Deep Learning HDL Toolbox™

User's Guide

MATLAB®

R2021b

MathWorks®

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— 2021 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)

Contents

1 What is Deep Learning?

2 Deep Learning Processor

3 Applications and Examples

4 Deep Learning on FPGA Overview

5 Workflow and APIS

6 Fast MATLAB to FPGA Connection Using LIBIIO/Ethernet

7 Networks and Layers

8 Custom Processor Configuration Workflow

9 Custom Processor Code Generation Workflow

10 Featured Examples

11 Deep Learning Quantization

12 Deep Learning Processor IP Core User Guide

1

What is Deep Learning?

Introduction to Deep Learning

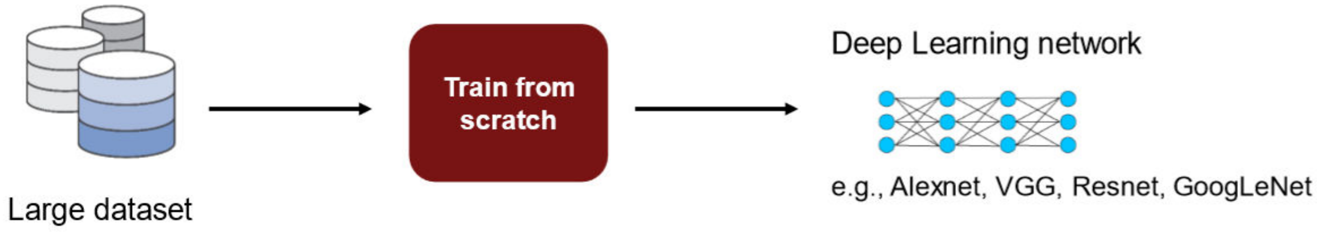
Deep learning is a branch of machine learning that teaches computers to do what comes naturally to humans: learn from experience. The learning algorithms use computational methods to “learn” information directly from data without relying on a predetermined equation as model. Deep learning uses neural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

Training Process

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

Training from Scratch

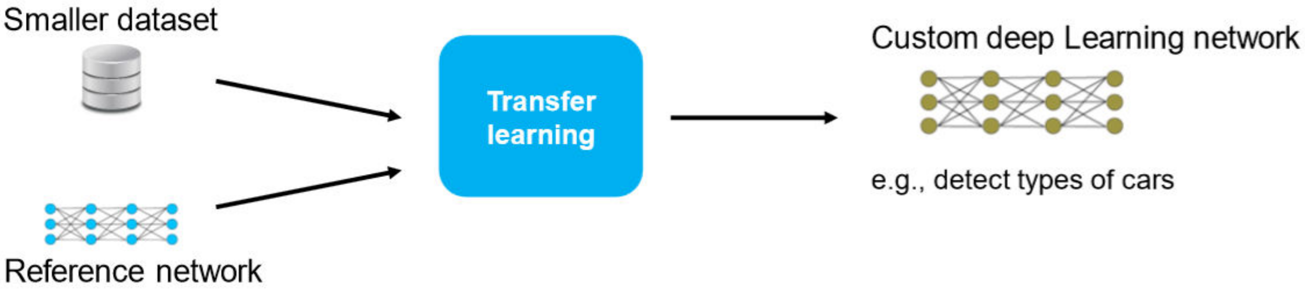
Training a deep learning neural network from scratch requires a large amount of labeled data.To create the network architecture by using Neural Network Toolbox™, you can use the built-in layers, define your own layers, or import layers from Caffe models. The neural network is then trained by using the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



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

Transfer Learning

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



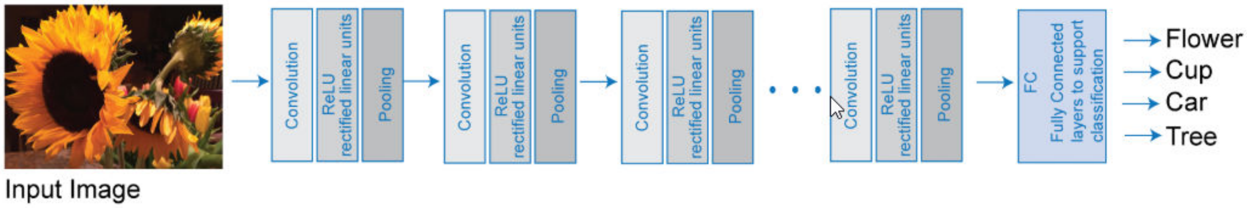
For more information, see “Get Started with Transfer Learning”

Feature Extraction

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

Convolutional Neural Networks

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



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

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

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

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

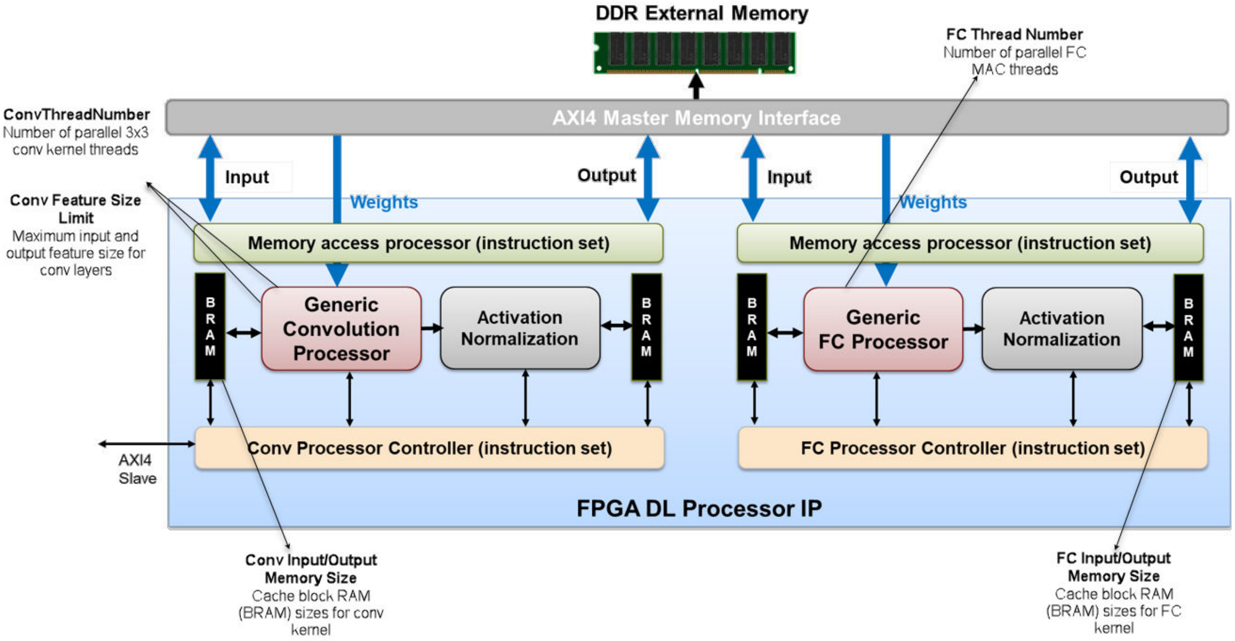
2

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.


DDR External Memory
FC Thread Number
Number of parallel FC MAC threads
ConvThreadNumber
Number of parallel 3x3 conv kernel threads
AXI4 Master Memory Interface
Input
Weights
Output
Input
Weights
Output
Conv Feature Size Limit
Maximum input and output feature size for conv layers
Memory access processor (instruction set)
Memory access processor (instruction set)
BRAM
Generic Convolution Processor
Activation Normalization
BRAM
BRAM
Generic FC Processor
Activation Normalization
BRAM
AXI4 Slave
Conv Processor Controller (instruction set)
FC Processor Controller (instruction set)
FPGA DL Processor IP
Conv Input/Output Memory Size
Cache block RAM (BRAM) sizes for conv kernel
FC Input/Output Memory Size
Cache block RAM (BRAM) sizes for FC kernel

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.

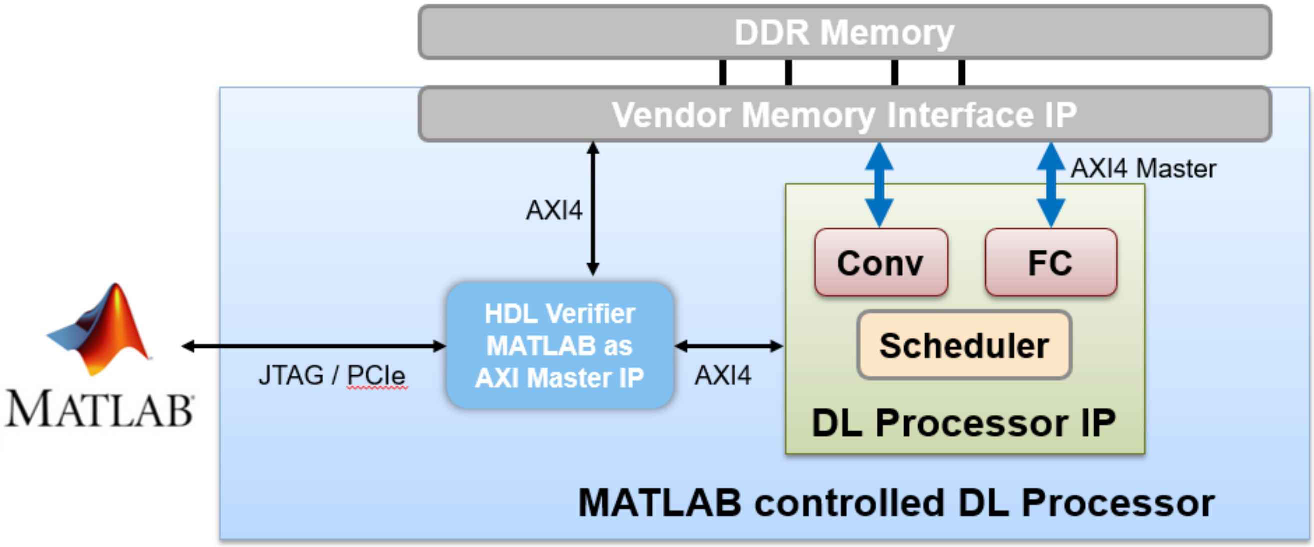
Applications and Examples

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 MATLAB AXI Master” (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.

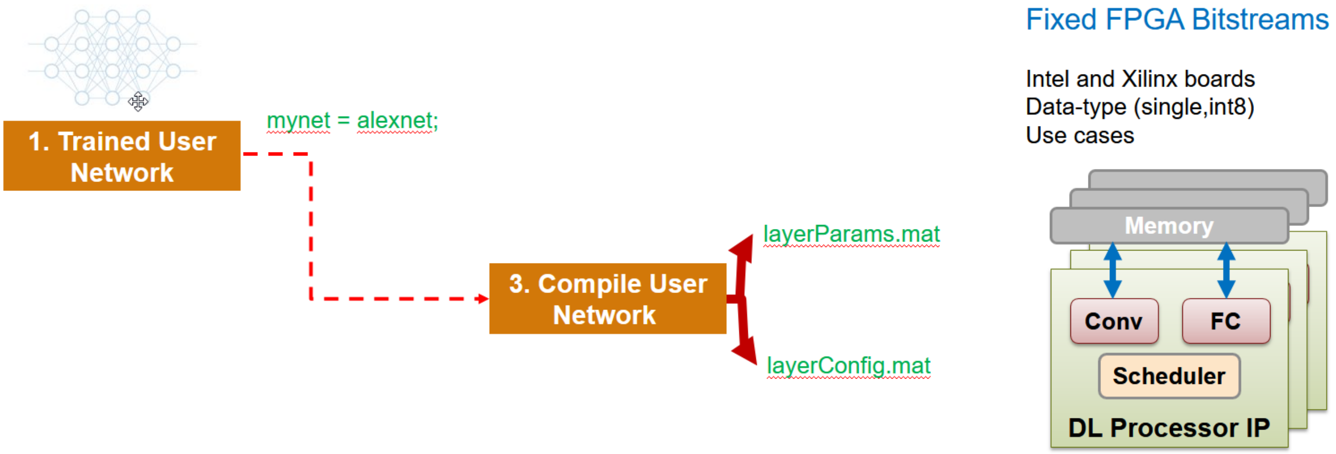

DDR Memory
Vendor Memory Interface IP
AXI4
AXI4 Master
Conv
FC
Scheduler
DL Processor IP
HDL Verifier MATLAB as AXI Master IP
JTAG / PCIe
AXI4
MATLAB
MATLAB controlled DL Processor

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

This figure illustrates deep learning on FPGA workflow.



To use the workflow:

1 **Load deep learning neural network**

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

2 **Modify pretrained neural network on MATLAB using transfer learning**

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

3 **Compile user network**

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

4 **Deploy on target FPGA board**

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

5 **Predict outcome**

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

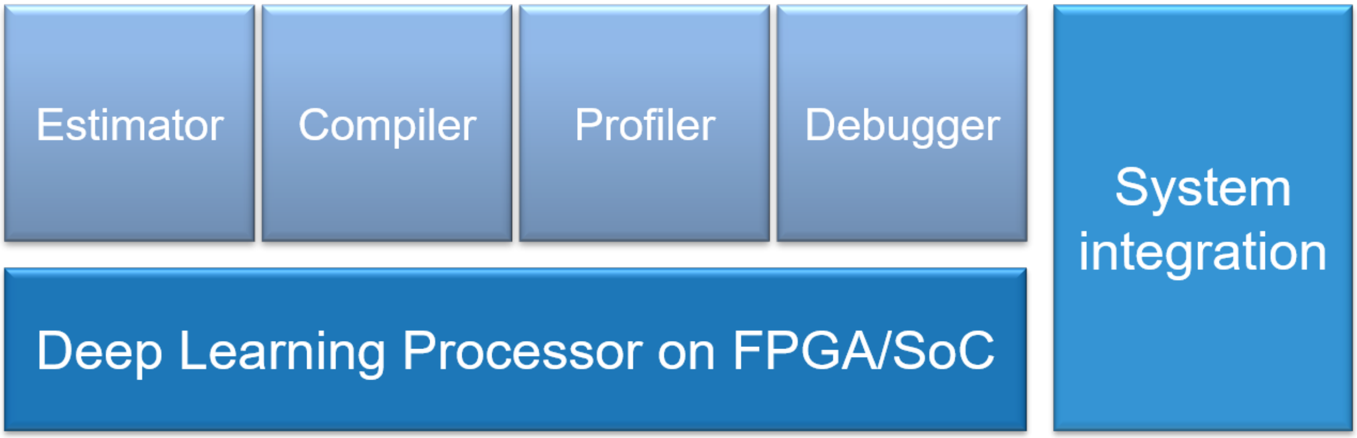
See Also

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

Deep Learning on FPGA Solution

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

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



The workflow is:

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

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

Advantages of Deep Learning on FPGA

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

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

Deep Learning on FPGA Workflows

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

Goal	Workflow
Run a pretrained series network on your target FPGA board.	“Prototype Deep Learning Networks on FPGA and 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 11-14
Run a quantized pretrained series network on your target FPGA board.	“Code Generation and Deployment” on page 11-17

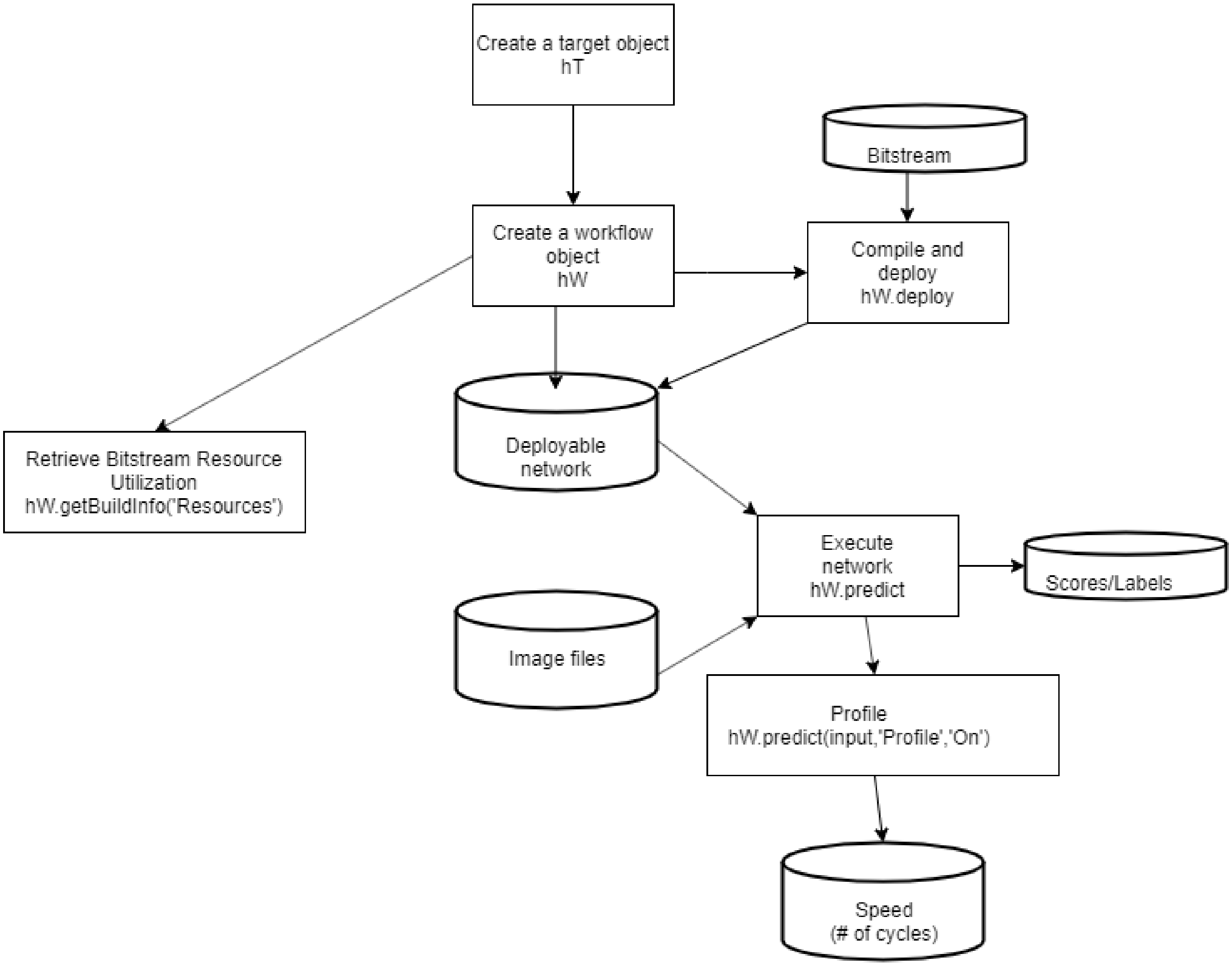
Workflow and APIS

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 10-5

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 `LastFrameLatency(cycles)/Clock Frequency`. For example the `conv_module` `LastFrameLatency(seconds)` is calculated as `2224115/(220*10^6)`.
- `FramesNum` — Total number of input frames to the network. This value will be used in the calculation of `Frames/s`.
- `Total Latency` — Total number of clock cycles to execute all the network layers and modules for `FramesNum`.
- `Frames/s` — Number of frames processed in one second by the network. The total `Frames/s` is calculated as `(FramesNum*Clock Frequency)/Total Latency`. For example the `Frames/s` in the example is calculated as `(1*220*10^6)/23659630`.

See Also

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

More About

- “Prototype Deep Learning Networks on FPGA and SoCs Workflow” on page 5-2

- “Profile Network for Performance Improvement” on page 10-43

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

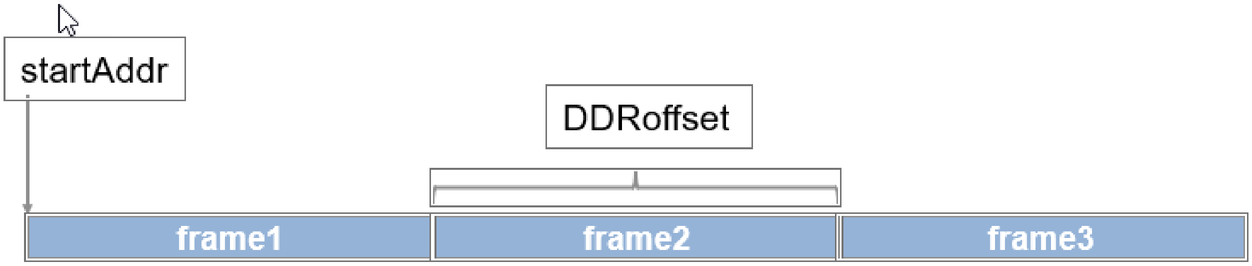
To format input images to meet the multiple frame input DDR format, you must have:

- 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.



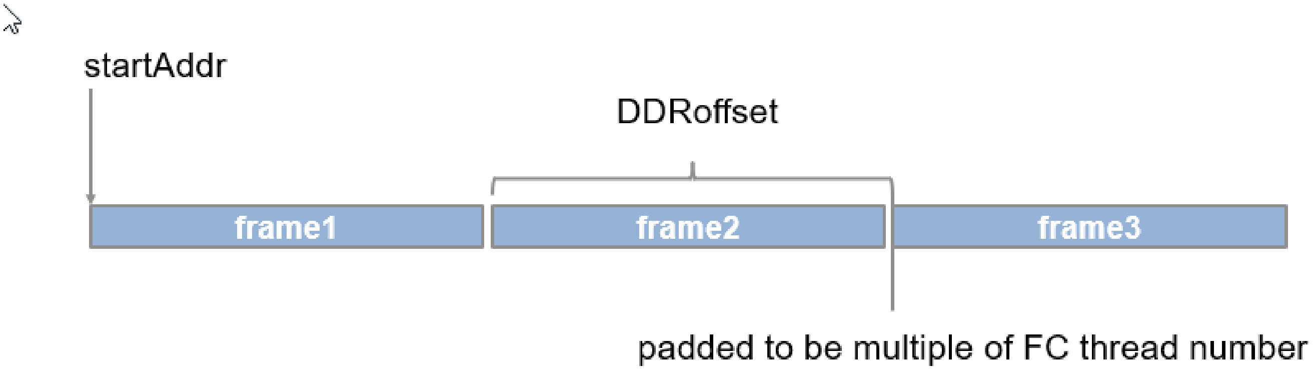
Output DDR Format

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

6

Fast MATLAB to FPGA Connection Using LIBIIO/Ethernet

LIBIIO/Ethernet Connection Based Deep Learning Network Deployment

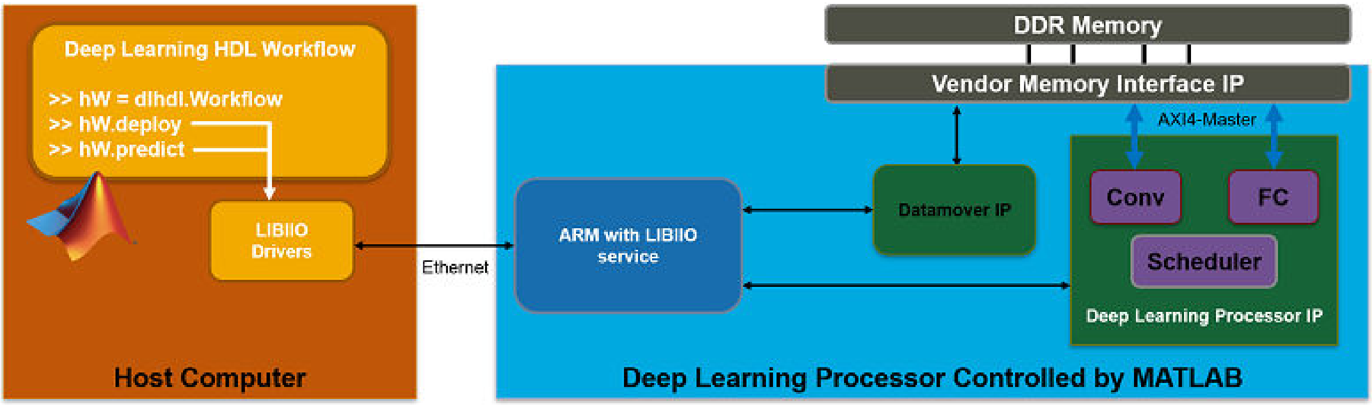
In this section...

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

`dlhdl.Target`

More About

- “Accelerate Prototyping Workflow for Large Networks by Using Ethernet” on page 10-78

7

Networks and Layers

Supported Networks, Layers, Boards, and Tools

In this section...

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**	Classification
AlexNet	AlexNet convolutional neural network.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	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 10-16.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	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 10-47.	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 10-26.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Classification

Defect Detection `snet_blemdetnet`	`snet_blemdetnet` is a custom convolutional neural network used to identify and classify defects. For more information, see “Defect Detection” on page 10-26.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Classification

YOLO v2 Vehicle Detection	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 YOLO v2 Deployed to FPGA” on page 10-88.	Series Network based	Yes	Yes	Yes	Yes	Yes	Yes	Object detection

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

ResNet-based YOLO v2	You 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 10-127.	Directed acyclic graph (DAG) network based	Yes	Yes	Yes	Yes	Yes	Yes	Object detection

MobileNetV2	MobileNet-v2 convolutional neural network. For the pretrained MobileNet-v2 model, see `mobilenetv2`.	Directed acyclic graph (DAG) network based	Yes	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 `googlenet`.								

