Learning 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: [1x1 DAGNetwork]
    Bitstream: 'zcu102_single'
    ProcessorConfig: []
    Target: [1x1 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' normal
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, 1, 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, 1, 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 [1, 1, 1]
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 [1, 1, 1]
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 [1, 1, 1]
28	'bn3a_branch1'	Batch Normalization	Batch normalization with 128 channels
29	'res3b_branch2a'	Convolution	128 3×3×128 convolutions with stride [1, 1, 1]
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 [1, 1, 1]
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 [1, 1, 1]
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 [1, 1, 1]
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 [1, 1, 1]
44	'bn4a_branch1'	Batch Normalization	Batch normalization with 256 channels
45	'res4b_branch2a'	Convolution	256 3×3×256 convolutions with stride [1, 1, 1]
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 [1, 1, 1]
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 [1, 1, 1]
53	'yolov2Batch1'	Batch Normalization	Batch normalization with 256 channels
54	'yolov2Relu1'	ReLU	ReLU
55	'yolov2Conv2'	Convolution	256 3×3×256 convolutions with stride [1, 1, 1]
56	'yolov2Batch2'	Batch Normalization	Batch normalization with 256 channels
57	'yolov2Relu2'	ReLU	ReLU
58	'yolov2ClassConv'	Convolution	42 1×1×256 convolutions with stride [1, 1, 1]
59	'yolov2Transform'	YOLO v2 Transform Layer.	YOLO v2 Transform Layer with 7 anchors.
60	'yolov2OutputLayer'	YOLO v2 Output	YOLO v2 Output with 7 anchors.

Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer'.
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: [1x1 struct]
  instructions: [1x1 struct]
  registers: [1x1 struct]
  syncInstructions: [1x1 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

```
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	TotalTime
	-----	-----	-----	-----
Network	16974672	0.07716	1	16974672
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)
```



Customize Bitstream Configuration to Meet Resource Use Requirements

The user wants 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 (shipping) 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 user resource budget and is on the higher end of the cost spectrum. You can achieve target performance and resource use budget by quantizing the target deep learning network and customizing the custom default bitstream configuration.

In this example create a custom bitstream configuration to match your resource budget and performance requirements.

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');
dlquantObj.calibrate(Image)
```

```
ans=21x5 table
```

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv_1_Weights' }	{'batchnorm_1'}	"Weights"	-0.017061
{'conv_1_Bias' }	{'batchnorm_1'}	"Bias"	-0.025344
{'conv_2_Weights' }	{'batchnorm_2'}	"Weights"	-0.54744
{'conv_2_Bias' }	{'batchnorm_2'}	"Bias"	-1.1787
{'conv_3_Weights' }	{'batchnorm_3'}	"Weights"	-0.39927
{'conv_3_Bias' }	{'batchnorm_3'}	"Bias"	-0.85118

```

{'fc_Weights'          }      {'fc'          }      "Weights"          -0.22558
{'fc_Bias'            }      {'fc'          }      "Bias"             -0.011837
{'imageinput'        }      {'imageinput'  }      "Activations"      0
{'imageinput_normalization'}  {'imageinput'  }      "Activations"      -22.566
{'conv_1'            }      {'batchnorm_1'}  "Activations"      -7.9196
{'relu_1'            }      {'relu_1'      }      "Activations"      0
{'maxpool_1'        }      {'maxpool_1'   }      "Activations"      0
{'conv_2'            }      {'batchnorm_2'}  "Activations"      -8.4641
{'relu_2'            }      {'relu_2'      }      "Activations"      0
{'maxpool_2'        }      {'maxpool_2'   }      "Activations"      0
:

```

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`.

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

```
hPC_reference =
```

```

Processing Module "conv"
  ModuleGeneration: 'on'
  LRNBlockGeneration: 'on'
  ConvThreadNumber: 64
  InputMemorySize: [227 227 3]
  OutputMemorySize: [227 227 3]
  FeatureSizeLimit: 2048

```

```

Processing Module "fc"
  ModuleGeneration: 'on'
  SoftmaxBlockGeneration: 'off'
  FCThreadNumber: 16
  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: 'int8'

```

```

System Level Properties
  TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation K
  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` function of the `dlhdl.ProcessorConfig` object. The function returns the estimated layer latency, network latency, and network performance in frames per second (Frames/s).

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

```
hPC_reference.estimatePerformance(dlquantObj)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer'
### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in :
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is impl
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
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

```
hPC_reference.estimateResources
```

Deep Learning Processor Estimator Resource Results

	DSPs	Block RAM*	LUTs(CLB/ALUT)
	-----	-----	-----
Available	2520	912	274080
DL_Processor	768(31%)	386(43%)	193454(71%)

* 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 - 768
- 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 the `dlhdl.ProcessorConfig` object. For more information, see `dlhdl.ProcessorConfig`. To learn about modifiable parameters of the processor configuration, see `getModuleProperty` and `setModuleProperty`.

To reduce the resource use for the custom bitstream, modify the `KernelDataType` for the `conv`, `fc`, and `adder` modules. Modify the `ConvThreadNumber` to reduce DSP slice count. Reduce the `InputMemorySize` and `OutputMemorySize` for the `conv` module to reduce BRAM count.

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

```
hPC_custom =
```

```
    Processing Module "conv"
      ModuleGeneration: 'on'
      LRNBlockGeneration: 'on'
      ConvThreadNumber: 4
      InputMemorySize: [30 30 1]
      OutputMemorySize: [30 30 1]
      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: '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

To estimate the performance of the digits 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).

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

```
hPC_custom.estimatePerformance(dlquantObj)
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.BatchNormalizationLayer'
### Notice: The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in 'nnet.cnn.layer.ImageInputLayer'
### Notice: The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in 'nnet.cnn.layer.ClassificationOutputLayer'
```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
Network	267467	0.00134	1	2
conv_1	25628	0.00013		
maxpool_1	42337	0.00021		
conv_2	30957	0.00015		
maxpool_2	68153	0.00034		
conv_3	56981	0.00028		
fc	43411	0.00022		

* The clock frequency of the DL processor is: 200MHz

```
hPC_custom.estimateResources
```

Deep Learning Processor Estimator Resource Results

	DSPs	Block RAM*	LUTs(CLB/ALUT)
Available	2520	912	274080
DL_Processor	120(5%)	108(12%)	69517(26%)

* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices

The estimated performance is 750 FPS and the estimated resource use counts are:

- Digital signal processor (DSP) slice count - 129
- 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.

Image Classification Using DAG Network Deployed to FPGA

This example shows how to train, compile, and deploy a `dlhdl.Workflow` object that has 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.

Required Products

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 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 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:
```

```

    Layers: [71x1 nnet.cnn.layer.Layer]
    Connections: [78x2 table]
    InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}

```

```
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 datastore 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 every `ValidationFrequency` iterations during training.

```

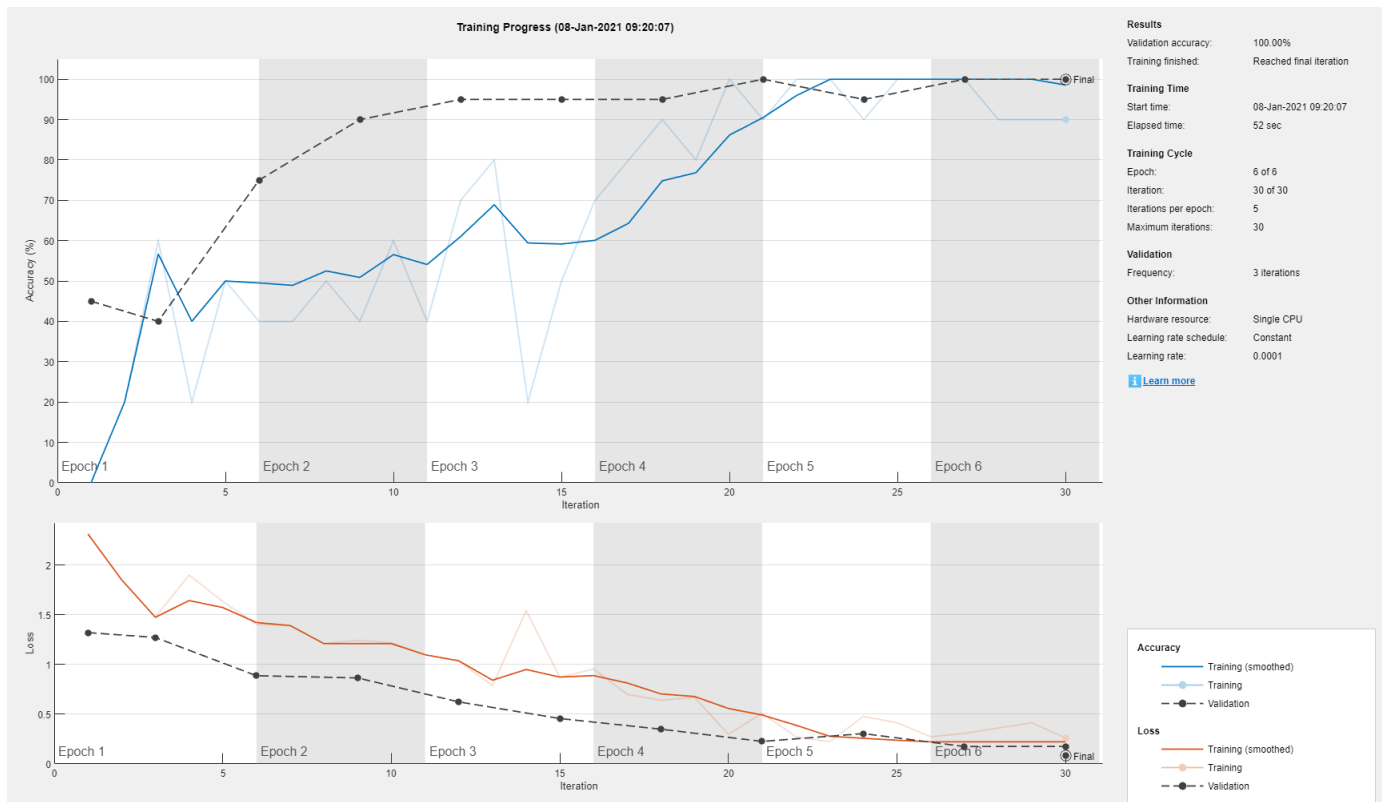
options = trainingOptions('sgdm', ...
    'MiniBatchSize',10, ...
    'MaxEpochs',6, ...
    'InitialLearnRate',1e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',3, ...
    'Verbose',false, ...
    'Plots','training-progress');

```

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 supported GPU device. For more

information, see “GPU Support by Release” (Parallel Computing Toolbox). 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 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 2019.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.l
```

```
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 `netTransfer` 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', netTransfer, 'Bitstream', 'zcu102_single', 'Target', hTarget);
```

Compile the netTransfer DAG network

To compile the netTransfer DAG network, run the compile method of the `dlhdl.Workflow` object. You can optionally specify the maximum number of input frames.

```
dn = hw.compile('InputFrameNumberLimit',15)
```

```
### 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' normaliz.
2	'conv1'	Convolution	64 7×7×3 convolutions with stride [2 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] and
6	'res2a_branch2a'	Convolution	64 3×3×64 convolutions with stride [1 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 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 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 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 [2 2]
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 [1 1]
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 [2 2]
28	'bn3a_branch1'	Batch Normalization	Batch normalization with 128 channels
29	'res3b_branch2a'	Convolution	128 3×3×128 convolutions with stride [1 1]
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 [1 1]
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 [2 2]
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 [1 1]
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 [2 2]
44	'bn4a_branch1'	Batch Normalization	Batch normalization with 256 channels
45	'res4b_branch2a'	Convolution	256 3×3×256 convolutions with stride [1 1]
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	'res5a_branch2a'	Convolution	512 3×3×256 convolutions with stride [
53	'bn5a_branch2a'	Batch Normalization	Batch normalization with 512 channels
54	'res5a_branch2a_relu'	ReLU	ReLU
55	'res5a_branch2b'	Convolution	512 3×3×512 convolutions with stride [
56	'bn5a_branch2b'	Batch Normalization	Batch normalization with 512 channels
57	'res5a'	Addition	Element-wise addition of 2 inputs
58	'res5a_relu'	ReLU	ReLU
59	'res5a_branch1'	Convolution	512 1×1×256 convolutions with stride [
60	'bn5a_branch1'	Batch Normalization	Batch normalization with 512 channels
61	'res5b_branch2a'	Convolution	512 3×3×512 convolutions with stride [
62	'bn5b_branch2a'	Batch Normalization	Batch normalization with 512 channels
63	'res5b_branch2a_relu'	ReLU	ReLU
64	'res5b_branch2b'	Convolution	512 3×3×512 convolutions with stride [
65	'bn5b_branch2b'	Batch Normalization	Batch normalization with 512 channels
66	'res5b'	Addition	Element-wise addition of 2 inputs
67	'res5b_relu'	ReLU	ReLU
68	'pool5'	Global Average Pooling	Global average pooling
69	'new_fc'	Fully Connected	5 fully connected layer
70	'prob'	Softmax	softmax
71	'new_classoutput'	Classification Output	crossentropyex with 'MathWorks Cap' and

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.lay
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: 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.
Skipping: prob
```

```

Skipping: new_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"	"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: [1x1 struct]
    instructions: [1x1 struct]
    registers: [1x1 struct]
    syncInstructions: [1x1 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.

```
hW.deploy
```

```

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target
### Deep learning network programming has been skipped as the same network is already loaded on the target

```

Load Image for Prediction

Load the example image.

```

imgFile = fullfile(pwd, 'MerchData', 'MathWorks Cube', 'Mathworks cube_0.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] = 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	Total
	-----	-----	-----	-----
Network	23470681	0.10668	1	23470681
conv1	2224133	0.01011		
pool1	573009	0.00260		
res2a_branch2a	972706	0.00442		
res2a_branch2b	972715	0.00442		
res2a	210584	0.00096		
res2b_branch2a	972670	0.00442		
res2b_branch2b	973171	0.00442		
res2b	210235	0.00096		
res3a_branch1	538433	0.00245		
res3a_branch2a	746681	0.00339		
res3a_branch2b	904757	0.00411		
res3a	104923	0.00048		
res3b_branch2a	904442	0.00411		
res3b_branch2b	904234	0.00411		
res3b	105019	0.00048		
res4a_branch1	485689	0.00221		
res4a_branch2a	486053	0.00221		
res4a_branch2b	880357	0.00400		
res4a	52814	0.00024		

res4b_branch2a	880122	0.00400
res4b_branch2b	880268	0.00400
res4b	52492	0.00024
res5a_branch1	1056215	0.00480
res5a_branch2a	1056269	0.00480
res5a_branch2b	2057399	0.00935
res5a	26272	0.00012
res5b_branch2a	2057349	0.00935
res5b_branch2b	2057639	0.00935
res5b	26409	0.00012
pool5	71402	0.00032
new_fc	24650	0.00011

* The clock frequency of the DL processor is: 220MHz

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

```
ans =  
'MathWorks Cube'
```

Classify Images on an FPGA Using a Quantized DAG Network

In this example, you use Deep Learning HDL Toolbox™ to deploy a quantized deep convolutional neural network and classify an image. The example uses the pretrained ResNet-18 convolutional neural network to demonstrate transfer learning, quantization, and deployment for the quantized network. 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.

Required Products

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 Libraries

Transfer Learning Using Resnet-18

To perform classification on a new set of images, you fine-tune a pretrained ResNet-18 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 ResNet-18 Network

To load the pretrained network ResNet-18, enter:

```
net = resnet18;
```

To view the layers of the pretrained network, enter:

```
analyzeNetwork(net);
```

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 = net.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');
validationData_FPGA = imdsValidation.subset(1:1);

```

Replace Final Layers

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:
        Layers: [71x1 nnet.cnn.layer.Layer]
        Connections: [78x2 table]
        InputNames: {'data'}
        OutputNames: {'ClassificationLayer_predictions'}

```

```
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.

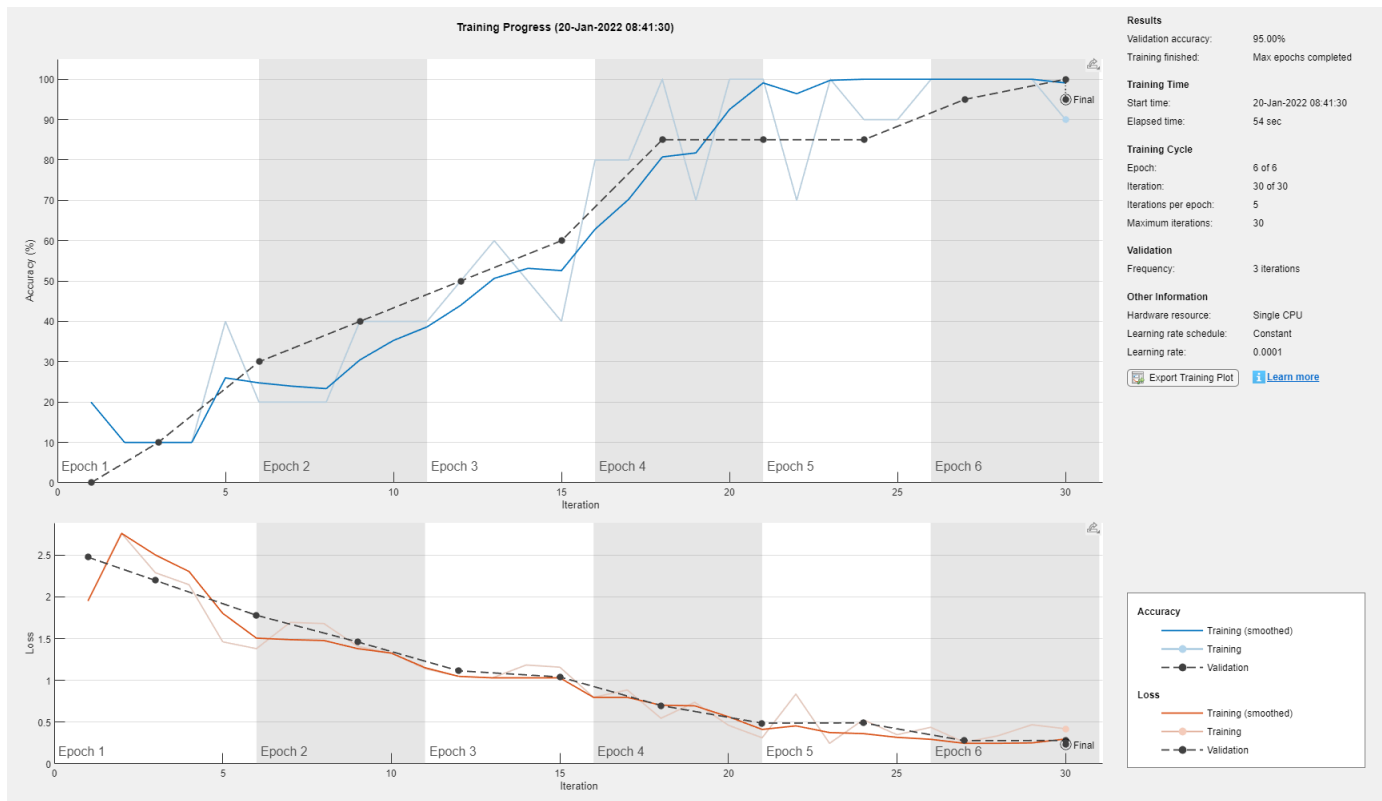
```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
'DataAugmentation',imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

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 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 supported GPU device. For more information, see “GPU Support by Release” (Parallel Computing Toolbox)). 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);
```



Quantize the Network

Create a `dlquantizer` object and specify the network to quantize.

```
dlquantObj = dlquantizer(netTransfer, 'ExecutionEnvironment', 'FPGA');
```

Calibrate the Quantized Network Object

Use the `calibrate` function to exercise the network with 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(augImdsTrain)
```

ans=95x5 table

Optimized Layer Name	Network Layer Name	Learnables / Activations	MinValue
{'conv1_Weights' }	{'conv1' }	"Weights"	-0.56885
{'conv1_Bias' }	{'conv1' }	"Bias"	-0.66869
{'res2a_branch2a_Weights' }	{'res2a_branch2a' }	"Weights"	-0.46037
{'res2a_branch2a_Bias' }	{'res2a_branch2a' }	"Bias"	-0.82446
{'res2a_branch2b_Weights' }	{'res2a_branch2b' }	"Weights"	-0.8002
{'res2a_branch2b_Bias' }	{'res2a_branch2b' }	"Bias"	-1.3954
{'res2b_branch2a_Weights' }	{'res2b_branch2a' }	"Weights"	-0.33991
{'res2b_branch2a_Bias' }	{'res2b_branch2a' }	"Bias"	-1.1367
{'res2b_branch2b_Weights' }	{'res2b_branch2b' }	"Weights"	-1.2616
{'res2b_branch2b_Bias' }	{'res2b_branch2b' }	"Bias"	-0.86662
{'res3a_branch2a_Weights' }	{'res3a_branch2a' }	"Weights"	-0.19675
{'res3a_branch2a_Bias' }	{'res3a_branch2a' }	"Bias"	-0.5063
{'res3a_branch2b_Weights' }	{'res3a_branch2b' }	"Weights"	-0.5385
{'res3a_branch2b_Bias' }	{'res3a_branch2b' }	"Bias"	-0.66884
{'res3a_branch1_Weights' }	{'res3a_branch1' }	"Weights"	-0.66715
{'res3a_branch1_Bias' }	{'res3a_branch1' }	"Bias"	-0.97269
:			

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 2019.2. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.l
```

```
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 alexnet 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', dlquantObj, 'Bitstream', 'zcu102_int8', 'Target', hTarget);
```

Compile the netTransfer DAG network

To compile the netTransfer DAG network, run the compile method of the dlhdl.Workflow object. You can optionally specify the maximum number of input frames.

```
dn = hW.compile('InputFrameNumberLimit', 15)
```

```
### Compiling network for Deep Learning FPGA prototyping ...
```

```
### Targeting FPGA bitstream zcu102_int8.
```

```
### The network includes the following layers:
```

1	'data'	Image Input	224x224x3 images with 'zscore' norm
2	'conv1'	Convolution	64 7x7x3 convolutions with stride
3	'bn_conv1'	Batch Normalization	Batch normalization with 64 channel
4	'conv1_relu'	ReLU	ReLU
5	'pool1'	Max Pooling	3x3 max pooling with stride [2 2]
6	'res2a_branch2a'	Convolution	64 3x3x64 convolutions with stride
7	'bn2a_branch2a'	Batch Normalization	Batch normalization with 64 channel
8	'res2a_branch2a_relu'	ReLU	ReLU
9	'res2a_branch2b'	Convolution	64 3x3x64 convolutions with stride
10	'bn2a_branch2b'	Batch Normalization	Batch normalization with 64 channel
11	'res2a'	Addition	Element-wise addition of 2 inputs
12	'res2a_relu'	ReLU	ReLU
13	'res2b_branch2a'	Convolution	64 3x3x64 convolutions with stride
14	'bn2b_branch2a'	Batch Normalization	Batch normalization with 64 channel
15	'res2b_branch2a_relu'	ReLU	ReLU
16	'res2b_branch2b'	Convolution	64 3x3x64 convolutions with stride
17	'bn2b_branch2b'	Batch Normalization	Batch normalization with 64 channel
18	'res2b'	Addition	Element-wise addition of 2 inputs
19	'res2b_relu'	ReLU	ReLU
20	'res3a_branch2a'	Convolution	128 3x3x64 convolutions with stride
21	'bn3a_branch2a'	Batch Normalization	Batch normalization with 128 channel
22	'res3a_branch2a_relu'	ReLU	ReLU
23	'res3a_branch2b'	Convolution	128 3x3x128 convolutions with stride
24	'bn3a_branch2b'	Batch Normalization	Batch normalization with 128 channel
25	'res3a'	Addition	Element-wise addition of 2 inputs
26	'res3a_relu'	ReLU	ReLU
27	'res3a_branch1'	Convolution	128 1x1x64 convolutions with stride
28	'bn3a_branch1'	Batch Normalization	Batch normalization with 128 channel
29	'res3b_branch2a'	Convolution	128 3x3x128 convolutions with stride
30	'bn3b_branch2a'	Batch Normalization	Batch normalization with 128 channel
31	'res3b_branch2a_relu'	ReLU	ReLU
32	'res3b_branch2b'	Convolution	128 3x3x128 convolutions with stride
33	'bn3b_branch2b'	Batch Normalization	Batch normalization with 128 channel
34	'res3b'	Addition	Element-wise addition of 2 inputs
35	'res3b_relu'	ReLU	ReLU
36	'res4a_branch2a'	Convolution	256 3x3x128 convolutions with stride
37	'bn4a_branch2a'	Batch Normalization	Batch normalization with 256 channel
38	'res4a_branch2a_relu'	ReLU	ReLU
39	'res4a_branch2b'	Convolution	256 3x3x256 convolutions with stride
40	'bn4a_branch2b'	Batch Normalization	Batch normalization with 256 channel
41	'res4a'	Addition	Element-wise addition of 2 inputs
42	'res4a_relu'	ReLU	ReLU
43	'res4a_branch1'	Convolution	256 1x1x128 convolutions with stride
44	'bn4a_branch1'	Batch Normalization	Batch normalization with 256 channel
45	'res4b_branch2a'	Convolution	256 3x3x256 convolutions with stride
46	'bn4b_branch2a'	Batch Normalization	Batch normalization with 256 channel

47	'res4b_branch2a_relu'	ReLU	ReLU
48	'res4b_branch2b'	Convolution	256 3×3×256 convolutions with stride 2
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	'res5a_branch2a'	Convolution	512 3×3×256 convolutions with stride 2
53	'bn5a_branch2a'	Batch Normalization	Batch normalization with 512 channels
54	'res5a_branch2a_relu'	ReLU	ReLU
55	'res5a_branch2b'	Convolution	512 3×3×512 convolutions with stride 2
56	'bn5a_branch2b'	Batch Normalization	Batch normalization with 512 channels
57	'res5a'	Addition	Element-wise addition of 2 inputs
58	'res5a_relu'	ReLU	ReLU
59	'res5a_branch1'	Convolution	512 1×1×256 convolutions with stride 1
60	'bn5a_branch1'	Batch Normalization	Batch normalization with 512 channels
61	'res5b_branch2a'	Convolution	512 3×3×512 convolutions with stride 2
62	'bn5b_branch2a'	Batch Normalization	Batch normalization with 512 channels
63	'res5b_branch2a_relu'	ReLU	ReLU
64	'res5b_branch2b'	Convolution	512 3×3×512 convolutions with stride 2
65	'bn5b_branch2b'	Batch Normalization	Batch normalization with 512 channels
66	'res5b'	Addition	Element-wise addition of 2 inputs
67	'res5b_relu'	ReLU	ReLU
68	'pool5'	2-D Global Average Pooling	2-D global average pooling
69	'new_fc'	Fully Connected	5 fully connected layer
70	'prob'	Softmax	softmax
71	'new_classoutput'	Classification Output	crossentropyex with 'MathWorks Cap

```

### 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 software.
### 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
### 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: [1x1 struct]  
    instructions: [1x1 struct]  
    registers: [1x1 struct]  
    syncInstructions: [1x1 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.

```
hw.deploy
```

```
### 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 20-Jan-2022 08:45:22  
### Loading weights to FC Processor.  
### FC Weights loaded. Current time is 20-Jan-2022 08:45:22
```

Load Image for Prediction

Load the example image.

```
imgFile = fullfile(pwd, 'MerchData', 'MathWorks Cube', 'Mathworks cube_0.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] = hW.predict(single(inputImg), 'Profile', 'on');
```

```
### Finished writing input activations.
```

```
### Running single input activation.
```

Deep Learning Processor Profiler Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	7389695	0.02956	1	7389695
conv1	1115359	0.00446		
pool1	237742	0.00095		
res2a_branch2a	269669	0.00108		
res2a_branch2b	270019	0.00108		
res2a	103095	0.00041		
res2b_branch2a	269716	0.00108		
res2b_branch2b	269895	0.00108		
res2b	102385	0.00041		
res3a_branch1	156246	0.00062		
res3a_branch2a	227373	0.00091		
res3a_branch2b	245201	0.00098		
res3a	52543	0.00021		
res3b_branch2a	244793	0.00098		
res3b_branch2b	244952	0.00098		
res3b	51176	0.00020		
res4a_branch1	135788	0.00054		
res4a_branch2a	135745	0.00054		
res4a_branch2b	237464	0.00095		
res4a	25612	0.00010		

```

res4b_branch2a      237244      0.00095
res4b_branch2b      237242      0.00095
res4b                25952       0.00010
res5a_branch1       311610      0.00125
res5a_branch2a      311719      0.00125
res5a_branch2b      596194      0.00238
res5a                13191       0.00005
res5b_branch2a      595890      0.00238
res5b_branch2b      596795      0.00239
res5b               14141       0.00006
pool5               36932       0.00015
new_fc              17825       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 single data type network.

```

options_FPGA = dlquantizationOptions('Bitstream','zcu102_int8','Target',hTarget);
prediction_FPGA = dlquantObj.validate(imdsValidation,options_FPGA)

```

```

### Compiling network for Deep Learning FPGA prototyping ...

```

```

### Targeting FPGA bitstream zcu102_int8.

```

```

### The network includes the following layers:

```

1	'data'	Image Input	224×224×3 images with 'zscore' norm
2	'conv1'	Convolution	64 7×7×3 convolutions with stride
3	'bn_conv1'	Batch Normalization	Batch normalization with 64 channel
4	'conv1_relu'	ReLU	ReLU
5	'pool1'	Max Pooling	3×3 max pooling with stride [2 2]
6	'res2a_branch2a'	Convolution	64 3×3×64 convolutions with stride
7	'bn2a_branch2a'	Batch Normalization	Batch normalization with 64 channel
8	'res2a_branch2a_relu'	ReLU	ReLU
9	'res2a_branch2b'	Convolution	64 3×3×64 convolutions with stride
10	'bn2a_branch2b'	Batch Normalization	Batch normalization with 64 channel
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
14	'bn2b_branch2a'	Batch Normalization	Batch normalization with 64 channel
15	'res2b_branch2a_relu'	ReLU	ReLU
16	'res2b_branch2b'	Convolution	64 3×3×64 convolutions with stride
17	'bn2b_branch2b'	Batch Normalization	Batch normalization with 64 channel
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 channel
22	'res3a_branch2a_relu'	ReLU	ReLU
23	'res3a_branch2b'	Convolution	128 3×3×128 convolutions with stri
24	'bn3a_branch2b'	Batch Normalization	Batch normalization with 128 channel
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 channel
29	'res3b_branch2a'	Convolution	128 3×3×128 convolutions with stri

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 1
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 1
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 1
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 1
44	'bn4a_branch1'	Batch Normalization	Batch normalization with 256 channels
45	'res4b_branch2a'	Convolution	256 3×3×256 convolutions with stride 1
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 1
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	'res5a_branch2a'	Convolution	512 3×3×256 convolutions with stride 1
53	'bn5a_branch2a'	Batch Normalization	Batch normalization with 512 channels
54	'res5a_branch2a_relu'	ReLU	ReLU
55	'res5a_branch2b'	Convolution	512 3×3×512 convolutions with stride 1
56	'bn5a_branch2b'	Batch Normalization	Batch normalization with 512 channels
57	'res5a'	Addition	Element-wise addition of 2 inputs
58	'res5a_relu'	ReLU	ReLU
59	'res5a_branch1'	Convolution	512 1×1×256 convolutions with stride 1
60	'bn5a_branch1'	Batch Normalization	Batch normalization with 512 channels
61	'res5b_branch2a'	Convolution	512 3×3×512 convolutions with stride 1
62	'bn5b_branch2a'	Batch Normalization	Batch normalization with 512 channels
63	'res5b_branch2a_relu'	ReLU	ReLU
64	'res5b_branch2b'	Convolution	512 3×3×512 convolutions with stride 1
65	'bn5b_branch2b'	Batch Normalization	Batch normalization with 512 channels
66	'res5b'	Addition	Element-wise addition of 2 inputs
67	'res5b_relu'	ReLU	ReLU
68	'pool5'	2-D Global Average Pooling	2-D global average pooling
69	'new_fc'	Fully Connected	5 fully connected layer
70	'prob'	Softmax	softmax
71	'new_classoutput'	Classification Output	crossentropyex with 'MathWorks Cap

```

### 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 software.
### 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 :
### 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 target.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 20-Jan-2022 08:46:40
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 20-Jan-2022 08:46:40
### Finished writing input activations.
### Running in multi-frame mode with 20 inputs.

```

Deep Learning Processor Bitstream Build Info

Resource	Utilized	Total	Percentage
LUTs (CLB/ALM)*	248190	274080	90.55
DSPs	384	2520	15.24
Block RAM	581	912	63.71

* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive IP blocks.

```

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Conv2D'
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data' and a
### 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

```

Deep Learning Processor Estimator Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
Network	23871185	0.10851	1	23871185
___data_norm_add	210750	0.00096		
___data_norm	210750	0.00096		
___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		
___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)*	168836	274080	61.60
DSPs	800	2520	31.75
Block RAM	453	912	49.67

* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive IP

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

prediction_FPGA = struct with fields:
 NumSamples: 20
 MetricResults: [1x1 struct]

```
Statistics: [2x7 table]
```

```
prediction_FPGA.Statistics.FramesPerSecond
```

```
ans = 2x1
```

```
    9.2161  
   33.8157
```

The first number is the frames per second performance for the single data type network and the second number is the frames per second performance for the quantized network.

Classify ECG Signals Using DAG Network Deployed To FPGA

This example shows how to classify human electrocardiogram (ECG) signals by deploying a trained directed acyclic graph (DAG) network.

Training a deep CNN from scratch is computationally expensive and requires a large amount of training data. In various applications, a sufficient amount of training data is not available, and synthesizing new realistic training examples is not feasible. In these cases, leveraging existing neural networks that have been trained on large data sets for conceptually similar tasks is desirable. This leveraging of existing neural networks is called transfer learning. In this example we adapt two deep CNNs, GoogLeNet and SqueezeNet, pretrained for image recognition to classify ECG waveforms based on a time-frequency representation.

GoogLeNet and SqueezeNet are deep CNNs originally designed to classify images in 1000 categories. We reuse the network architecture of the CNN to classify ECG signals based on images from the CWT of the time series data. The data used in this example are publicly available from PhysioNet.

Data Description

In this example, you use ECG data obtained from three groups of people: persons with cardiac arrhythmia (ARR), persons with congestive heart failure (CHF), and persons with normal sinus rhythms (NSR). In total you use 162 ECG recordings from three PhysioNet databases: MIT-BIH Arrhythmia Database [3][7], MIT-BIH Normal Sinus Rhythm Database [3], and The BIDMC Congestive Heart Failure Database [1][3]. More specifically, 96 recordings from persons with arrhythmia, 30 recordings from persons with congestive heart failure, and 36 recordings from persons with normal sinus rhythms. The goal is to train a classifier to distinguish between ARR, CHF, and NSR.

Download Data

The first step is to download the data from the GitHub repository. To download the data from the website, click Code and select Download ZIP. Save the file `physionet_ECG_data-main.zip` in a folder where you have write permission. The instructions for this example assume you have downloaded the file to your temporary directory, `tempdir`, in MATLAB. Modify the subsequent instructions for unzipping and loading the data if you choose to download the data in folder different from `tempdir`. If you are familiar with Git, you can download the latest version of the tools (`git`) and obtain the data from a system command prompt using `git clone https://github.com/mathworks/physionet_ECG_data/`.

After downloading the data from GitHub, unzip the file in your temporary directory.

```
unzip(fullfile(tempdir, 'physionet_ECG_data-main.zip'), tempdir)
```

Unzipping creates the folder `physionet-ECG_data-main` in your temporary directory. This folder contains the text file `README.md` and `ECGData.zip`. The `ECGData.zip` file contains

- `ECGData.mat`
- `Modified_physionet_data.txt`
- `License.txt`

`ECGData.mat` holds the data used in this example. The text file, `Modified_physionet_data.txt`, is required by PhysioNet's copying policy and provides the source attributions for the data as well as a description of the preprocessing steps applied to each ECG recording.

Unzip `ECGData.zip` in `physionet-ECG_data-main`. Load the 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'))
```

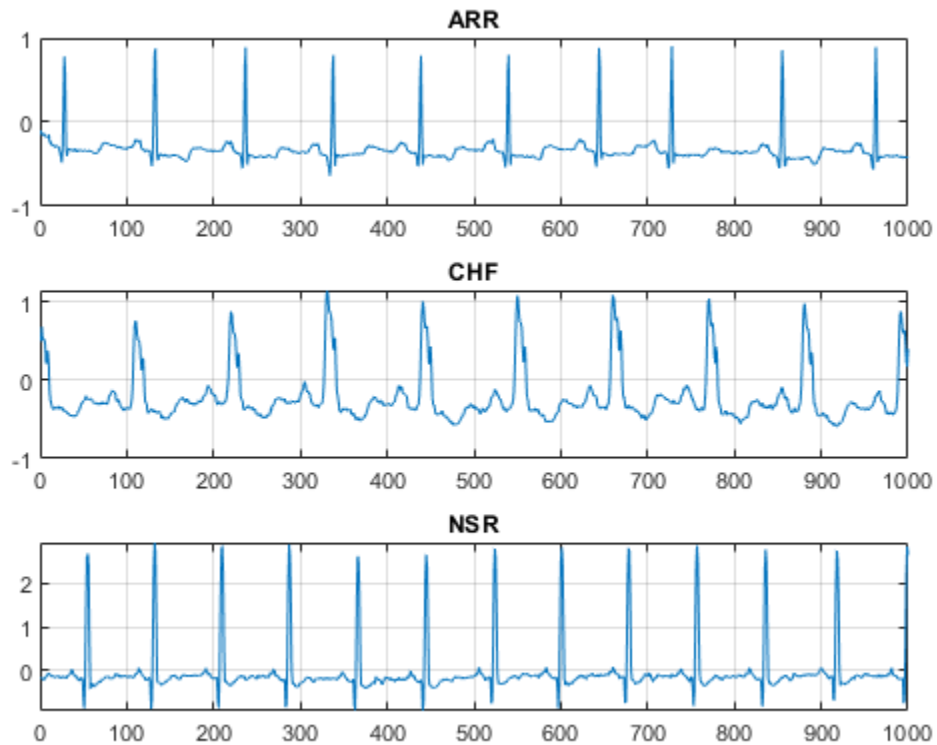
`ECGData` is a structure array with two fields: `Data` and `Labels`. The `Data` field is a 162-by-65536 matrix where each row is an ECG recording sampled at 128 hertz. `Labels` is a 162-by-1 cell array of diagnostic labels, one for each row of `Data`. The three diagnostic categories are: 'ARR', 'CHF', and 'NSR'.

To store the preprocessed data of each category, first create an ECG data directory `dataDir` inside `tempdir`. Then create three subdirectories in 'data' named after each ECG category. The helper function `helperCreateECGDirectories` does this. `helperCreateECGDirectories` accepts `ECGData`, the name of an ECG data directory, and the name of a parent directory as input arguments. You can replace `tempdir` with another directory where you have write permission. 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 a representative of each ECG category. The helper function `helperPlotReps` does this. `helperPlotReps` accepts `ECGData` as input. 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. These representations are called scalograms. A scalogram is the absolute value of the CWT coefficients of a signal.

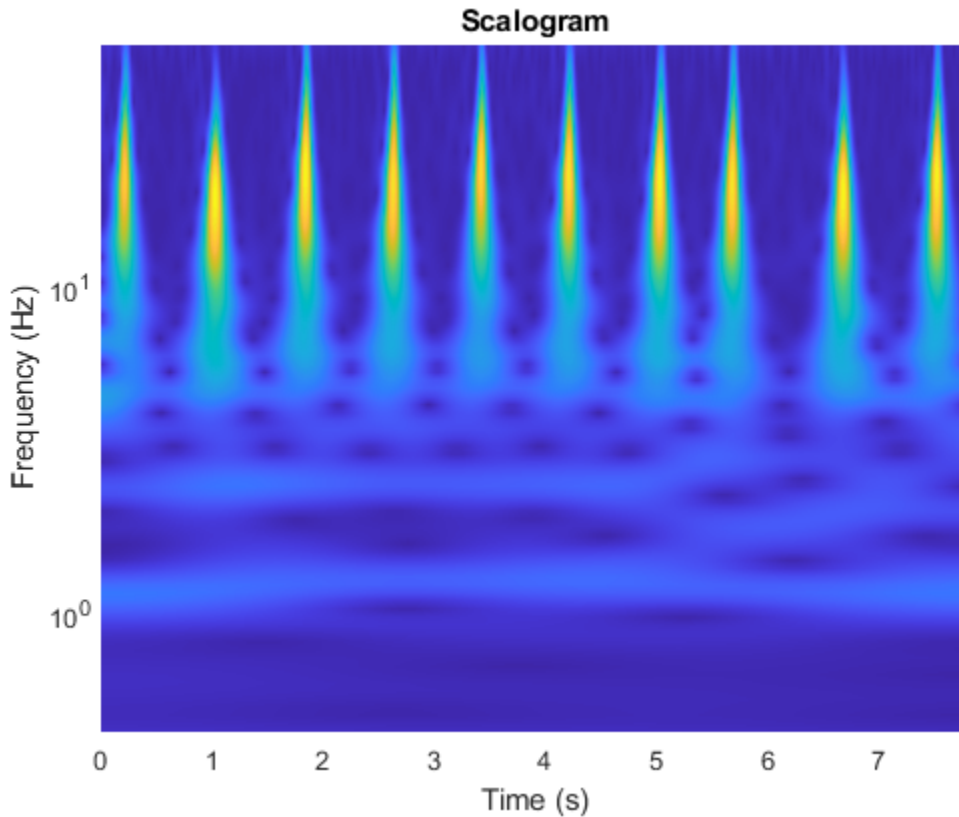
To create the scalograms, precompute a CWT filter bank. Precomputing the CWT filter bank is the preferred method when obtaining the CWT of many signals using the same parameters.

Before generating the scalograms, examine one of them. Create a CWT filter bank using `cwtfilterbank` (Wavelet Toolbox) for a signal with 1000 samples. Use the filter bank to take the CWT of the first 1000 samples of the signal and obtain the scalogram from the coefficients.

```

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 helper function `helperCreateRGBfromTF` 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 GoogLeNet architecture, each RGB image is an array of size 224-by-224-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, one for training and the other for validation. 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))]);
```

```
Number of training images: 130
```

```
disp(['Number of validation images: ', num2str(numel(imgsValidation.Files))]);
```

```
Number of validation images: 32
```

SqueezeNet

SqueezeNet is a deep CNN whose architecture supports images of size 227-by-227-by-3. Even though the image dimensions are different for GoogLeNet, you do not have to generate new RGB images at the SqueezeNet dimensions. You can use the original RGB images.

Load

Load a custom SqueezeNet neural network.

```
sqz = squeezeNet;
```

Extract the layer graph from the network. Confirm SqueezeNet has fewer layers than GoogLeNet. Also confirm that SqueezeNet is configured for images of size 227-by-227-by-3

```
lgraphSqz = layerGraph(sqz);
disp(['Number of Layers: ', num2str(numel(lgraphSqz.Layers))])
```

```
Number of Layers: 68
```

```
disp(lgraphSqz.Layers(1).InputSize)
```

```
    227    227     3
```

Modify SqueezeNet Network Parameters

To retrain SqueezeNet to classify new images, make changes similar to those made for GoogLeNet.

Inspect the last six network layers.

```
lgraphSqz.Layers(end-5:end)
```

```
ans =
```

```
    6×1 Layer array with layers:
```

1	'drop9'	Dropout	50% dropout
2	'conv10'	Convolution	1000 1×1×512 convolutions v
3	'relu_conv10'	ReLU	ReLU
4	'pool10'	Global Average Pooling	Global average pooling
5	'prob'	Softmax	softmax
6	'ClassificationLayer_predictions'	Classification Output	crossentropyex with 'tench

Replace the 'drop9' layer, the last dropout layer in the network, with a dropout layer of probability 0.6.

```
tmpLayer = lgraphSqz.Layers(end-5);
newDropoutLayer = dropoutLayer(0.6, 'Name', 'new_dropout');
lgraphSqz = replaceLayer(lgraphSqz, tmpLayer.Name, newDropoutLayer);
```

Unlike GoogLeNet, the last learnable layer in SqueezeNet is a 1-by-1 convolutional layer, 'conv10', and not a fully connected layer. Replace the 'conv10' layer with a new convolutional layer with the number of filters equal to the number of classes. As was done with GoogLeNet, increase the learning rate factors of the new layer.

```
numClasses = numel(categories(imgsTrain.Labels));
tmpLayer = lgraphSqz.Layers(end-4);
```

```

newLearnableLayer = convolution2dLayer(1,numClasses, ...
    'Name','new_conv', ...
    'WeightLearnRateFactor',10, ...
    'BiasLearnRateFactor',10);
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newLearnableLayer);

```

Replace the classification layer with a new one without class labels.

```

tmpLayer = lgraphSqz.Layers(end);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newClassLayer);

```

Inspect the last six layers of the network. Confirm the dropout, convolutional, and output layers have been changed.

```
lgraphSqz.Layers(63:68)
```

```

ans =
    6×1 Layer array with layers:

     1  'new_dropout'      Dropout      60% dropout
     2  'new_conv'        Convolution  3 1×1 convolutions with stride [1 1] and
     3  'relu_conv10'     ReLU        ReLU
     4  'pool10'          Global Average Pooling  Global average pooling
     5  'prob'            Softmax     softmax
     6  'new_classoutput' Classification Output  crossentropyex

```

Prepare RGB Data for SqueezeNet

The RGB images have dimensions appropriate for the GoogLeNet architecture. Create augmented image datastores that automatically resize the existing RGB images for the SqueezeNet architecture. For more information, see `augmentedImageDatastore`.

```

augimgsTrain = augmentedImageDatastore([227 227],imgsTrain);
augimgsValidation = augmentedImageDatastore([227 227],imgsValidation);

```

Set Training Options and Train SqueezeNet

Create a new set of training options to use with SqueezeNet. Set the random seed to the default value and train the network. The training process usually takes 1-5 minutes on a desktop CPU.