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.	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**

convolution2dLayer	HW	Convolution (Conv)	A 2-D convolutional layer applies sliding convolutional filters to the input. When generating code for a network using this layer, these limitations apply: • Filter size must be 1-15 and square. For example [1 1] or [15 15]. • 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

`groupedConvolution2dLayer`	HW	Convolution (Conv)	A 2-D grouped convolutional layer separates the input channels into groups and applies sliding convolutional filters. Use grouped convolutional layers for channel-wise separable (also known as depth-wise separable) convolution. Code generation is now supported for a 2-D grouped convolution layer that has the `NumGroups` property set as `'channel-wise'`. When generating code for a network using this layer, these limitations apply: • Filter size must be 1-15 and square. For example [1 1] or [14 14]. When the `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

			• Dilation factor must be [1 1]. • 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`.	
`fullyConnectedLayer`	HW	Fully Connected (FC)	A fully connected layer multiplies the input by a weight matrix, and then adds a bias vector. When generating code for a network using this layer, these limitations apply: • The layer input and output size are limited by the values specified in “InputMemorySize” and “OutputMemorySize”.	Yes

Activation Layers

Layer	Layer Type Hardware (HW) or Software(SW)	Layer Output Format	Description and Limitations	INT8 Compatible
reluLayer	HW	Layer is fused.	A ReLU layer performs a threshold operation to each element of the input where any value less than zero is set to zero. A ReLU layer is supported only when it is preceded by any of these layers: • Convolution • Fully Connected • Adder	Yes
leakyReluLayer	HW	Layer is fused.	A leaky ReLU layer performs a threshold operation where any input value less than zero is multiplied by a fixed scalar. A leaky ReLU layer is supported only when it is preceded by any of these layers: • Convolution • Fully Connected • Adder	Yes

`clippedReluLayer`	HW	Layer is fused.	A clipped ReLU layer performs a threshold operation where any input value less than zero is set to zero and any value above the clipping ceiling is set to that clipping ceiling value. A clipped ReLU layer is supported only when it is preceded by any of these layers: • Convolution • Fully Connected • Adder	Yes

Normalization, Dropout, and Cropping Layers

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

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

Pooling and Unpooling Layers

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

`maxPooling2dLayer`	HW	Convolution (Conv)	A max pooling layer performs downsampling by dividing the layer input into rectangular pooling regions and computing the maximum of each region. When generating code for a network using this layer, these limitations apply: • Pool size must be 1-15 and square. For example [1 1] or [12 12]. • Stride size must be 1-15 and square. • Padding size must be in the range 0-2.	Yes

`averagePooling2dLayer`	HW	Convolution (Conv)	An average pooling layer performs downsampling by dividing the layer input into rectangular pooling regions and computing the average values of each region. When generating code for a network using this layer, these limitations apply: • Pool size must be 1-15 and square. For example [3 3] • Stride size must be 1-15 and square. • Padding size must be in the range 0-2.	Yes

`globalAveragePooling2dLayer`	HW	Convolution (Conv) or Fully Connected (FC). When the input activation size is lower than the memory threshold, the layer output format is FC.	A global average pooling layer performs downsampling by computing the mean of the height and width dimensions of the input. When generating code for a network using this layer, these limitations apply: • Can accept inputs of sizes up to 15-by-15-by-N. • Total activation pixel size must be smaller than the deep learning processor convolution module input memory size. For more information, see "InputMemorySize"	Yes

Combination Layers

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

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

`depthConcatenationLayer`	HW	Inherit from input.	A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension). When generating code for a network using this layer, these limitations apply: • The input activation feature number must be a multiple of the square root of the “ConvThreadNumber”. • Inputs to the depth concatenation layer must be exclusive to the depth concatenation layer. • Layers that have a conv output format and layers that have an FC output format cannot be concatenated together.	Yes

Output Layer

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

`softmax`	SW and HW	A softmax layer applies a softmax function to the input. If the softmax layer is implemented in hardware: • The inputs must be in the range `exp(-87)` to `exp(88)`. • Softmax layer followed by adder layer or depth concatenation layer is not supported.	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

Keras and ONNX Layers

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

`nnet.keras.layer.ZeroPadding2dLayer`	HW	Layer will be fused.	Zero padding layer for 2-D input. A `nnet.keras.layer.ZeroPadding2dLayer` is supported only when it is followed by a convolution layer or a maxpool layer.	Yes

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

See Also

More About

- “Configure FPGA Boards”

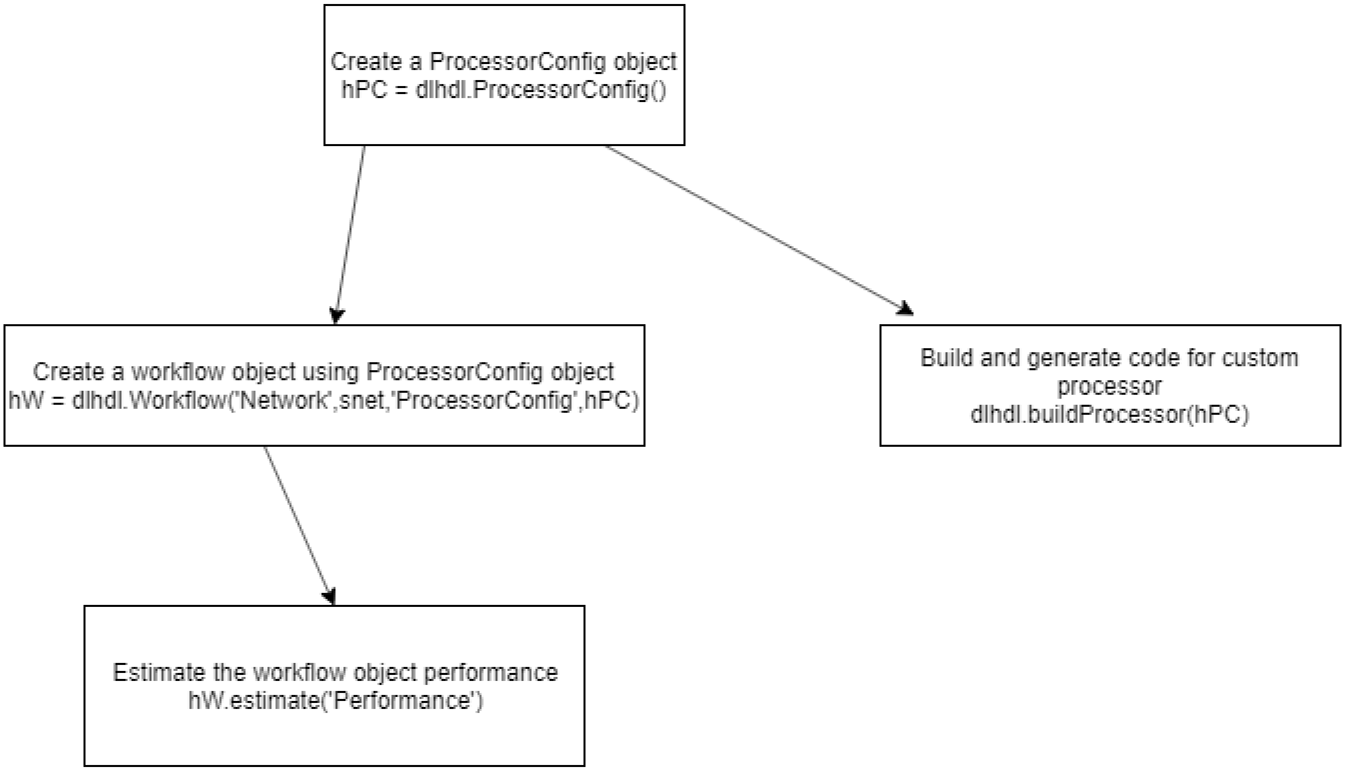
Custom Processor Configuration Workflow

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.


Create a ProcessorConfig object
hPC = dlhdl.ProcessorConfig()
Create a workflow object using ProcessorConfig object
hW = dlhdl.Workflow('Network',snet,'ProcessorConfig',hPC)
Build and generate code for custom processor
dlhdl.buildProcessor(hPC)
Estimate the workflow object performance
hW.estimate('Performance')

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/gpucoder/cnn_models/logo_detection/LogoNet.mat';
        websave('LogoNet.mat',url);
    end
    data = load('LogoNet.mat');
    net  = data.convnet;
end
```

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

```
snet = getLogoNetwork;
```

2 Create a `dlhdl.ProcessorConfig` object.

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

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

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

```
3 Memory Regions created.


              Deep Learning Processor Estimator Performance Results

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

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

Evaluate Performance of Deep Learning Network on Custom Processor Configuration

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

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

Prerequisites

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

Load Pretrained Network

Load the pretrained network.

```
snet = resnet18;
```

Retrieve `zcu102_single Bitstream Configuration`

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

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

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

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

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

              Processor Top Level Properties
                              RunTimeControl: 'register'
                          InputDataInterface: 'External Memory'
```

```
                         OutputDataInterface: 'External Memory'
                           ProcessorDataType: 'single'

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

Estimate ResNet-18 Performance for `zcu102_single` Bitstream Configuration

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

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

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


              Deep Learning Processor Estimator Performance Results

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

Create Custom Processor Configuration

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

```
hPC_custom = dlhdl.ProcessorConfig
```

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

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

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

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

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

Estimate ResNet-18 Performance for Custom Bitstream Configuration

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

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

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


              Deep Learning Processor Estimator Performance Results

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

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

Modify Custom Processor Configuration

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

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

```
hPC_custom = 
                    Processing Module "conv"
```

```
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

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

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

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

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

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

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

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

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


              Deep Learning Processor Estimator Performance Results

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

See Also

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

More About

- “Estimate 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'
                          LRNBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

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

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

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

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

2 Call `estimateResources` to retrieve resource utilization.

```
hPC.estimateResources
```

```
             Deep Learning Processor Estimator Resource Results

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

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

Customize Bitstream Configuration to Meet Resource Use Requirements

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=21×5 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 Ki
                             TargetFrequency: 250
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

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

To estimate the performance of the digits series network, use the `estimatePerformance` 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.laye
### 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 imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                      58101                  0.00023                       1
    ____conv_1                4391                  0.00002
    ____maxpool_1             2877                  0.00001
    ____conv_2                2351                  0.00001
    ____maxpool_2             2265                  0.00001
    ____conv_3                2651                  0.00001
    ____fc                   43566                  0.00017
 * The clock frequency of the DL processor is: 250MHz
```

```
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.laye
### 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 imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        -
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.
“ProcessorDataType”	`Top Level`	Change data type to int8.	Improves performance. There could be a drop in accuracy.	Reduces 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.

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

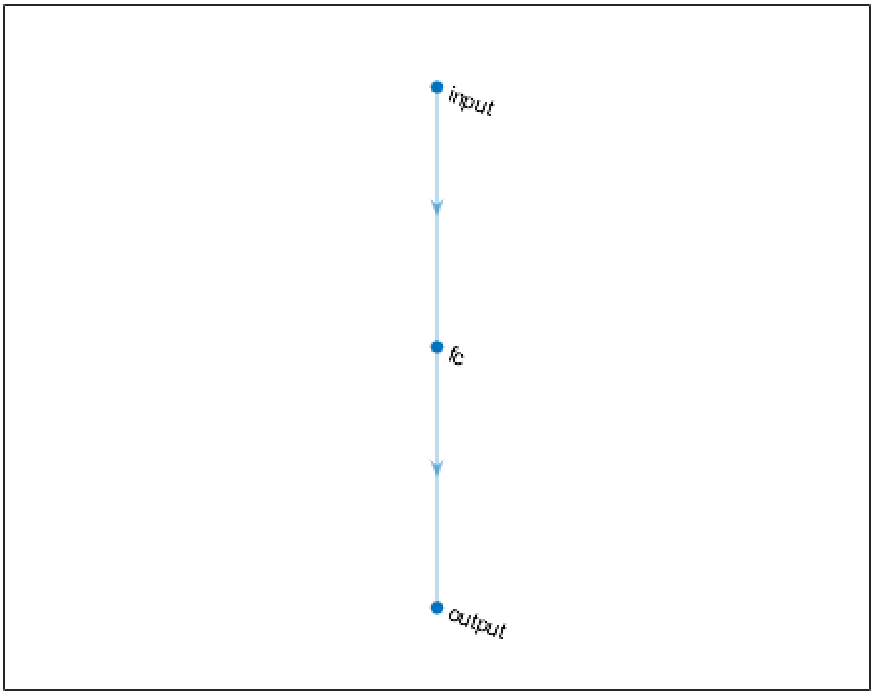
```
hPC = 
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'on'
                            ConvThreadNumber: 16
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [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 Ki
                             TargetFrequency: 200
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

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

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

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



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

```
hPC.estimateResources
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912            274080
                        -------------    -------------    -------------
DL_Processor                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 softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in


              Deep Learning Processor Estimator Performance Results

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

The target device resource counts are:

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

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

- Digital signal processor (DSP) slice count — 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 softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in


              Deep Learning Processor Estimator Performance Results

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

The target device resource counts are:

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

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

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

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

Customize Processor Configuration by Using optimizeConfigurationForNetwork

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

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

```
### Optimizing processor configuration for deep learning network begin.
### Note: Processing module "conv" property "ModuleGeneration" changed from "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" becaus

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

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

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


              Deep Learning Processor Estimator Performance Results

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

The target device resource counts are:

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

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

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

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

Generate Custom Bitstream

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

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

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

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

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

See Also

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

More About

- "Estimate Performance of Deep Learning Network" on page 8-3
- "Estimate Resource Utilization for Custom Processor Configuration" on page 8-10
- "Effects of Custom Deep Learning Processor Parameters on Performance and Resource Utilization" on page 8-16

9

Custom Processor Code Generation Workflow

Generate Custom Bitstream

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

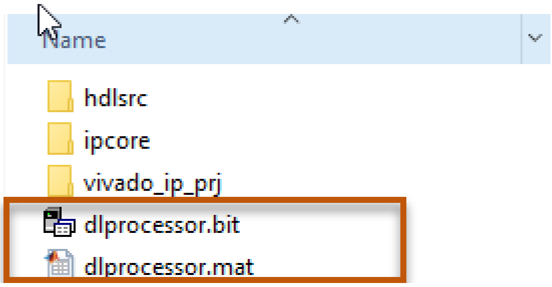
1. Create a `dlhdl.ProcessorConfig` object.

   ```
   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.

   Name

   hdlsrc

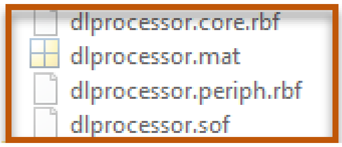
   ipcore

   vivado_ip_prj

   dlprocessor.bit

   dlprocessor.mat

   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.

   dlprocessor.core.rbf

   dlprocessor.mat

   dlprocessor.periph.rbf

   dlprocessor.sof
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

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: [1×1 SeriesNetwork]
          Bitstream: 'arria10soc_single'
    ProcessorConfig: []
             Target: [1×1 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.laye
          offset_name          offset_address    allocated_space 
    _______________________    ______________    _________________

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

Program Bitstream onto FPGA and Download Network Weights

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

```
hW.deploy
```

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

Run Prediction for Example Image

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

```
inputImg = imread('five_28x28.pgm');
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      Tota
                         -------------             -------------              ---------        --
Network                      49243                 0.00033                       1
    conv_module              25983                 0.00017
        conv_1                6813                 0.00005
        maxpool_1             4705                 0.00003
        conv_2                5205                 0.00003
        maxpool_2             3839                 0.00003
        conv_3                5481                 0.00004
    fc_module                23260                 0.00016
        fc                   23260                 0.00016
 * The clock frequency of the DL processor is: 150MHz
```

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

```
The prediction result is 5
```

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: [1×1 SeriesNetwork]
          Bitstream: 'zcu102_single'
    ProcessorConfig: []
             Target: [1×1 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             28×28×1 images with 'zerocenter' normalization
     2   'conv_1'        Convolution             8 3×3×1 convolutions with stride [1  1] and padd
     3   'batchnorm_1'   Batch Normalization     Batch normalization with 8 channels
     4   'relu_1'        ReLU                    ReLU
     5   'maxpool_1'     Max Pooling             2×2 max pooling with stride [2  2] and padding 
     6   'conv_2'        Convolution             16 3×3×8 convolutions with stride [1  1] and pad
     7   'batchnorm_2'   Batch Normalization     Batch normalization with 16 channels
     8   'relu_2'        ReLU                    ReLU
     9   'maxpool_2'     Max Pooling             2×2 max pooling with stride [2  2] and padding 
    10   'conv_3'        Convolution             32 3×3×16 convolutions with stride [1  1] and pa
    11   'batchnorm_3'   Batch Normalization     Batch normalization with 32 channels
    12   'relu_3'        ReLU                    ReLU
    13   'fc'            Fully Connected         10 fully connected layer
    14   'softmax'       Softmax                 softmax
    15   'classoutput'   Classification Output   crossentropyex with '0' and 9 other classes

3 Memory Regions created.

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

```
Emitting Status Table...complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space 
    _______________________    ______________    _________________

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

### Network compilation complete.
```

Program Bitstream onto FPGA and Download Network Weights

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

```
hW.deploy
```

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

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


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);
```

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

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

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


              Deep Learning Processor Profiler Performance Results

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

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

```
The prediction result is 5
```

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 detction series networkas 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.b
```

```
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.laye
          offset_name          offset_address    allocated_space
    _______________________    ______________    ________________
    "InputDataOffset"           "0x00000000"     "4.0 MB"
    "OutputResultOffset"        "0x00400000"     "4.0 MB"
    "SystemBufferOffset"        "0x00800000"     "28.0 MB"
    "InstructionDataOffset"     "0x02400000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x02800000"     "4.0 MB"
    "FCWeightDataOffset"        "0x02c00000"     "4.0 MB"
    "EndOffset"                 "0x03000000"     "Total: 48.0 MB"

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

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx 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      Tota
                         -------------             -------------              ---------        --
Network                      80141                  0.00160                       1
    conv_module              47601                  0.00095
        conv_1               10047                  0.00020
        maxpool_1             6999                  0.00014
        conv_2               11367                  0.00023
        maxpool_2             5465                  0.00011
        conv_3               13783                  0.00028
    fc_module                32540                  0.00065
        fc                   32540                  0.00065
 * The clock frequency of the DL processor is: 50MHz
```

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

```
The prediction result is 5
```

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™

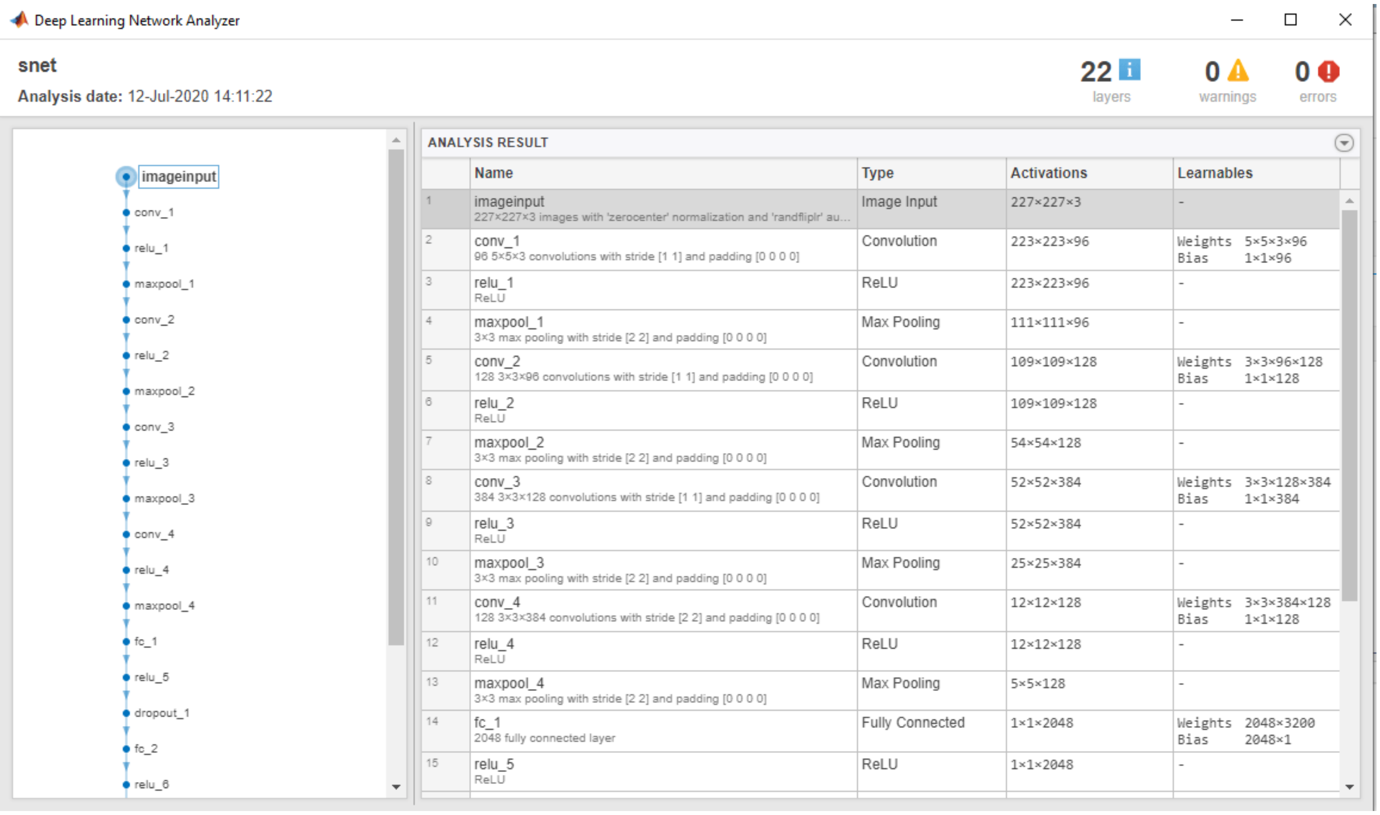
Load the Pretrained Series Network

To load the pretrained series network logonet, enter:

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

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

```
analyzeNetwork(snet)
```



snet

Analysis date: 12-Jul-2020 14:11:22

ANALYSIS RESULT

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

Create Target Object

Create a target object that has a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use JTAG, install Xilinx™ Vivado™ Design Suite 2019.2. To set the Xilinx Vivado toolpath, enter:

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

To create the target object, enter:

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

Create WorkFlow Object

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

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

Compile the Logo Recognition Network

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

```
dn = hW.compile
```

```
          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

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

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

Program Bitstream onto FPGA and Download Network Weights

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

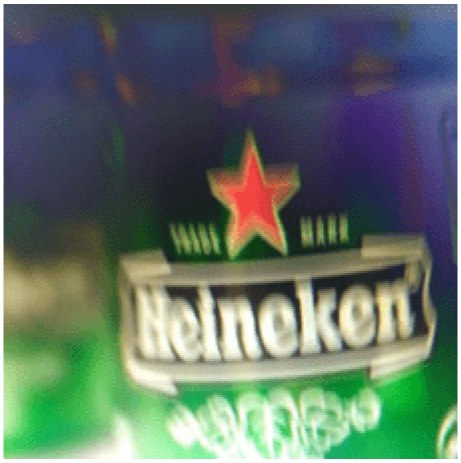
```
 hW.deploy
```

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

Load the Example Image

Load the example image.

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



Run the Prediction

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

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

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

              Deep Learning Processor Profiler Performance Results

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

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

```
ans =
'heineken'
```

Deploy Transfer Learning Network for Lane Detection

This example shows how to create, compile, and deploy a `dlhdl.Workflow` object that has a convolutional neural network. 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();
```

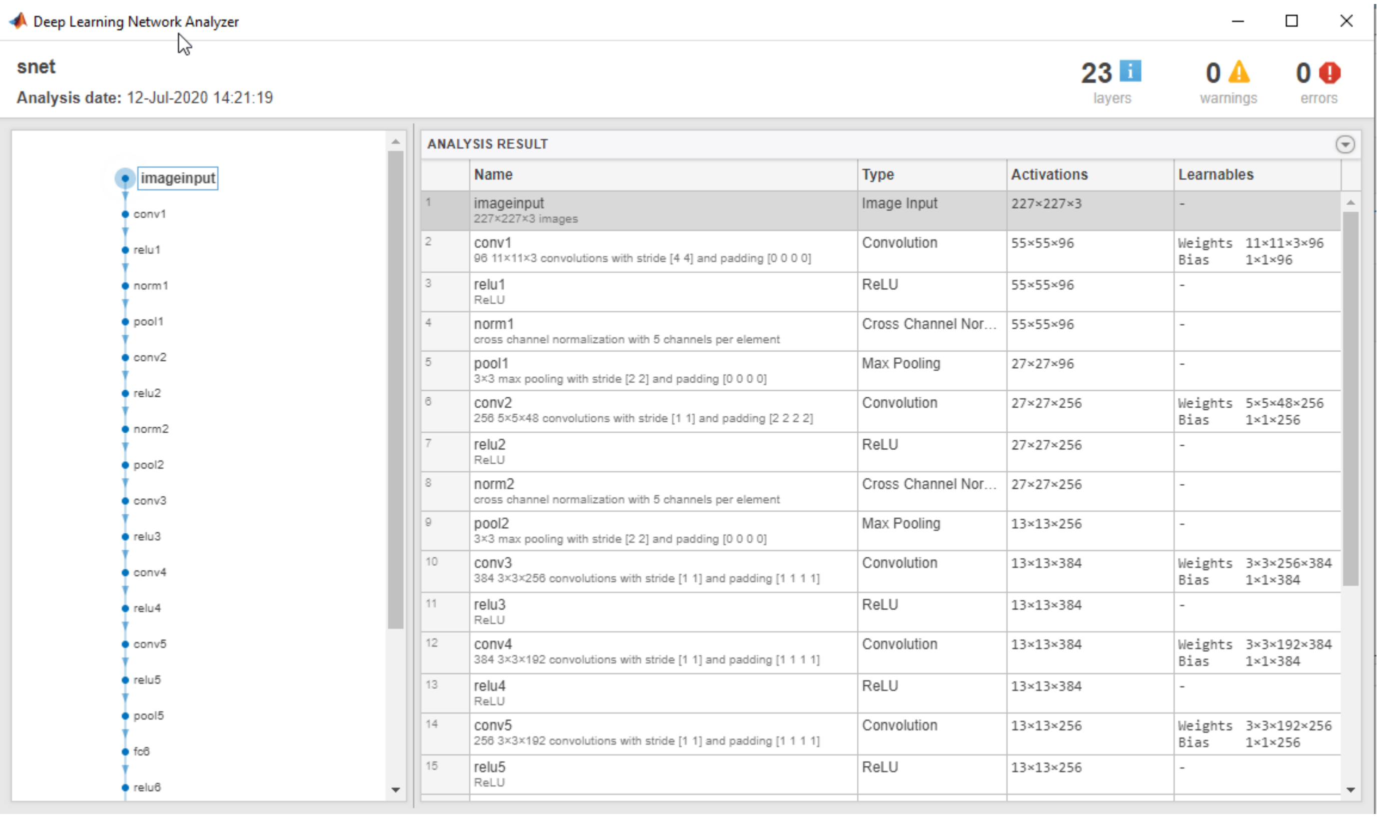
Normalize the Input Layer

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 normali
% max pool, fully connected, and the regression output layers.
```



snet

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

23 layers, 0 warnings, 0 errors

ANALYSIS RESULT

	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');
```

Create WorkFlow Object

Create an object of the `dlhdl.Workflow` class. When you create the class, specify the network and the bitstream name. Specify the saved pretrained lanenet 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 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 ta
### Loading weights to FC Processor.
### 13% finished, current time is 28-Jun-2020 12:36:09.
### 25% finished, current time is 28-Jun-2020 12:36:10.
### 38% finished, current time is 28-Jun-2020 12:36:11.
### 50% finished, current time is 28-Jun-2020 12:36:12.
### 63% finished, current time is 28-Jun-2020 12:36:13.
### 75% finished, current time is 28-Jun-2020 12:36:14.
### 88% finished, current time is 28-Jun-2020 12:36:14.
### FC Weights loaded. Current time is 28-Jun-2020 12:36:15
```

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      Tota
                         -------------             -------------              ---------        --
Network                   24904175                  0.11320                       1          249
    conv_module            8967009                  0.04076
        conv1              1396633                  0.00635
        norm1               623003                  0.00283
        pool1               226855                  0.00103
        conv2              3410044                  0.01550
        norm2               378531                  0.00172
        pool2               233635                  0.00106
        conv3              1139419                  0.00518
        conv4               892918                  0.00406
        conv5               615897                  0.00280
        pool5                50189                  0.00023
```

```
   fc_module             15937166                  0.07244
       fc6               15819257                  0.07191
       fcLane1             117125                  0.00053
       fcLane2                782                  0.00000
* The clock frequency of the DL processor is: 220MHz
```