```

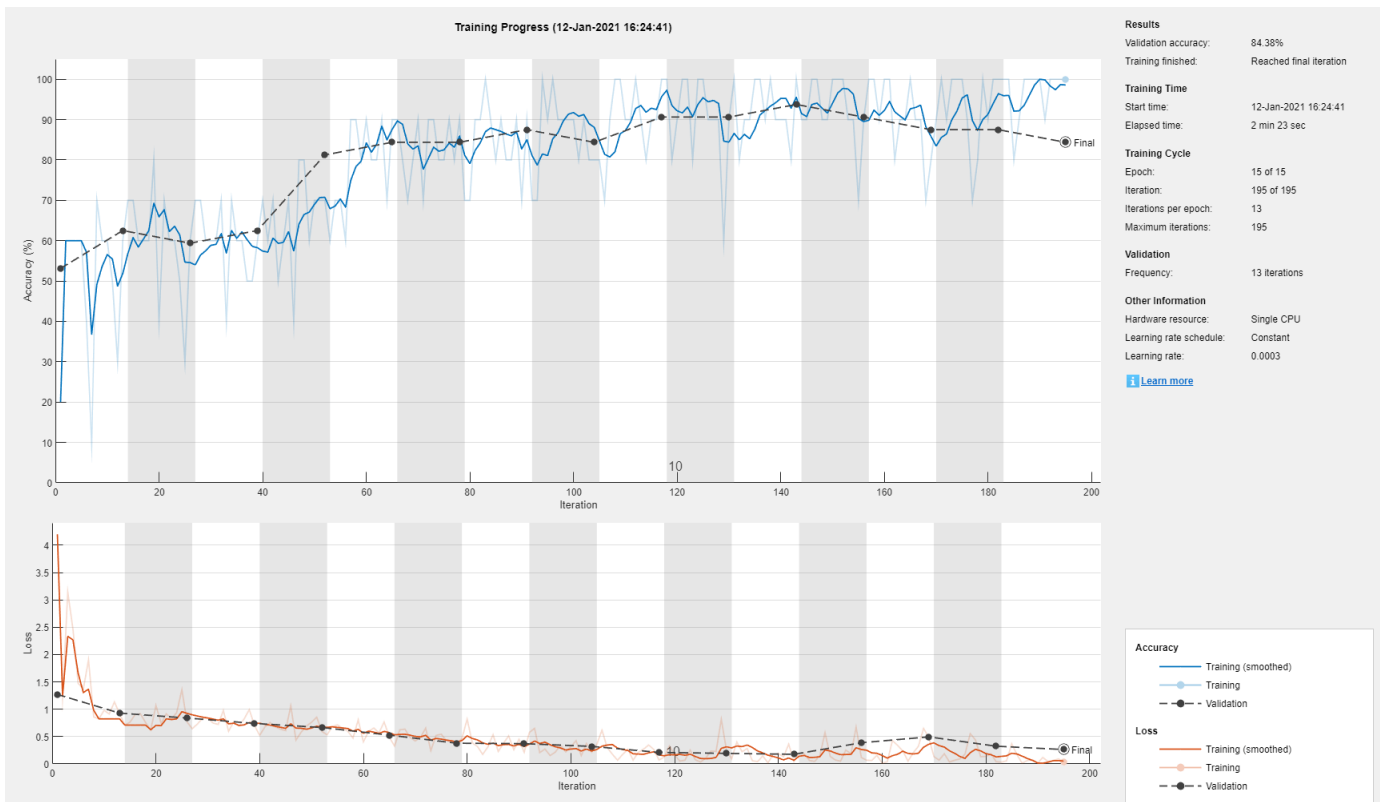
ilr = 3e-4;
miniBatchSize = 10;
maxEpochs = 15;
valFreq = floor(numel(augimgsTrain.Files)/miniBatchSize);
opts = trainingOptions('sgdm',...
    'MiniBatchSize',miniBatchSize,...
    'MaxEpochs',maxEpochs,...
    'InitialLearnRate',ilr,...
    'ValidationData',augimgsValidation,...
    'ValidationFrequency',valFreq,...
    'Verbose',1,...
    'ExecutionEnvironment','cpu',...
    'Plots','training-progress');

rng default
trainedSN = trainNetwork(augimgsTrain,lgraphSqz,opts);

```

Initializing input data normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Validation Accuracy	Mini-batch Loss	Validation Loss
1	1	00:00:06	20.00%	53.12%	4.2000	1.2000
1	13	00:00:16	60.00%	62.50%	0.9170	0.9170
2	26	00:00:25	60.00%	59.38%	0.7670	0.7670
3	39	00:00:35	60.00%	62.50%	0.7033	0.7033
4	50	00:00:42	70.00%		0.7629	
4	52	00:00:44	70.00%	81.25%	0.5941	0.5941
5	65	00:00:53	90.00%	84.38%	0.4883	0.4883
6	78	00:01:02	90.00%	84.38%	0.3627	0.3627
7	91	00:01:11	90.00%	87.50%	0.2145	0.2145
8	100	00:01:17	90.00%		0.3157	
8	104	00:01:20	80.00%	84.38%	0.2166	0.2166
9	117	00:01:29	100.00%	90.62%	0.0720	0.0720
10	130	00:01:38	90.00%	90.62%	0.2510	0.2510
11	143	00:01:47	100.00%	93.75%	0.0443	0.0443
12	150	00:01:51	100.00%		0.1377	
12	156	00:01:56	90.00%	90.62%	0.1190	0.1190
13	169	00:02:04	80.00%	87.50%	0.4859	0.4859
14	182	00:02:13	100.00%	87.50%	0.0395	0.0395
15	195	00:02:22	100.00%	84.38%	0.0399	0.0399



Inspect the last layer of the network. Confirm the Classification Output layer includes the three classes.

```

trainedSN.Layers(end)

ans =
  ClassificationOutputLayer with properties:

      Name: 'new_classoutput'
      Classes: [ARR    CHF    NSR]
      ClassWeights: 'none'
      OutputSize: 3

  Hyperparameters
      LossFunction: 'crossentropyex'

```

Evaluate SqueezeNet Accuracy

Evaluate the network using the validation data.

```

[YPred,probs] = classify(trainedSN,augimgsValidation);
accuracy = mean(YPred==imgsValidation.Labels);
disp(['SqueezeNet Accuracy: ',num2str(100*accuracy), '%'])

```

SqueezeNet Accuracy: 84.375%

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 2019.2. To set the Xilinx Vivado toolpath, enter:

```

% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.1
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 alexnet 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', trainedSN, 'Bitstream', 'zcu102_single','Target',hTarget)

hW =
  Workflow with properties:

      Network: [1x1 DAGNetwork]
      Bitstream: 'zcu102_single'
      ProcessorConfig: []
      Target: [1x1 dlhdl.Target]

```

Compile the Modified SqueezeNet DAG network

To compile the `trainedSN` DAG network, run 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' no
2	'conv1'	Convolution	64 3×3×3 convolutions with stride [2
3	'relu_conv1'	ReLU	ReLU
4	'pool1'	Max Pooling	3×3 max pooling with stride [2 2] ar
5	'fire2-squeeze1x1'	Convolution	16 1×1×64 convolutions with stride [1
6	'fire2-relu_squeeze1x1'	ReLU	ReLU
7	'fire2-expand1x1'	Convolution	64 1×1×16 convolutions with stride [1
8	'fire2-relu_expand1x1'	ReLU	ReLU
9	'fire2-expand3x3'	Convolution	64 3×3×16 convolutions with stride [1
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 stride
13	'fire3-relu_squeeze1x1'	ReLU	ReLU
14	'fire3-expand1x1'	Convolution	64 1×1×16 convolutions with stride [1
15	'fire3-relu_expand1x1'	ReLU	ReLU
16	'fire3-expand3x3'	Convolution	64 3×3×16 convolutions with stride [1
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] ar
20	'fire4-squeeze1x1'	Convolution	32 1×1×128 convolutions with stride
21	'fire4-relu_squeeze1x1'	ReLU	ReLU
22	'fire4-expand1x1'	Convolution	128 1×1×32 convolutions with stride
23	'fire4-relu_expand1x1'	ReLU	ReLU
24	'fire4-expand3x3'	Convolution	128 3×3×32 convolutions with stride
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 stride
28	'fire5-relu_squeeze1x1'	ReLU	ReLU
29	'fire5-expand1x1'	Convolution	128 1×1×32 convolutions with stride
30	'fire5-relu_expand1x1'	ReLU	ReLU
31	'fire5-expand3x3'	Convolution	128 3×3×32 convolutions with stride
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] ar
35	'fire6-squeeze1x1'	Convolution	48 1×1×256 convolutions with stride
36	'fire6-relu_squeeze1x1'	ReLU	ReLU
37	'fire6-expand1x1'	Convolution	192 1×1×48 convolutions with stride
38	'fire6-relu_expand1x1'	ReLU	ReLU
39	'fire6-expand3x3'	Convolution	192 3×3×48 convolutions with stride
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 stride
43	'fire7-relu_squeeze1x1'	ReLU	ReLU
44	'fire7-expand1x1'	Convolution	192 1×1×48 convolutions with stride
45	'fire7-relu_expand1x1'	ReLU	ReLU
46	'fire7-expand3x3'	Convolution	192 3×3×48 convolutions with stride
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 stride
50	'fire8-relu_squeeze1x1'	ReLU	ReLU
51	'fire8-expand1x1'	Convolution	256 1×1×64 convolutions with stride
52	'fire8-relu_expand1x1'	ReLU	ReLU
53	'fire8-expand3x3'	Convolution	256 3×3×64 convolutions with stride
54	'fire8-relu_expand3x3'	ReLU	ReLU

55	'fire8-concat'	Depth concatenation	Depth concatenation of 2 inputs
56	'fire9-squeeze1x1'	Convolution	64 1x1x512 convolutions with stride
57	'fire9-relu_squeeze1x1'	ReLU	ReLU
58	'fire9-expand1x1'	Convolution	256 1x1x64 convolutions with stride
59	'fire9-relu_expand1x1'	ReLU	ReLU
60	'fire9-expand3x3'	Convolution	256 3x3x64 convolutions with stride
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 1x1x512 convolutions with stride [
65	'relu_conv10'	ReLU	ReLU
66	'pool10'	Global Average Pooling	Global average pooling
67	'prob'	Softmax	softmax
68	'new_classoutput'	Classification Output	crossentropyex with 'ARR' and 2 other

4 Memory Regions created.

```

Skipping: data
Compiling leg: conv1>>fire2-relu_squeeze1x1 ...
Compiling leg: conv1>>fire2-relu_squeeze1x1 ... complete.
Compiling leg: fire2-expand1x1>>fire2-relu_expand1x1 ...
Compiling leg: fire2-expand1x1>>fire2-relu_expand1x1 ... complete.
Compiling leg: fire2-expand3x3>>fire2-relu_expand3x3 ...
Compiling leg: fire2-expand3x3>>fire2-relu_expand3x3 ... complete.
Do nothing: fire2-concat
Compiling leg: fire3-squeeze1x1>>fire3-relu_squeeze1x1 ...
Compiling leg: fire3-squeeze1x1>>fire3-relu_squeeze1x1 ... complete.
Compiling leg: fire3-expand1x1>>fire3-relu_expand1x1 ...
Compiling leg: fire3-expand1x1>>fire3-relu_expand1x1 ... complete.
Compiling leg: fire3-expand3x3>>fire3-relu_expand3x3 ...
Compiling leg: fire3-expand3x3>>fire3-relu_expand3x3 ... complete.
Do nothing: fire3-concat
Compiling leg: pool3>>fire4-relu_squeeze1x1 ...
Compiling leg: pool3>>fire4-relu_squeeze1x1 ... complete.
Compiling leg: fire4-expand1x1>>fire4-relu_expand1x1 ...
Compiling leg: fire4-expand1x1>>fire4-relu_expand1x1 ... complete.
Compiling leg: fire4-expand3x3>>fire4-relu_expand3x3 ...
Compiling leg: fire4-expand3x3>>fire4-relu_expand3x3 ... complete.
Do nothing: fire4-concat
Compiling leg: fire5-squeeze1x1>>fire5-relu_squeeze1x1 ...
Compiling leg: fire5-squeeze1x1>>fire5-relu_squeeze1x1 ... complete.
Compiling leg: fire5-expand1x1>>fire5-relu_expand1x1 ...
Compiling leg: fire5-expand1x1>>fire5-relu_expand1x1 ... complete.
Compiling leg: fire5-expand3x3>>fire5-relu_expand3x3 ...
Compiling leg: fire5-expand3x3>>fire5-relu_expand3x3 ... complete.
Do nothing: fire5-concat
Compiling leg: pool5>>fire6-relu_squeeze1x1 ...
Compiling leg: pool5>>fire6-relu_squeeze1x1 ... complete.
Compiling leg: fire6-expand1x1>>fire6-relu_expand1x1 ...
Compiling leg: fire6-expand1x1>>fire6-relu_expand1x1 ... complete.
Compiling leg: fire6-expand3x3>>fire6-relu_expand3x3 ...
Compiling leg: fire6-expand3x3>>fire6-relu_expand3x3 ... complete.
Do nothing: fire6-concat
Compiling leg: fire7-squeeze1x1>>fire7-relu_squeeze1x1 ...
Compiling leg: fire7-squeeze1x1>>fire7-relu_squeeze1x1 ... complete.
Compiling leg: fire7-expand1x1>>fire7-relu_expand1x1 ...
Compiling leg: fire7-expand1x1>>fire7-relu_expand1x1 ... complete.
Compiling leg: fire7-expand3x3>>fire7-relu_expand3x3 ...

```



```

Compiling leg: fire7-expand3x3>>fire7-relu_expand3x3 ... complete.
Do nothing: fire7-concat
Compiling leg: fire8-squeeze1x1>>fire8-relu_squeeze1x1 ...
Compiling leg: fire8-squeeze1x1>>fire8-relu_squeeze1x1 ... complete.
Compiling leg: fire8-expand1x1>>fire8-relu_expand1x1 ...
Compiling leg: fire8-expand1x1>>fire8-relu_expand1x1 ... complete.
Compiling leg: fire8-expand3x3>>fire8-relu_expand3x3 ...
Compiling leg: fire8-expand3x3>>fire8-relu_expand3x3 ... complete.
Do nothing: fire8-concat
Compiling leg: fire9-squeeze1x1>>fire9-relu_squeeze1x1 ...
Compiling leg: fire9-squeeze1x1>>fire9-relu_squeeze1x1 ... complete.
Compiling leg: fire9-expand1x1>>fire9-relu_expand1x1 ...
Compiling leg: fire9-expand1x1>>fire9-relu_expand1x1 ... complete.
Compiling leg: fire9-expand3x3>>fire9-relu_expand3x3 ...
Compiling leg: fire9-expand3x3>>fire9-relu_expand3x3 ... complete.
Do nothing: fire9-concat
Compiling leg: new_conv>>relu_conv10 ...
Compiling leg: new_conv>>relu_conv10 ... complete.
Compiling leg: pool10 ...
Compiling leg: pool10 ... complete.
Skipping: prob
Skipping: new_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"	"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"
"FCWeightDataOffset"	"0x04c00000"	"0.0 MB"
"EndOffset"	"0x04c00000"	"Total: 76.0 MB"

```
### Network compilation complete.
```

```

dn = struct with fields:
    weights: [1x1 struct]
    instructions: [1x1 struct]
    registers: [1x1 struct]
    syncInstructions: [1x1 struct]

```

Program Bitstream onto FPGA and Download Network Weights

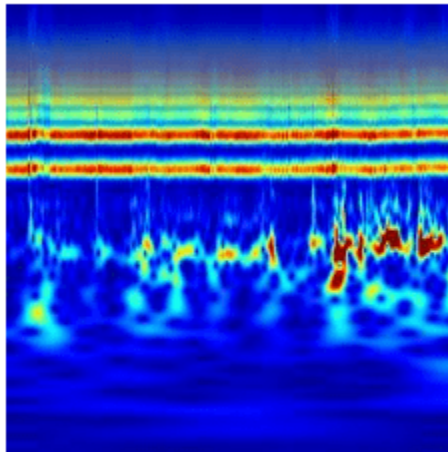
To deploy the network on the Xilinx ZCU102 hardware, run the `deploy` function of the `dlhdl.Workflow` object. This function uses the output of the `compile` function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The `deploy` function starts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

```
hw.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the target
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 12-Jan-2021 16:28:17
```

Load Image for Prediction and Run prediction

```
idx=randi(32);
testim=readimage(imgsValidation,idx);
im=imresize(testim,[227 227]);
imshow(testim)
```



```
[YPred1,probs1] = classify(trainedSN,im);
accuracy1 = (YPred1==imgsValidation.Labels);
[YPred2,probs2] = hw.predict(single(im),'profile','on');
```

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

Deep Learning Processor Profiler Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
Network	8847610	0.04022	1	8847610
conv1	626502	0.00285		

pool1	579473	0.00263
fire2-squeeze1x1	308065	0.00140
fire2-expand1x1	305121	0.00139
fire2-expand3x3	305091	0.00139
fire3-squeeze1x1	624849	0.00284
fire3-expand1x1	305136	0.00139
fire3-expand3x3	305587	0.00139
pool3	290789	0.00132
fire4-squeeze1x1	262881	0.00119
fire4-expand1x1	263129	0.00120
fire4-expand3x3	262617	0.00119
fire5-squeeze1x1	703951	0.00320
fire5-expand1x1	262552	0.00119
fire5-expand3x3	262599	0.00119
pool5	216737	0.00099
fire6-squeeze1x1	192738	0.00088
fire6-expand1x1	142333	0.00065
fire6-expand3x3	142132	0.00065
fire7-squeeze1x1	286437	0.00130
fire7-expand1x1	142363	0.00065
fire7-expand3x3	142079	0.00065
fire8-squeeze1x1	364915	0.00166
fire8-expand1x1	240660	0.00109
fire8-expand3x3	240946	0.00110
fire9-squeeze1x1	483766	0.00220
fire9-expand1x1	240624	0.00109
fire9-expand3x3	241242	0.00110
new_conv	93673	0.00043
pool10	6135	0.00003

* The clock frequency of the DL processor is: 220MHz

```
accuracy2 = (YPred==imgsValidation.Labels);
[val,idx]= max(YPred2);
trainedSN.Layers(end).ClassNames{idx}
```

```
ans =
'CHF'
```

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)
% This function is only intended to support the ECGAndDeepLearningExample.
% It may change or be removed in a future release.
```

```
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)
% This function is only intended to support the ECGAndDeepLearningExample.
% It may change or be removed in a future release.

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)
% This function is only intended to support the ECGAndDeepLearningExample.
% It may change or be removed in a future release.

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,[224 224]),fullfile(imgLoc,imFileName));
end
end
```

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 Libraries™

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:

    Layers: [71x1 nnet.cnn.layer.Layer]
    Connections: [78x2 table]
    InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}

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.

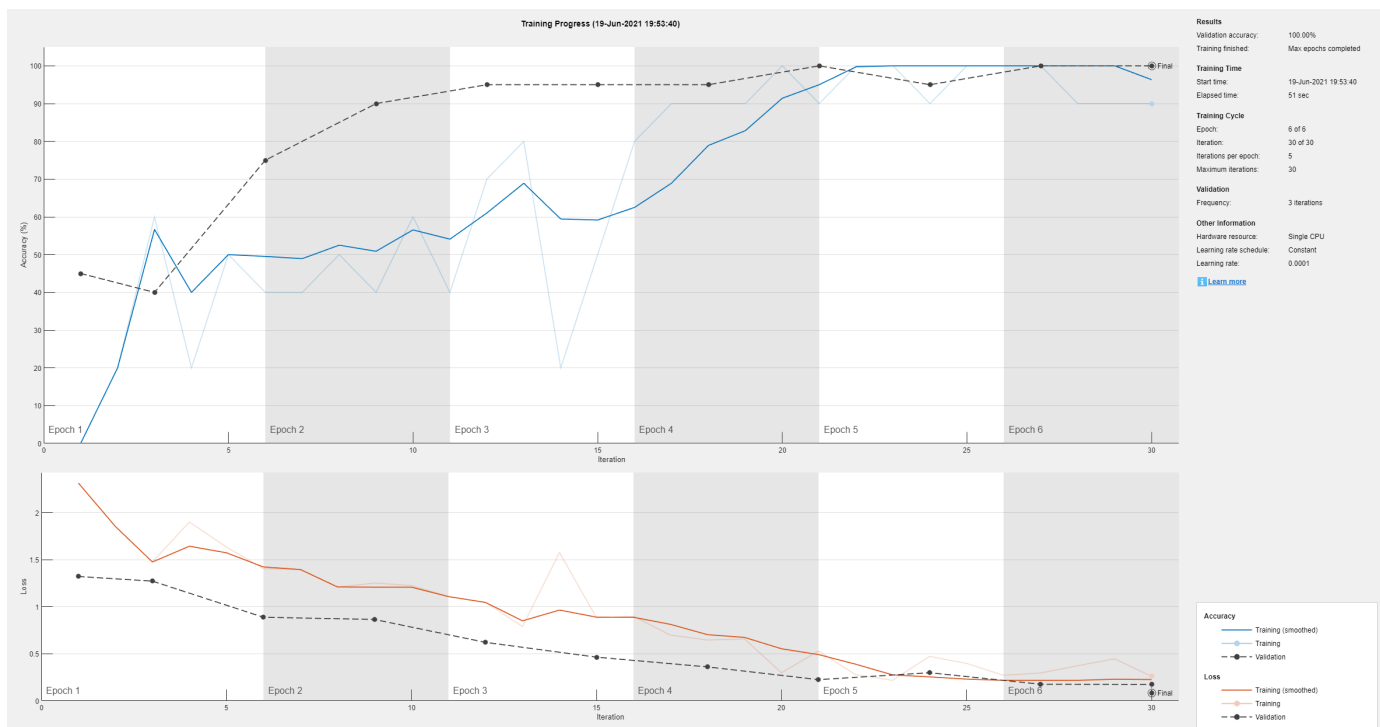
```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

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 Support by Release” (Parallel Computing Toolbox)). 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);
```



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 series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.lay
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'out_data' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented
Compiling leg: conv1>>pool1 ...
### 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
Compiling leg: conv1>>pool1 ... complete.
Compiling leg: res2a_branch2a>>res2a_branch2b ...
### Notice: The layer 'pool1' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res2a_branch2a>>res2a_branch2b ... complete.
Compiling leg: res2b_branch2a>>res2b_branch2b ...
### Notice: The layer 'res2a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res2b_branch2a>>res2b_branch2b ... complete.
Compiling leg: res3a_branch1 ...
### Notice: The layer 'res2b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res3a_branch1 ... complete.
Compiling leg: res3a_branch2a>>res3a_branch2b ...
### Notice: The layer 'res2b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res3a_branch2a>>res3a_branch2b ... complete.
Compiling leg: res3b_branch2a>>res3b_branch2b ...
### Notice: The layer 'res3a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res3b_branch2a>>res3b_branch2b ... complete.
Compiling leg: res4a_branch1 ...
### Notice: The layer 'res3b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res4a_branch1 ... complete.
Compiling leg: res4a_branch2a>>res4a_branch2b ...
### Notice: The layer 'res3b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res4a_branch2a>>res4a_branch2b ... complete.
Compiling leg: res4b_branch2a>>res4b_branch2b ...
### Notice: The layer 'res4a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res4b_branch2a>>res4b_branch2b ... complete.
Compiling leg: res5a_branch1 ...
### Notice: The layer 'res4b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res5a_branch1 ... complete.
Compiling leg: res5a_branch2a>>res5a_branch2b ...
### Notice: The layer 'res4b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res5a_branch2a>>res5a_branch2b ... complete.
Compiling leg: res5b_branch2a>>res5b_branch2b ...
### Notice: The layer 'res5a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res5b_branch2a>>res5b_branch2b ... complete.
Compiling leg: pool5 ...
### Notice: The layer 'res5b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: pool5 ... complete.
Compiling leg: new_fc ...
### Notice: The layer 'pool5' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
```



```

    Compiling leg: new_fc ... complete.
    ### Should not enter here. It means a component is unaccounted for in MATLAB Emulation.
    ### Notice: The layer 'new_fc' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softw
    ### 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
    
```

Load Image for Prediction and Intermediate Layer Activation Results

Load the example image. Save its 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.

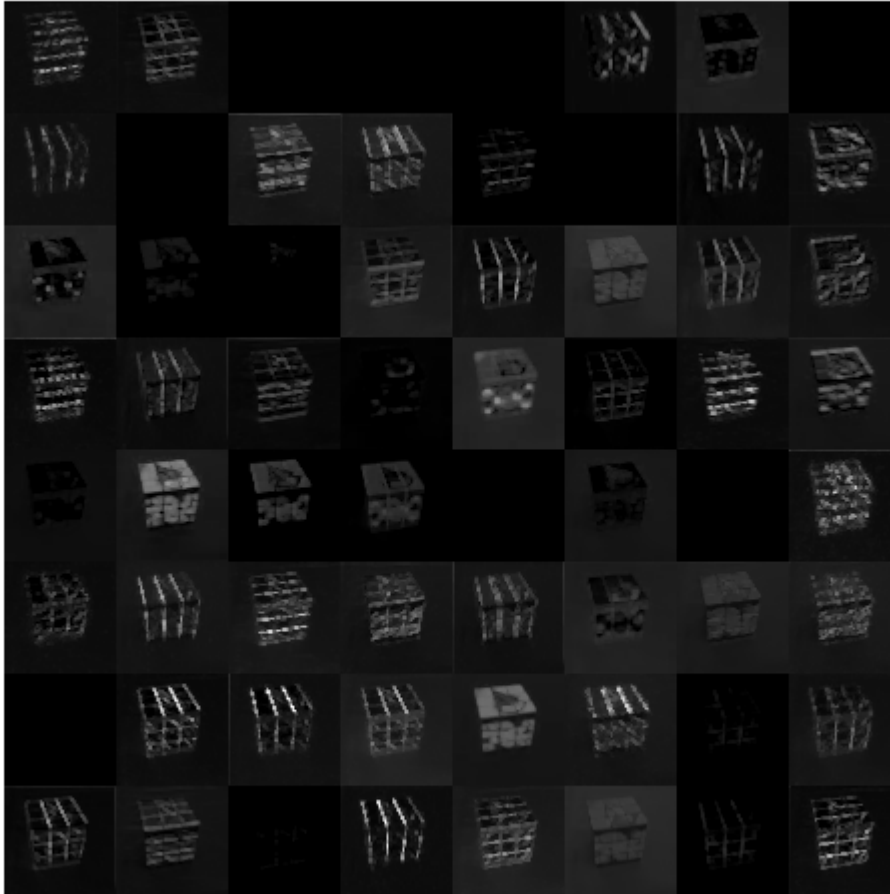
```
act1 = simObj.activations(single(inputImg), 'pool1');
```

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)
```

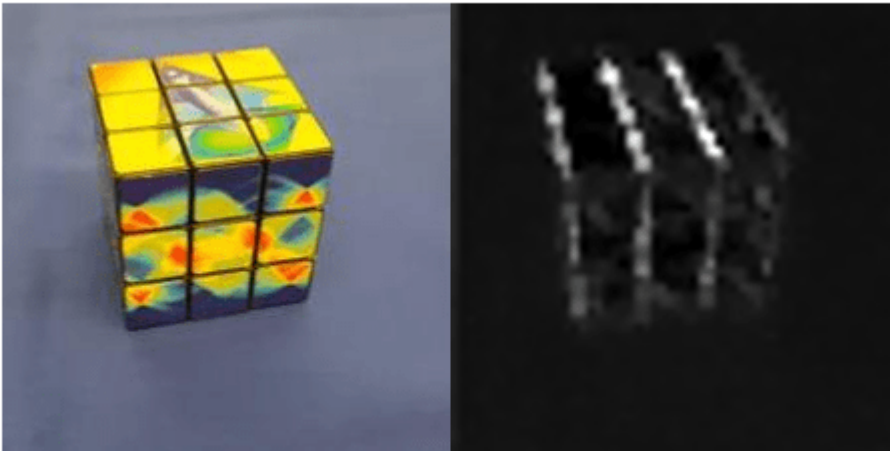


Find Strongest Activation Channel

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 shows 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*).

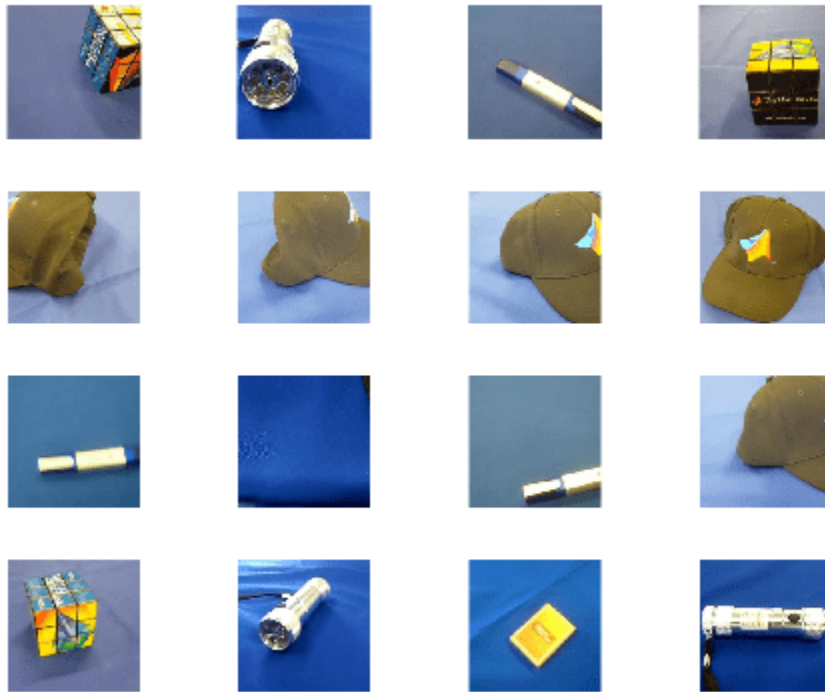
```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

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: [144x1 nnet.cnn.layer.Layer]
    Connections: [170x2 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', 'newFC'),
    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. For more information, see GPU Support by Release (Parallel Computing Toolbox)). 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 18.1 executable file. You must have already installed Altera® Quartus II.

```
% hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\intel\18.1\quartus\bin\quartus.
```

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');
```

Alternatively, you can use an Ethernet interface.

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

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 w
7	'conv2-relu_3x3_reduce'	ReLU	ReLU
8	'conv2-3x3'	Convolution	192 3×3×64 convolutions v
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 v
13	'inception_3a-relu_1x1'	ReLU	ReLU
14	'inception_3a-3x3_reduce'	Convolution	96 1×1×192 convolutions v
15	'inception_3a-relu_3x3_reduce'	ReLU	ReLU
16	'inception_3a-3x3'	Convolution	128 3×3×96 convolutions v
17	'inception_3a-relu_3x3'	ReLU	ReLU
18	'inception_3a-5x5_reduce'	Convolution	16 1×1×192 convolutions v
19	'inception_3a-relu_5x5_reduce'	ReLU	ReLU
20	'inception_3a-5x5'	Convolution	32 5×5×16 convolutions w
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 v
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 w
35	'inception_3b-relu_5x5'	ReLU	ReLU
36	'inception_3b-pool'	Max Pooling	3×3 max pooling with str:
37	'inception_3b-pool_proj'	Convolution	64 1×1×256 convolutions v
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 str:
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 v
44	'inception_4a-relu_3x3_reduce'	ReLU	ReLU
45	'inception_4a-3x3'	Convolution	208 3×3×96 convolutions v
46	'inception_4a-relu_3x3'	ReLU	ReLU
47	'inception_4a-5x5_reduce'	Convolution	16 1×1×480 convolutions v
48	'inception_4a-relu_5x5_reduce'	ReLU	ReLU
49	'inception_4a-5x5'	Convolution	48 5×5×16 convolutions w
50	'inception_4a-relu_5x5'	ReLU	ReLU
51	'inception_4a-pool'	Max Pooling	3×3 max pooling with str:
52	'inception_4a-pool_proj'	Convolution	64 1×1×480 convolutions v
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 v
62	'inception_4b-relu_5x5_reduce'	ReLU	ReLU
63	'inception_4b-5x5'	Convolution	64 5×5×24 convolutions w
64	'inception_4b-relu_5x5'	ReLU	ReLU
65	'inception_4b-pool'	Max Pooling	3×3 max pooling with str:
66	'inception_4b-pool_proj'	Convolution	64 1×1×512 convolutions v
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 v
76	'inception_4c-relu_5x5_reduce'	ReLU	ReLU
77	'inception_4c-5x5'	Convolution	64 5×5×24 convolutions w
78	'inception_4c-relu_5x5'	ReLU	ReLU
79	'inception_4c-pool'	Max Pooling	3×3 max pooling with str:
80	'inception_4c-pool_proj'	Convolution	64 1×1×512 convolutions v
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 1x1x512 convolutions w
90	'inception_4d-relu_5x5_reduce'	ReLU	ReLU
91	'inception_4d-5x5'	Convolution	64 5x5x32 convolutions w
92	'inception_4d-relu_5x5'	ReLU	ReLU
93	'inception_4d-pool'	Max Pooling	3x3 max pooling with str
94	'inception_4d-pool_proj'	Convolution	64 1x1x512 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 1x1x528 convolutions
98	'inception_4e-relu_1x1'	ReLU	ReLU
99	'inception_4e-3x3_reduce'	Convolution	160 1x1x528 convolutions
100	'inception_4e-relu_3x3_reduce'	ReLU	ReLU
101	'inception_4e-3x3'	Convolution	320 3x3x160 convolutions
102	'inception_4e-relu_3x3'	ReLU	ReLU
103	'inception_4e-5x5_reduce'	Convolution	32 1x1x528 convolutions
104	'inception_4e-relu_5x5_reduce'	ReLU	ReLU
105	'inception_4e-5x5'	Convolution	128 5x5x32 convolutions
106	'inception_4e-relu_5x5'	ReLU	ReLU
107	'inception_4e-pool'	Max Pooling	3x3 max pooling with str
108	'inception_4e-pool_proj'	Convolution	128 1x1x528 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	3x3 max pooling with str
112	'inception_5a-1x1'	Convolution	256 1x1x832 convolutions
113	'inception_5a-relu_1x1'	ReLU	ReLU
114	'inception_5a-3x3_reduce'	Convolution	160 1x1x832 convolutions
115	'inception_5a-relu_3x3_reduce'	ReLU	ReLU
116	'inception_5a-3x3'	Convolution	320 3x3x160 convolutions
117	'inception_5a-relu_3x3'	ReLU	ReLU
118	'inception_5a-5x5_reduce'	Convolution	32 1x1x832 convolutions
119	'inception_5a-relu_5x5_reduce'	ReLU	ReLU
120	'inception_5a-5x5'	Convolution	128 5x5x32 convolutions
121	'inception_5a-relu_5x5'	ReLU	ReLU
122	'inception_5a-pool'	Max Pooling	3x3 max pooling with str
123	'inception_5a-pool_proj'	Convolution	128 1x1x832 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 1x1x832 convolutions
127	'inception_5b-relu_1x1'	ReLU	ReLU
128	'inception_5b-3x3_reduce'	Convolution	192 1x1x832 convolutions
129	'inception_5b-relu_3x3_reduce'	ReLU	ReLU
130	'inception_5b-3x3'	Convolution	384 3x3x192 convolutions
131	'inception_5b-relu_3x3'	ReLU	ReLU
132	'inception_5b-5x5_reduce'	Convolution	48 1x1x832 convolutions
133	'inception_5b-relu_5x5_reduce'	ReLU	ReLU
134	'inception_5b-5x5'	Convolution	128 5x5x48 convolutions
135	'inception_5b-relu_5x5'	ReLU	ReLU
136	'inception_5b-pool'	Max Pooling	3x3 max pooling with str
137	'inception_5b-pool_proj'	Convolution	128 1x1x832 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 pool
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 software
### Notice: The layer 'newClassOutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is in
### 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: [1x1 struct]
    instructions: [1x1 struct]
    registers: [1x1 struct]
    syncInstructions: [1x1 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	Total
	-----	-----	-----	-----
Network	15836394	0.10558	1	15
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=75x5 table

	Latency(cycles)	Latency(seconds)	NumFrames	Total Latency
Network	1.5836e+07	0.10558	"1"	"158"
___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.

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 = 'Kintex Ultrascale+';
hPC.SynthesisToolDeviceName = 'xcku1lp-ffva1156-1-e';
hPC.estimateResources
```

Warning: Device family "Kintex Ultrascale+" is not currently supported for LUT Estimation. Support

Deep Learning Processor Estimator Resource Results

	DSPs	Block RAM*	LUTs(CLB/ALUT)
Available	2928	600	298560
DL_Processor	381(14%)	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%)	255074(94%)
ReferenceDesign	3(1%)	78(9%)	35000(13%)
DL_Processor	381(16%)	508(56%)	220074(81%)

* 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 dldhdl_device_registration.m
```

```
function hFPGADeviceFamily = dldhdl_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', 'Kintex Ultrascale+');
hFPGADeviceFamily.Vendor = 'Xilinx';
hFPGADeviceFamily.DSPWidth = [27, 18];
hFPGADeviceFamily.RAMWidth = 36;
```

```
hFPGADeviceFamily.SplitRAMWidth = 18;  
hFPGADeviceFamily.RAMDepth = 1024;  
end
```

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 “Speech Command Recognition 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(classWeights)` 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`.

```
ads = audioDataStore(fullfile(dataFolder, 'train'), ...
    'IncludeSubfolders',true, ...
    'FileExtensions','.wav', ...
    'LabelSource','foldernames')
```

```
ads =
    audioDatastore with properties:
```

```
        Files: {
            '...\AppData\Local\Temp\google_speech\train\bed\00176480_nohash_0
            '...\AppData\Local\Temp\google_speech\train\bed\004ae714_nohash_0
            '...\AppData\Local\Temp\google_speech\train\bed\004ae714_nohash_1
            ... and 40393 more
        }
        Folders: {
            'C:\Users\skapali\AppData\Local\Temp\google_speech\train'
        }
        Labels: [bed; bed; bed ... and 40393 more categorical]
AlternateFileSystemRoots: {}
        OutputDataType: 'double'
SupportedOutputFormats: ["wav"      "flac"      "ogg"      "mp4"      "m4a"]
DefaultOutputFormat: "wav"
```

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). 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=11x2 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')

```

```

ads =
    audioDatastore with properties:

```

```

        Files: {
            '...\AppData\Local\Temp\google_speech\validation\bed\026290a7_noha
            '...\AppData\Local\Temp\google_speech\validation\bed\060cd039_noha
            '...\AppData\Local\Temp\google_speech\validation\bed\060cd039_noha
            ... and 5457 more
        }
        Folders: {
            'C:\Users\skapali\AppData\Local\Temp\google_speech\validation'
        }
        Labels: [bed; bed; bed ... and 5457 more categorical]
AlternateFileSystemRoots: {}
        OutputDataType: 'double'
SupportedOutputFormats: ["wav"    "flac"    "ogg"    "mp4"    "m4a"]
DefaultOutputFormat: "wav"

```

```

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=11x2 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) / numUniqueLabels));
end

```

Compute Auditory Spectrograms

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) 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(x,1))/2),1)];
        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(x,1))/2),1)];
        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'))
```

```
adsBkg =
    audioDatastore with properties:
```

```
Files: {
    ' ... \AppData\Local\Temp\google_speech\background\doing_the_dishes
    ' ... \skapali\AppData\Local\Temp\google_speech\background\dude_mia
    ' ... \skapali\AppData\Local\Temp\google_speech\background\exercise
```

```

        ... and 3 more
    }
    Folders: {
        'C:\Users\skapali\AppData\Local\Temp\google_speech\background'
    }
    AlternateFileSystemRoots: {}
    OutputDataType: 'double'
    Labels: {}
    SupportedOutputFormats: ["wav"    "flac"    "ogg"    "mp4"    "m4a"]
    DefaultOutputFormat: "wav"

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           98x50x1 images with 'zerocenter' normalization
 2  'conv_1'       Convolution           12 3x3x1 convolutions with stride [1 1] and padding
 3  'batchnorm_1' Batch Normalization  Batch normalization with 12 channels
 4  'relu_1'       ReLU                  ReLU
 5  'maxpool_1'    Max Pooling           3x3 max pooling with stride [2 2] and padding
 6  'conv_2'       Convolution           24 3x3x12 convolutions with stride [1 1] and padding
 7  'batchnorm_2' Batch Normalization  Batch normalization with 24 channels
 8  'relu_2'       ReLU                  ReLU
 9  'maxpool_2'    Max Pooling           3x3 max pooling with stride [2 2] and padding
10  'conv_3'       Convolution           48 3x3x24 convolutions with stride [1 1] and padding
11  'batchnorm_3' Batch Normalization  Batch normalization with 48 channels
12  'relu_3'       ReLU                  ReLU
13  'maxpool_3'    Max Pooling           3x3 max pooling with stride [2 2] and padding
14  'conv_4'       Convolution           48 3x3x48 convolutions with stride [1 1] and padding
15  'batchnorm_4' Batch Normalization  Batch normalization with 48 channels
16  'relu_4'       ReLU                  ReLU
17  'conv_5'       Convolution           48 3x3x48 convolutions with stride [1 1] and padding
18  'batchnorm_5' Batch Normalization  Batch normalization with 48 channels
19  'relu_5'       ReLU                  ReLU
20  'maxpool_4'    Max Pooling           13x1 max pooling with stride [1 1] and padding

```



```
# 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 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	Total
	-----	-----	-----	-----
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 SoCs Workflow" 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: [28x1 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) PAM4-modulate the bits
- `rcosdesign` (Signal Processing Toolbox): Design a square-root raised cosine pulse shaping filter
- `filter`: Pulse shape the symbols
- `comm.RicianChannel` (Communications Toolbox): Apply Rician multipath channel
- `comm.PhaseFrequencyOffset` (Communications Toolbox): Apply phase and frequency shift due to clock offset
- `interp1`: Apply timing drift due to clock offset
- `awgn` (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);

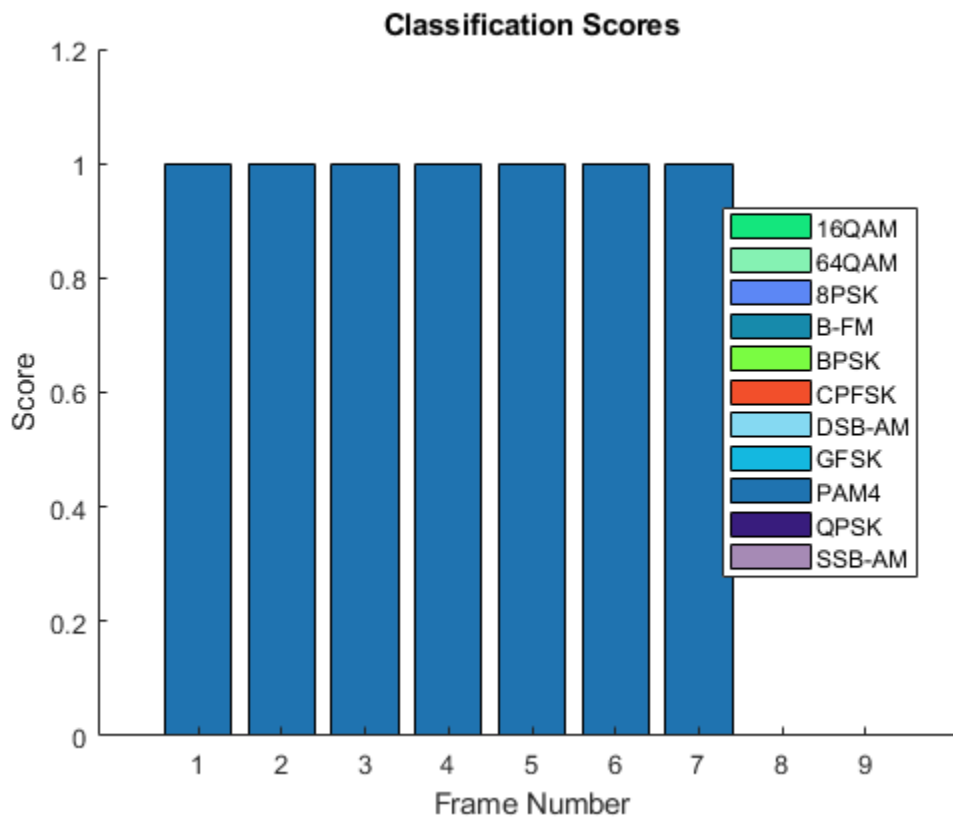
% Classification
[prediction1,score1] = classify(trainedNet,unknownFrames);

```

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) function.

Rician Multipath

The channel passes the signals through a Rician multipath fading channel by using the `comm.RicianChannel` (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 $C=1+\Delta\text{clock}/10^6$, 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 $[-\text{max}\Delta\text{clock} \text{max}\Delta\text{clock}]$, where $\text{max}\Delta\text{clock}$ 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).

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)

Data file directory is C:\Users\skapali\AppData\Local\Temp\ModClassDataFiles

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

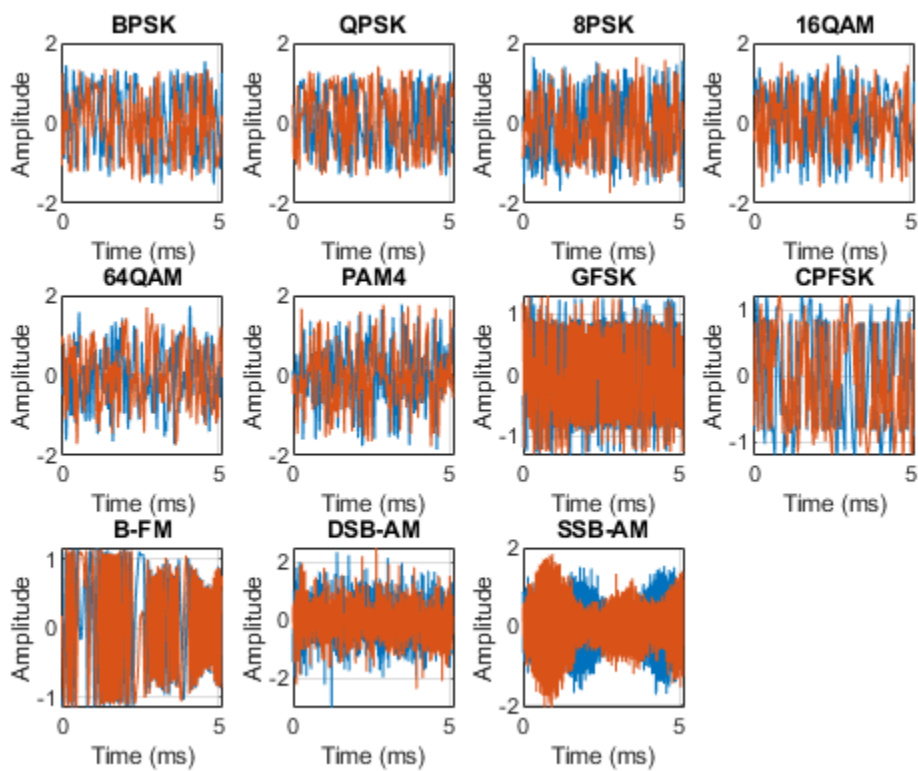
```