Image Category Classification by Using Deep Learning

This example shows you how to create, compile, and deploy a `dlhdl.Workflow` object with alexnet 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. Alexnet 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 Alexnet
- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™

Load the Pretrained Series Network

To load the pretrained series network alexnet, enter:

```
snet = alexnet;
```

To load the pretrained series network vgg19, enter:

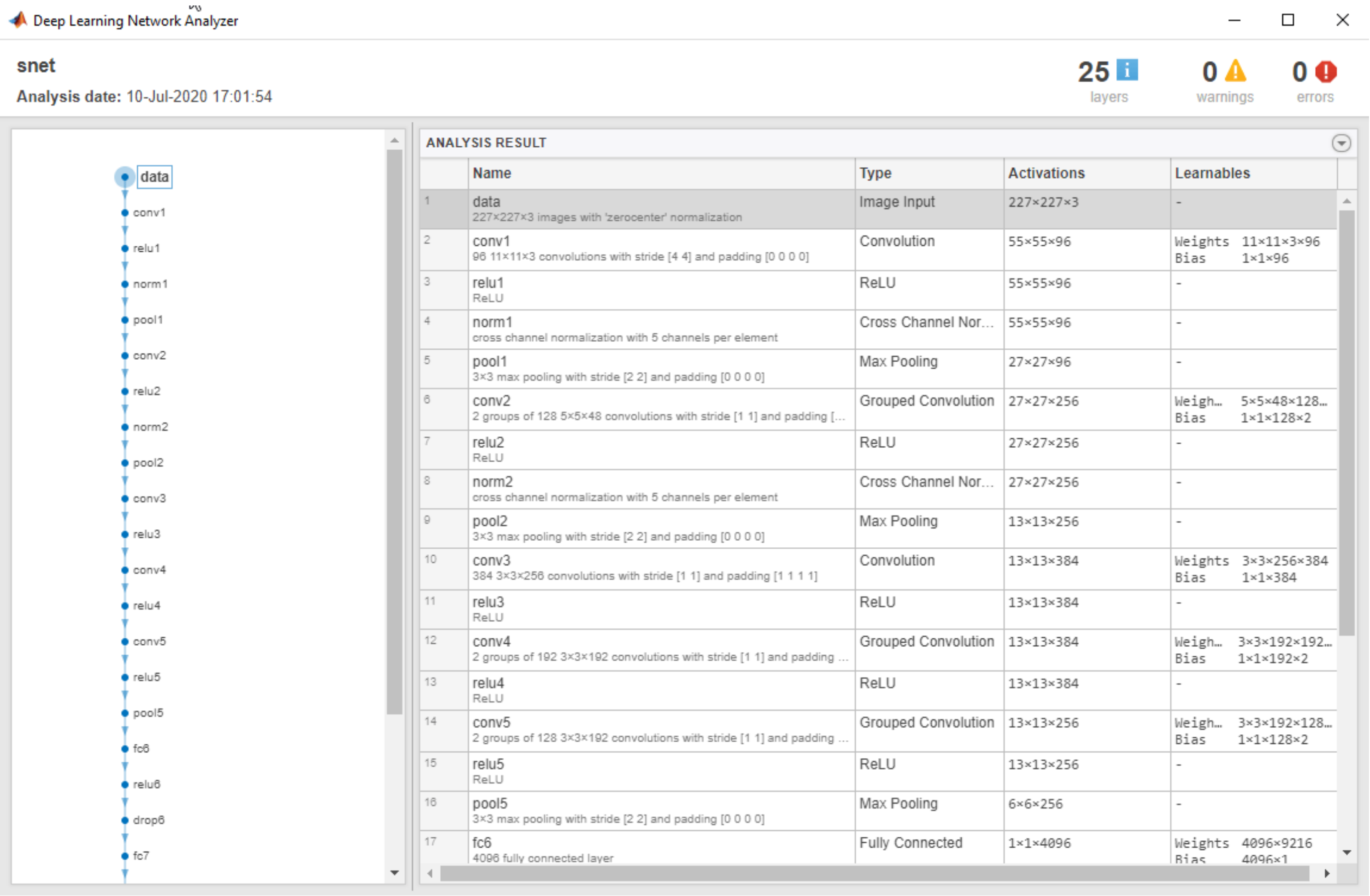
```
% snet = vgg19;
```

To load the pretrained series network darknet19, enter:

```
% snet = darknet19;
```

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

```
analyzeNetwork(snet)
% The saved network contains 25 layers including input, convolution, ReLU, cross channel normaliz
% max pool, fully connected, and the softmax output layers.
```



snet

Analysis date: 10-Jul-2020 17:01:54

25 layers 0 warnings 0 errors

ANALYSIS RESULT

	Name	Type	Activations	Learnables
1	data 227×227×3 images with 'zerocenter' normalization	Image Input	227×227×3	-
2	conv1 96 11×11×3 convolutions with stride [4 4] and padding [0 0 0 0]	Convolution	55×55×96	Weights 11×11×3×96 Bias 1×1×96
3	relu1 ReLU	ReLU	55×55×96	-
4	norm1 cross channel normalization with 5 channels per element	Cross Channel Nor...	55×55×96	-
5	pool1 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	27×27×96	-
6	conv2 2 groups of 128 5×5×48 convolutions with stride [1 1] and padding [...	Grouped Convolution	27×27×256	Weigh... 5×5×48×128... Bias 1×1×128×2
7	relu2 ReLU	ReLU	27×27×256	-
8	norm2 cross channel normalization with 5 channels per element	Cross Channel Nor...	27×27×256	-
9	pool2 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	13×13×256	-
10	conv3 384 3×3×256 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	13×13×384	Weights 3×3×256×384 Bias 1×1×384
11	relu3 ReLU	ReLU	13×13×384	-
12	conv4 2 groups of 192 3×3×192 convolutions with stride [1 1] and padding ...	Grouped Convolution	13×13×384	Weigh... 3×3×192×192... Bias 1×1×192×2
13	relu4 ReLU	ReLU	13×13×384	-
14	conv5 2 groups of 128 3×3×192 convolutions with stride [1 1] and padding ...	Grouped Convolution	13×13×256	Weigh... 3×3×192×128... Bias 1×1×128×2
15	relu5 ReLU	ReLU	13×13×256	-
16	pool5 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	6×6×256	-
17	fc6 4096 fully connected layer	Fully Connected	1×1×4096	Weights 4096×9216 Bias 4096×1

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.b
```

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

Compile the Alexnet Series network

To compile the Alexnet series 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)
```

```
          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "12.0 MB"
    "OutputResultOffset"        "0x00c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x01000000"     "28.0 MB"
    "InstructionDataOffset"     "0x02c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03000000"     "16.0 MB"
    "FCWeightDataOffset"        "0x04000000"     "224.0 MB"
    "EndOffset"                 "0x12000000"     "Total: 288.0 MB"

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

Program Bitstream onto FPGA and Download Network Weights

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

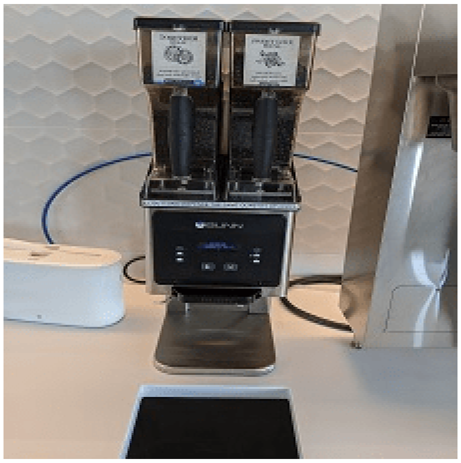
```
hW.deploy
```

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

Load Image for Prediction

Load the example image.

```
imgFile = 'espressomaker.jpg';
inputImg = imresize(imread(imgFile), [227,227]);
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

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        -
Network                   33531964                  0.15242                       1           335
    conv_module            8965629                  0.04075
        conv1              1396567                  0.00635
        norm1               622836                  0.00283
        pool1               226593                  0.00103
        conv2              3409730                  0.01550
        norm2               378491                  0.00172
        pool2               233223                  0.00106
        conv3              1139273                  0.00518
        conv4               892869                  0.00406
        conv5               615895                  0.00280
        pool5                50267                  0.00023
    fc_module             24566335                  0.11167
        fc6               15819119                  0.07191
        fc7                7030644                  0.03196
        fc8                1716570                  0.00780
 * The clock frequency of the DL processor is: 220MHz
```

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

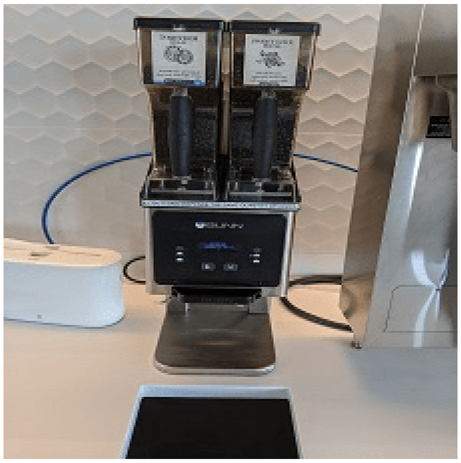
```
ans =
'espresso maker'
```

Run Prediction for Multiple Images

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

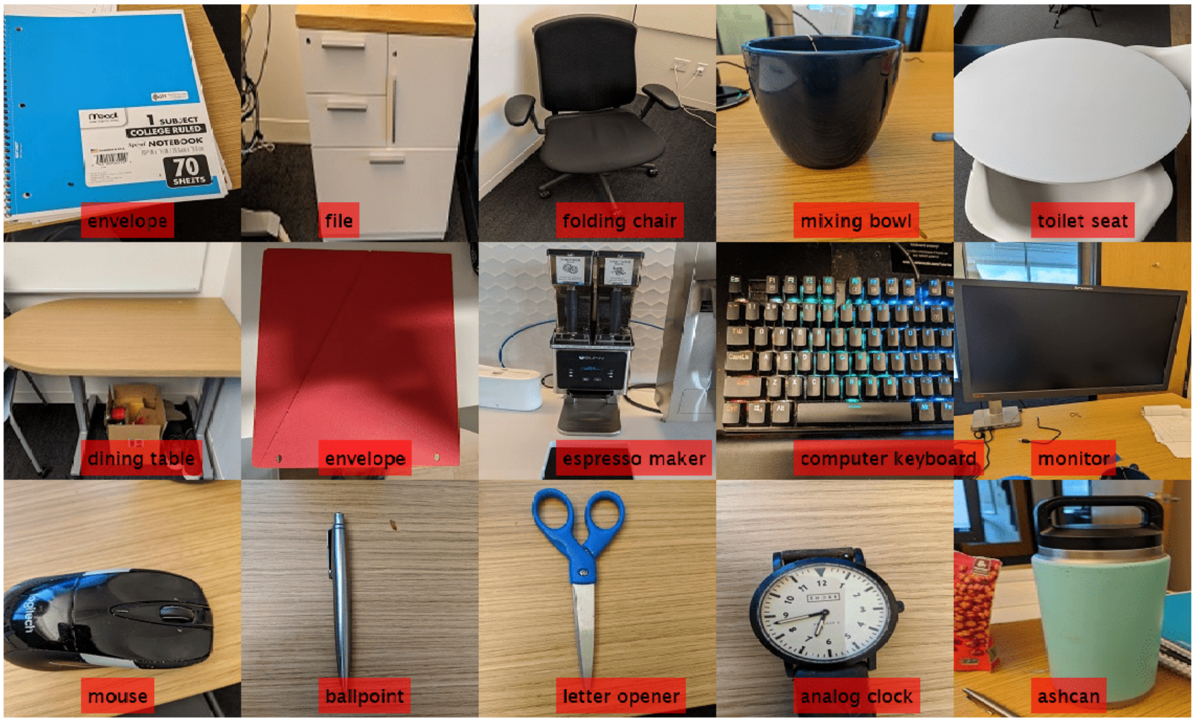
The `demoOnImage` function loads multiple images and retrieves their prediction results. The `annotateresults` function displays the image prediction result on top of the images which are assembled into a 3-by-5 array.

```
imshow(inputImg)
```



```
 demoOnImage;
```

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

FPGA PREDICTION: envelope
FPGA PREDICTION: file
FPGA PREDICTION: folding chair
FPGA PREDICTION: mixing bowl
FPGA PREDICTION: toilet seat
FPGA PREDICTION: dining table
FPGA PREDICTION: envelope
FPGA PREDICTION: espresso maker
FPGA PREDICTION: computer keyboard
FPGA PREDICTION: monitor
FPGA PREDICTION: mouse
FPGA PREDICTION: ballpoint
FPGA PREDICTION: letter opener
FPGA PREDICTION: analog clock
FPGA PREDICTION: ashcan
```


envelope
file
folding chair
mixing bowl
toilet seat
dining table
envelope
espresso maker
computer keyboard
monitor
mouse
ballpoint
letter opener
analog clock
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_blemdetnet`.

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

```
snet_blemdetnet = 
  SeriesNetwork with properties:

         Layers: [12×1 nnet.cnn.layer.Layer]
```

```
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

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

```
    analyzeNetwork(snet_blemdetnet)
```

Create Target Object

Create a target object that has a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use the JTAG connection, install the Xilinx™ Vivado™ Design Suite 2020.1.

Set the Xilinx Vivado toolpath.

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2020.1\bin\vivado.b
    hT = dlhdl.Target('Xilinx','Interface','Ethernet')
```

```
hT = 
  Target with properties:

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

Create Workflow Object for trainedDefNet Network

Create an object of the `dlhdl.Workflow` class. Specify the network and the bitstream name. Specify the saved pretrained network `trainedDefNet` 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 the single data type.

```
hW = dlhdl.Workflow('Network',snet_defnet,'Bitstream','zcu102_single','Target',hT)
```

```
hW = 
  Workflow with properties:

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

Compile trainedDefNet Series Network

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

```
hW.compile
```

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

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

3 Memory Regions created.

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

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

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

### Network compilation complete.

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

Program Bitstream onto FPGA and Download Network Weights

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

```
hW.deploy
```

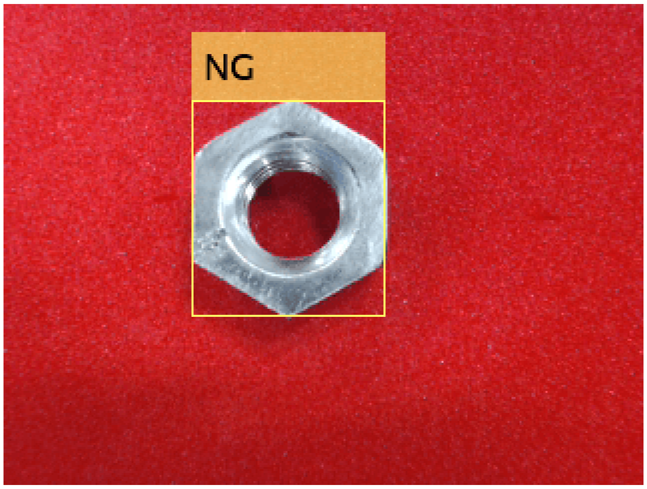
```
### Programming FPGA Bitstream using Ethernet...
Downloading target FPGA device configuration over Ethernet to SD card ...
# Copied /tmp/hdlcoder_rd to /mnt/hdlcoder_rd
# Copying Bitstream hdlcoder_system.bit to /mnt/hdlcoder_rd
# Set Bitstream to hdlcoder_rd/hdlcoder_system.bit
# Copying Devicetree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

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


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Dec-2020 16:16:31
### Loading weights to FC Processor.
### 20% finished, current time is 16-Dec-2020 16:16:32.
### 40% finished, current time is 16-Dec-2020 16:16:32.
### 60% finished, current time is 16-Dec-2020 16:16:33.
### 80% finished, current time is 16-Dec-2020 16:16:34.
### FC Weights loaded. Current time is 16-Dec-2020 16:16:34
```

Run Prediction for One Image

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

```
wi = uint32(320);
he = uint32(240);
ch = uint32(3);
filename = fullfile(pwd,'ng1.png');
```

```
img=imread(filename);
img = imresize(img, [he, wi]);
img = mat2ocv(img);

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

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

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

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


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        -
Network                   12231156                  0.05560                       1           122
    conv1                   414021                  0.00188
    norm1                   172325                  0.00078
    pool1                    56747                  0.00026
    conv2                   654112                  0.00297
    norm2                   119403                  0.00054
    pool2                    43611                  0.00020
    conv3                   777446                  0.00353
    conv4                   595551                  0.00271
    conv5                   404425                  0.00184
    pool5                    17831                  0.00008
    fc6                    1759699                  0.00800
    fc7                    7030188                  0.03196
    fc8                     185672                  0.00084
 * The clock frequency of the DL processor is: 220MHz

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

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

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



Create Workflow Object for `trainedBlemDetNet` Network

Create an object of the `dlhdl.Workflow` class. Specify the network and the bitstream name. Specify the saved pretrained network `trainedblemDetNet` 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 the single data type.

```
hW = dlhdl.Workflow('Network',snet_blemdetnet,'Bitstream','zcu102_single','Target',hT)
```

```
hW = 
  Workflow with properties:

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

Compile trainedBlemDetNet Series Network

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

```
hW.compile
```

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

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

3 Memory Regions created.

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

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    ________________

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

### Network compilation complete.

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

Program Bitstream onto FPGA and Download Network Weights

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

```
 hW.deploy
```

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

Run Prediction for One Image

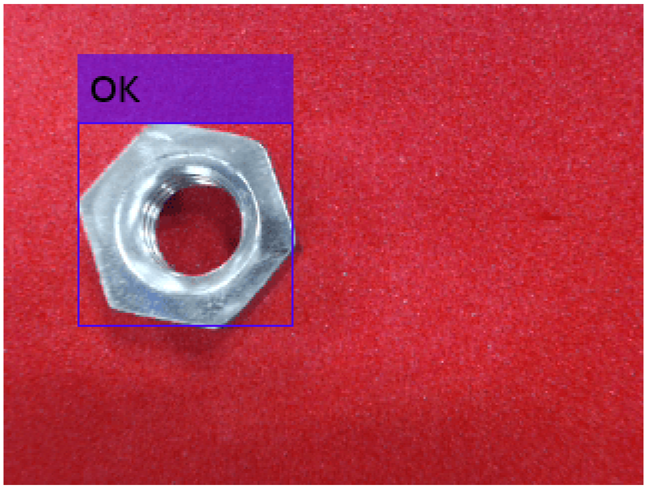
Load an image from the attached `testImages` folder and resize the image to match the network image input layer dimensions. Run the predict function of the `dlhdl.Workflow` object to retrieve and display the defect prediction from the FPGA.

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

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

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

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

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

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


              Deep Learning Processor Profiler Performance Results

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

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

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

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

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



Quantize and Deploy trainedBlemDetNet Network

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

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

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

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

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

```
dlQuantObj = 
  dlquantizer with properties:

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

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

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

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

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

⋮

Create an object of the `dlhdl.Workflow` class. Specify the network and the bitstream name. Specify the saved pretrained quantized `trainedblemDetNet` 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 Xilinx ZCU102 SOC board. The bitstream uses the `int8` data type.

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

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

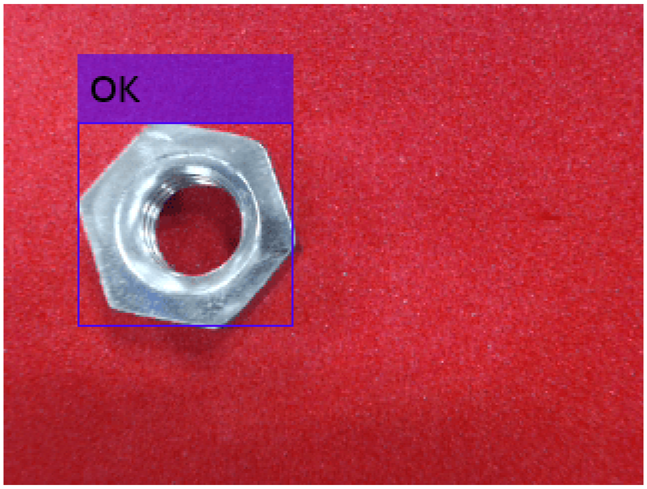
```
hW.compile('InputFrameNumberLimit',30)
```

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

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

3 Memory Regions created.

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

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space
    _______________________    ______________    ________________
    "InputDataOffset"           "0x00000000"     "16.0 MB"
    "OutputResultOffset"        "0x01000000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01400000"     "4.0 MB"
    "SystemBufferOffset"        "0x01800000"     "28.0 MB"
    "InstructionDataOffset"     "0x03400000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03800000"     "4.0 MB"
    "FCWeightDataOffset"        "0x03c00000"     "12.0 MB"
    "EndOffset"                 "0x04800000"     "Total: 72.0 MB"

### Network compilation complete.

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

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

```
hW.deploy
```

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

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


System is rebooting .
```



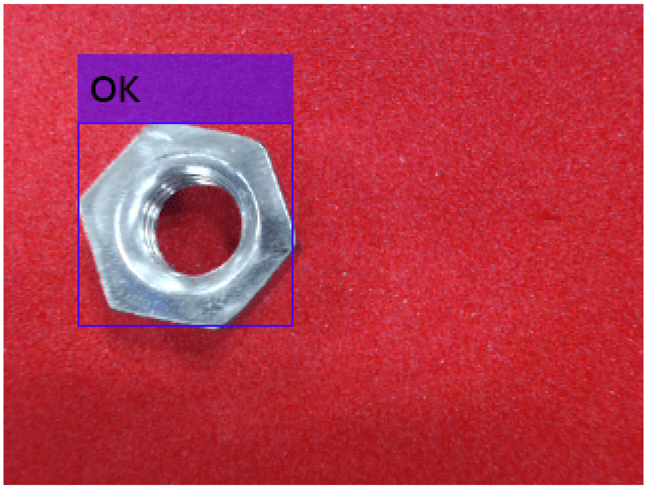
```
  . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Dec-2020 16:18:03
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 16-Dec-2020 16:18:03
```

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

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

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

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

    % row-major > column-major conversion
    imgPacked2 = zeros([128,128,4],'uint8');
    for c = 1:4
        for i = 1:128
            for j = 1:128
                imgPacked2(i,j,c) = imgPacked((i-1)*128 + (j-1) + (c-1)*128*128 + 1);
            end
        end
    end
    % classify detected nuts by using CNN
    scores = zeros(2,4);
    for i = 1:num
         [scores(:,i), speed] = hW.predict(single(imgPacked2(:,:,i)),'Profile','on');
    end
```

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


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        -
Network                    1754969                  0.00798                       1           17
    conv_1                  271340                  0.00123
    maxpool_1                87533                  0.00040
    crossnorm               125737                  0.00057
    conv_2                  149972                  0.00068
    maxpool_2                19657                  0.00009
    fc_1                   1085683                  0.00493
    fc_2                     14928                  0.00007
 * The clock frequency of the DL processor is: 220MHz
```

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

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

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



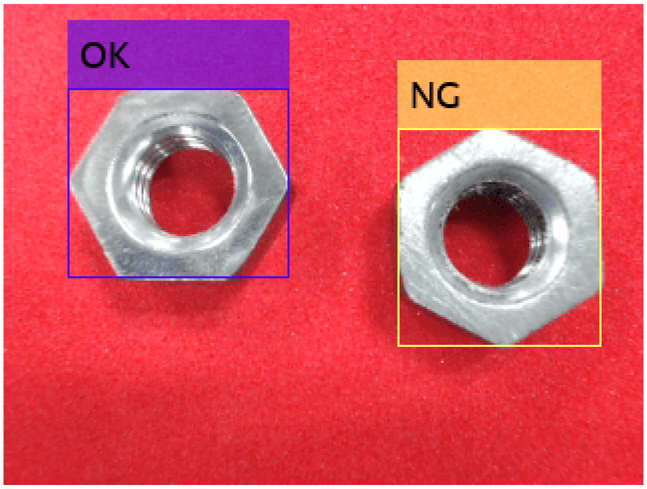
To test that the quantized network can identify all test cases deploy an additional image, resize the image to match the network image input layer dimensions, and run the predict function of the `dlhdl.Workflow` object to retrieve and display the defect prediction from the FPGA.

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

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

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

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

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

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

```
              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------       --
Network                    1754614                  0.00798                       1           17
    conv_1                  271184                  0.00123
    maxpool_1                87557                  0.00040
    crossnorm               125768                  0.00057
    conv_2                  149819                  0.00068
    maxpool_2                19602                  0.00009
    fc_1                   1085664                  0.00493
    fc_2                     14930                  0.00007
 * The clock frequency of the DL processor is: 220MHz


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


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------       --
Network                    1754486                  0.00797                       1           17
    conv_1                  271014                  0.00123
    maxpool_1                87662                  0.00040
    crossnorm               125835                  0.00057
    conv_2                  149789                  0.00068
    maxpool_2                19661                  0.00009
    fc_1                   1085505                  0.00493
    fc_2                     14930                  0.00007
 * The clock frequency of the DL processor is: 220MHz
```

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

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

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



Quantizing the network improves the performance from 45 frames per second to 125 frames per second and reduces the deployed network size from 88 MB to 72 MB.

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 = 
  15×1 Layer array with layers:

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

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.b
```

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.laye
          offset_name          offset_address    allocated_space 
    _______________________    ______________    _________________

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

Program Bitstream onto FPGA and Download Network Weights

To deploy the network on the Xilinx ZCU102 SoC hardware, run the deploy 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      Tota
                         -------------             -------------              ---------        --
Network                       73231                  0.00033                       1
    conv_module               26847                  0.00012
        conv_1                 6618                  0.00003
        maxpool_1              4823                  0.00002
        conv_2                 4876                  0.00002
        maxpool_2              3551                  0.00002
        conv_3                 7039                  0.00003
    fc_module                 46384                  0.00021
        fc                    46384                  0.00021
 * The clock frequency of the DL processor is: 220MHz
```

Identify and Display the 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
```

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

```
### Bottleneck layer index is 13.
```

```
% disply 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.
```

```
% disply 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: [10×1568 single]
          Bias: [10×1 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`.