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)

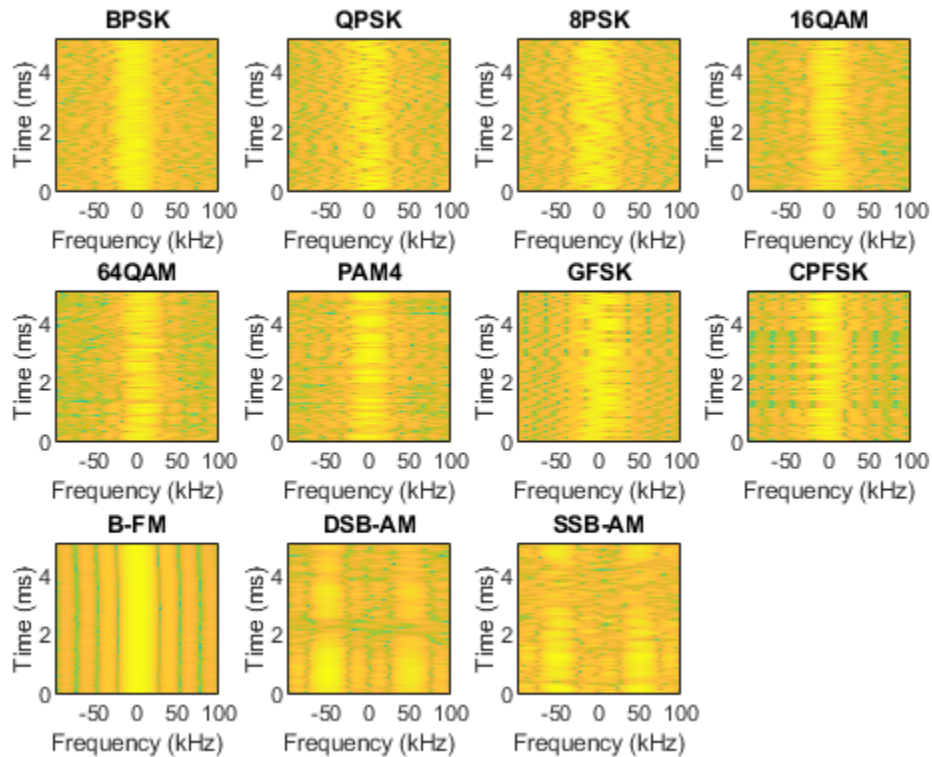
```



```

% 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      1x1024x2 images
  2 'CNN1'        Convolution      16 1x8x2 convolutions with stride [1 1] and padding
  3 'BN1'         Batch Normalization Batch normalization with 16 channels
  4 'ReLU1'       ReLU              ReLU
  5 'MaxPool1'    Max Pooling        1x2 max pooling with stride [1 2] and padding
  6 'CNN2'        Convolution      24 1x8x16 convolutions with stride [1 1] and padding
  7 'BN2'         Batch Normalization Batch normalization with 24 channels
  8 'ReLU2'       ReLU              ReLU
  9 'MaxPool2'    Max Pooling        1x2 max pooling with stride [1 2] and padding
 10 'CNN3'        Convolution      32 1x8x24 convolutions with stride [1 1] and padding
 11 'BN3'         Batch Normalization Batch normalization with 32 channels
 12 'ReLU3'       ReLU              ReLU
 13 'MaxPool3'    Max Pooling        1x2 max pooling with stride [1 2] and padding
 14 'CNN4'        Convolution      48 1x8x32 convolutions with stride [1 1] and padding
 15 'BN4'         Batch Normalization Batch normalization with 48 channels
 16 'ReLU4'       ReLU              ReLU
 17 'MaxPool4'    Max Pooling        1x2 max pooling with stride [1 2] and padding
 18 'CNN5'        Convolution      64 1x8x48 convolutions with stride [1 1] and padding
 19 'BN5'         Batch Normalization Batch normalization with 64 channels
 20 'ReLU5'       ReLU              ReLU
 21 'MaxPool5'    Max Pooling        1x2 max pooling with stride [1 2] and padding
 22 'CNN6'        Convolution      96 1x8x64 convolutions with stride [1 1] and padding
 23 'BN6'         Batch Normalization Batch normalization with 96 channels
 24 'ReLU6'       ReLU              ReLU
 25 'AP1'         Average Pooling    1x32 average pooling with stride [1 1] and padding
 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.ConvolutionLayer'
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool1' made symmetric as it is
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool2' made symmetric as it is
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool3' made symmetric as it is
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool4' made symmetric as it is
### Optimizing network: Non-symmetric stride of layer with name 'MaxPool5' made symmetric as it is
### Notice: The layer 'Input Layer' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in hardware
### Notice: The layer 'SoftMax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software
### Notice: The layer 'Output' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software
### 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: [1x1 struct]
    instructions: [1x1 struct]
    registers: [1x1 struct]
    syncInstructions: [1x1 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 target
### 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	Total
--------------------------	---------------------------	-----------	-------

Network	-----	-----	-----
Network	656546	0.00298	5
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 SoCs Workflow" 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      3x3x4 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 software
### Notice: The layer 'output' with type 'nnet.cnn.layer.ReggressionOutputLayer' is implemented in software

### 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: [1x1 struct]
      registers: []
      syncInstructions: [1x1 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	3aWRD.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.l
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 FPGAI0 i.e, Write/Read FPGA mu
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 software.

```

```

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

```

```

### 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.

```

Eventhough 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: [75x1 nnet.cnn.layer.Layer]
    Connections: [82x2 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 2020.1. To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.1\bin\vivado.l
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 stride
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 stride
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 stride
10	'bn2a_branch2b'	Batch Normalization	Batch normalization with 64 chan
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
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 stride
17	'bn2b_branch2b'	Batch Normalization	Batch normalization with 64 chan
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 chan
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 chan
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 chan
29	'res3b_branch2a'	Convolution	128 3×3×128 convolutions with stride
30	'bn3b_branch2a'	Batch Normalization	Batch normalization with 128 chan
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 chan
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 chan
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 chan

41	'res4a'	Addition	Element-wise addition of 2 inputs
42	'res4a_relu'	ReLU	ReLU
43	'res4a_branch1'	Convolution	256 1x1x128 convolutions with stride 1
44	'bn4a_branch1'	Batch Normalization	Batch normalization with 256 channels
45	'res4b_branch2a'	Convolution	256 3x3x256 convolutions with stride 1
46	'bn4b_branch2a'	Batch Normalization	Batch normalization with 256 channels
47	'res4b_branch2a_relu'	ReLU	ReLU
48	'res4b_branch2b'	Convolution	256 3x3x256 convolutions with stride 1
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	'res5a_branch2a'	Convolution	512 3x3x256 convolutions with stride 1
53	'bn5a_branch2a'	Batch Normalization	Batch normalization with 512 channels
54	'res5a_branch2a_relu'	ReLU	ReLU
55	'res5a_branch2b'	Convolution	512 3x3x512 convolutions with stride 1
56	'bn5a_branch2b'	Batch Normalization	Batch normalization with 512 channels
57	'res5a'	Addition	Element-wise addition of 2 inputs
58	'res5a_relu'	ReLU	ReLU
59	'res5a_branch1'	Convolution	512 1x1x256 convolutions with stride 1
60	'bn5a_branch1'	Batch Normalization	Batch normalization with 512 channels
61	'res5b_branch2a'	Convolution	512 3x3x512 convolutions with stride 1
62	'bn5b_branch2a'	Batch Normalization	Batch normalization with 512 channels
63	'res5b_branch2a_relu'	ReLU	ReLU
64	'res5b_branch2b'	Convolution	512 3x3x512 convolutions with stride 1
65	'bn5b_branch2b'	Batch Normalization	Batch normalization with 512 channels
66	'res5b'	Addition	Element-wise addition of 2 inputs
67	'res5b_relu'	ReLU	ReLU
68	'transposed-conv_1'	Transposed Convolution	256 4x4x512 transposed convolution
69	'relu_1'	ReLU	ReLU
70	'transposed-conv_2'	Transposed Convolution	256 4x4x256 transposed convolution
71	'relu_2'	ReLU	ReLU
72	'transposed-conv_3'	Transposed Convolution	256 4x4x256 transposed convolution
73	'relu_3'	ReLU	ReLU
74	'conv2d_final'	Convolution	17 1x1x256 convolutions with stride 1
75	'RegressionLayer_conv15_fwd'	Regression Output	mean-squared-error

```

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.ConvolutionLayer'
### Notice: The layer 'transposed-conv_1' of type 'nnet.cnn.layer.TransposedConvolution2DLayer' is fused into 'conv1'
### Notice: The layer 'transposed-conv_2' of type 'nnet.cnn.layer.TransposedConvolution2DLayer' is fused into 'conv2'
### Notice: The layer 'transposed-conv_3' of type 'nnet.cnn.layer.TransposedConvolution2DLayer' is fused into 'conv3'
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data' and a constant layer 'data_offset'
### Notice: The layer 'RegressionLayer_conv15_fwd' with type 'nnet.cnn.layer.RegistrationOutputLayer' is fused into 'conv15'
### 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
```

```
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(ircrop(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.
```

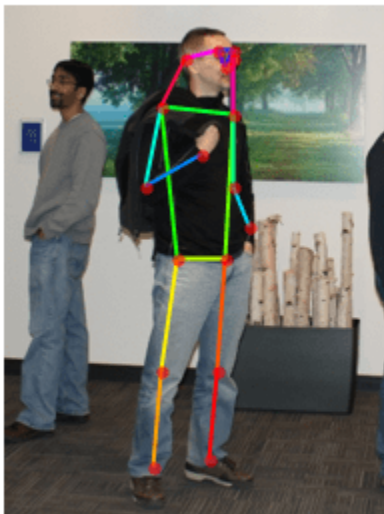
Deep Learning Processor Profiler Performance Results

	LastFrameLatency(cycles)	LastFrameLatency(seconds)	FramesNum	Total
	-----	-----	-----	-----
Network	106379104	0.48354	1	106379104
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 shows 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 Libraries

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 10-0]. 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 = 'http://www.cis.rit.edu/~rmk6217/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=103x5 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 v
2	'Encoder-Section-1-Conv-1'	Convolution	64 3×3×6 convolut
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 w
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	2x2 max pooling w/
12	'Encoder-Section-3-Conv-1'	Convolution	256 3x3x128 convol
13	'Encoder-Section-3-ReLU-1'	ReLU	ReLU
14	'Encoder-Section-3-Conv-2'	Convolution	256 3x3x256 convol
15	'Encoder-Section-3-ReLU-2'	ReLU	ReLU
16	'Encoder-Section-3-MaxPool'	Max Pooling	2x2 max pooling w/
17	'Encoder-Section-4-Conv-1'	Convolution	512 3x3x256 convol
18	'Encoder-Section-4-ReLU-1'	ReLU	ReLU
19	'Encoder-Section-4-Conv-2'	Convolution	512 3x3x512 convol
20	'Encoder-Section-4-ReLU-2'	ReLU	ReLU
21	'Encoder-Section-4-DropOut'	Dropout	50% dropout
22	'Encoder-Section-4-MaxPool'	Max Pooling	2x2 max pooling w/
23	'Mid-Conv-1'	Convolution	1024 3x3x512 convol
24	'Mid-ReLU-1'	ReLU	ReLU
25	'Mid-Conv-2'	Convolution	1024 3x3x1024 convol
26	'Mid-ReLU-2'	ReLU	ReLU
27	'Mid-DropOut'	Dropout	50% dropout
28	'Decoder-Section-1-UpConv'	Transposed Convolution	512 2x2x1024 transp
29	'Decoder-Section-1-UpReLU'	ReLU	ReLU
30	'Decoder-Section-1-DepthConcatenation'	Depth concatenation	Depth concatenati
31	'Decoder-Section-1-Conv-1'	Convolution	512 3x3x1024 convol
32	'Decoder-Section-1-ReLU-1'	ReLU	ReLU
33	'Decoder-Section-1-Conv-2'	Convolution	512 3x3x512 convol
34	'Decoder-Section-1-ReLU-2'	ReLU	ReLU
35	'Decoder-Section-2-UpConv'	Transposed Convolution	256 2x2x512 transp
36	'Decoder-Section-2-UpReLU'	ReLU	ReLU
37	'Decoder-Section-2-DepthConcatenation'	Depth concatenation	Depth concatenati
38	'Decoder-Section-2-Conv-1'	Convolution	256 3x3x512 convol
39	'Decoder-Section-2-ReLU-1'	ReLU	ReLU
40	'Decoder-Section-2-Conv-2'	Convolution	256 3x3x256 convol
41	'Decoder-Section-2-ReLU-2'	ReLU	ReLU
42	'Decoder-Section-3-UpConv'	Transposed Convolution	128 2x2x256 transp
43	'Decoder-Section-3-UpReLU'	ReLU	ReLU
44	'Decoder-Section-3-DepthConcatenation'	Depth concatenation	Depth concatenati
45	'Decoder-Section-3-Conv-1'	Convolution	128 3x3x256 convol
46	'Decoder-Section-3-ReLU-1'	ReLU	ReLU
47	'Decoder-Section-3-Conv-2'	Convolution	128 3x3x128 convol
48	'Decoder-Section-3-ReLU-2'	ReLU	ReLU
49	'Decoder-Section-4-UpConv'	Transposed Convolution	64 2x2x128 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 3x3x128 convol
53	'Decoder-Section-4-ReLU-1'	ReLU	ReLU
54	'Decoder-Section-4-Conv-2'	Convolution	64 3x3x64 convolut
55	'Decoder-Section-4-ReLU-2'	ReLU	ReLU
56	'Final-ConvolutionLayer'	Convolution	18 1x1x64 convolut
57	'Softmax-Layer'	Softmax	softmax
58	'Segmentation-Layer'	Pixel Classification Layer	Cross-entropy loss

```

### Notice: The layer 'Decoder-Section-1-UpConv' of type 'nnet.cnn.layer.TransposedConvolution2D' is implemented in 'nnet.cnn.layer.TransposedConvolution2D'
### Notice: The layer 'Decoder-Section-2-UpConv' of type 'nnet.cnn.layer.TransposedConvolution2D' is implemented in 'nnet.cnn.layer.TransposedConvolution2D'
### Notice: The layer 'Decoder-Section-3-UpConv' of type 'nnet.cnn.layer.TransposedConvolution2D' is implemented in 'nnet.cnn.layer.TransposedConvolution2D'
### Notice: The layer 'ImageInputLayer' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in 'nnet.cnn.layer.ImageInputLayer'
### Notice: The layer 'Softmax-Layer' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in 'nnet.cnn.layer.SoftmaxLayer'
### Notice: The layer 'Segmentation-Layer' with type 'nnet.cnn.layer.PixelClassificationLayer' is implemented in 'nnet.cnn.layer.PixelClassificationLayer'
### 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: [1x1 struct]
    instructions: [1x1 struct]
    registers: [1x1 struct]
    syncInstructions: [1x1 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	Total
	-----	-----	-----	-----
Network	175391449	1.16928	20	35076
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

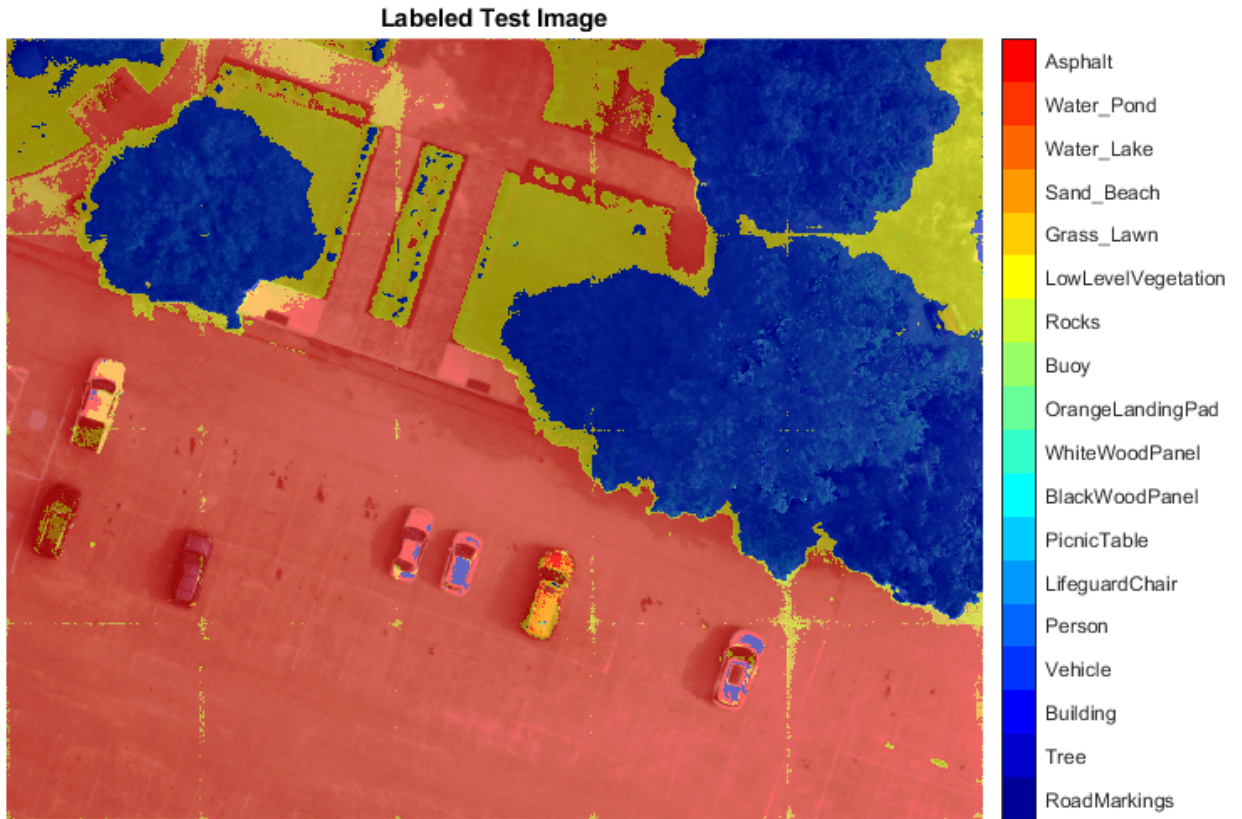
Overlay the segmented image on the histogram-equalized RGB test image and display the prediction results.

```
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, C

figure
imshow(B);
title('Labeled Test Image')
colorbar('TickLabels', cellstr(classNames), 'Ticks', ticks, 'TickLength', 0, 'TickLabelInterpret
colormap(cmap)
```

References

[1] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918. 2017.

See Also

`dlhdl.Target` | `dlhdl.Workflow` | `dlquantizer` | `calibrate` | `compile` | `deploy` | `predict` | `classify`

Deep Learning Quantization

- “Quantization of Deep Neural Networks” on page 11-2
- “Quantization Workflow Prerequisites” on page 11-10
- “Calibration” on page 11-13
- “Validation” on page 11-15
- “Code Generation and Deployment” on page 11-18

Quantization of Deep Neural Networks

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). The data type defines how hardware components or software functions interpret this sequence of 1's and 0's. Numbers are represented as either scaled integer (usually referred to as fixed-point) or floating-point data types.

Most pretrained neural networks and neural networks trained using Deep Learning Toolbox use single-precision floating point data types. Even small trained neural networks require a considerable amount of memory, and require hardware that can perform floating-point arithmetic. These restrictions can inhibit deployment of deep learning capabilities to low-power microcontrollers and FPGAs.

Using the Deep Learning Toolbox Model Quantization Library support package, you can quantize a network to use 8-bit scaled integer data types.

To learn about the products required to quantize and deploy the deep learning network to a GPU, FPGA, or CPU environment, see “Quantization Workflow Prerequisites”.

Precision and Range

Scaled 8-bit integer data types have limited precision and range when compared to single-precision floating point data types. There are several numerical considerations when casting a number from a larger floating-point data type to a smaller data type of fixed length.

- Precision loss: Precision loss is a rounding error. When precision loss occurs, the value is rounded to the nearest number that is representable by the data type. In the case of a tie it rounds:
 - Positive numbers to the closest representable value in the direction of positive infinity.
 - Negative numbers to the closest representable value in the direction of negative infinity.

In MATLAB you can perform this type of rounding using the `round` function.

- Underflow: Underflow is a type of precision loss. Underflows occur when the value is smaller than the smallest value representable by the data type. When this occurs, the value saturates to zero.
- Overflow: When a value is larger than the largest value that a data type can represent, an overflow occurs. When an overflow occurs, the value saturates to the largest value representable by the data type.

Histograms of Dynamic Ranges

Use the **Deep Network Quantizer** app to collect and visualize the dynamic ranges of the weights and biases of the convolution layers and fully connected layers of a network, and the activations of all layers in the network. The app assigns a scaled 8-bit integer data type for the weights, biases, and activations of the convolution layers of the network. The app displays a histogram of the dynamic range for each of these parameters. The following steps describe how these histograms are produced.

- 1 Consider the following values logged for a parameter while exercising a network.

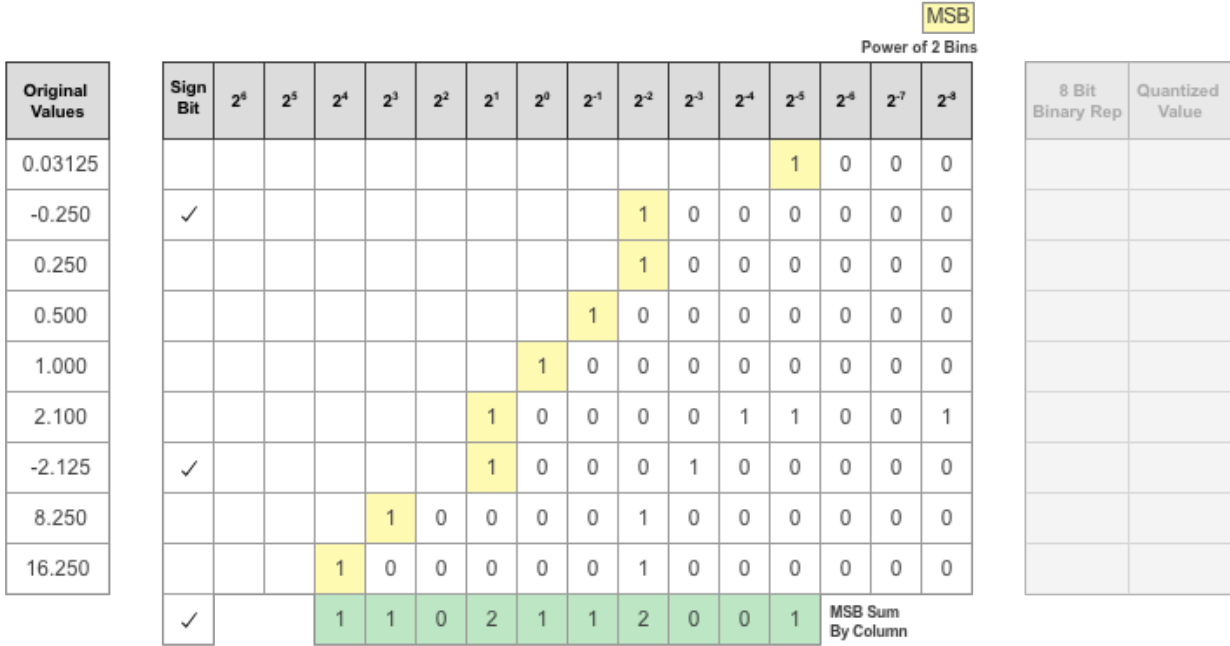
Original Values	Power of 2 Bins															8 Bit Binary Rep	Quantized Value		
	Sign Bit	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}			2^{-8}	
0.03125																			
-0.250																			
0.250																			
0.500																			
1.000																			
2.100																			
-2.125																			
8.250																			
16.250																			

2 Find the ideal binary representation of each logged value of the parameter.

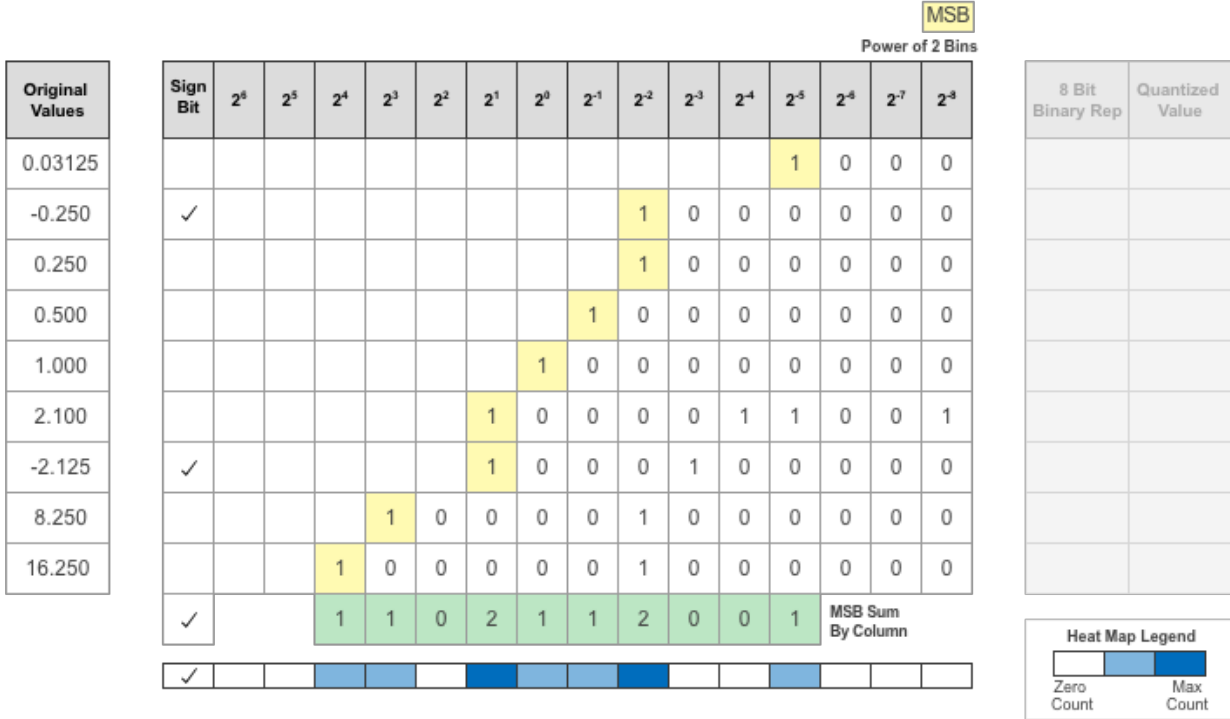
The most significant bit (MSB) is the left-most bit of the binary word. This bit contributes most to the value of the number. The MSB for each value is highlighted in yellow.

Original Values	Sign Bit	Power of 2 Bins														8 Bit Binary Rep	Quantized Value			
		2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}			2^{-8}		
0.03125																				
-0.250	✓																			
0.250																				
0.500																				
1.000																				
2.100																				
-2.125	✓																			
8.250																				
16.250																				

- 3 By aligning the binary words, you can see the distribution of bits used by the logged values of a parameter. The sum of MSB's in each column, highlighted in green, give an aggregate view of the logged values.

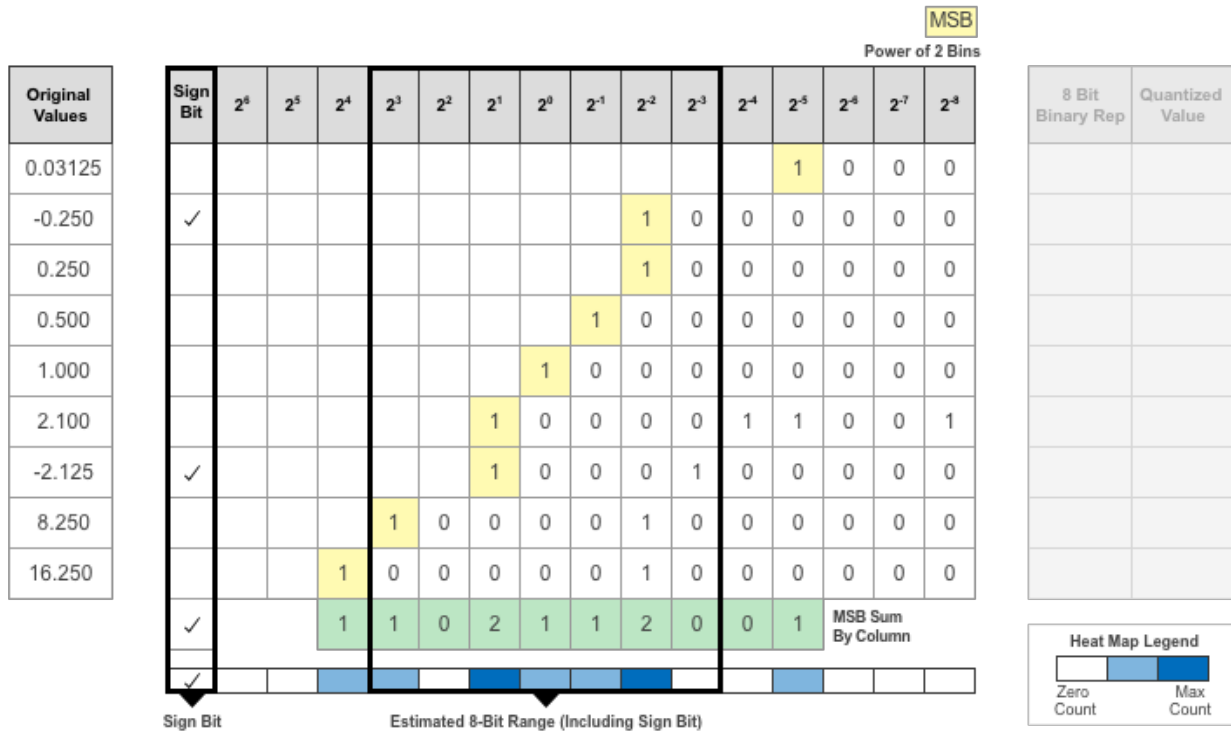


4 The MSB counts of each bit location are displayed as a heat map. In this heat map, darker blue regions correspond to a larger number of MSB's in the bit location.

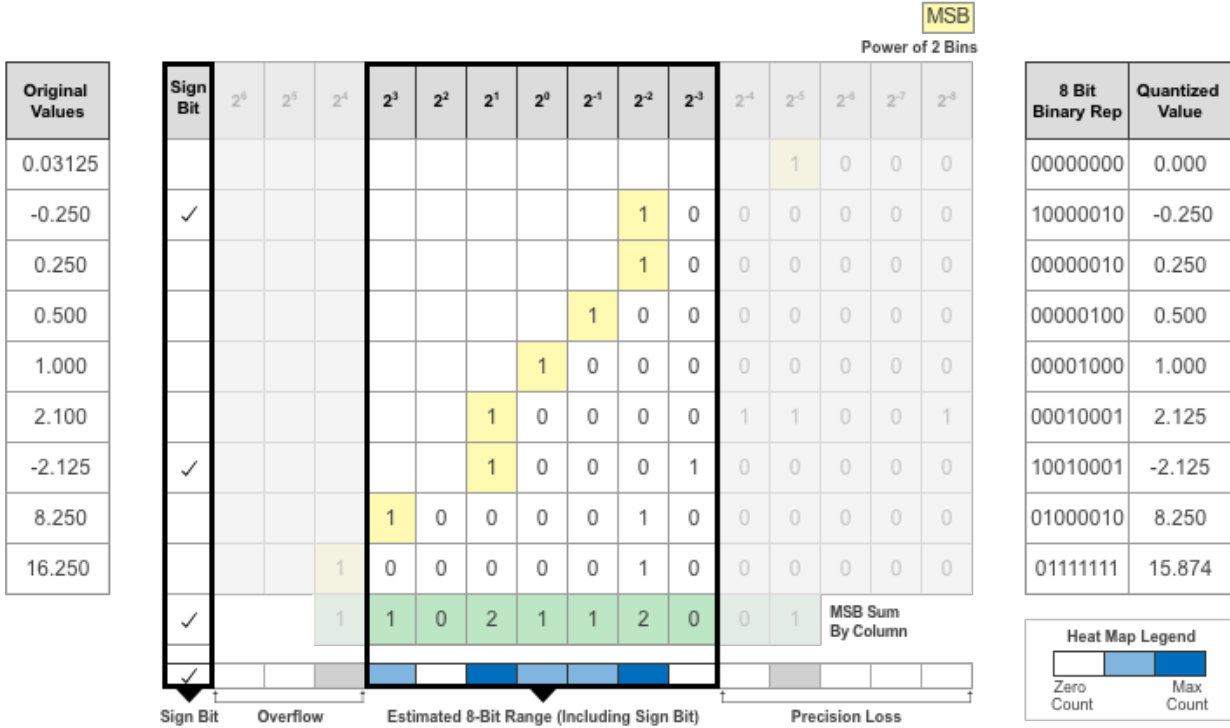


- The **Deep Network Quantizer** app assigns a data type that can avoid overflow, cover the range, and allow underflow. An additional sign bit is required to represent the signedness of the value.

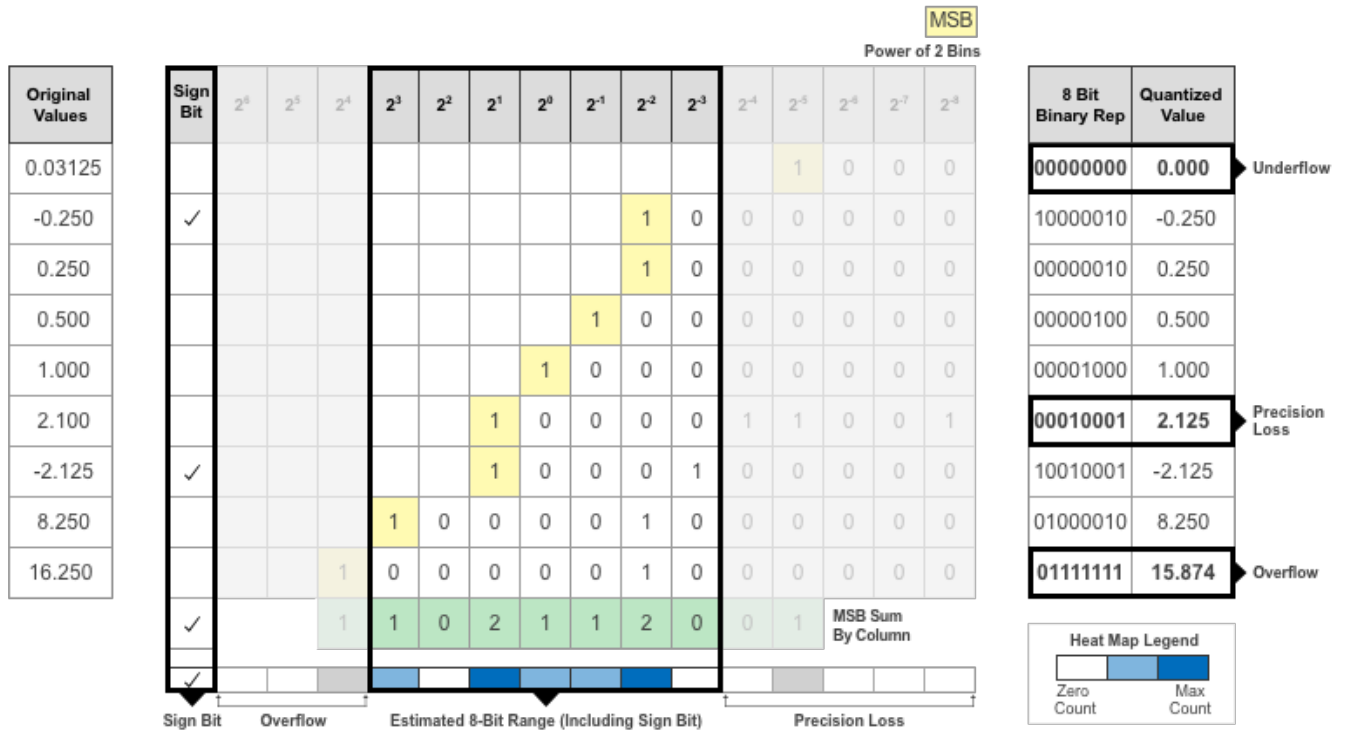
The figure below shows an example of a data type that represents bits from 2^3 to 2^{-3} , including the sign bit.



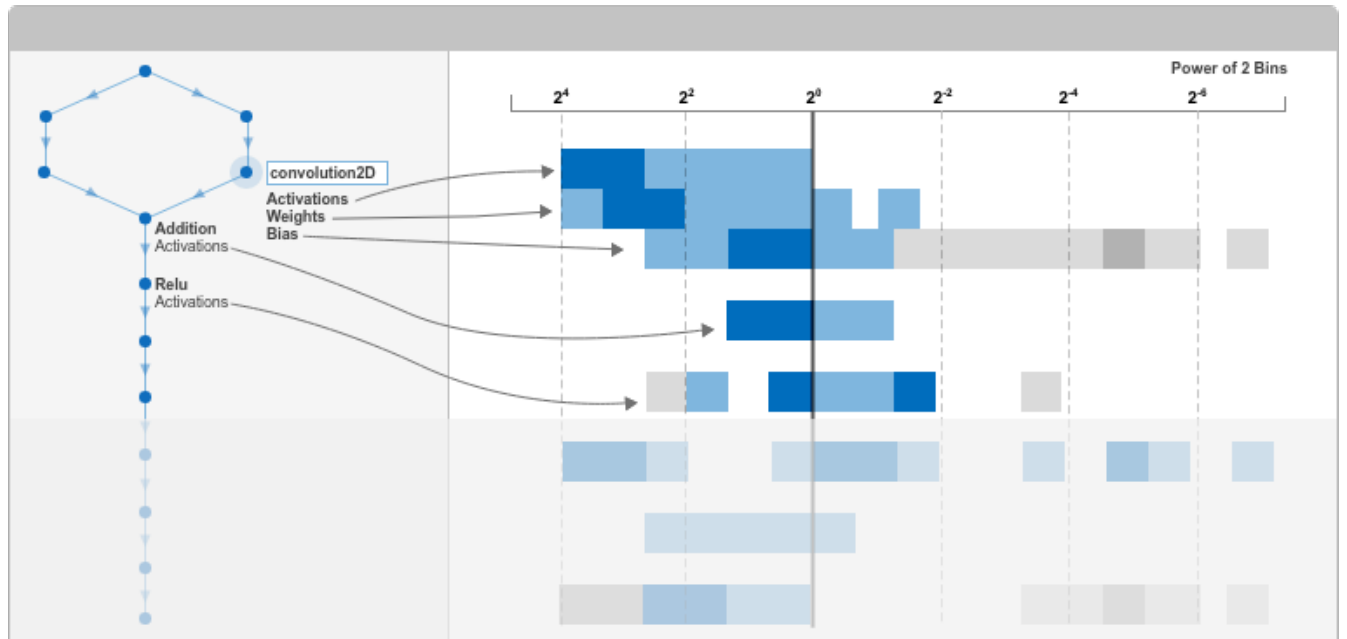
- After assigning the data type, any bits outside of that data type are removed. Due to the assignment of a smaller data type of fixed length, precision loss, overflow, and underflow can occur for values that are not representable by the data type.



In this example, the value 0.03125, suffers from an underflow, so the quantized value is 0. The value 2.1 suffers some precision loss, so the quantized value is 2.125. The value 16.250 is larger than the largest representable value of the data type, so this value overflows and the quantized value saturates to 15.874.



- 7 The **Deep Network Quantizer** app displays this heat map histogram for each learnable parameter in the convolution layers and fully connected layers of the network. The gray regions of the histogram show the bits that cannot be represented by the data type.



See Also

Apps

Deep Network Quantizer

Functions

calibrate | validate | dlquantizer | dlquantizationOptions

Quantization Workflow Prerequisites

This table lists the products required to quantize and deploy deep learning networks.

	Execution Environment			
Development Host Requirements	FPGA	GPU	CPU	MATLAB
Setup Toolkit Environment	<p>hdlsetuptoolpath (HDL Coder)</p> <p>The quantization workflow does not support the MinGW C/C++ compiler. Use Microsoft Visual C++ 2017 or Microsoft Visual C++ 2015.</p> <p>For a list of supported compilers, see Supported and Compatible Compilers.</p>	<p>“Setting Up the Prerequisite Products” (GPU Coder)</p> <p>The quantization workflow does not support the MinGW C/C++ compiler. Use Microsoft Visual C++ 2017 or Microsoft Visual C++ 2015.</p> <p>For a list of supported compilers, see Supported and Compatible Compilers.</p>	<p>“Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder)</p> <p>The quantization workflow does not support the MinGW C/C++ compiler. Use Microsoft Visual C++ 2017 or Microsoft Visual C++ 2015.</p> <p>For a list of supported compilers, see Supported and Compatible Compilers.</p> <p>Only Raspberry Pi™ with ARM® v7 architecture is supported.</p> <p>The ARM Compute Library version 20.02.1 for deep learning quantized inference.</p>	<p>The quantization workflow does not support the MinGW C/C++ compiler. Use Microsoft Visual C++ 2017 or Microsoft Visual C++ 2015.</p> <p>For a list of supported compilers, see Supported and Compatible Compilers.</p>
Required Products	<ul style="list-style-type: none"> Deep Learning Toolbox Deep Learning HDL Toolbox 	Deep Learning Toolbox	<ul style="list-style-type: none"> Deep Learning Toolbox MATLAB Coder Embedded Coder 	Deep Learning Toolbox

Required Support Packages	<ul style="list-style-type: none"> • Deep Learning Toolbox Model Quantization Library • Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices • Deep Learning HDL Toolbox Support Package for Intel FPGA and SoC Devices 	Deep Learning Toolbox Model Quantization Library	Deep Learning Toolbox Model Quantization Library	Deep Learning Toolbox Model Quantization Library
Required Add Ons	MATLAB Coder™ Interface for Deep Learning Libraries	<ul style="list-style-type: none"> • GPU Coder™ Interface for Deep Learning Libraries • CUDA® enabled NVIDIA® GPU with compute capability 6.1, 6.3 or higher. 	MATLAB Coder Interface for Deep Learning Libraries	MATLAB Coder Interface for Deep Learning Libraries
Supported Networks and Layers	“Supported Networks, Layers, Boards, and Tools” on page 7-2	“Supported Networks, Layers, and Classes” (GPU Coder)	“Networks and Layers Supported for Code Generation” (MATLAB Coder)	“Networks and Layers Supported for Code Generation” (MATLAB Coder) Note When MATLAB is the Execution Environment, only the layers for the Intel MKL-DNN deep learning library are supported.

Deployment	Deep Learning HDL Toolbox	GPU Coder	MATLAB Coder	None
			Note Before validation, you must create a <code>raspi</code> object to establish connection to hardware.	