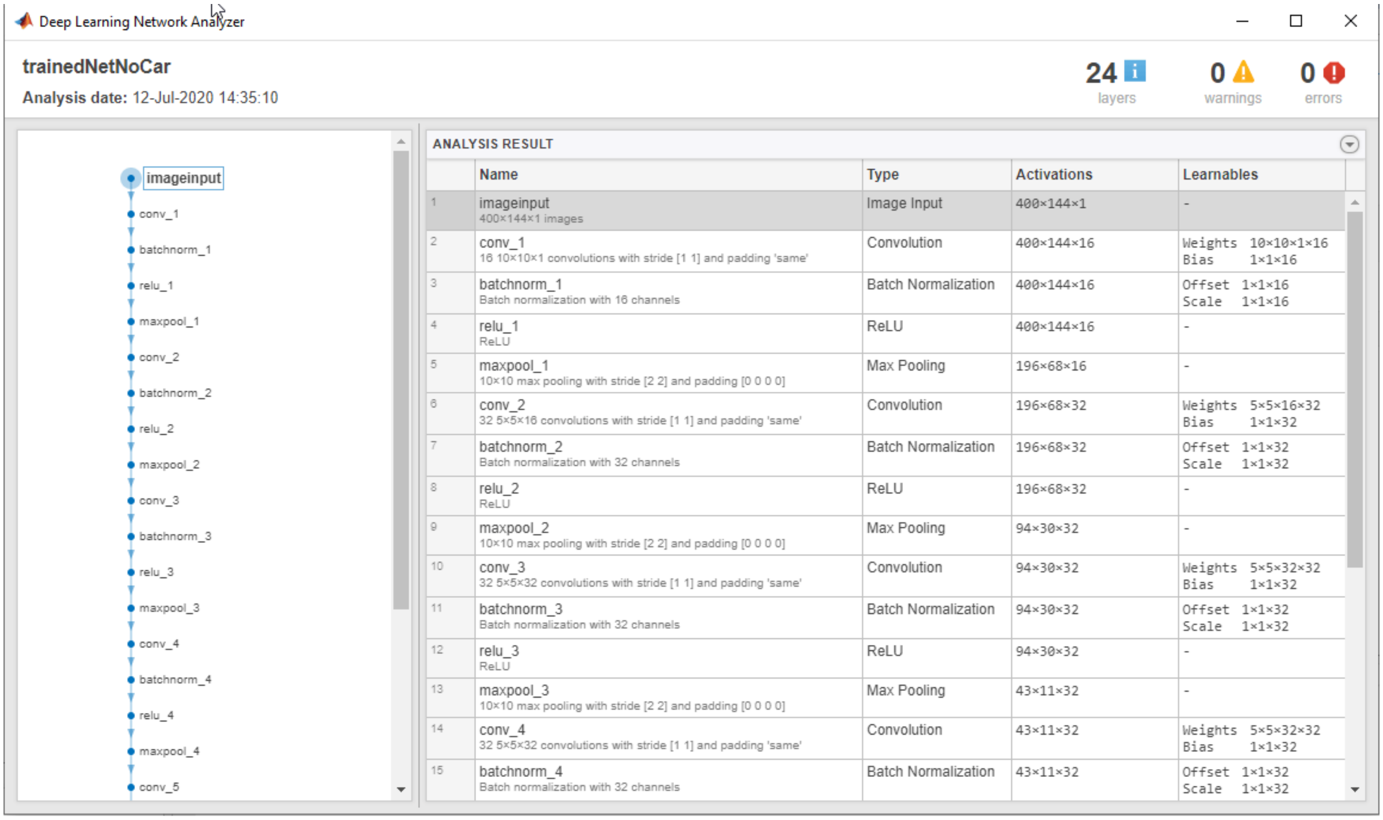
Load Data and Network

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);
```



	Name	Type	Activations	Learnables
1	imageinput 400×144×1 images	Image Input	400×144×1	-
2	conv_1 16 10×10×1 convolutions with stride [1 1] and padding 'same'	Convolution	400×144×16	Weights 10×10×1×16 Bias 1×1×16
3	batchnorm_1 Batch normalization with 16 channels	Batch Normalization	400×144×16	Offset 1×1×16 Scale 1×1×16
4	relu_1 ReLU	ReLU	400×144×16	-
5	maxpool_1 10×10 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	196×68×16	-
6	conv_2 32 5×5×16 convolutions with stride [1 1] and padding 'same'	Convolution	196×68×32	Weights 5×5×16×32 Bias 1×1×32
7	batchnorm_2 Batch normalization with 32 channels	Batch Normalization	196×68×32	Offset 1×1×32 Scale 1×1×32
8	relu_2 ReLU	ReLU	196×68×32	-
9	maxpool_2 10×10 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	94×30×32	-
10	conv_3 32 5×5×32 convolutions with stride [1 1] and padding 'same'	Convolution	94×30×32	Weights 5×5×32×32 Bias 1×1×32
11	batchnorm_3 Batch normalization with 32 channels	Batch Normalization	94×30×32	Offset 1×1×32 Scale 1×1×32
12	relu_3 ReLU	ReLU	94×30×32	-
13	maxpool_3 10×10 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	43×11×32	-
14	conv_4 32 5×5×32 convolutions with stride [1 1] and padding 'same'	Convolution	43×11×32	Weights 5×5×32×32 Bias 1×1×32
15	batchnorm_4 Batch normalization with 32 channels	Batch Normalization	43×11×32	Offset 1×1×32 Scale 1×1×32

Set up HDL Toolpath

Set up the path to your installed Xilinx™ Vivado™ Design Suite 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.laye
          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "28.0 MB"
    "OutputResultOffset"        "0x01c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x02000000"     "28.0 MB"
    "InstructionDataOffset"     "0x03c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04000000"     "4.0 MB"
    "FCWeightDataOffset"        "0x04400000"     "4.0 MB"
    "EndOffset"                 "0x04800000"     "Total: 72.0 MB"
```

Program the Bitstream onto FPGA and Download Network Weights

To deploy the network on the Zynq® UltraScale+™ MPSoC ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object . This function uses the output of the compile function to program the FPGA board by using the programming file.The function also downloads the network weights and biases. The deploy function checks for the Xilinx Vivado tool and the supported tool version. It then starts programming the FPGA device by using the bitstream, displays progress messages and the time it takes to deploy the network.

```
hW.deploy;
```

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

Run Predictions on Micro-Doppler Signatures

Classify one input from the sample test data set by using the predict function of the `dlhdl.Workflow` object and display the label. The inputs to the network correspond to the sonograms of the micro-Doppler signatures for a pedestrian or a bicyclist or a combination of both.

```
testImg = single(testDataNoCar(:, :, :, 1));
testLabel = testLabelNoCar(1);
classnames = trainedNetNoCar.Layers(end).Classes;

% Get predictions from network on single test input
score = hW.predict(testImg, 'Profile', 'On')
```

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


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    9430692                  0.04287                       1           94
    conv_module            9411355                  0.04278
        conv_1             4178753                  0.01899
        maxpool_1          1394883                  0.00634
        conv_2             1975197                  0.00898
        maxpool_2           706156                  0.00321
        conv_3              813598                  0.00370
        maxpool_3           121790                  0.00055
        conv_4              148165                  0.00067
        maxpool_4            22255                  0.00010
        conv_5               41999                  0.00019
        avgpool2d             8674                  0.00004
    fc_module                19337                  0.00009
        fc                   19337                  0.00009
 * The clock frequency of the DL processor is: 220MHz

score = 1×5 single row vector

    0.9956    0.0000    0.0000    0.0044    0.0000
```

```
[~, idx1] = max(score);
predTestLabel = classnames(idx1)
```

```
predTestLabel = categorical
     ped
```

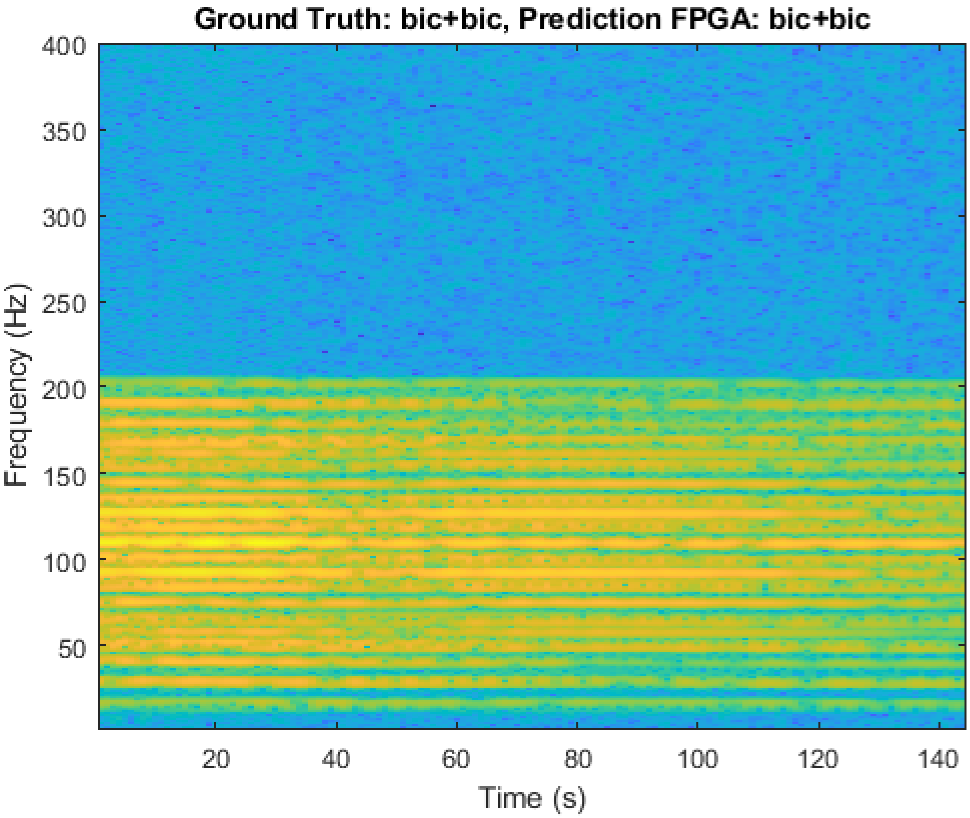
Load five random images from the sample test data set and execute the predict function of the `dlhdl.Workflow` object to display the labels alongside the signatures. The predictions will happen at once since the input is concatenated along the fourth dimension.

```
numTestFrames = size(testDataNoCar, 4);
numView = 5;
listIndex = randperm(numTestFrames, numView);
testImgBatch = single(testDataNoCar(:, :, :, listIndex));
testLabelBatch = testLabelNoCar(listIndex);

% Get predictions from network using DL HDL Toolbox on FPGA
[scores, speed] = hW.predict(testImgBatch, 'Profile', 'On');
```

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


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        -
Network                    9446929                  0.04294                       5           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(predTestLab
    drawnow;
    pause(3);
end
```



The image shows the micro-Doppler signatures of two bicyclists (bic+bic) which is the ground truth. The ground truth is the classification of the image against which the network prediction is compared. The network prediction retrieved from the FPGA correctly predicts that the image has two bicyclists.

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      Tota
                         -------------             -------------              ---------        -
Network                   10182024                  0.06788                       1           101
    conv_module           10182024                  0.06788
        conv_1             7088885                  0.04726
        maxpool_1          3093166                  0.02062
 * The clock frequency of the DL processor is: 150MHz
```

The activations are returned as a 3-D array, with the third dimension indexing the channel on the `maxpool_1` layer. To show these activations using the `imtile` function, reshape the array to 4-D. The third dimension in the input to `imtile` represents the image color. Set the third dimension to have size 1 because the activations do not have color. The fourth dimension indexes the channel.

```
sz = size(act1);
act1 = reshape(act1,[sz(1) sz(2) 1 sz(3)]);
```

Display the activations. Each activation can take any value, so normalize the output using the `mat2gray`. All activations are scaled so that the minimum activation is 0 and the maximum activation is 1. Display the 96 images on an 12-by-8 grid, one for each channel in the layer.

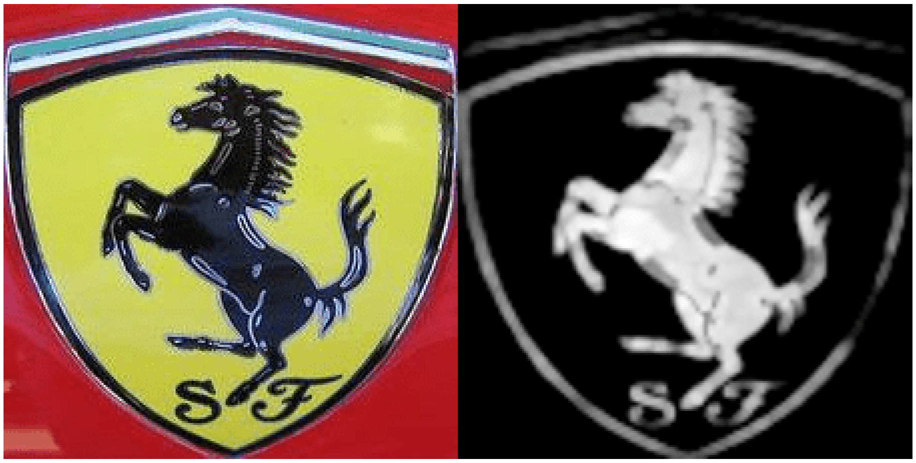
```
I = imtile(mat2gray(act1),'GridSize',[12 8]);
imshow(I)
```

Investigate Activations in Specific Channels

Each tile in the activations grid is the output of a channel in the `maxpool_1` layer. White pixels represent strong positive activations and black pixels represent strong negative activations. A channel that is mostly gray does not activate as strongly on the input image. The position of a pixel in the activation of a channel corresponds to the same position in the original image. A white pixel at a location in a channel indicates that the channel is strongly activated at that position.

Resize the activations in channel 33 to be the same size as the original image and display the activations.

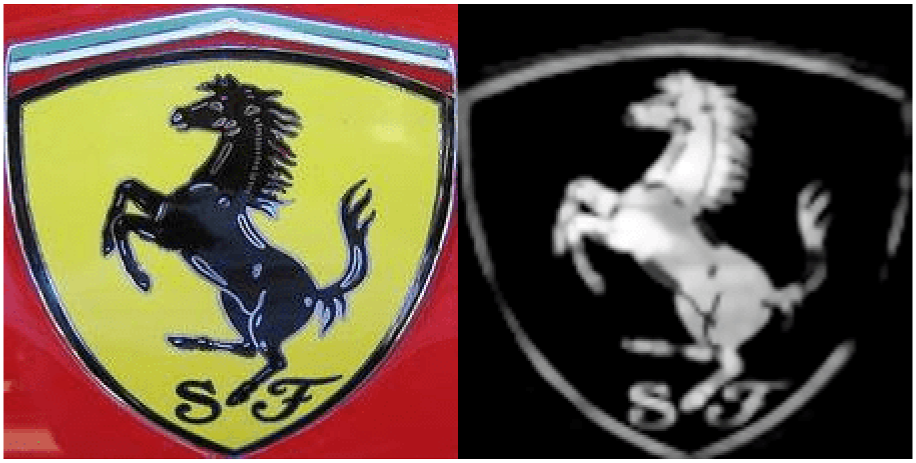
```
act1ch33 = act1(:,:,:,22);
act1ch33 = mat2gray(act1ch33);
act1ch33 = imresize(act1ch33,imgSize);

I = imtile({im,act1ch33});
imshow(I)
```



Find Strongest Activation Channel

Find interesting channels by programmatically investigating channels with large activations. Find the channel that has the largest activation by using the `max` function, resize the channel output, and display the activations.

```
[maxValue,maxValueIndex] = max(max(max(act1)));
act1chMax = act1(:,:,:,maxValueIndex);
act1chMax = mat2gray(act1chMax);
act1chMax = imresize(act1chMax,imgSize);

I = imtile({im,act1chMax});
imshow(I)
```



Compare the strongest activation channel image to the original image. This channel activates on edges. It activates positively on light left/dark right edges and negatively on dark left/light right edges.

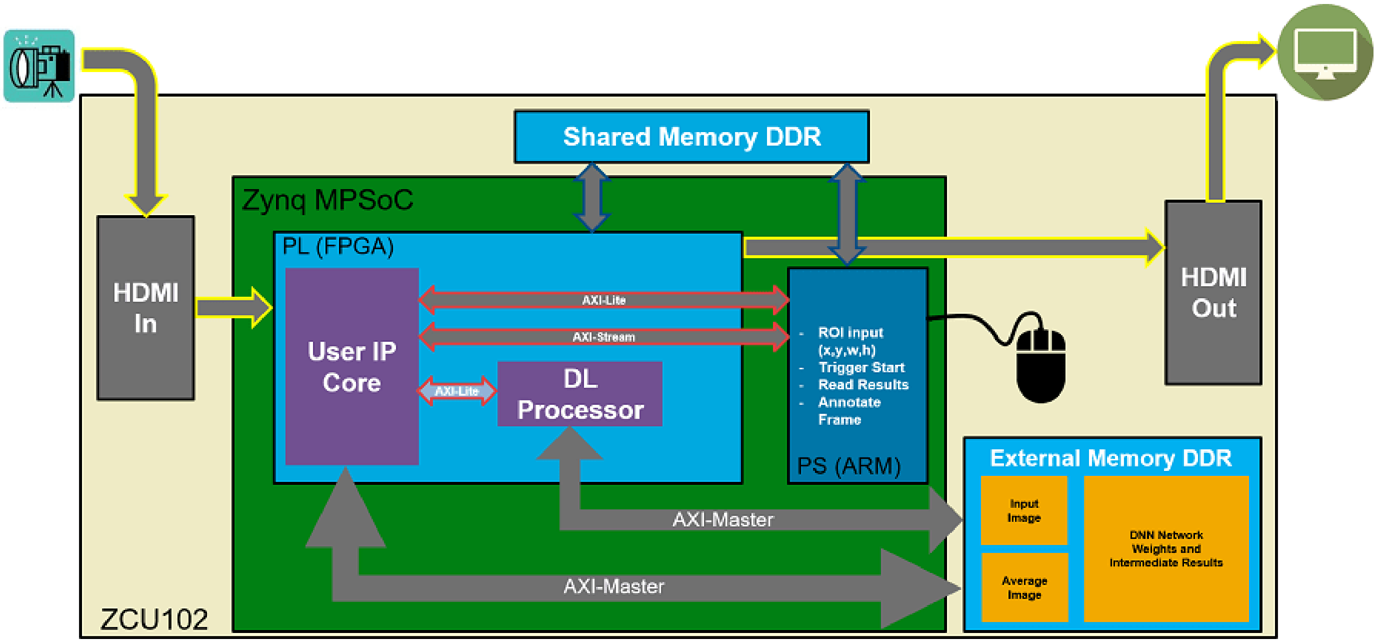
See Also

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 10-63.

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
```

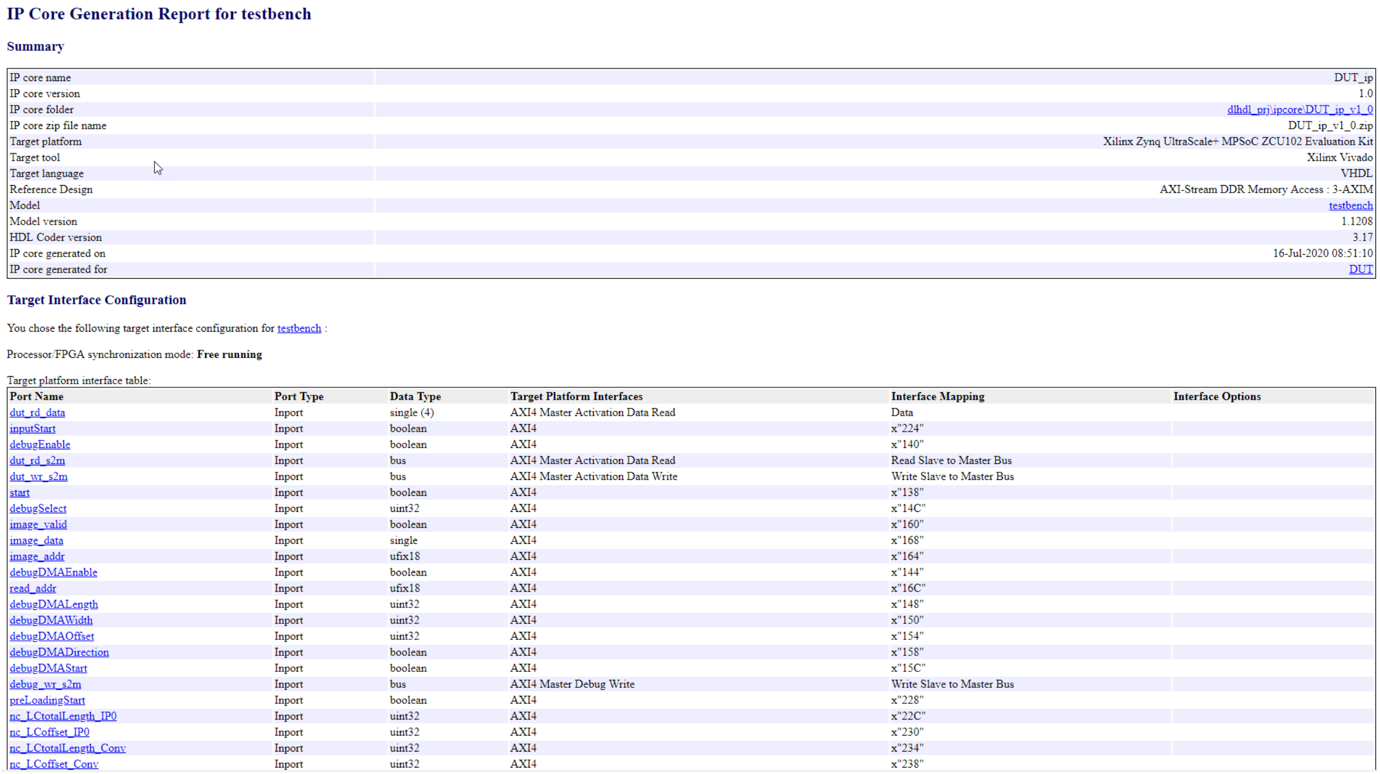
To learn more about the deep learning processor architecture, see “Deep Learning Processor Architecture” on page 2-2. To get information about the custom processor configuration parameters and modifying the parameters, see `getModuleProperty` and `setModuleProperty`.

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_prj\ipcore\DUT_ip_v1_0
IP core zip file name	DUT_ip_v1_0.zip
Target platform	Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
Target tool	Xilinx Vivado
Target language	VHDL
Reference Design	AXI-Stream DDR Memory Access : 3-AXIM
Model	testbench
Model version	1.1208
HDL Coder version	3.17
IP core generated on	16-Jul-2020 08:51:10
IP core generated for	DUT

Target Interface Configuration

You chose the following target interface configuration for testbench :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
dut_rd_data	Inport	single (4)	AXI4 Master Activation Data Read	Data	
inputStart	Inport	boolean	AXI4	x"224"	
debugEnable	Inport	boolean	AXI4	x"140"	
dut_rd_s2m	Inport	bus	AXI4 Master Activation Data Read	Read Slave to Master Bus	
dut_wr_s2m	Inport	bus	AXI4 Master Activation Data Write	Write Slave to Master Bus	
start	Inport	boolean	AXI4	x"138"	
debugSelect	Inport	uint32	AXI4	x"14C"	
image_valid	Inport	boolean	AXI4	x"160"	
image_data	Inport	single	AXI4	x"168"	
image_addr	Inport	ufix18	AXI4	x"164"	
debugDMAEnable	Inport	boolean	AXI4	x"144"	
read_addr	Inport	ufix18	AXI4	x"16C"	
debugDMALength	Inport	uint32	AXI4	x"148"	
debugDMAWidth	Inport	uint32	AXI4	x"150"	
debugDMAOffset	Inport	uint32	AXI4	x"154"	
debugDMADirection	Inport	boolean	AXI4	x"158"	
debugDMAStart	Inport	boolean	AXI4	x"15C"	
debug_wr_s2m	Inport	bus	AXI4 Master Debug Write	Write Slave to Master Bus	
preLoadingStart	Inport	boolean	AXI4	x"228"	
nc_LCtotalLength_IP0	Inport	uint32	AXI4	x"22C"	
nc_LCoffset_IP0	Inport	uint32	AXI4	x"230"	
nc_LCtotalLength_Conv	Inport	uint32	AXI4	x"234"	
nc_LCoffset_Conv	Inport	uint32	AXI4	x"238"	

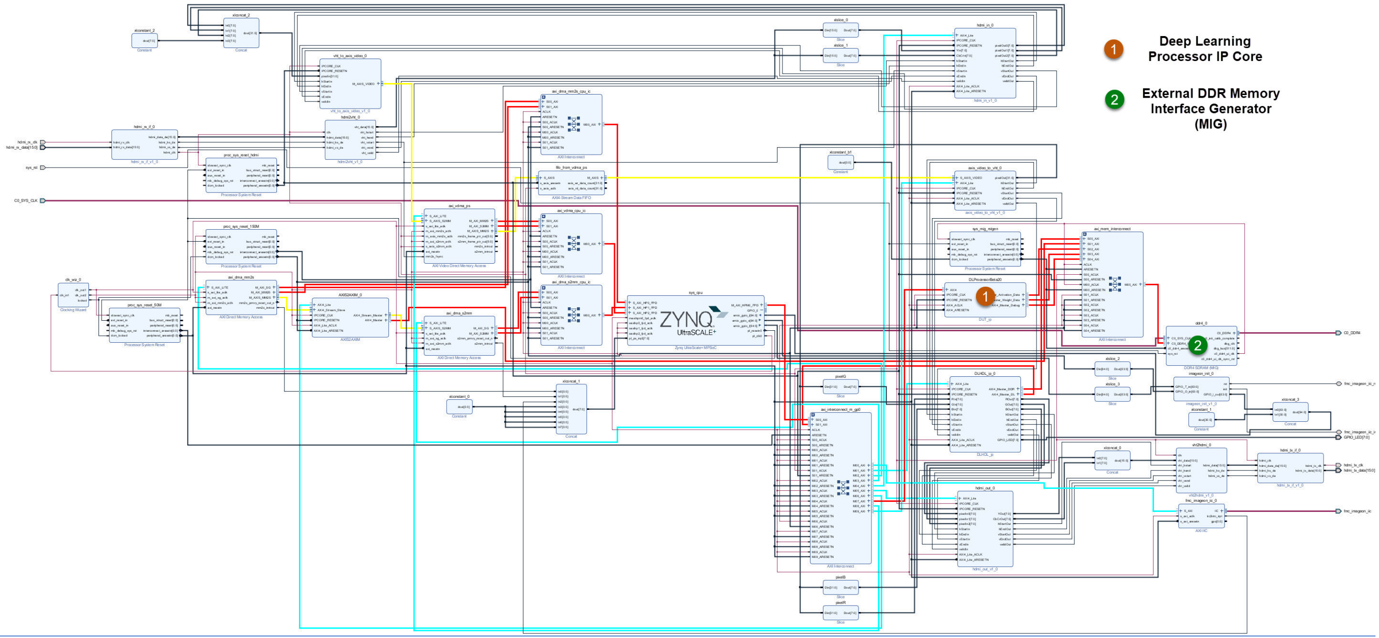
The HTML report contains a description of the deep learning processor IP core, instructions for using the core and integrating the core into your Vivado™ reference design, and a list of AXI4 registers. You will need the AXI4 register list to enter addresses into the Vivado™ Address Mapping tool. For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

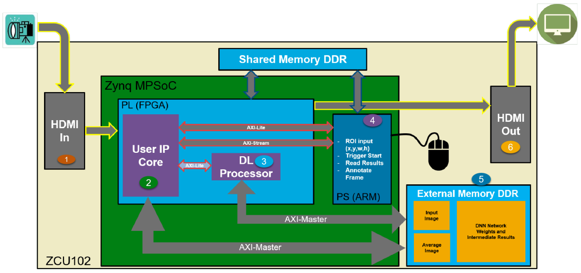
- Connect the generated deep learning processor IP core AXI4 slave interface to an AXI4 master device such as a JTAG AXI master IP core or a Zynq™ processing system (PS). Use the AXI4 master device to communicate with the deep learning processor IP core.
- Connect the vendor provided external memory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

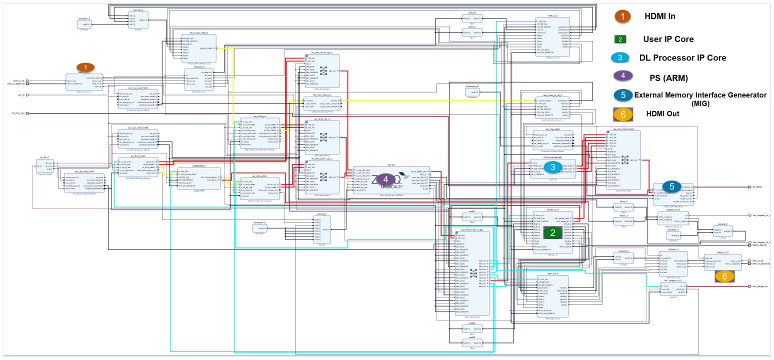
The deep learning processor IP core uses the external memory interface to access the external DDR memory. The image shows the deep learning processor IP core integrated into the Vivado™ reference design and connected to the DDR memory interface generator (MIG) IP.


1 Deep Learning Processor IP Core
2 External DDR Memory Interface Generator (MIG)
ZYNQ UltraSCALE+

Connect the External Memory Interface Generator

In your Vivado™ reference design add an external memory interface generator (MIG) block and connect the generated deep learning processor IP core to the MIG module. The MIG module is connected to the processor IP core through an AXI interconnect module. The image shows the high level architectural design and the Vivado™ reference design implementation.





Create the Reference Design Definition File

The following code describes the contents of the ZCU102 reference design definition file **plugin_rd.m** for the above Vivado™ reference design. For more details on how to define and register the custom board, refer to the “Define Custom Board and Reference Design for Zynq Workflow” (HDL Coder).

```
function hRD = plugin_rd(varargin)

% Parse config
config = ZynqVideoPSP.common.parse_config(...
   'ToolVersion', '2019.1', ...
   'Board', 'zcu102', ...
```

```
    'Design', 'visionzynq_base', ...
   'ColorSpace', 'RGB' ...
);
% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.BoardName = ZynqVideoPSP.ZCU102Hdmicam.BoardName();
hRD.ReferenceDesignName = 'HDMI RGB with DL Processor';
% Tool information
hRD.SupportedToolVersion = {'2019.1'}
...
```

Verify the Reference Design

After creating the reference design, use the HDL Coder™ IP core generation workflow to generate the bitstream and program the ZCU102 board. You can then use MATLAB® and a `dlhdl.Workflow` object to verify the deep learning processor IP core or you can use the HDL Coder™ workflow to prototype the entire system. To verify the reference design, see “Run a Deep Learning Network on FPGA with Live Camera Input” on page 10-63.

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:

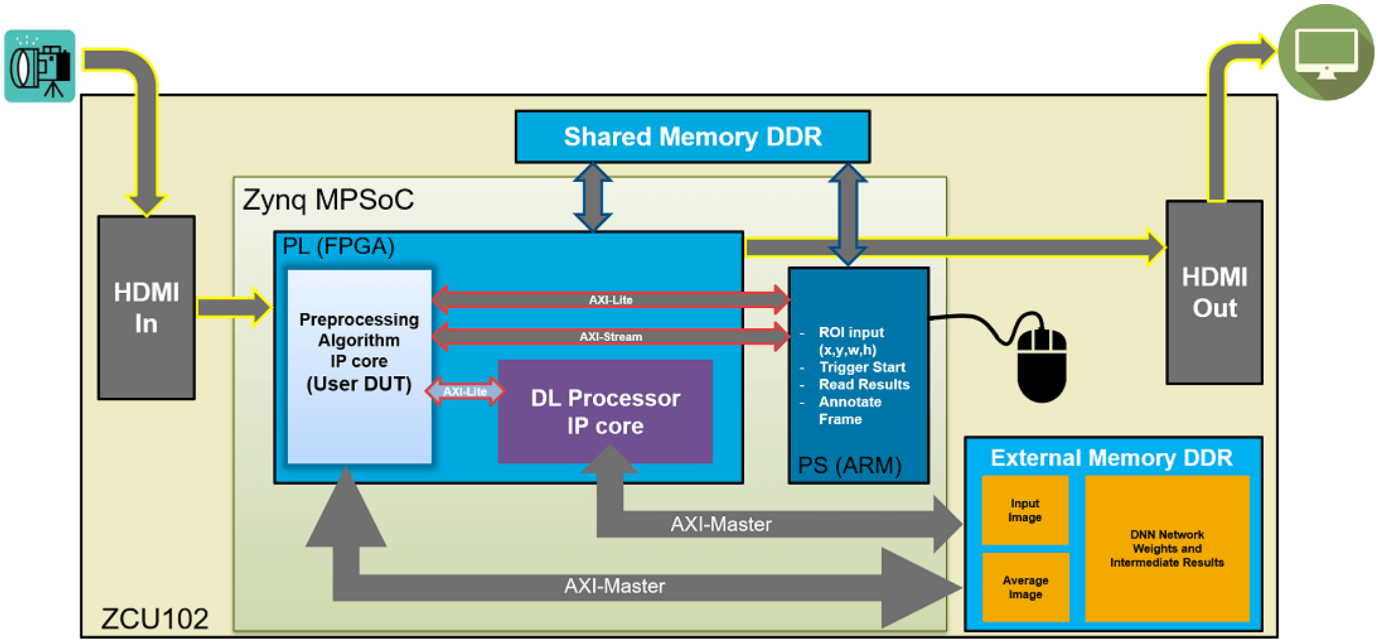
- Deep Learning HDL Toolbox™
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- Deep Learning Toolbox™
- HDL Coder™
- Simulink™

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.



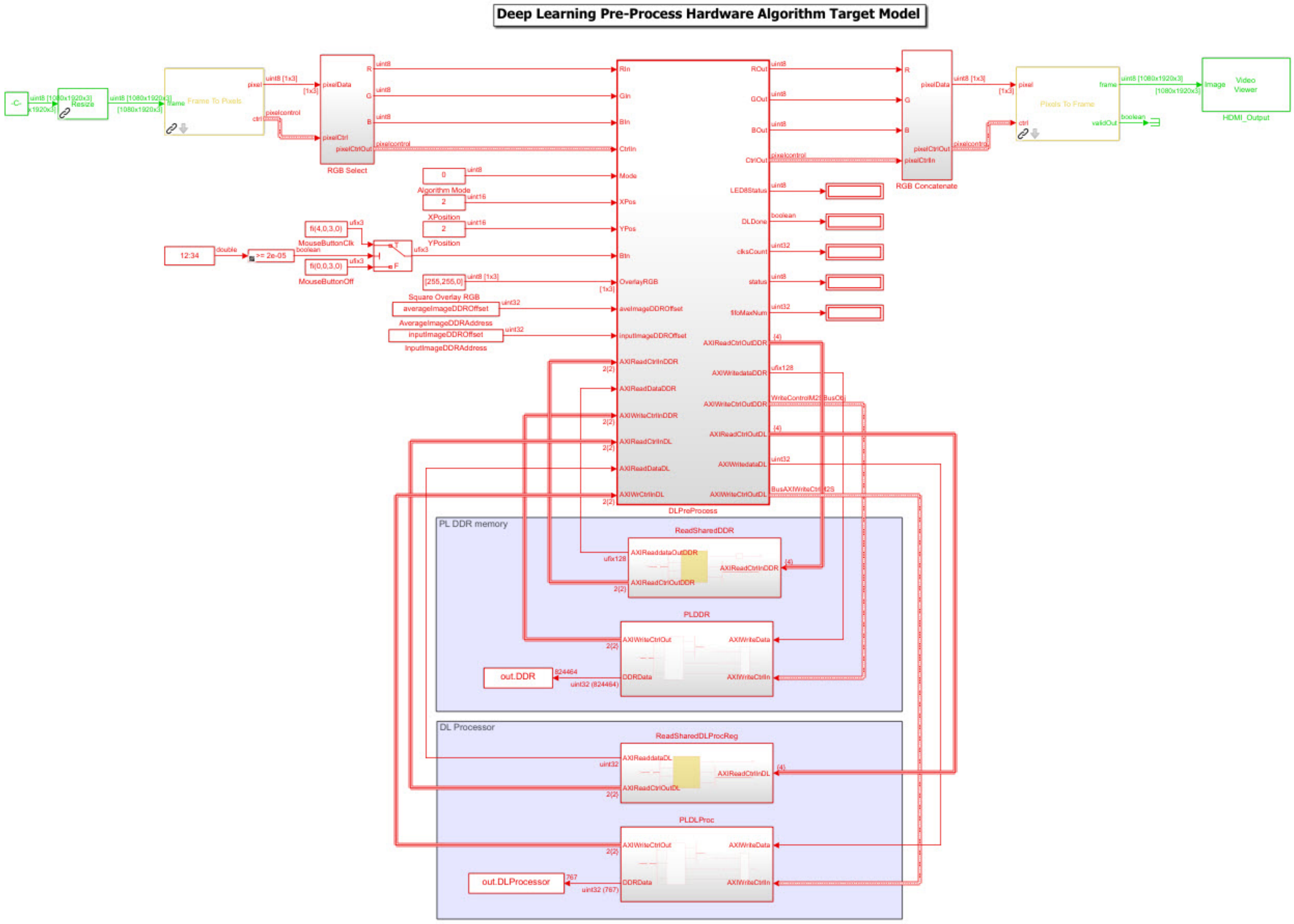
The objective of this system is to receive the live camera input through the HDMI input of the FMC daughter card on the ZCU102 board. You design the preprocessing logic in Simulink® to select and resize the region of interest (ROI). You then transmit the processed image frame to the deep learning processor 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.



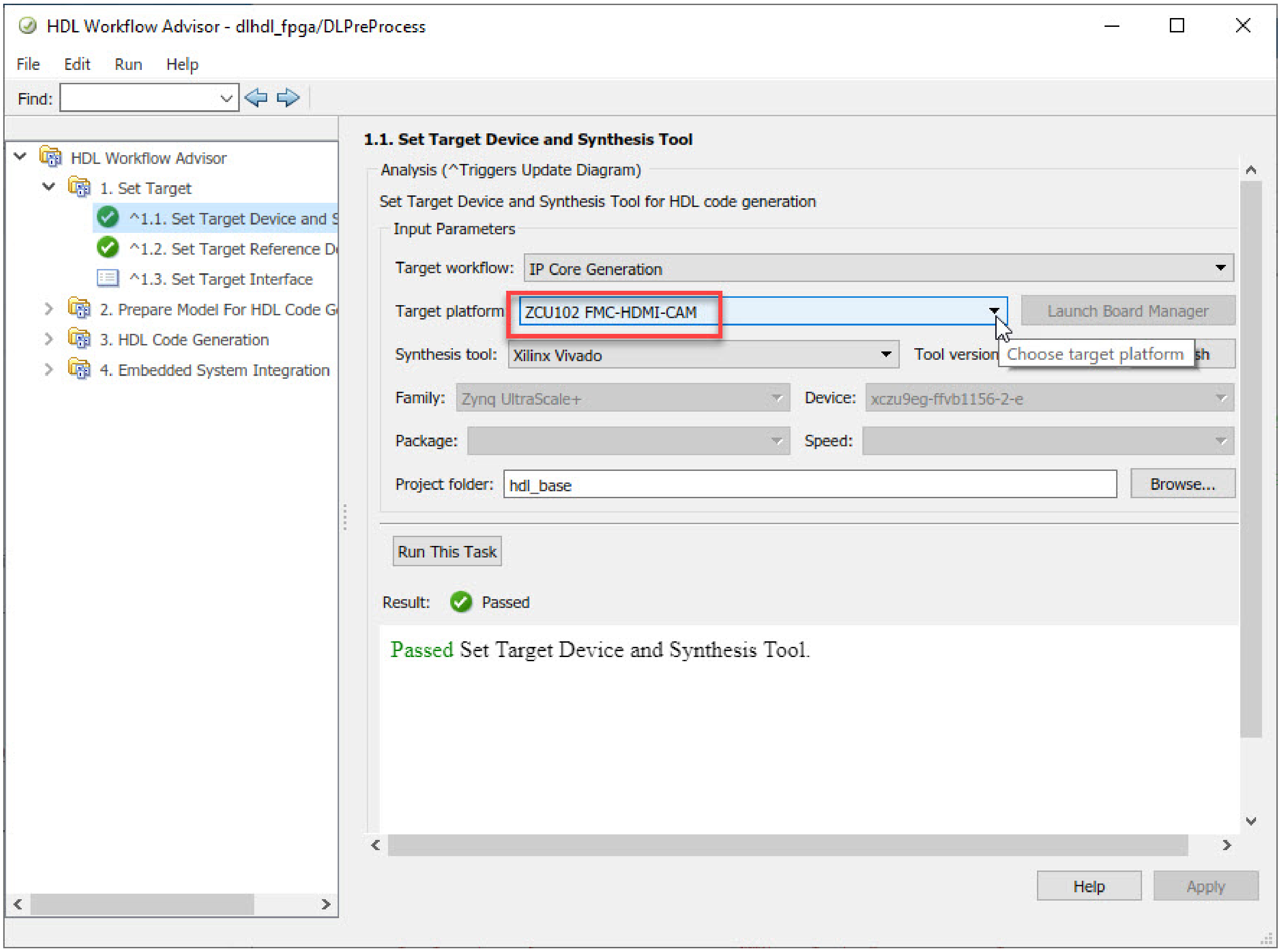
Generate Preprocessing Logic HDL IP Core

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 10-58.

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**.



HDL Workflow Advisor - dlhdl_fpga/DLPreProcess

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: IP Core Generation

Target platform: ZCU102 FMC-HDMI-CAM

Synthesis tool: Xilinx Vivado

Family: Zynq UltraScale+

Device: xczu9eg-ffvb1156-2-e

Package:

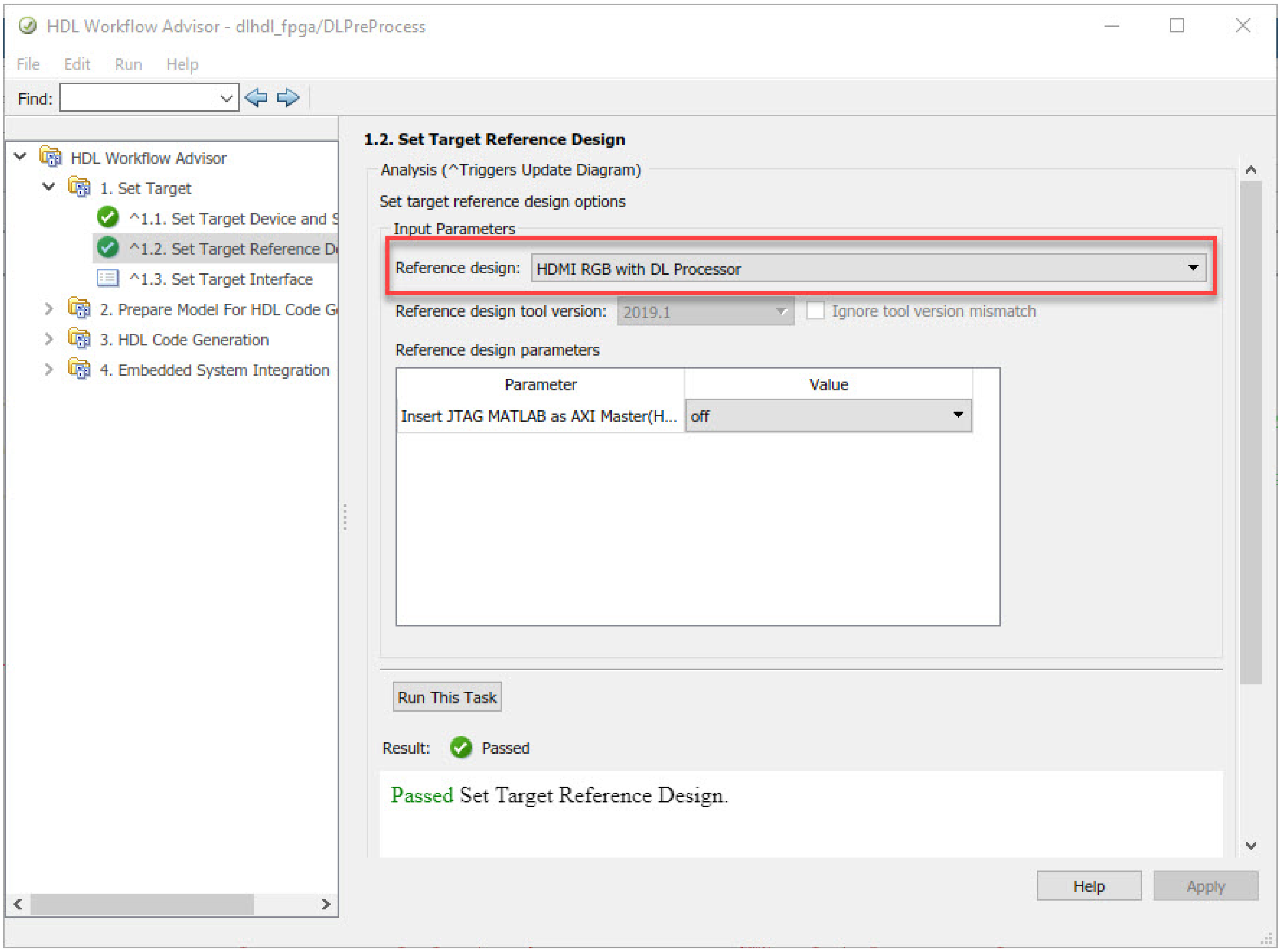
Speed:

Project folder: hdl_base

Result: Passed

Passed Set Target Device and Synthesis Tool.

In Task 1.2, **HDMI RGB with DL Processor** is selected for **Reference Design**.



HDL Workflow Advisor - dlhdl_fpga/DLPreProcess

File
Edit
Run
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and S
^1.2. Set Target Reference D
^1.3. Set Target Interface
2. Prepare Model For HDL Code G
3. HDL Code Generation
4. Embedded System Integration

1.2. Set Target Reference Design

Analysis (^Triggers Update Diagram)

Set target reference design options

Input Parameters

Reference design: HDMI RGB with DL Processor

Reference design tool version: 2019.1 □ Ignore tool version mismatch

Reference design parameters

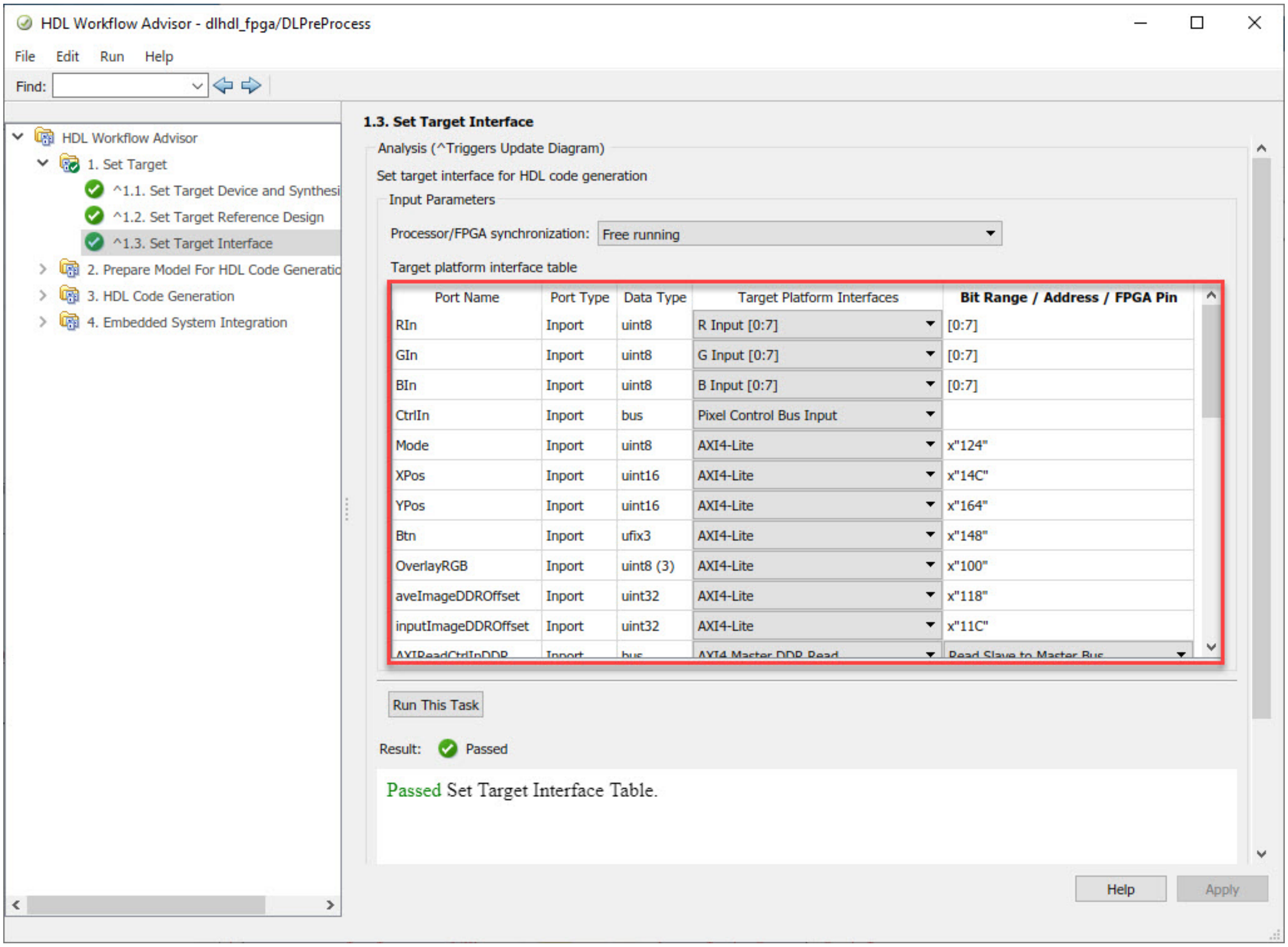
Parameter	Value
Insert JTAG MATLAB as AXI Master(H...	off

Run This Task

Result: ✓ Passed

Passed Set Target 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.



HDL Workflow Advisor - dlhdl_fpga/DLPreProcess

File
Edit
Run
Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesi
^1.2. Set Target Reference Design
^1.3. Set Target Interface
2. Prepare Model For HDL Code Generatio
3. HDL Code Generation
4. Embedded System Integration

1.3. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Processor/FPGA synchronization: Free running

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
RIn	Inport	uint8	R Input [0:7]	[0:7]
GIn	Inport	uint8	G Input [0:7]	[0:7]
BIn	Inport	uint8	B Input [0:7]	[0:7]
CtrlIn	Inport	bus	Pixel Control Bus Input	
Mode	Inport	uint8	AXI4-Lite	x"124"
XPos	Inport	uint16	AXI4-Lite	x"14C"
YPos	Inport	uint16	AXI4-Lite	x"164"
Btn	Inport	ufix3	AXI4-Lite	x"148"
OverlayRGB	Inport	uint8 (3)	AXI4-Lite	x"100"
aveImageDDROffset	Inport	uint32	AXI4-Lite	x"118"
inputImageDDROffset	Inport	uint32	AXI4-Lite	x"11C"
AXIReadCtrlInDDR	Inport	bus	AXI4 Master DDR Read	Read Slave to Master Bus

Result: Passed

Passed Set Target Interface Table.

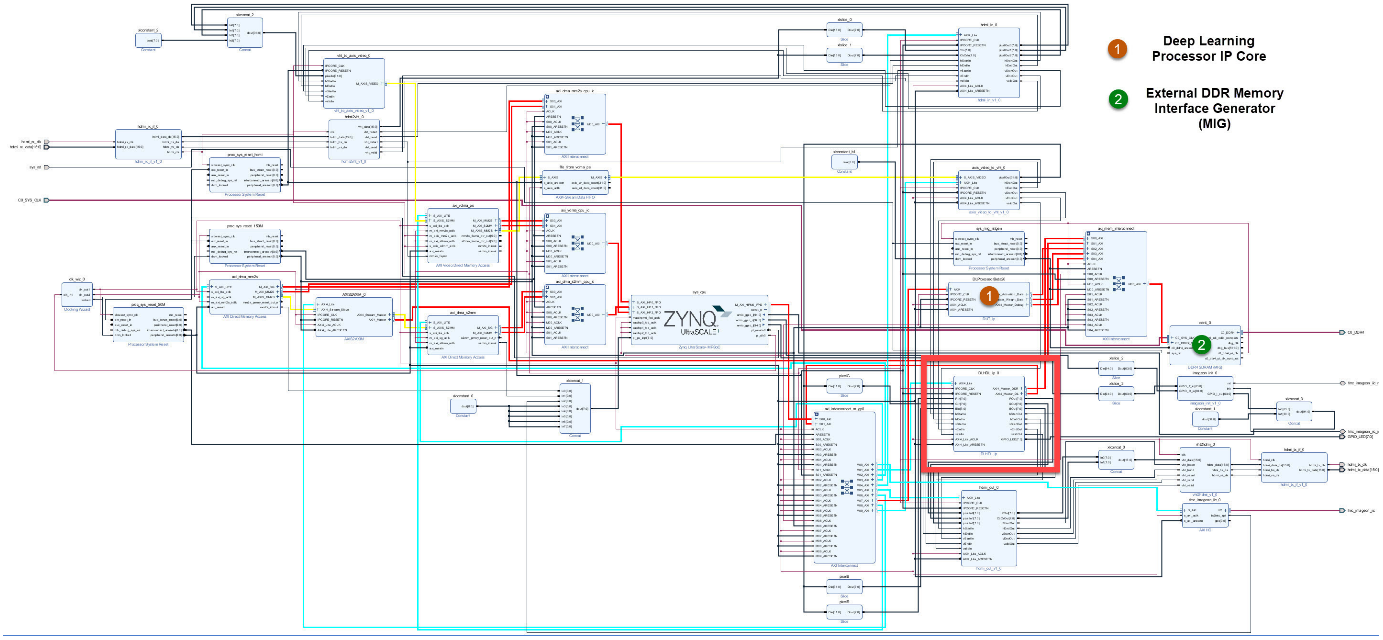
2. Right-click Task 3.2, **Generate RTL Code and IP Core**, and then select **Run to Selected Task**. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

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 10-58.

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.