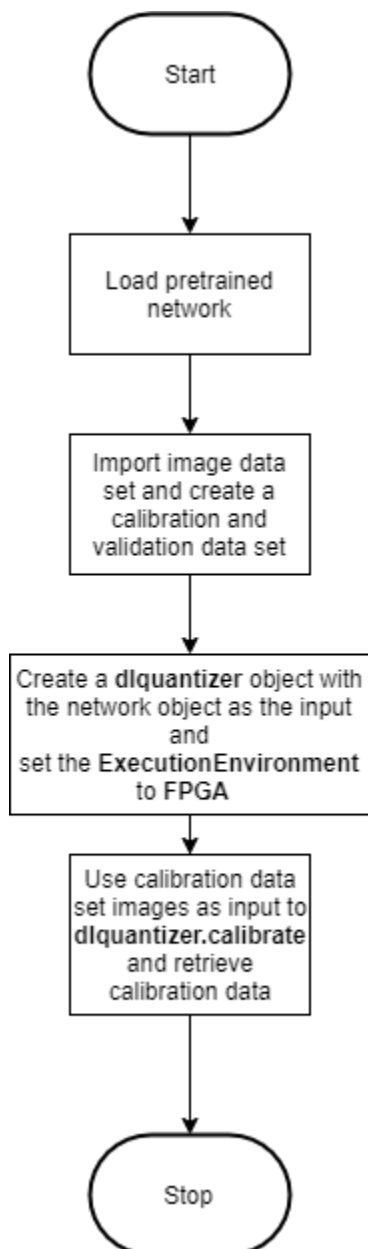
Calibration

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

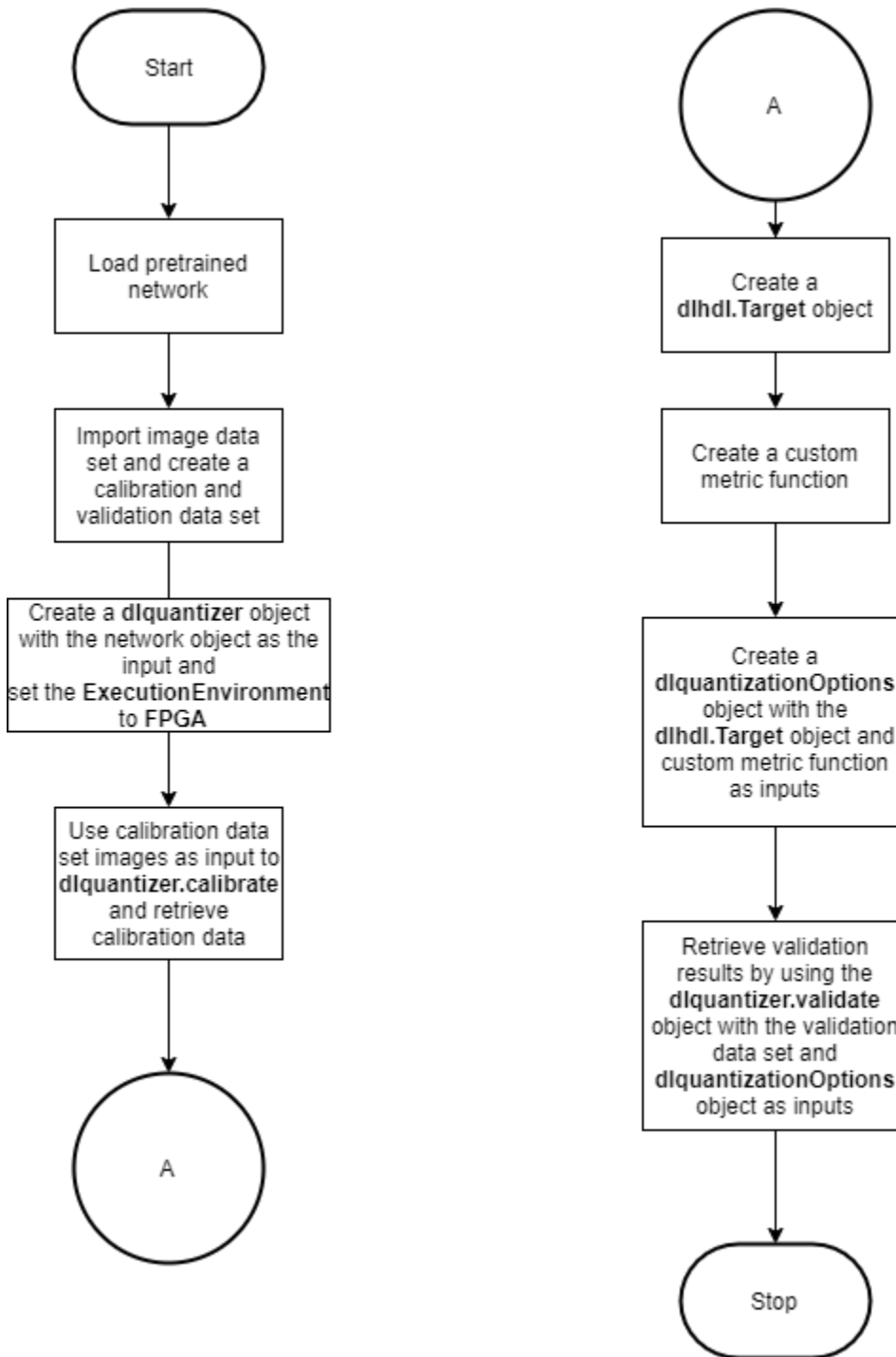
- “Quantization of Deep Neural Networks” on page 11-2
- “Validation” on page 11-15
- “Code Generation and Deployment” on page 11-18

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

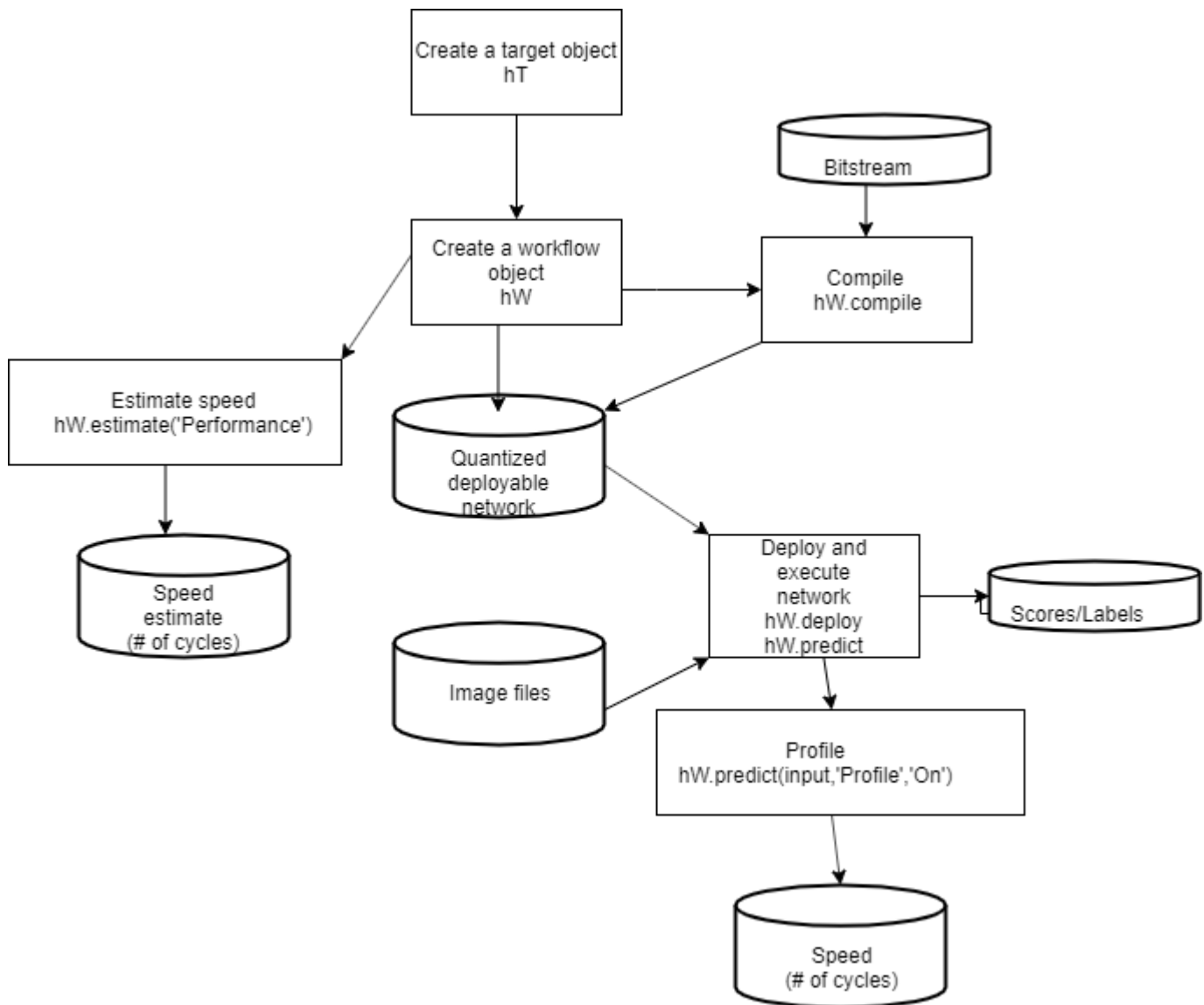
- “Quantization of Deep Neural Networks” on page 11-2
- “Calibration” on page 11-13
- “Code Generation and Deployment” on page 11-18

Code Generation and Deployment

To generate 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” on page 11-2
- “Calibration” on page 11-13
- “Validation” on page 11-15

Deep Learning Processor IP Core User Guide

- “Deep Learning Processor IP Core” on page 12-2
- “Use the Compiler Output for System Integration” on page 12-3
- “External Memory Data Format” on page 12-6
- “Deep Learning Processor Register Map” on page 12-9
- “Interface with the Deep Learning Processor IP Core” on page 12-11
- “Define Custom Board and Reference Design for Deep Learning Processor IP Core Workflow” on page 12-25

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 generate the DL processor IP core, use 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 Register Map” on page 12-9.

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 12-3.

The input and output data stored in the external memory in a predefined format. For more information, see “External Memory Data Format” on page 12-6.

See Also

More About

- “Custom IP Core Generation” (HDL Coder)
- “Use the Compiler Output for System Integration” on page 12-3
- “External Memory Data Format” on page 12-6
- “Deep Learning Processor Register Map” on page 12-9
- “Interface with the Deep Learning Processor IP Core” on page 12-11

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`).

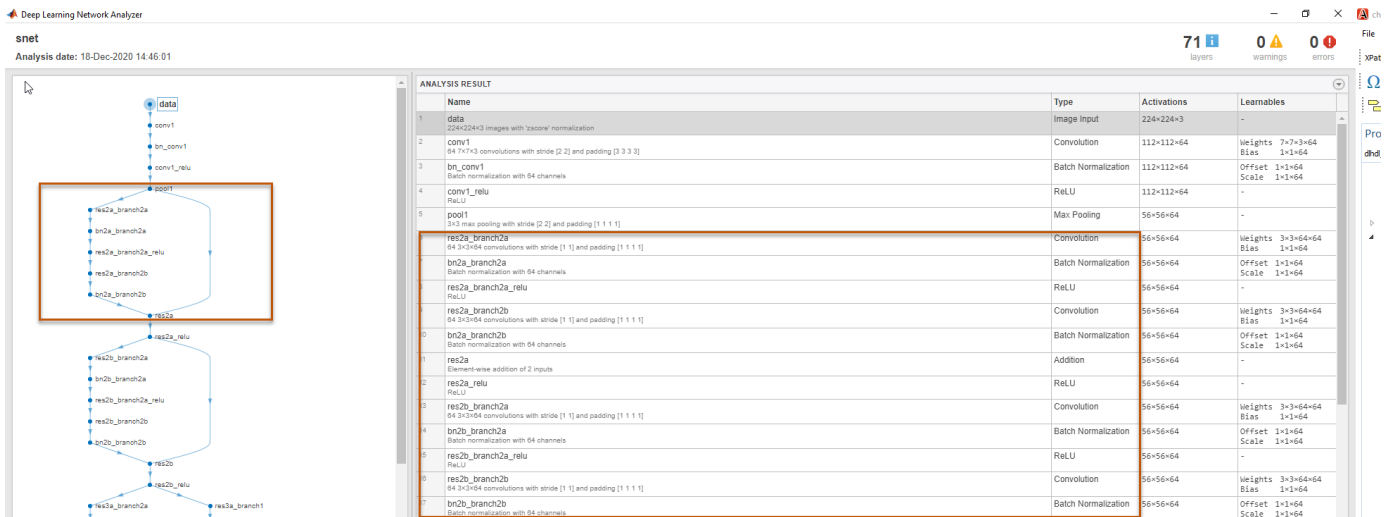
- 2-D zero padding layer (`nnet.keras.layer.ZeroPadding2dLayer`) and 2-D convolution layer (`convolution2dLayer`).
- 2-D zero padding layer (`nnet.keras.layer.ZeroPadding2dLayer`) and 2-D max polling layer (`maxPooling2dLayer`).

This code output is an example compiler optimization in the compiler log.

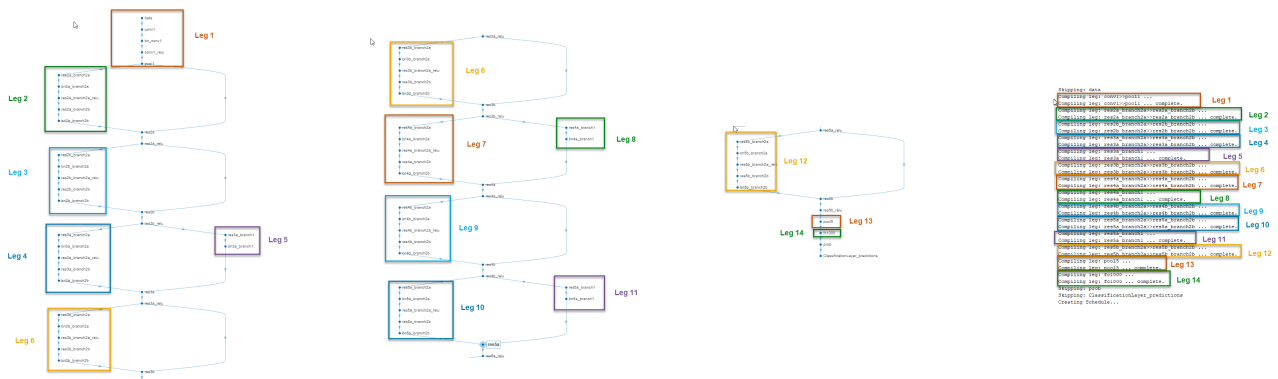
Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convolution2dLayer'

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 7-10.



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 12-2
- “External Memory Data Format” on page 12-6
- “Deep Learning Processor Register Map” on page 12-9
- “Interface with the Deep Learning Processor IP Core” on page 12-11

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

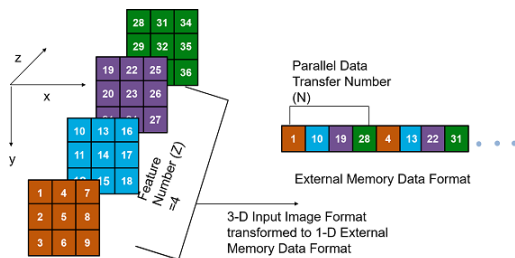
- Parallel Data Transfer Number refers to the number of pixels that are transferred every clock cycle through the AXI master interface. Use the letter N in place of the Parallel Data Transfer Number. Mathematically N is calculated as $\text{nextpow2}(\text{sqrt}(\text{ConvThreadNumber}))$. For example, 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.

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 Send N number of data in the z dimension of the matrix.
- 2 Send the image information along the x dimension of the input image.
- 3 Send 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.

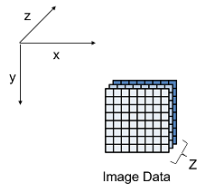
The image demonstrates how the data stored in a 3-by-3-by-4 matrix is translated into a 1-by-36 matrix that is then stored in the external memory.



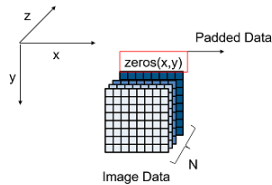
When the image Feature Number (Z) is not a multiple of the Parallel Data Transfer Number (N), then we must pad a zeroes matrix of size x-by-y along the z dimension of the matrix to make the image Z value a multiple of N.

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.

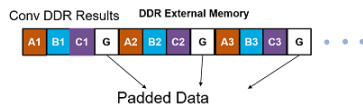
This image is the input image format before padding.



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.

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 12-2
- “Use the Compiler Output for System Integration” on page 12-3
- “Deep Learning Processor Register Map” on page 12-9

- “Interface with the Deep Learning Processor IP Core” on page 12-11

See Also

Deep Learning Processor Register Map

During custom 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.

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 ← Table of AXI Registers and offsets

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
AXI4_Master_Activation_Data_Wr_BaseAddr	0xC	Base Address offset for AXI4 Master Activation Data Write
AXI4_Master_Weight_Data_Rd_BaseAddr	0x10	Base Address offset for AXI4 Master Weight Data Read
AXI4_Master_Debug_Rd_BaseAddr	0x14	Base Address offset for AXI4 Master Debug Read
AXI4_Master_Debug_Wr_BaseAddr	0x18	Base Address offset for AXI4 Master Debug Write
IPCore_Timestamp	0x1C	contains unique IP timestamp (yymmddHHMM): 2006132129
start_Data	0x138	data register for Import start
debugEnable_Data	0x140	data register for Import debugEnable
debugDMAEnable_Data	0x144	data register for Import debugDMAEnable
debugDMAlength_Data	0x148	data register for Import debugDMAlength
debugSelect_Data	0x14C	data register for Import debugSelect
debugDMAWidth_Data	0x150	data register for Import debugDMAWidth
debugDMAOffset_Data	0x154	data register for Import debugDMAOffset
debugDMAIDirection_Data	0x158	data register for Import debugDMAIDirection
debugDMAStart_Data	0x15C	data register for Import debugDMAStart
image_valid_Data	0x160	data register for Import image_valid
image_addr_Data	0x164	data register for Import image_addr
image_data_Data	0x168	data register for Import image_data
read_addr_Data	0x16C	data register for Import read_addr
debug_read_data_Data	0x17C	data register for Import debug_read_data
dma_from_ddr4_done_Data	0x184	data register for Import dma_from_ddr4_done

The image contains all the AXI4 registers created during IP core generation.

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
AXI4_Master_Activation_Data_Wr_BaseAddr	0xC	Base Address offset for AXI4 Master Activation Data Write
AXI4_Master_Weight_Data_Rd_BaseAddr	0x10	Base Address offset for AXI4 Master Weight Data Read
AXI4_Master_Debug_Rd_BaseAddr	0x14	Base Address offset for AXI4 Master Debug Read
AXI4_Master_Debug_Wr_BaseAddr	0x18	Base Address offset for AXI4 Master Debug Write
IPCore_Timestamp	0x1C	contains unique IP timestamp (yymmddHHMM): 2005151208
start_Data	0x138	data register for Import start
debugEnable_Data	0x140	data register for Import debugEnable
debugDMAEnable_Data	0x144	data register for Import debugDMAEnable
debugDMAlength_Data	0x148	data register for Import debugDMAlength
debugSelect_Data	0x14C	data register for Import debugSelect
debugDMAWidth_Data	0x150	data register for Import debugDMAWidth
debugDMAOffset_Data	0x154	data register for Import debugDMAOffset
debugDMAIDirection_Data	0x158	data register for Import debugDMAIDirection
debugDMAStart_Data	0x15C	data register for Import debugDMAStart
image_valid_Data	0x160	data register for Import image_valid
image_addr_Data	0x164	data register for Import image_addr
image_data_Data	0x168	data register for Import image_data
read_addr_Data	0x16C	data register for Import read_addr
debug_read_data_Data	0x17C	data register for Import debug_read_data
dma_from_ddr4_done_Data	0x184	data register for Import dma_from_ddr4_done
dma_to_ddr4_done_Data	0x188	data register for Import dma_to_ddr4_done
done_Data	0x220	data register for Import done
importStart_Data	0x224	data register for Import importStart
preLoadingStart_Data	0x228	data register for Import preLoadingStart
nc_L1CtoAllLength_IP0_Data	0x22C	data register for Import nc_L1CtoAllLength_IP0
nc_L1CtoFFet_IP0_Data	0x230	data register for Import nc_L1CtoFFet_IP0
nc_L1CtoAllLength_Conv_Data	0x234	data register for Import nc_L1CtoAllLength_Conv
nc_L1CtoFFet_Conv_Data	0x238	data register for Import nc_L1CtoFFet_Conv
nc_L1CtoAllLength_OPO_Data	0x23C	data register for Import nc_L1CtoAllLength_OPO
nc_L1CtoFFet_OPO_Data	0x240	data register for Import nc_L1CtoFFet_OPO
nc_op_image_count_Data	0x24C	data register for Import nc_op_image_count
convDone_Data	0x278	data register for Import convDone
importDDROffset_Data	0x27C	data register for Import importDDROffset
nc_haaSFC_Data	0x280	data register for Import nc_haaSFC
convResultDDROffset_Data	0x284	data register for Import convResultDDROffset
nc_haaSIS_Data	0x288	data register for Import nc_haaSIS
HS_ddr_addr_Data	0x28C	data register for Import HS_ddr_addr
conv_weight_ddr_addr_Data	0x290	data register for Import conv_weight_ddr_addr
fc_weight_ddr_addr_Data	0x294	data register for Import fc_weight_ddr_addr
fc_fc_ddr_addr_Data	0x298	data register for Import fc_fc_ddr_addr
fc_fc_ddr_addr_Data	0x29C	data register for Import fc_fc_ddr_addr
fc_layerNum_Data	0x2A0	data register for Import fc_layerNum

See Also

More About

- “Deep Learning Processor IP Core” on page 12-2
- “Use the Compiler Output for System Integration” on page 12-3
- “External Memory Data Format” on page 12-6
- “Interface with the Deep Learning Processor IP Core” on page 12-11

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	Requires 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	register	Signal from the user to the deep learning processor IP core to start processing the data.	Input
FrameCount	integer	RunTimeControl	register	Signal from the user to the deep learning processor IP core specifying the number of input data frames.	Input
InputStop	logical	RunTimeControl	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	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	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 <code>InputValid</code> 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 of the next input data frame. Use this signal when the InputValid signal is high.	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. 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.

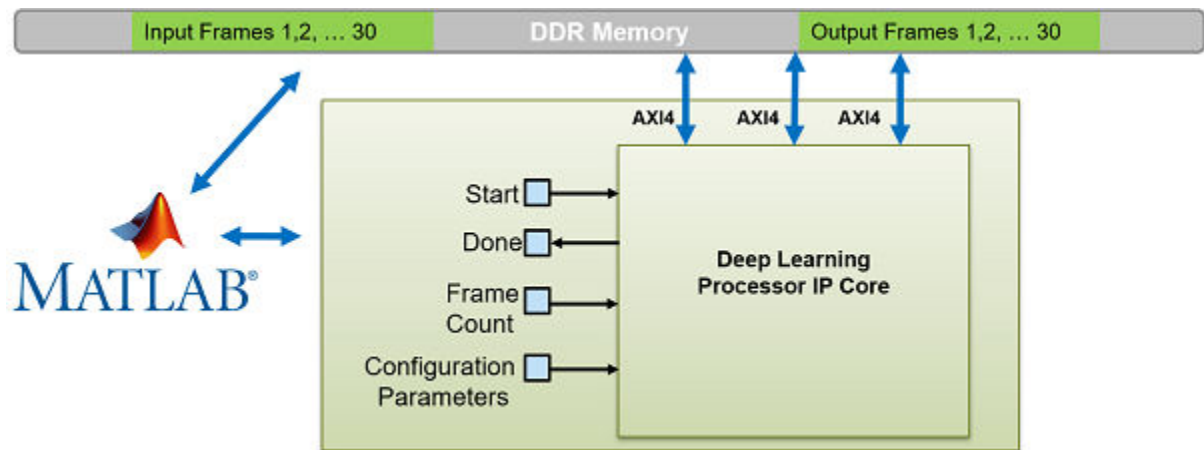
Signal Name	Data Type	Configuration Control Parameter	Interface Type (Port or Register)	Description	Interface Direction (Input or Output)
StreamingMode	logical	SetupControl	register	Signal from the user to the deep learning processor IP core specifying the data processing mode. <code>false</code> selects buffer mode and <code>true</code> selects streaming mode.	Input

Signal Name	Data Type	Configuration Control Parameter	Interface Type (Port or Register)	Description	Interface Direction (Input or Output)
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. <code>true</code> selects user addresses and <code>false</code> selects compiler generated addresses.	Input
InputBaseAddr	uint32	SetupControl	port/register	User provided input base address. Specify the address before you toggle the <code>InputStart</code> signal.	Input
OutputBaseAddr	uint32	SetupControl	port/register	User provided output base address. Specify the address before you toggle the <code>InputStart</code> signal.	Input

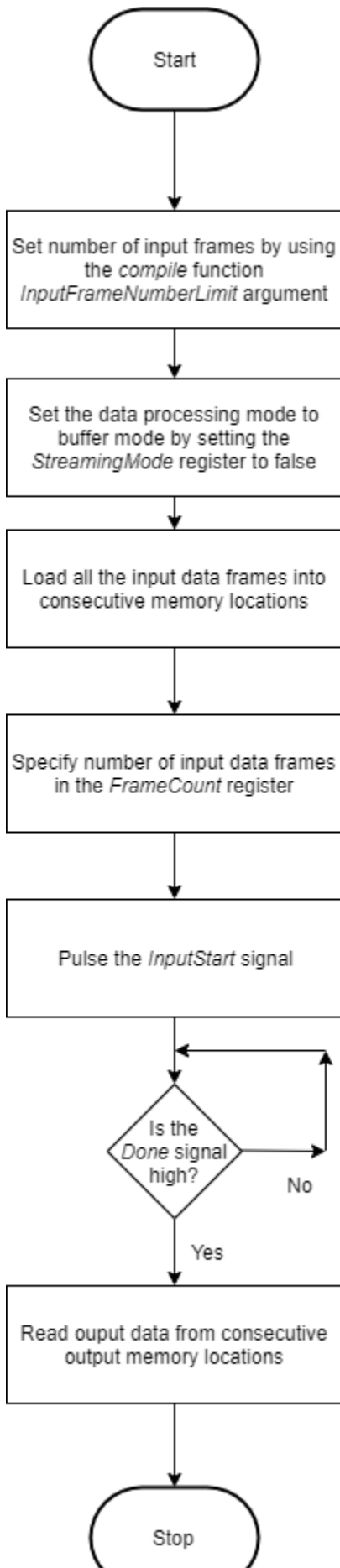
Design Batch Processing Mode Interface

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.