1
Deep Learning Processor IP Core
2
External DDR Memory Interface Generator (MIG)
ZYNQ
UltraSCALE+

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.

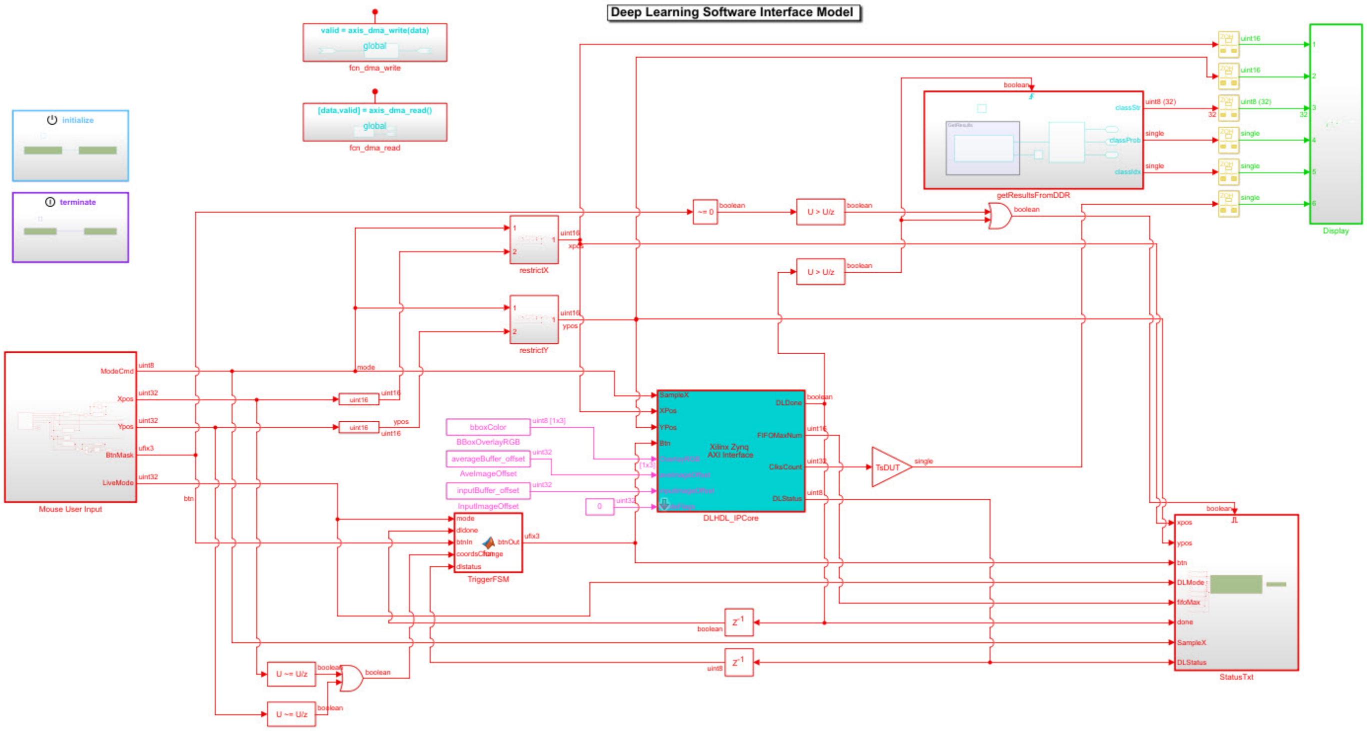
Deploy and Validate the Integrated Reference Design

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.

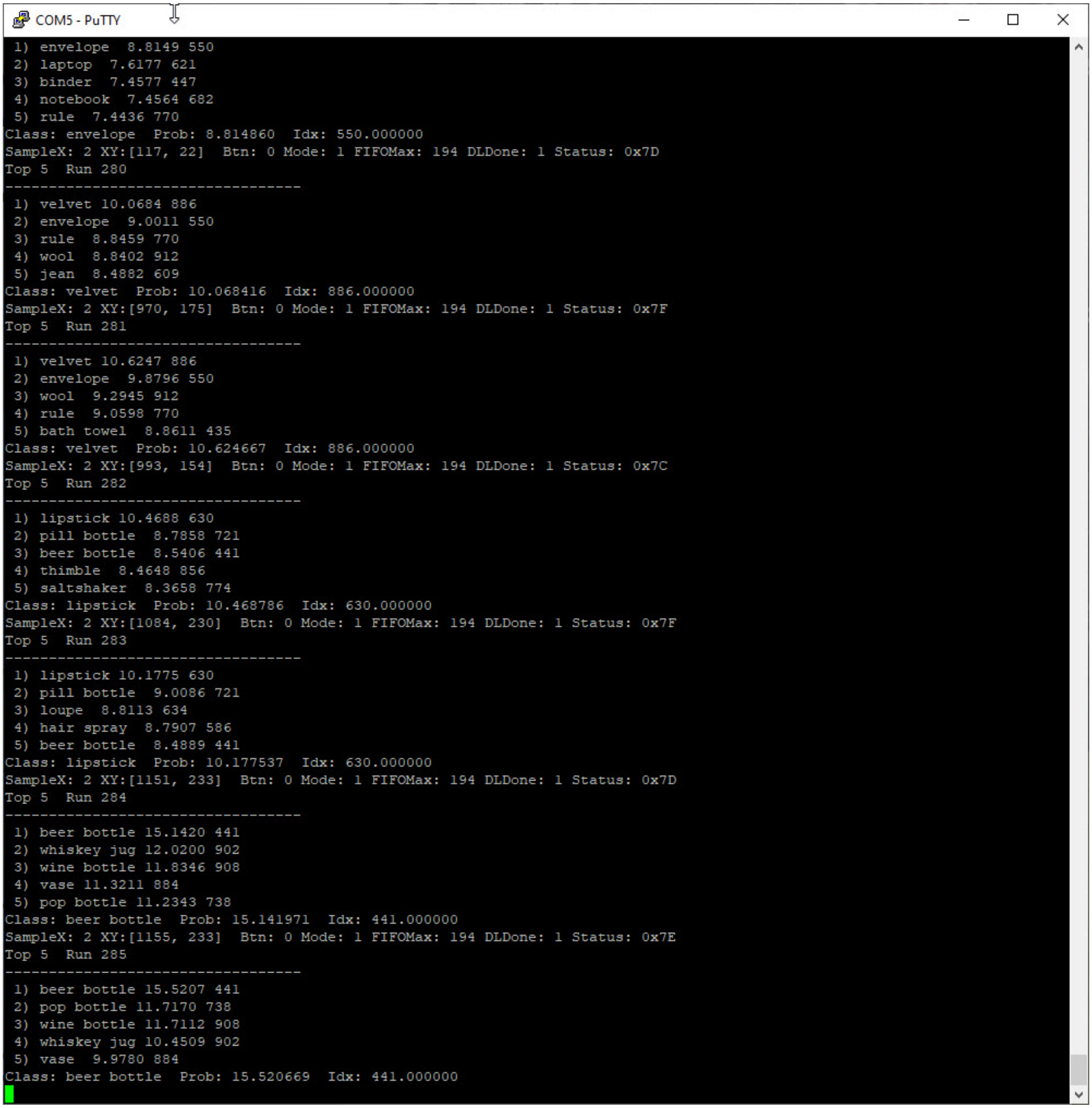
You can then use `fpga` object to create a connection from MATLAB to the ZCU102 board and read the contents of the external DDR memory into MATLAB for validation. to use the `fpga` object, see “Create Software Interface Script to Control and Rapidly Prototype HDL IP Core” (HDL Coder).



2. The generated deep learning processor IP core has Ethernet and JTAG interfaces for communications in the generated bitstream. You can individually validate the deep learning processor IP core by using the `dlhdl.Workflow` object.

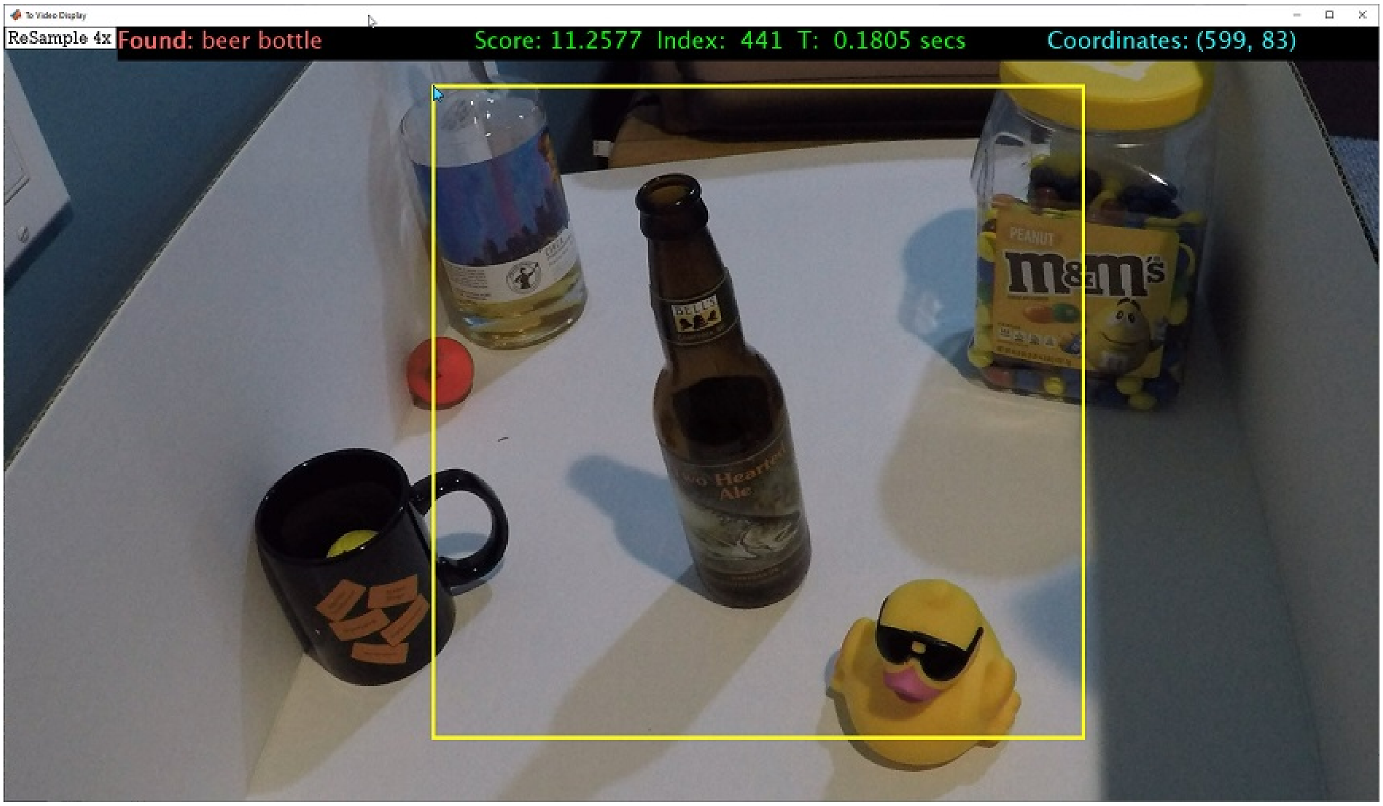
3. After you individually validate the preprocessing logic IP core and the deep learning processor IP core, you can prototype the entire integrated system on the FPGA board. Using Simulink External mode, instruct the FPGA preprocessing logic to send a processed input image frame to the DDR buffer, instruct the deep learning processor IP core to read from the same DDR buffer, and execute the prediction.

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



```
 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.



ReSample 4x Found: beer bottle Score: 11.2577 Index: 441 T: 0.1805 secs Coordinates: (599, 83)

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.laye
          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "24.0 MB"
    "SystemBufferOffset"        "0x03000000"     "28.0 MB"
    "InstructionDataOffset"     "0x04c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x05000000"     "4.0 MB"
    "EndOffset"                 "0x05400000"     "Total: 84.0 MB"

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

Program Bitstream onto FPGA and Download Network Weights

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

```
hW.deploy
```

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

Load Example Image

Load and display an image to use as an input image to the series network.

```
I = imread('daisy.jpg');
imshow(I)
```

Run the Prediction

Execute the predict function of the `dlhdl.Workflow` object.

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

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

              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------      --
Network                    2813005                  0.01279                       1            28
    conv_module            2813005                  0.01279
        conv1              2224168                  0.01011
        max_pooling2d_1     588864                  0.00268
 * The clock frequency of the DL processor is: 220MHz
```

The result data is returned as a 3-D array, with the third dimension indexing across the 64 feature images.

```
sz = size(P)
```

```
sz = 1×3

    56    56    64
```

To visualize all 64 features in a single image, the data is reshaped into four dimensions, which is appropriate input to the `imtile` function

```
R = reshape(P, [sz(1) sz(2) 1 sz(3)]);
sz = size(R)
```

```
sz = 1×4

    56    56     1    64
```

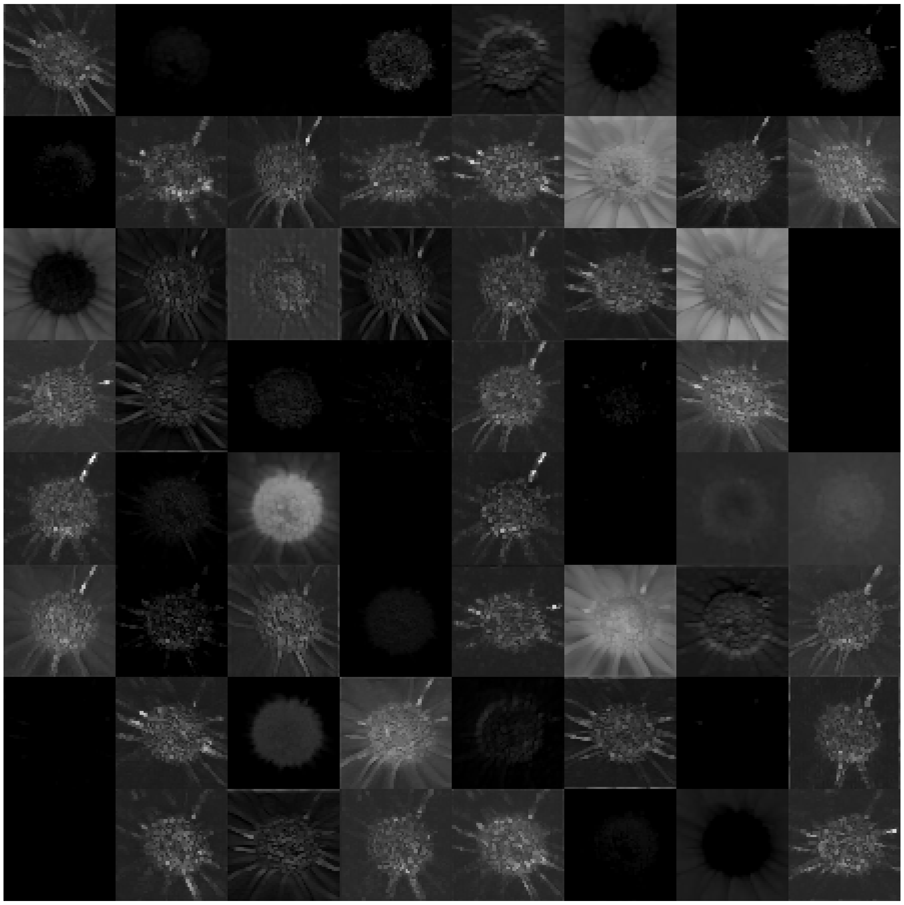
The third dimension in the input to `imtile` function represents the image color. Set the third dimension to size 1 because the activation signals in this example are scalars and do not include color. The fourth dimension indexes the channel.

The input to `imtile` is normalized using `mat2gray`. All values are scaled so that the minimum activation is 0 and the maximum activation is 1.

```
J = imtile(mat2gray(R), 'GridSize', [8 8]);
```

A grid size of 8-by-8 is selected because there are 64 features to display.

```
imshow(J)
```



The image shows activation data for each of the 64 features. Bright features indicate a strong activation.

The output from the convolutional layers only network differs from that of a network with convolution and fully connected layers. Convolution layers are used to reduce the input image size while maintaining features which are needed to get a good prediction. Convolution only layer networks are used to study feature extraction. Earlier convolution layers are used to extract low level features such as edges, colors, gradients and so on. Later convolution layers are used to extract high level features such as patterns, curves, lines and so on. These high level features can then be used to identify objects.

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 AlexNet 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

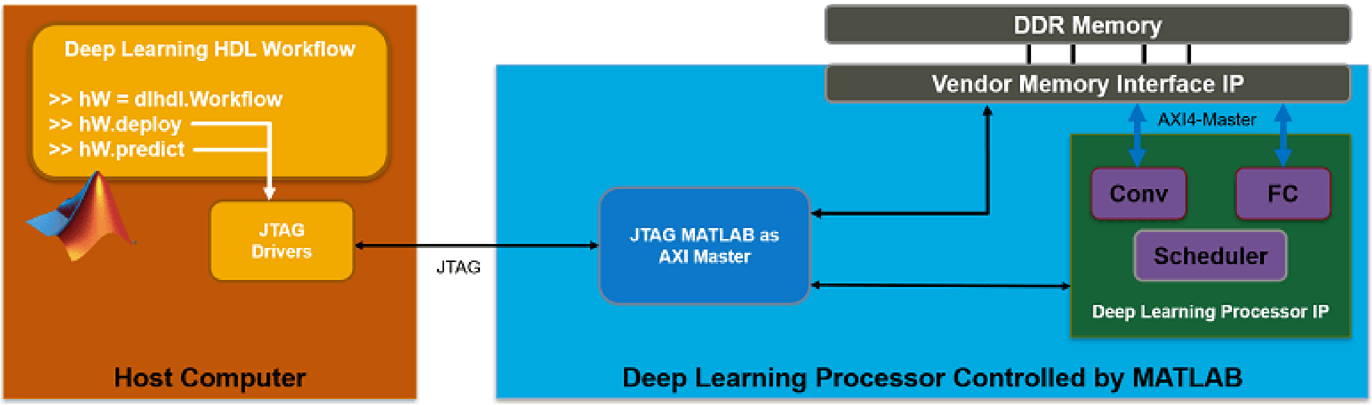
There are two hardware interfaces for establishing a connection between the host computer and FPGA board: JTAG and Ethernet.

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.

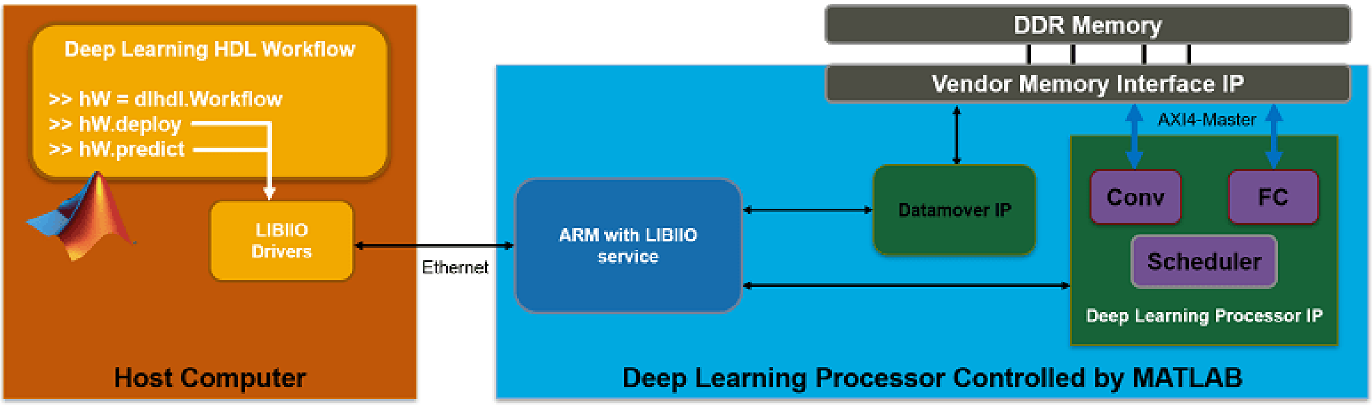

Deep Learning HDL Workflow
>> hW = dlhdl.Workflow
>> hW.deploy
>> hW.predict
JTAG Drivers
Host Computer
JTAG
DDR Memory
Vendor Memory Interface IP
AXI4-Master
JTAG MATLAB as AXI Master
Conv
FC
Scheduler
Deep Learning Processor IP
Deep Learning Processor Controlled by MATLAB

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.\



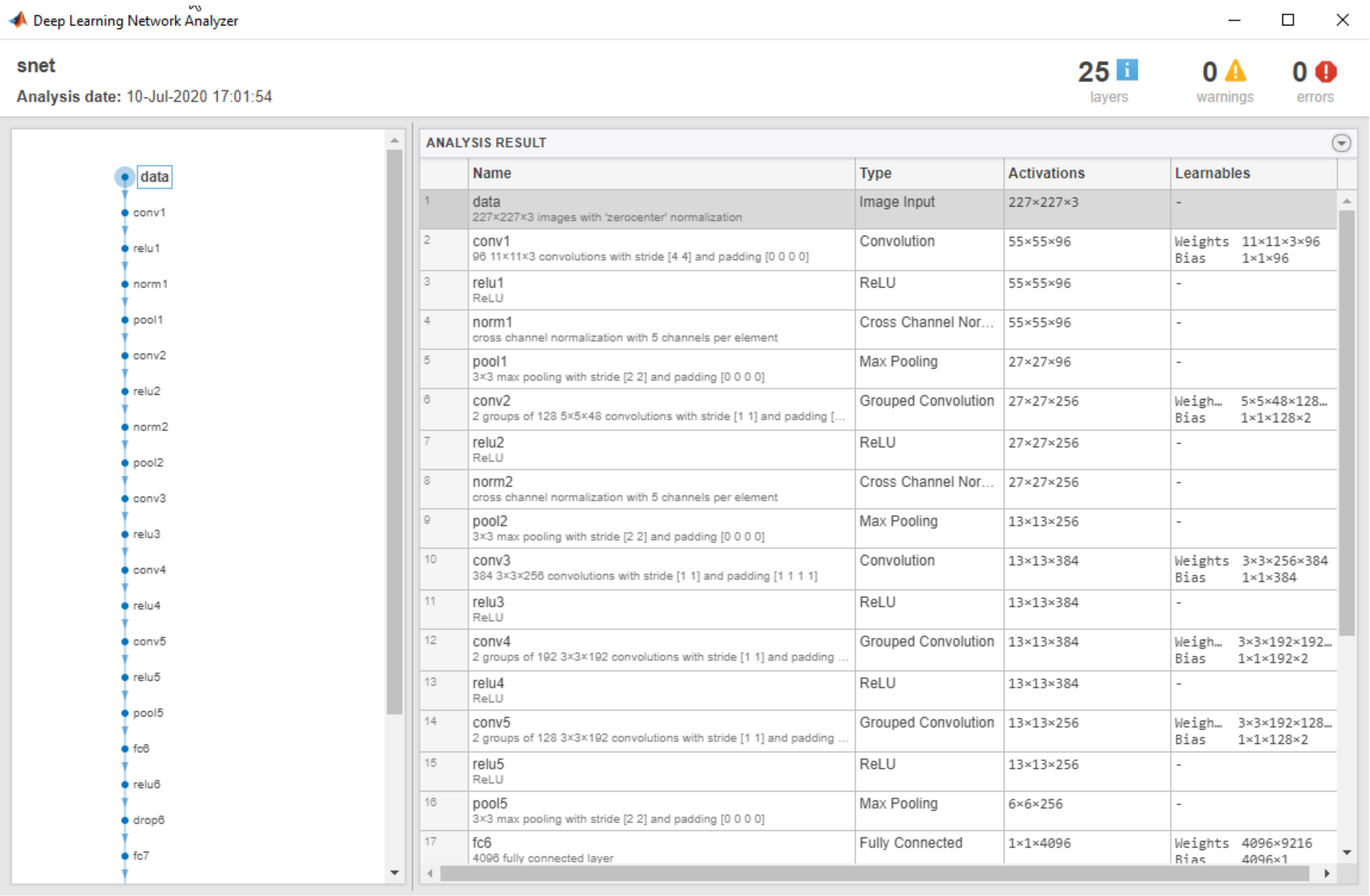
Load and Compile Deep Learning Network

This example uses the pretrained series network `alexnet`. This network is a larger network that has significant improvement in transfer time when deploying it to the FPGA by using Ethernet. To load `alexnet`, run the command:

```
snet = alexnet;
```

To view the layers of the network enter:

```
analyzeNetwork(snet);
% The saved network contains 25 layers including input, convolution, ReLU, cross channel normali
% max pool, fully connected, and the softmax output layers.
```



snet

Analysis date: 10-Jul-2020 17:01:54

25 layers 0 warnings 0 errors

ANALYSIS RESULT

	Name	Type	Activations	Learnables
1	data 227×227×3 images with 'zerocenter' normalization	Image Input	227×227×3	-
2	conv1 96 11×11×3 convolutions with stride [4 4] and padding [0 0 0 0]	Convolution	55×55×96	Weights 11×11×3×96 Bias 1×1×96
3	relu1 ReLU	ReLU	55×55×96	-
4	norm1 cross channel normalization with 5 channels per element	Cross Channel Nor...	55×55×96	-
5	pool1 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	27×27×96	-
6	conv2 2 groups of 128 5×5×48 convolutions with stride [1 1] and padding [...	Grouped Convolution	27×27×256	Weigh... 5×5×48×128... Bias 1×1×128×2
7	relu2 ReLU	ReLU	27×27×256	-
8	norm2 cross channel normalization with 5 channels per element	Cross Channel Nor...	27×27×256	-
9	pool2 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	13×13×256	-
10	conv3 384 3×3×256 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	13×13×384	Weights 3×3×256×384 Bias 1×1×384
11	relu3 ReLU	ReLU	13×13×384	-
12	conv4 2 groups of 192 3×3×192 convolutions with stride [1 1] and padding ...	Grouped Convolution	13×13×384	Weigh... 3×3×192×192... Bias 1×1×192×2
13	relu4 ReLU	ReLU	13×13×384	-
14	conv5 2 groups of 128 3×3×192 convolutions with stride [1 1] and padding ...	Grouped Convolution	13×13×256	Weigh... 3×3×192×128... Bias 1×1×128×2
15	relu5 ReLU	ReLU	13×13×256	-
16	pool5 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	6×6×256	-
17	fc6 4096 fully connected layer	Fully Connected	1×1×4096	Weights 4096×9216 Bias 4096×1

To deploy the deep learning network on the target FPGA board, create a `dlhdl.Workflow` object that has the pretrained network `snet` 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', snet, 'Bitstream', 'zcu102_single');
```

`Compile` the alexnet network for deployment to the FPGA.

```
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"     "224.0 MB"
    "EndOffset"                     "0x12c00000"     "Total: 300.0 MB"
```

The output displays the size of the compiled network, which is 300 MB. The entire 300 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 = 
  Target with properties:

       Vendor: 'Xilinx'
    Interface: JTAG
```

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

```
hTargetEthernet = 
  Target with properties:

       Vendor: 'Xilinx'
    Interface: Ethernet
    IPAddress: '192.168.1.100'
     Username: 'root'
         Port: 22
```

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;
hW.deploy;
```

```
### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to FC Processor.
### 8% finished, current time is 29-Jun-2020 16:33:14.
### 17% finished, current time is 29-Jun-2020 16:34:20.
### 25% finished, current time is 29-Jun-2020 16:35:38.
### 33% finished, current time is 29-Jun-2020 16:36:56.
### 42% finished, current time is 29-Jun-2020 16:38:13.
### 50% finished, current time is 29-Jun-2020 16:39:31.
### 58% finished, current time is 29-Jun-2020 16:40:48.
### 67% finished, current time is 29-Jun-2020 16:42:02.
### 75% finished, current time is 29-Jun-2020 16:43:10.
### 83% finished, current time is 29-Jun-2020 16:44:23.
### 92% finished, current time is 29-Jun-2020 16:45:39.
### FC Weights loaded. Current time is 29-Jun-2020 16:46:31
```

```
elapsedTimeJTAG = toc
```

```
elapsedTimeJTAG = 1.0614e+03
```

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;
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.
### 8% finished, current time is 29-Jun-2020 16:47:08.
### 17% finished, current time is 29-Jun-2020 16:47:08.
### 25% finished, current time is 29-Jun-2020 16:47:09.
### 33% finished, current time is 29-Jun-2020 16:47:10.
### 42% finished, current time is 29-Jun-2020 16:47:10.
### 50% finished, current time is 29-Jun-2020 16:47:11.
### 58% finished, current time is 29-Jun-2020 16:47:13.
### 67% finished, current time is 29-Jun-2020 16:47:13.
### 75% finished, current time is 29-Jun-2020 16:47:15.
### 83% finished, current time is 29-Jun-2020 16:47:16.
### 92% finished, current time is 29-Jun-2020 16:47:18.
### FC Weights loaded. Current time is 29-Jun-2020 16:47:18
```

```
elapsedTimeEthernet = toc
```

```
elapsedTimeEthernet = 47.5854
```

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 = 'zebra.JPEG';
inputImg = imresize(imread(imgFile), [227,227]);
imshow(inputImg)
```

```
prediction = hW.predict(single(inputImg));
```

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

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

```
result = 
'zebra'
```

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');
```

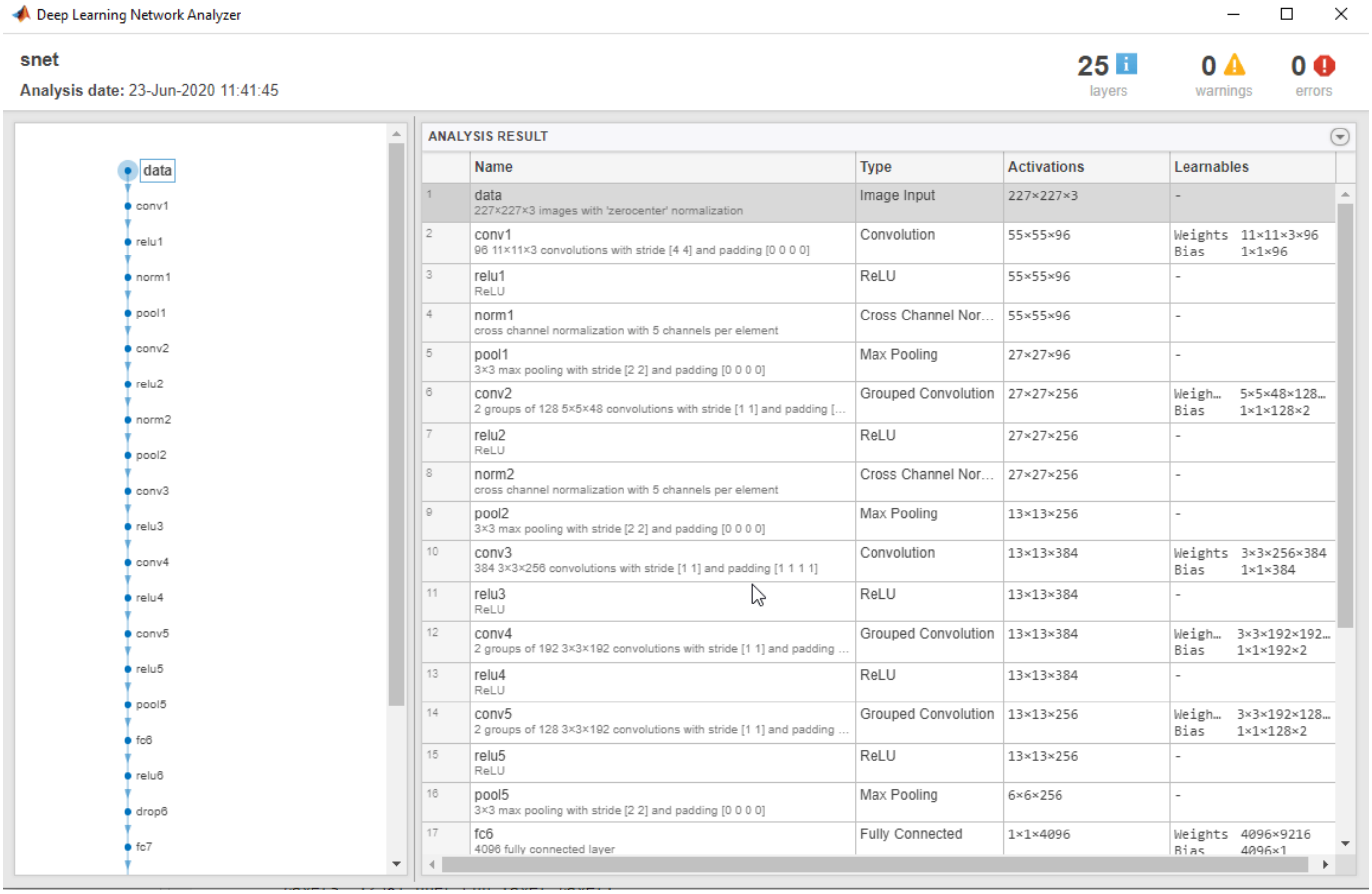
Load Pretrained Network

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)
```



snet

Analysis date: 23-Jun-2020 11:41:45

25 layers · 0 warnings · 0 errors

ANALYSIS RESULT

	Name	Type	Activations	Learnables
1	data 227×227×3 images with 'zerocenter' normalization	Image Input	227×227×3	-
2	conv1 96 11×11×3 convolutions with stride [4 4] and padding [0 0 0 0]	Convolution	55×55×96	Weights 11×11×3×96 Bias 1×1×96
3	relu1 ReLU	ReLU	55×55×96	-
4	norm1 cross channel normalization with 5 channels per element	Cross Channel Nor…	55×55×96	-
5	pool1 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	27×27×96	-
6	conv2 2 groups of 128 5×5×48 convolutions with stride [1 1] and padding […	Grouped Convolution	27×27×256	Weigh… 5×5×48×128… Bias 1×1×128×2
7	relu2 ReLU	ReLU	27×27×256	-
8	norm2 cross channel normalization with 5 channels per element	Cross Channel Nor…	27×27×256	-
9	pool2 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	13×13×256	-
10	conv3 384 3×3×256 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	13×13×384	Weights 3×3×256×384 Bias 1×1×384
11	relu3 ReLU	ReLU	13×13×384	-
12	conv4 2 groups of 192 3×3×192 convolutions with stride [1 1] and padding …	Grouped Convolution	13×13×384	Weigh… 3×3×192×192… Bias 1×1×192×2
13	relu4 ReLU	ReLU	13×13×384	-
14	conv5 2 groups of 128 3×3×192 convolutions with stride [1 1] and padding …	Grouped Convolution	13×13×256	Weigh… 3×3×192×128… Bias 1×1×128×2
15	relu5 ReLU	ReLU	13×13×256	-
16	pool5 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	6×6×256	-
17	fc6 4096 fully connected layer	Fully Connected	1×1×4096	Weights 4096×9216 Bias 4096×1

The first layer, the image input layer, requires input images of size 227-by-227-by-3, where 3 is the number of color channels.

```
inputSize = snet.Layers(1).InputSize
```

```
inputSize = 1×3

   227   227     3
```

Replace Final Layers

The last three layers of the pretrained network `net` are configured for 1000 classes. These three layers must be fine-tuned for the new classification problem. Extract all layers, except the last three, from the pretrained network.

```
layersTransfer = snet.Layers(1:end-3);
```

Transfer the layers to the new classification task by replacing the last three layers with a fully connected layer, a softmax layer, and a classification output layer. Specify the options of the new fully connected layer according to the new data. Set the fully connected layer to have the same size as the number of classes in the new data. To learn faster in the new layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the fully connected layer.

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

```
layers = [
    layersTransfer
    fullyConnectedLayer(numClasses,'WeightLearnRateFactor',20,'BiasLearnRateFactor',20)
    softmaxLayer
    classificationLayer];
```

Train Network

The network requires input images of size 227-by-227-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

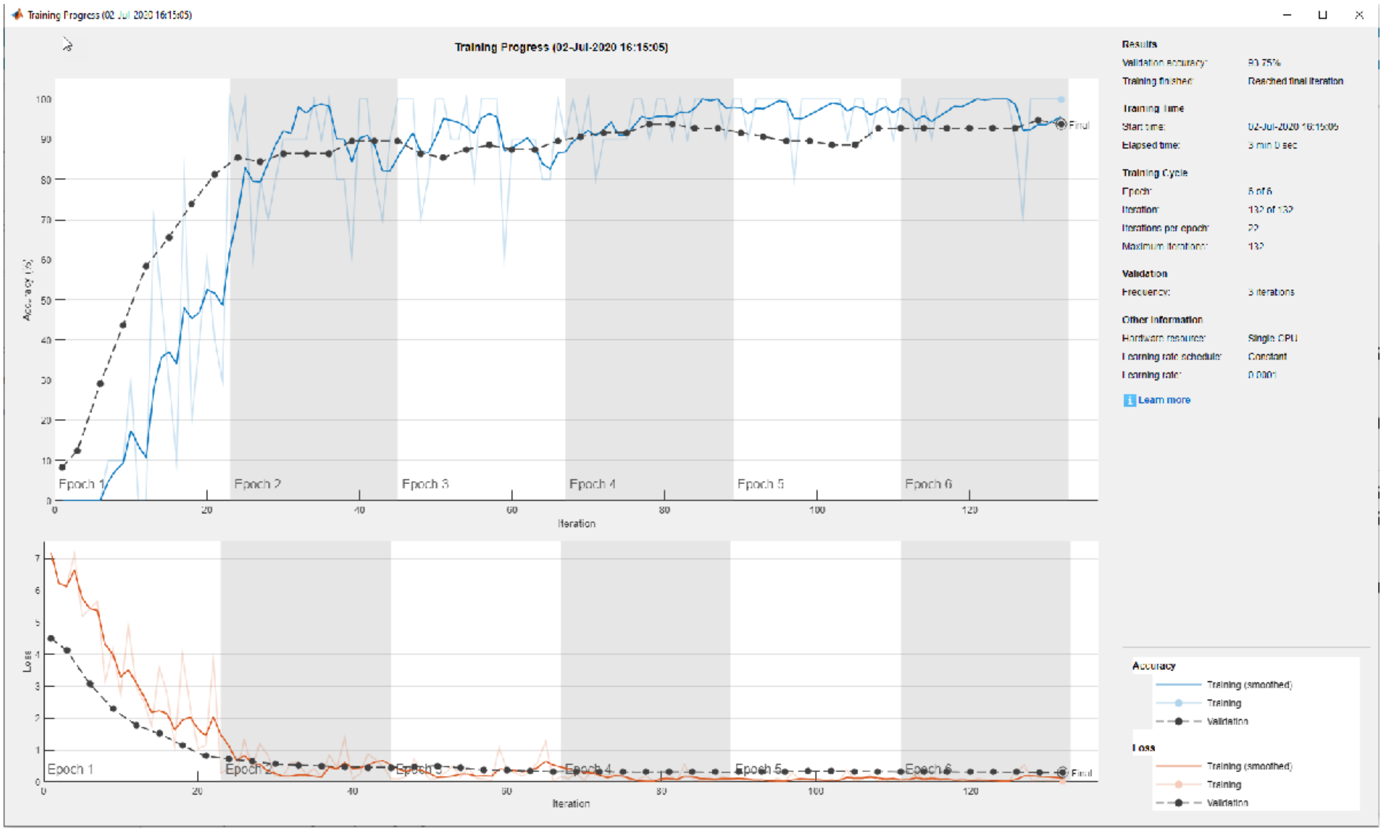
```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate factors for the fully connected layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size and validation data. The software validates the network every `ValidationFrequency` iterations during training.

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

Train the network that consists of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 6.1, 6.3, or higher). Otherwise, it uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

```
netTransfer = trainNetwork(augimdsTrain,layers,options);
```



Results
Validation accuracy: 93.75%
Training finished: Reached final iteration

Training Time
Start time: 02-Jul-2020 16:15:05
Elapsed time: 3 min 0 sec

Training Cycle
Epoch: 6 of 6
Iteration: 132 of 132
Iterations per epoch: 22
Maximum iterations: 132

Validation
Frequency: 3 iterations

Other Information
Hardware resource: Single CPU
Learning rate schedule: Constant
Learning rate: 0.0001

Learn more

Vehicle Detection Using YOLO v2 Deployed to FPGA

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.

Train and deploy a you look only once (YOLO) v2 object detector by using the `dlhdl.Workflow` object.

Load Data Set

This example uses a small vehicle data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, used with permission and available at the Caltech Computational Vision website, created by Pietro Perona. 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. Extract 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 data set into training and test sets. Select 60% of the data for training and use 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 the image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
testData = combine(imdsTest,bldsTest);
```

Create 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 AlexNet for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-18 can also be used 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. The `yolov2Layers` funcvtion 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-by-224-by-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;
```

The training images used in this example are larger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use the `estimateAnchorBoxes` function 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 the `transform` function to preprocess the training data and then define the number of anchor boxes and estimate the anchor boxes. Resize the training data to the input image size of the network by using the supporting function `yolo_preprocessData`, attached to this example.

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).

```
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
```

Use the `alexnet` function to load a pretrained model.

```
featureExtractionNetwork = alexnet
```

```
featureExtractionNetwork = 
  SeriesNetwork with properties:

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

Select `'relu5'` as the feature extraction layer to replace the layers after `'relu5'` 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 farther down the network encode stronger image features at the cost of spatial resolution.

```
featureLayer = 'relu5';
```

Create the YOLO v2 object detection network.

```
lgraph = yolov2Layers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer);
```

You can visualize the network by using the `analyzeNetwork` function or the Deep Network Designer from Deep Learning Toolbox™.

If you require more control over the YOLO v2 network architecture, use the 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 the `transform` function 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 is representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@yolo_augmentData);
```

Preprocess Training Data and Train YOLO v2 Object Detector

Preprocess the augmented training data and the validation data to prepare for training.

```
preprocessedTrainingData = transform(augmentedTrainingData,@(data)yolo_preprocessData(data,inputS
```

Use the `trainingOptions` function to specify network training options. Set `'ValidationData'` to the preprocessed validation data. Set `'CheckpointPath'` to a temporary location. Thse settings 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 the `trainYOLOv2ObjectDetector` function to train the YOLO v2 object detector.

```
[detector,info] = trainYOLOv2ObjectDetector(preprocessedTrainingData,lgraph,options);
```

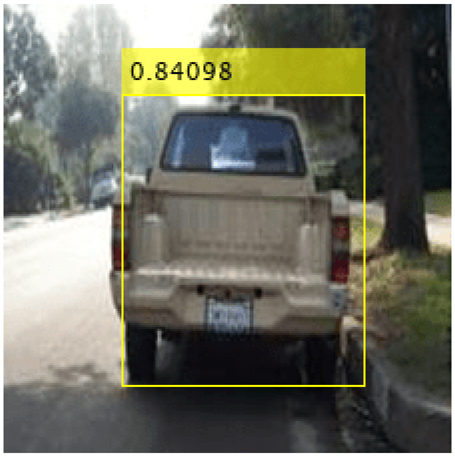
```
*************************************************************************
Training a YOLO v2 Object Detector for the following object classes:

* vehicle

Training on single CPU.
Initializing input data normalization.
|========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |     RMSE     |     Loss     |      Rate       |
|========================================================================================|
|       1 |           1 |       00:00:02 |         7.23 |         52.3 |          0.0010 |
|       5 |          50 |       00:00:43 |         0.99 |          1.0 |          0.0010 |
|      10 |         100 |       00:01:24 |         0.77 |          0.6 |          0.0010 |
|      14 |         150 |       00:02:03 |         0.64 |          0.4 |          0.0010 |
|      19 |         200 |       00:02:41 |         0.57 |          0.3 |          0.0010 |
|      20 |         220 |       00:02:55 |         0.58 |          0.3 |          0.0010 |
|========================================================================================|
Detector training complete.
*************************************************************************
```

As a quick test, run the detector on one test image. Make sure that 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)
```



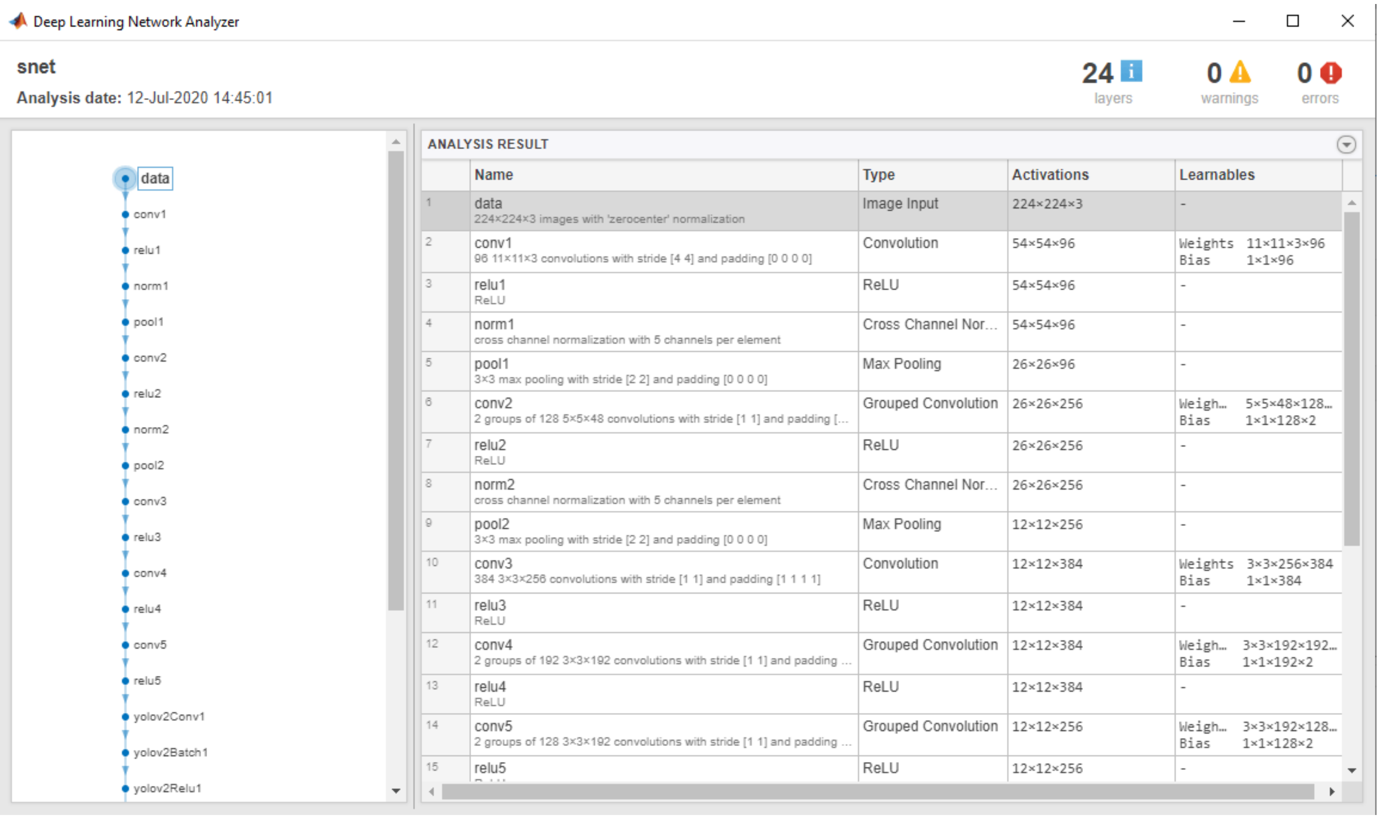
Load Pretrained Network

Load the pretrained network.

```
snet=detector.Network;
I_pre=yolo_pre_proc(I);
```

Use the `analyzeNetwork` function to obtain information about the network layers.

```
analyzeNetwork(snet)
```



snet

Analysis date: 12-Jul-2020 14:45:01

ANALYSIS RESULT

	Name	Type	Activations	Learnables
1	data 224×224×3 images with 'zerocenter' normalization	Image Input	224×224×3	-
2	conv1 96 11×11×3 convolutions with stride [4 4] and padding [0 0 0 0]	Convolution	54×54×96	Weights 11×11×3×96 Bias 1×1×96
3	relu1 ReLU	ReLU	54×54×96	-
4	norm1 cross channel normalization with 5 channels per element	Cross Channel Nor...	54×54×96	-
5	pool1 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	26×26×96	-
6	conv2 2 groups of 128 5×5×48 convolutions with stride [1 1] and padding [...	Grouped Convolution	26×26×256	Weigh… 5×5×48×128… Bias 1×1×128×2
7	relu2 ReLU	ReLU	26×26×256	-
8	norm2 cross channel normalization with 5 channels per element	Cross Channel Nor...	26×26×256	-
9	pool2 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	12×12×256	-
10	conv3 384 3×3×256 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	12×12×384	Weights 3×3×256×384 Bias 1×1×384
11	relu3 ReLU	ReLU	12×12×384	-
12	conv4 2 groups of 192 3×3×192 convolutions with stride [1 1] and padding ...	Grouped Convolution	12×12×384	Weigh… 3×3×192×192… Bias 1×1×192×2
13	relu4 ReLU	ReLU	12×12×384	-
14	conv5 2 groups of 128 3×3×192 convolutions with stride [1 1] and padding ...	Grouped Convolution	12×12×256	Weigh… 3×3×192×128… Bias 1×1×128×2
15	relu5	ReLU	12×12×256	-

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. Specify the network and the bitstream name. Specify the saved pretrained 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 the single data type.

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

```
hW = 
  Workflow with properties:

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

Compile YOLO v2 Object Detector

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

```
dn = hW.compile
```

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

     1   'data'                Image Input                   224×224×3 images with 'zerocenter' n
     2   'conv1'               Convolution                   96 11×11×3 convolutions with stride
     3   'relu1'               ReLU                          ReLU
     4   'norm1'               Cross Channel Normalization   cross channel normalization with 5 c
     5   'pool1'               Max Pooling                   3×3 max pooling with stride [2  2] a
     6   'conv2'               Grouped Convolution           2 groups of 128 5×5×48 convolutions
     7   'relu2'               ReLU                          ReLU
     8   'norm2'               Cross Channel Normalization   cross channel normalization with 5 c
     9   'pool2'               Max Pooling                   3×3 max pooling with stride [2  2] a
    10   'conv3'               Convolution                   384 3×3×256 convolutions with stride
    11   'relu3'               ReLU                          ReLU
    12   'conv4'               Grouped Convolution           2 groups of 192 3×3×192 convolutions
    13   'relu4'               ReLU                          ReLU
    14   'conv5'               Grouped Convolution           2 groups of 128 3×3×192 convolutions
    15   'relu5'               ReLU                          ReLU
    16   'yolov2Conv1'         Convolution                   256 3×3×256 convolutions with stride
    17   'yolov2Batch1'        Batch Normalization           Batch normalization with 256 channel
    18   'yolov2Relu1'         ReLU                          ReLU
    19   'yolov2Conv2'         Convolution                   256 3×3×256 convolutions with stride
    20   'yolov2Batch2'        Batch Normalization           Batch normalization with 256 channel
    21   'yolov2Relu2'         ReLU                          ReLU
    22   'yolov2ClassConv'     Convolution                   42 1×1×256 convolutions with stride
    23   'yolov2Transform'     YOLO v2 Transform Layer.      YOLO v2 Transform Layer with 7 ancho
    24   'yolov2OutputLayer'   YOLO v2 Output                YOLO v2 Output with 7 anchors.

### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
2 Memory Regions created.

Skipping: data
Compiling leg: conv1>>yolov2ClassConv ...
Compiling leg: conv1>>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"     "0.0 MB"
    "SystemBufferOffset"        "0x01c00000"     "28.0 MB"
    "InstructionDataOffset"     "0x03800000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03c00000"     "16.0 MB"
    "EndOffset"                 "0x04c00000"     "Total: 76.0 MB"

### Network compilation complete.

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

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 checks for the Xilinx Vivado tool and the supported tool version. It then starts programming the FPGA device by using the bitstream and displays progress messages and the time it takes to deploy the network.

```
hW.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 20-Dec-2020 15:26:28
```

Load Example Image and Run Prediction

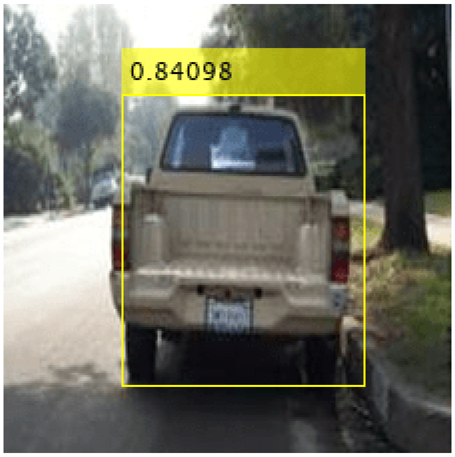
Execute the predict function on the `dlhdl.Workflow` object and display the result.

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

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


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    8615567                  0.03916                       1           86
    conv1                  1357049                  0.00617
    norm1                   569406                  0.00259
    pool1                   205869                  0.00094
    conv2                  2207222                  0.01003
    norm2                   360973                  0.00164
    pool2                   197444                  0.00090
    conv3                   976419                  0.00444
    conv4                   761188                  0.00346
    conv5                   521782                  0.00237
    yolov2Conv1             660213                  0.00300
    yolov2Conv2             661162                  0.00301
    yolov2ClassConv         136816                  0.00062
 * 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)
```



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

Estimate LogoNet Performance

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

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

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


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        -
Network                   39926006                  0.18148                       1           399
    ____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 Ki
                             TargetFrequency: 300
                               SynthesisTool: 'Xilinx Vivado'
                             ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
                     SynthesisToolChipFamily: 'Zynq UltraScale+'
                     SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                    SynthesisToolPackageName: ''
                     SynthesisToolSpeedValue: ''
```

Quantize LogoNet Series Network

To quantize the LogoNet network, enter:

```
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 imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                   14306694                  0.04769                       1          143
    ____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.b
% 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 Alexnet 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 HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- MATLAB Coder Interface for Deep Learning Libraries.

Load Pretrained SeriesNetwork

To load the pretrained series network AlexNet, enter:

```
snet = alexnet;
```

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

```
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
```

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.