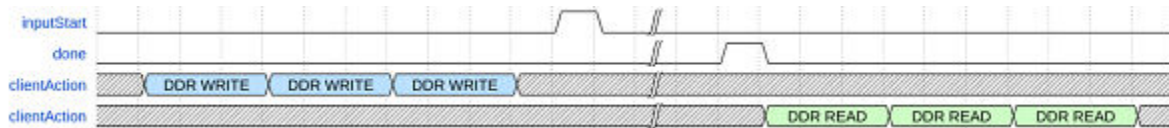
To process a single data frame set the `FrameCount` register value to one. If the `FrameCount` is set to zero the deep learning processor runs intermittently and the `Done` signal does not become `true`.



This flowchart shows the operation of the batch processing mode.



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

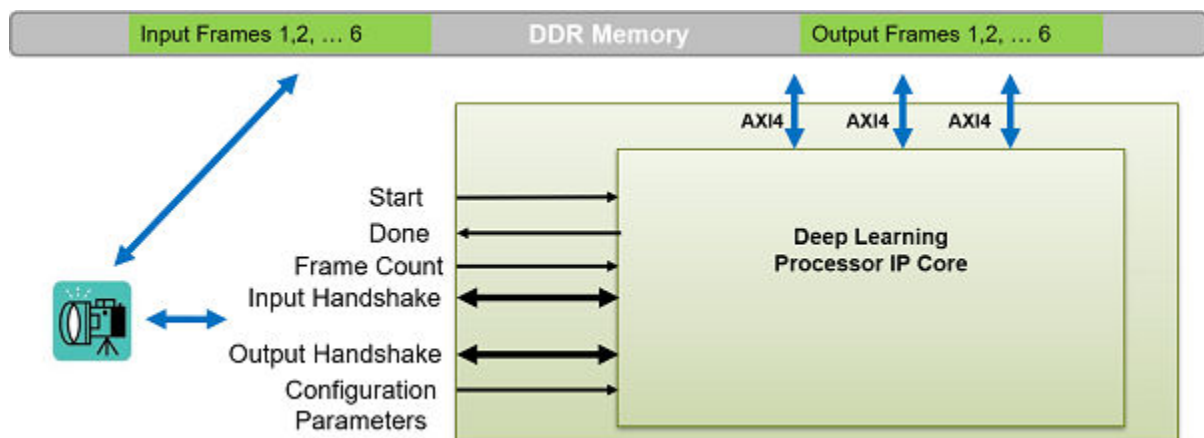
When your input data is streaming in, when you have access to limited DDR memory space, and when your application requires handshaking protocols, process multiple frames by using the streaming mode. The figure shows the generated deep learning processor IP core with interface signals for the streaming 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.

Data 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.

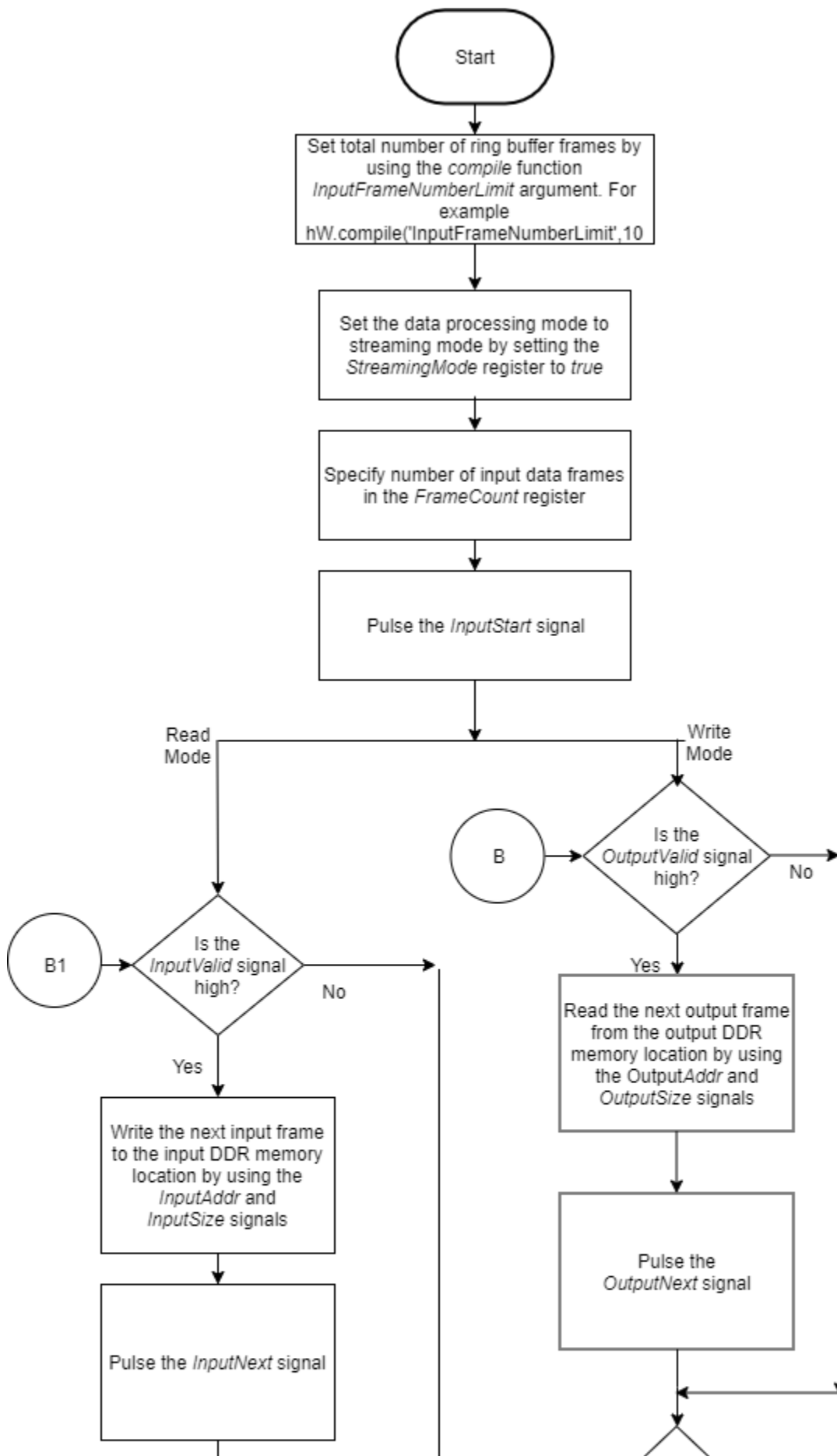


Streaming Mode up to a Frame Count

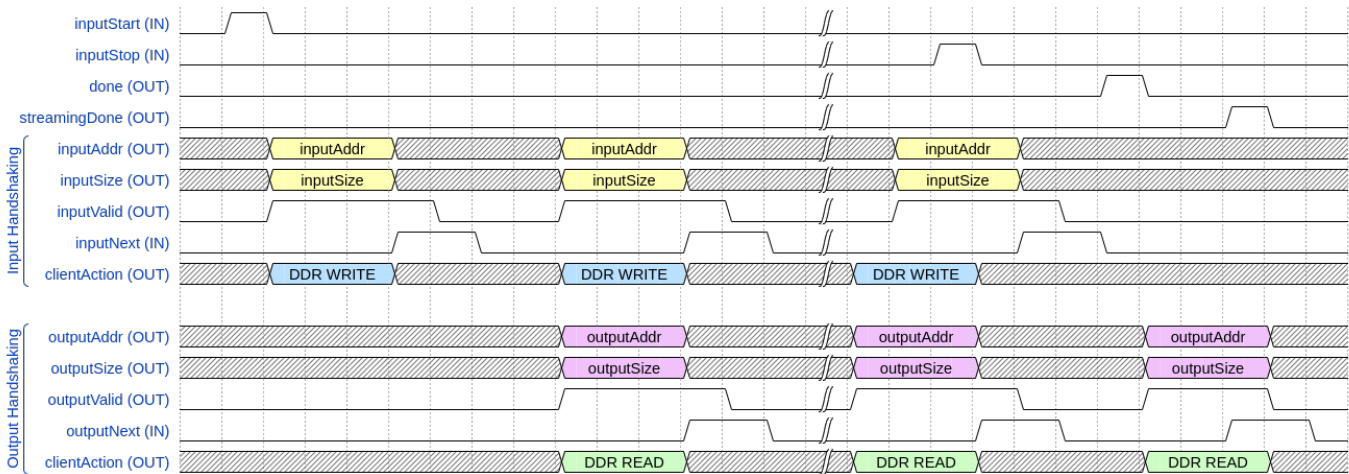
This flowchart shows the operation of the streaming mode data processing mode. The read and write operations occur in parallel. The value set in the `InputFrameNumberLimit` specifies the number of

spaces in the DDR for the input and output ring buffers. If you use a larger ring buffer size the deep learning processor IP core processes more input images before you read the first output result.

This flowchart shows the operation of the streaming mode up to a frame count. The read and write operations occur in parallel.



This timing diagram shows the operation of the streaming mode up to a frame count.



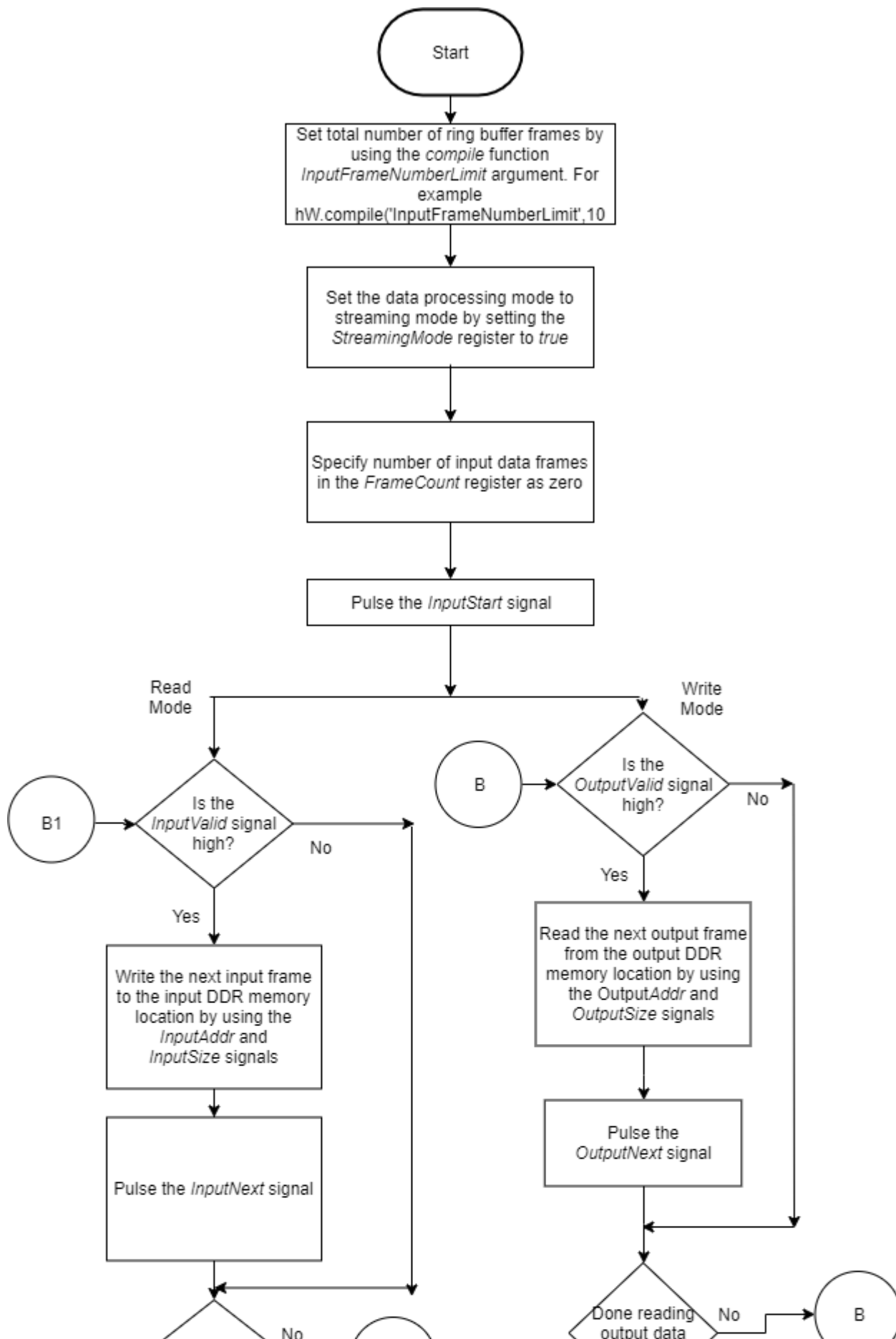
- 1 Set the `InputFrameNumberLimit` argument of the `compile` method to a value greater than two.
- 2 Set the `StreamingMode` signal to `true`.
- 3 Set the number of data frames to process in the `FrameCount` register.
- 4 Pulse the `inputStart` signal. These next actions can be performed in parallel:
 - 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.

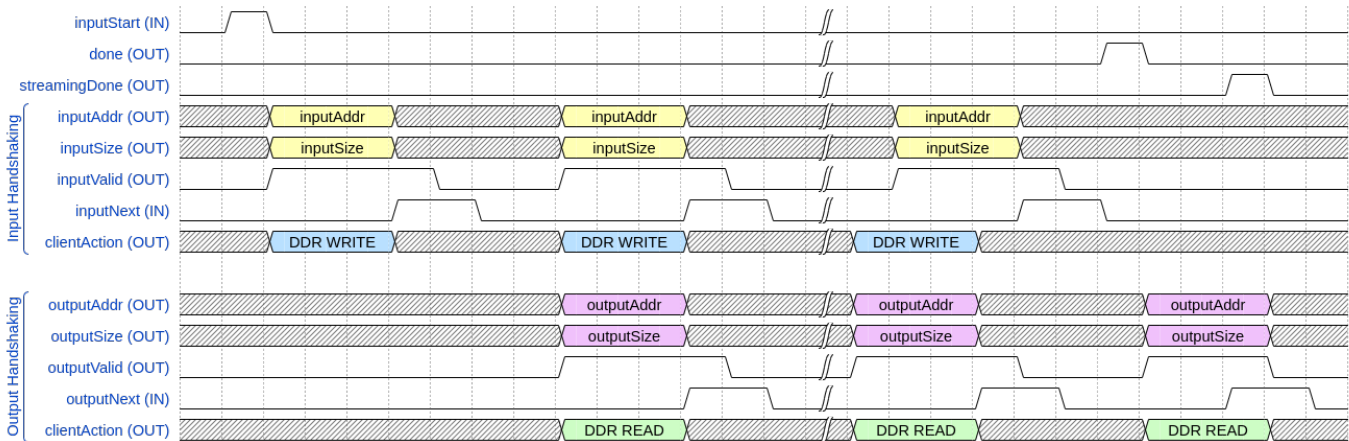
Continuous Streaming Mode

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`.

This flowchart shows the operation of the continuous streaming mode. The read and write operations occur in parallel.



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 12-2 | “Use the Compiler Output for System Integration” on page 12-3 | “External Memory Data Format” on page 12-6

Define Custom Board and Reference Design for Deep Learning Processor IP Core Workflow

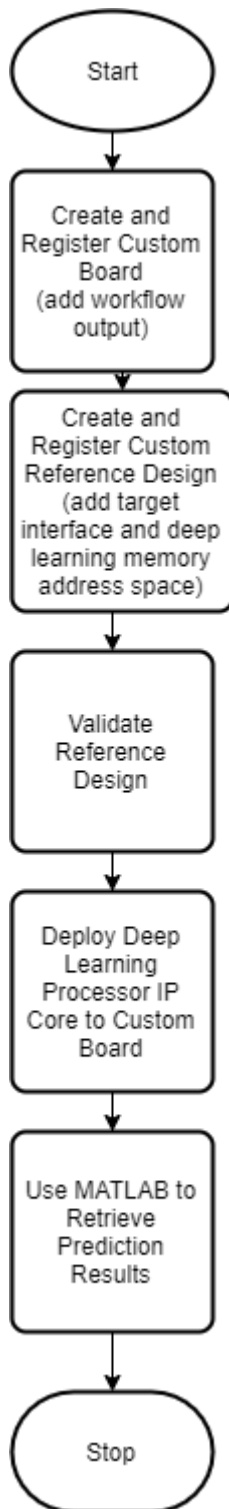
This example shows how to define and register a custom board and reference design for the deep learning processor IP core workflow. If you have a system that has preprocessing and postprocessing design under test (DUT) blocks, you can iterate on those blocks to improve your performance and resource utilization by using HDL Coder™. Create a custom reference design that includes the DUT blocks and the definition for your custom deep learning processor IP core.

Introduction

First create a custom board and reference design. Register the deep learning processor IP core in your custom board and reference design. Deploy the deep learning processor IP core to your custom board by generating a custom bitstream. Use MATLAB® to interface with the deep learning processor IP core.

Workflow

The image shows the workflow to deploy the deep learning processor IP core to a custom board.



Register Custom Board

Register a custom board in HDL Workflow Advisor.

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 workflow. Set the workflow to:

```
workflow = hdlcoder.Workflow.DeepLearningProcessor;

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 2019 The MathWorks, Inc.

boardList = { ...
    'DLZCU102.plugin_board', ...
    'DLZC706.plugin_board', ...
};

workflow = hdlcoder.Workflow.DeepLearningProcessor;

end
```

2. Create the board definition file.

```
%   Copyright 2019 The MathWorks, Inc.

% Board definition of ZCU102
function hB = plugin_board

% Construct board object
hB = hdlcoder.Board;

hB.BoardName    = 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit';

% FPGA device information
hB.FPGAVendor   = 'Xilinx';
hB.FPGAFamily   = 'Zynq UltraScale+';
hB.FPGADevice   = 'xczu9eg-ffvb1156-2-e';
hB.FGPAPackage  = '';
hB.FPGASpeed    = '';

% Tool information
hB.SupportedTool = {'Xilinx Vivado'};

% FPGA JTAG chain position
hB.JTAGChainPosition = 1;

%% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS18'});

% GPIO LEDs
hB.addExternalIOInterface( ...
    'InterfaceID',    'LEDs General Purpose', ...
    'InterfaceType', 'OUT', ...
    'PortName',       'GPIO_LED', ...
```

```

    'PortWidth',      8, ...
    'FPGAPin',       {'AG14', 'AF13', 'AE13', 'AJ14', 'AJ15', 'AH13', 'AH14', 'AL12'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

```

Register Custom Reference Design

Register the custom reference design in HDL Workflow Advisor.

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` containing a list of reference design plugins associated with an SoC board.

```

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 2019 The MathWorks, Inc.

```

```

rd = {...
    'DLZCU102.matlab_libiio_3axi4_master_2019_1.plugin_rd', ...
};

```

```

boardName = 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit';

```

```

end

```

2. Create the reference design definition file.

```

function hRD = plugin_rd
% Reference design definition
%
% Copyright 2019-2020 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 Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit';

% Tool information
hRD.SupportedToolVersion = {'2019.1','2019.2','2020.1'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl','system_top.tcl');

% Add constraint files
hRD.CustomConstraints = {'ZCU102_DDR4.xdc'};

% Post Create Project Callback function to improve timing closure
hRD.PostCreateProjectFcn = @DLZCU102.matlab_libiio_3axi4_master_2019_1.callback_PostCreateProj;

% Add HDL Verifier JTAG as AXI Master IP from support package
msg = message('hdlcommon:plugin:IPRepositoryHDLVerifierXilinxNotFound').getString;

```

```

hRD.addIPRepository( ...
    'IPListFunction', 'hdlverifier.fpga.vivado.iplist', ...
    'NotExistMessage', msg);

% Add AXI4-Stream to AXI4-Master DDR Access IP
hRD.addIPRepository( ...
    'IPListFunction', 'hdlcoder.fpga.vivado.hdlcoder_axis2axim_iplist', ...
    'NotExistMessage', 'AXI4-Stream to AXI4-Master IP not found.');
```

```

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection', 'core_clkwiz/clk_out1', ...
    'ResetConnection', 'sys_dut_rstgen/peripheral_aresetn',...
    'DefaultFrequencyMHz', 50,...
    'MinFrequencyMHz', 5,...
    'MaxFrequencyMHz', 500,...
    'ClockModuleInstance', 'core_clkwiz',...
    'ClockNumber', 1);

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_cpu_interconnect/M03_AXI', ...
    'BaseAddress', {'0xA0000000', '0x00A0000000'}, ...
    'MasterAddressSpace', {'hdlverifier_axi_master_0/axi4m', 'sys_cpu/Data'}, ...
    'InterfaceType', 'AXI4', ...
    'IDWidth', 13);
```

3. Add deep learning processor information to the reference design file. 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.

```

% Deep learning specific properties
hRD.registerDeepLearningTargetInterface("JTAG");
hRD.registerDeepLearningTargetInterface("Ethernet");
hRD.registerDeepLearningMemoryAddressSpace(0x80000000, 0x20000000); % 512 MB

hRD.ResourcesUsed.LogicElements = 35000;
hRD.ResourcesUsed.DSP = 3;
hRD.ResourcesUsed.RAM = 77.5;
```

The completed board and reference design registration files for the Xilinx® Zynq® Ultrascale+(R) MPSoC ZCU102 board can be found at,

supportpackages\R2021b\toolbox\dnnfpga\supportpackages\xilinx\boards and

supportpackages\R2021b\toolbox\dnnfpga\supportpackages\xilinx\boards\+DLZCU102.

For more details on how to set up, define, and register the custom board refer to the “Getting Started with Targeting Zynq UltraScale+ MPSoC Platform” (HDL Coder).

See Also

hdlcoder.ReferenceDesign | registerDeepLearningMemoryAddressSpace | registerDeepLearningTargetInterface | validateReferenceDesignForDeepLearning

More About

- “Deep Learning Processor IP Core” on page 12-2
- “Use the Compiler Output for System Integration” on page 12-3
- “External Memory Data Format” on page 12-6
- “Interface with the Deep Learning Processor IP Core” on page 12-11
- “Register a Custom Board” (HDL Coder)
- “Register a Custom Reference Design” (HDL Coder)