```
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. 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
```

```
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, 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.

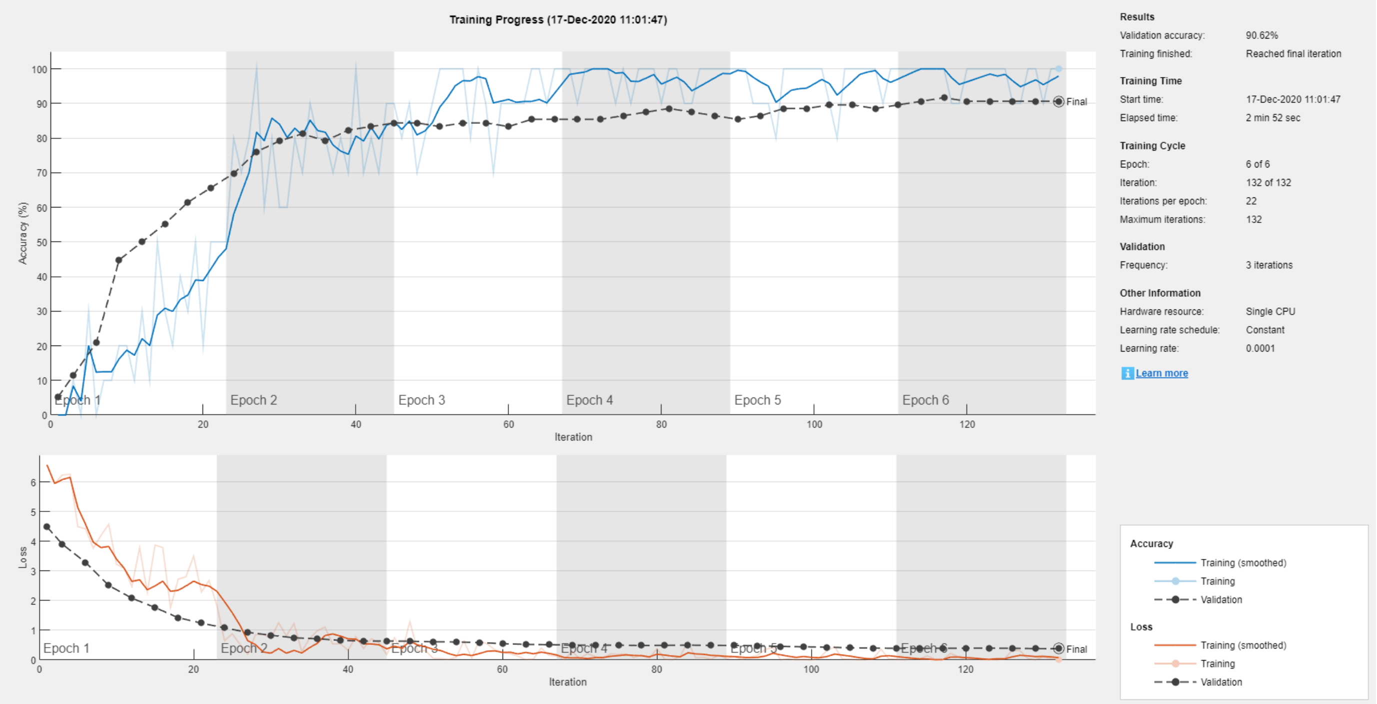
```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

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,layers,options);
```



Training Progress (17-Dec-2020 11:01:47)

Results
Validation accuracy: 90.62%
Training finished: Reached final iteration

Training Time
Start time: 17-Dec-2020 11:01:47
Elapsed time: 2 min 52 sec

Training Cycle
Epoch: 6 of 6
Iteration: 132 of 132
Iterations per epoch: 22
Maximum iterations: 132

Validation
Frequency: 3 iterations

Other Information
Hardware resource: Single CPU
Learning rate schedule: Constant
Learning rate: 0.0001

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)
```

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.b
```

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 Series Network

To compile the quantized AlexNet 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_int8 ...
### The network includes the following layers:

     1   'data'          Image Input                   227×227×3 images with 'zerocenter' normali
     2   'conv1'         Convolution                   96 11×11×3 convolutions with stride [4  4
     3   'relu1'         ReLU                          ReLU
     4   'norm1'         Cross Channel Normalization   cross channel normalization with 5 channel
     5   'pool1'         Max Pooling                   3×3 max pooling with stride [2  2] and pad
     6   'conv2'         Grouped Convolution           2 groups of 128 5×5×48 convolutions with s
     7   'relu2'         ReLU                          ReLU
     8   'norm2'         Cross Channel Normalization   cross channel normalization with 5 channel
     9   'pool2'         Max Pooling                   3×3 max pooling with stride [2  2] and pad
    10   'conv3'         Convolution                   384 3×3×256 convolutions with stride [1  1
    11   'relu3'         ReLU                          ReLU
    12   'conv4'         Grouped Convolution           2 groups of 192 3×3×192 convolutions with
    13   'relu4'         ReLU                          ReLU
    14   'conv5'         Grouped Convolution           2 groups of 128 3×3×192 convolutions with
    15   'relu5'         ReLU                          ReLU
    16   'pool5'         Max Pooling                   3×3 max pooling with stride [2  2] and pad
    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   'fc'            Fully Connected                32 fully connected layer
    24   'softmax'       Softmax                        softmax
    25   'classoutput'   Classification Output          crossentropyex with 'adidas' and 31 other

3 Memory Regions created.

Skipping: data
Compiling leg: conv1>>pool5 ...
Compiling leg: conv1>>pool5 ... complete.
Compiling leg: fc6>>fc ...
Compiling leg: fc6>>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"     "48.0 MB"        
    "OutputResultOffset"        "0x03000000"     "4.0 MB"         
    "SchedulerDataOffset"       "0x03400000"     "4.0 MB"         
    "SystemBufferOffset"        "0x03800000"     "28.0 MB"        
    "InstructionDataOffset"     "0x05400000"     "4.0 MB"         
    "ConvWeightDataOffset"      "0x05800000"     "8.0 MB"         
    "FCWeightDataOffset"        "0x06000000"     "56.0 MB"        
    "EndOffset"                 "0x09800000"     "Total: 152.0 MB"

### Network compilation complete.

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

Program Bitstream onto FPGA and Download Network Weights

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

```
hW.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 17-Dec-2020 11:06:56
### Loading weights to FC Processor.
### 33% finished, current time is 17-Dec-2020 11:06:57.
### 67% finished, current time is 17-Dec-2020 11:06:59.
### FC Weights loaded. Current time is 17-Dec-2020 11:06:59
```

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));
    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 activations.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------            -------------              ---------        -
Network                    9088267                  0.04131                       1           90
    conv1                   713071                  0.00324
    norm1                   460546                  0.00209
    pool1                    88791                  0.00040
    conv2                   911059                  0.00414
    norm2                   270230                  0.00123
    pool2                    92782                  0.00042
    conv3                   297066                  0.00135
    conv4                   238155                  0.00108
    conv5                   166248                  0.00076
    pool5                    19576                  0.00009
    fc6                    3955696                  0.01798
    fc7                    1757863                  0.00799
    fc                      117059                  0.00053
 * The clock frequency of the DL processor is: 220MHz

ans =
'ford'

label =
'ford'

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


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    9088122                  0.04131                       1           90
    conv1                   713003                  0.00324
    norm1                   460513                  0.00209
    pool1                    89083                  0.00040
    conv2                   910726                  0.00414
    norm2                   270238                  0.00123
    pool2                    92773                  0.00042
    conv3                   297151                  0.00135
    conv4                   238132                  0.00108
    conv5                   166415                  0.00076
    pool5                    19561                  0.00009
    fc6                    3955517                  0.01798
    fc7                    1757860                  0.00799
    fc                      117054                  0.00053
 * The clock frequency of the DL processor is: 220MHz

ans =
'bmw'

label =
'bmw'

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


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    9088305                  0.04131                       1           90
    conv1                   713031                  0.00324
    norm1                   460263                  0.00209
    pool1                    88948                  0.00040
    conv2                   911216                  0.00414
    norm2                   270247                  0.00123
    pool2                    92514                  0.00042
    conv3                   297124                  0.00135
    conv4                   238252                  0.00108
    conv5                   166320                  0.00076
    pool5                    19519                  0.00009
    fc6                    3955853                  0.01798
    fc7                    1757867                 0.00799
    fc                      117055                 0.00053
 * The clock frequency of the DL processor is: 220MHz

ans = 
'aldi'

label = 
'aldi'

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


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    9088168                 0.04131                       1           90
    conv1                   713087                 0.00324
    norm1                   460226                 0.00209
    pool1                    89136                 0.00041
    conv2                   910865                 0.00414
    norm2                   270243                 0.00123
    pool2                    92511                 0.00042
    conv3                   297117                 0.00135
    conv4                   238363                 0.00108
    conv5                   166485                 0.00076
    pool5                    19504                 0.00009
    fc6                    3955608                 0.01798
    fc7                    1757867                 0.00799
    fc                      117060                 0.00053
 * The clock frequency of the DL processor is: 220MHz

ans = 
'corona'

label = 
'corona'
```

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.

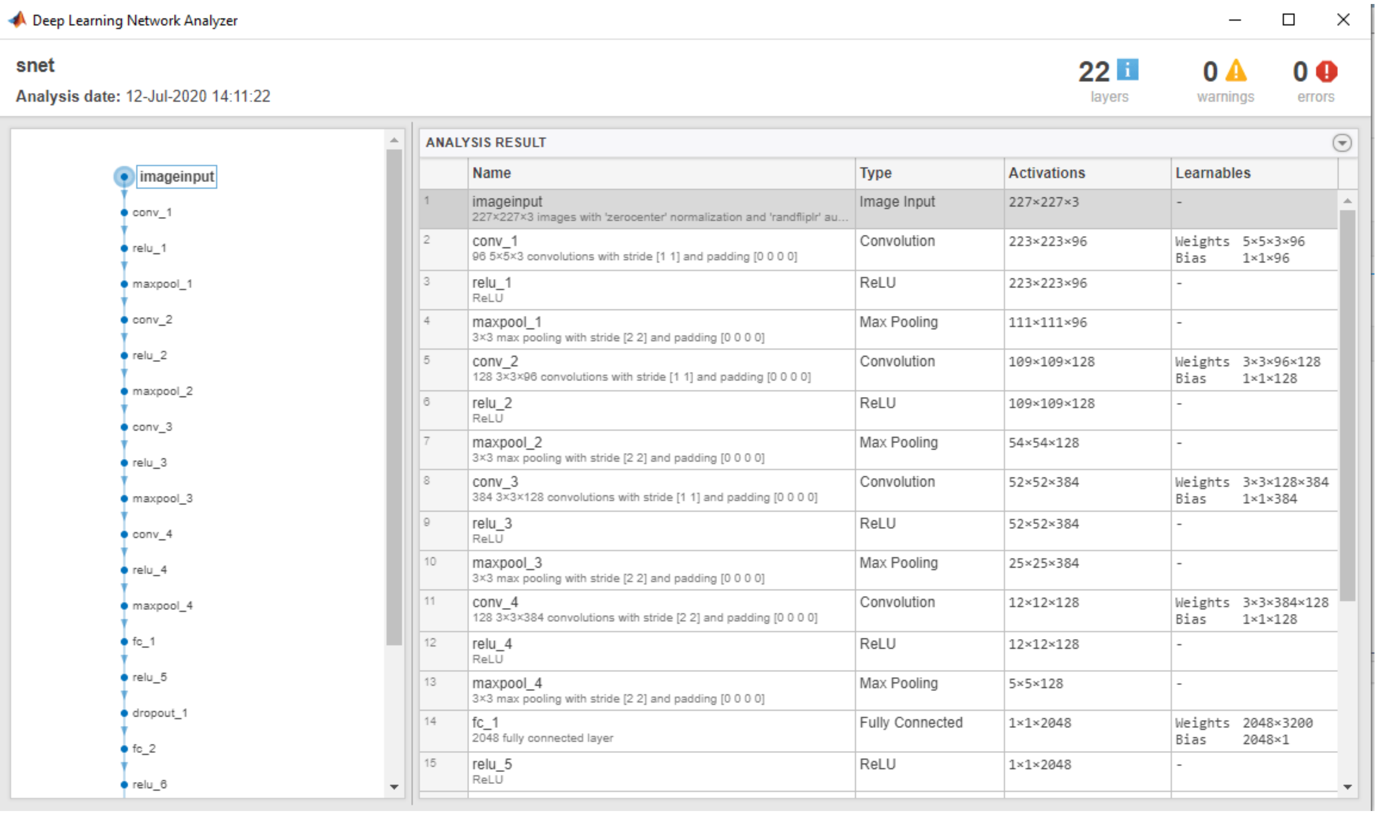
For this example, you need:

- Deep Learning Toolbox ™
- Deep Learning HDL Toolbox ™
- Deep Learning Toolbox Model Quantization Library
- Deep Learning HDL Toolbox Support Package for Xlinx 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);
```



snet

Analysis date: 12-Jul-2020 14:11:22

ANALYSIS RESULT

	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=35×5 table
        Optimized Layer Name          Network Layer Name     Learnables / Activations    MinValue
    ____________________________    __________________    ________________________    __________

    {'conv_1_Weights'          }      {'conv_1'     }           "Weights"              -0.048978
    {'conv_1_Bias'             }      {'conv_1'     }           "Bias"                   0.99996
    {'conv_2_Weights'          }      {'conv_2'     }           "Weights"              -0.055518
    {'conv_2_Bias'             }      {'conv_2'     }           "Bias"                -0.00061171
    {'conv_3_Weights'          }      {'conv_3'     }           "Weights"              -0.045942
    {'conv_3_Bias'             }      {'conv_3'     }           "Bias"                 -0.0013998
    {'conv_4_Weights'          }      {'conv_4'     }           "Weights"              -0.045967
    {'conv_4_Bias'             }      {'conv_4'     }           "Bias"                   -0.00164
    {'fc_1_Weights'            }      {'fc_1'       }           "Weights"              -0.051394
    {'fc_1_Bias'               }      {'fc_1'       }           "Bias"                -0.00052319
    {'fc_2_Weights'            }      {'fc_2'       }           "Weights"               -0.05016
    {'fc_2_Bias'               }      {'fc_2'       }           "Bias"                 -0.0017564
    {'fc_3_Weights'            }      {'fc_3'       }           "Weights"              -0.050706
    {'fc_3_Bias'               }      {'fc_3'       }           "Bias"                  -0.02951
    {'imageinput'              }      {'imageinput'}            "Activations"                  0
    {'imageinput_normalization'}      {'imageinput'}            "Activations"            -139.34
      ⋮
```

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

```
ans=35×5 table
         Optimized Layer Name          Network Layer Name    Learnables / Activations     MinValue
    ____________________________    __________________    ________________________    __________

    {'conv_1_Weights'           }      {'conv_1'     }           "Weights"                  -0.048978
    {'conv_1_Bias'              }      {'conv_1'     }           "Bias"                       0.99996
    {'conv_2_Weights'           }      {'conv_2'     }           "Weights"                  -0.055518
    {'conv_2_Bias'              }      {'conv_2'     }           "Bias"                   -0.00061171
    {'conv_3_Weights'           }      {'conv_3'     }           "Weights"                  -0.045942
    {'conv_3_Bias'              }      {'conv_3'     }           "Bias"                    -0.0013998
    {'conv_4_Weights'           }      {'conv_4'     }           "Weights"                  -0.045967
    {'conv_4_Bias'              }      {'conv_4'     }           "Bias"                      -0.00164
    {'fc_1_Weights'             }      {'fc_1'       }           "Weights"                  -0.051394
    {'fc_1_Bias'                }      {'fc_1'       }           "Bias"                   -0.00052319
    {'fc_2_Weights'             }      {'fc_2'       }           "Weights"                   -0.05016
    {'fc_2_Bias'                }      {'fc_2'       }           "Bias"                    -0.0017564
    {'fc_3_Weights'             }      {'fc_3'       }           "Weights"                  -0.050706
    {'fc_3_Bias'                }      {'fc_3'       }           "Bias"                      -0.02951
    {'imageinput'               }      {'imageinput'}           "Activations"                       0
    {'imageinput_normalization'}      {'imageinput'}           "Activations"                 -139.34
      ⋮
```

Create Target Object

Create a target object with a custom name for your target device and an interface to connect your target device to the host computer. Interface options are JTAG and Ethernet. To use JTAG, install Xilinx™ Vivado™ Design Suite 2020.1. 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,validatio
```

Validate Quantized Network

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network. The `validate` function simulates the quantized network in MATLAB. The `validate` function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the single data type network object to the results of the quantized network object.

```
prediction_simulation = dlQuantObj_simulation.validate(validationData_simulation,options_simulati
```

```
### Notice: (Layer  1) The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is imple
### Notice: (Layer  2) The layer 'out_imageinput' with type 'nnet.cnn.layer.RegressionOutputLayer
Compiling leg: conv_1>>maxpool_4 ...
### Notice: (Layer  1) The layer 'imageinput' with type 'nnet.cnn.layer.ImageInputLayer' is imple
### Notice: (Layer 14) The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is imp
Compiling leg: conv_1>>maxpool_4 ... complete.
Compiling leg: fc_1>>fc_3 ...
### Notice: (Layer  1) The layer 'maxpool_4' with type 'nnet.cnn.layer.ImageInputLayer' is implem
### Notice: (Layer  7) The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is imp
Compiling leg: fc_1>>fc_3 ... complete.
### Should not enter here. It means a component is unaccounted for in MATLAB Emulation.
### Notice: (Layer  1) The layer 'fc_3' with type 'nnet.cnn.layer.ImageInputLayer' is implemented
### Notice: (Layer  2) The layer 'softmax' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented
### Notice: (Layer  3) The layer 'classoutput' with type 'nnet.cnn.layer.ClassificationOutputLaye

prediction_simulation = struct with fields:
       NumSamples: 5
    MetricResults: [1×1 struct]
```

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            227×227×3 images with 'zerocenter' normalizatio
     2   'conv_1'        Convolution            96 5×5×3 convolutions with stride [1  1] and pad
     3   'relu_1'        ReLU                   ReLU
     4   'maxpool_1'     Max Pooling            3×3 max pooling with stride [2  2] and padding 
     5   'conv_2'        Convolution            128 3×3×96 convolutions with stride [1  1] and p
     6   'relu_2'        ReLU                   ReLU
     7   'maxpool_2'     Max Pooling            3×3 max pooling with stride [2  2] and padding 
     8   'conv_3'        Convolution            384 3×3×128 convolutions with stride [1  1] and 
     9   'relu_3'        ReLU                   ReLU
    10   'maxpool_3'     Max Pooling            3×3 max pooling with stride [2  2] and padding 
    11   'conv_4'        Convolution            128 3×3×384 convolutions with stride [2  2] and 
    12   'relu_4'        ReLU                   ReLU
    13   'maxpool_4'     Max Pooling            3×3 max pooling with stride [2  2] and padding 
    14   'fc_1'          Fully Connected        2048 fully connected layer
    15   'relu_5'        ReLU                   ReLU
    16   'dropout_1'     Dropout                50% dropout
    17   'fc_2'          Fully Connected        2048 fully connected layer
    18   'relu_6'        ReLU                   ReLU
    19   'dropout_2'     Dropout                50% dropout
    20   'fc_3'          Fully Connected        32 fully connected layer
    21   'softmax'       Softmax                 softmax
    22   'classoutput'   Classification Output   crossentropyex with 'adidas' and 31 other classe

3 Memory Regions created.

Skipping: imageinput
Compiling leg: conv_1>>maxpool_4 ...
Compiling leg: conv_1>>maxpool_4 ... complete.
Compiling leg: fc_1>>fc_3 ...
Compiling leg: fc_1>>fc_3 ... 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"     "48.0 MB"        
    "OutputResultOffset"        "0x03000000"     "4.0 MB"         
    "SchedulerDataOffset"       "0x03400000"     "4.0 MB"         
    "SystemBufferOffset"        "0x03800000"     "60.0 MB"        
    "InstructionDataOffset"     "0x07400000"     "8.0 MB"         
    "ConvWeightDataOffset"      "0x07c00000"     "12.0 MB"        
    "FCWeightDataOffset"        "0x08800000"     "12.0 MB"        
    "EndOffset"                 "0x09400000"     "Total: 148.0 MB"

### Network compilation complete.

### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Deep learning network programming has been skipped as the same network is already loaded on t
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------              -------------              ---------        --
Network                   12722978                  0.05783                       1          127
    conv_1                 3437086                  0.01562 
    maxpool_1              1296014                  0.00589 
    conv_2                 2813632                  0.01279 
    maxpool_2               477861                  0.00217 
    conv_3                 2462903                  0.01120 
    maxpool_3               535330                  0.00243 
    conv_4                 504820                  0.00229
    maxpool_4                8965                  0.00004
    fc_1                   687629                  0.00313
    fc_2                   439923                  0.00200
    fc_3                    58721                  0.00027
 * The clock frequency of the DL processor is: 220MHz

prediction_FPGA = struct with fields:
             NumSamples: 1
          MetricResults: [1×1 struct]
    QuantizedNetworkFPS: 17.2915
```

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=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
     {'Quantized'     }           1
```

```
prediction_FPGA.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
     {'Quantized'     }           1
```

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

```
prediction_FPGA.QuantizedNetworkFPS
```

```
ans = 17.2915
```

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

Estimate ResNet-18 Performance for `zcu102_single` Bitstream Configuration

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

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

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


              Deep Learning Processor Estimator Performance Results

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

Create Custom Processor Configuration

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

```
hPC_custom = dlhdl.ProcessorConfig
```

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

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

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

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

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

Estimate ResNet-18 Performance for Custom Bitstream Configuration

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

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

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


              Deep Learning Processor Estimator Performance Results

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

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

Modify Custom Processor Configuration

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

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

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

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

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

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

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation 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.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.Classification
```

```
              Deep Learning Processor Estimator Performance Results

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

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=21×5 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.laye
### 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 imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                     58101                  0.00023                       1
    ____conv_1               4391                  0.00002
    ____maxpool_1            2877                  0.00001
    ____conv_2               2351                  0.00001
    ____maxpool_2            2265                  0.00001
    ____conv_3               2651                  0.00001
    ____fc                  43566                  0.00017
 * The clock frequency of the DL processor is: 250MHz
```

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

Estimate 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.laye
### 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 imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
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 look only 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. 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,inputS
```

Use `trainingOptions` to specify network training options. Set `'ValidationData'` to the preprocessed validation data. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...
        'MiniBatchSize', 16, ....
        'InitialLearnRate',1e-3, ...
        'MaxEpochs',20,...
        'CheckpointPath', tempdir, ...
        'Shuffle','never');
```

Use `trainYOLOv2ObjectDetector` function to train YOLO v2 object detector.

```
[detector,info] = trainYOLOv2ObjectDetector(preprocessedTrainingData,lgraph,options);
```

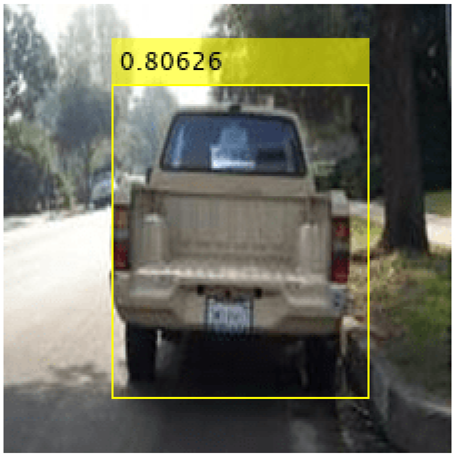
```
*************************************************************************
Training a YOLO v2 Object Detector for the following object classes:

* vehicle

Training on single CPU.
Initializing input data normalization.
|========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |     RMSE     |     Loss     |      Rate       |
|========================================================================================|
|       1 |           1 |       00:00:02 |         8.43 |         71.1 |          0.0010 |
|       5 |          50 |       00:01:26 |         0.71 |          0.5 |          0.0010 |
|      10 |         100 |       00:02:46 |         0.75 |          0.6 |          0.0010 |
|      14 |         150 |       00:04:04 |         0.53 |          0.3 |          0.0010 |
|      19 |         200 |       00:05:23 |         0.48 |          0.2 |          0.0010 |
|      20 |         220 |       00:05:53 |         0.57 |          0.3 |          0.0010 |
|========================================================================================|
Detector training complete.
*************************************************************************
```

As a quick test, run the detector on one test image. Make sure you resize the image to the same size as the training images.

```
I = imread(testDataTbl.imageFilename{2});
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
```

Display the results.

```
I_new = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I_new)
```



Load Pretrained Network

Load the pretrained network.

```
snet=detector.Network;
I_pre=yolo_pre_proc(I);
```

Use `analyzeNetwork` to obtain information about the network layers:

```
analyzeNetwork(snet)
```

Create Target Object

Create a target object for your target device with a vendor name and an interface to connect your target device to the host computer. Interface options are JTAG (default) and Ethernet. Vendor options are Intel or Xilinx. Use the installed Xilinx Vivado Design Suite over an Ethernet connection to program the device.

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

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pre-trained series network, `trainedNetNoCar,` as the network. Make sure the bitstream name matches the data type and the FPGA board that you are targeting. In this example, the target FPGA board is the Zynq UltraScale+ MPSoC ZCU102 board. The bitstream uses a single data type.

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

```
hW = 
  Workflow with properties:

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

Compile YOLO v2 Object Detector

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

```
dn = hW.compile
```

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

     1   'data'                    Image Input              224×224×3 images with 'zscore' norma
     2   'conv1'                   Convolution              64 7×7×3 convolutions with stride [2
     3   'bn_conv1'                Batch Normalization      Batch normalization with 64 channels
     4   'conv1_relu'              ReLU                     ReLU
     5   'pool1'                   Max Pooling              3×3 max pooling with stride [2  2] a
     6   'res2a_branch2a'          Convolution              64 3×3×64 convolutions with stride [
     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 [
    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 [
    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 channel
    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 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 stride
    30   'bn3b_branch2a'         Batch Normalization       Batch normalization with 128 channel
    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 channel
    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 channel
    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 channel
    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 channel
    45   'res4b_branch2a'        Convolution               256 3×3×256 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
    49   'bn4b_branch2b'         Batch Normalization       Batch normalization with 256 channel
    50   'res4b'                 Addition                  Element-wise addition of 2 inputs
    51   'res4b_relu'            ReLU                      ReLU
    52   'yolov2Conv1'           Convolution               256 3×3×256 convolutions with stride
    53   'yolov2Batch1'          Batch Normalization       Batch normalization with 256 channel
    54   'yolov2Relu1'           ReLU                      ReLU
    55   'yolov2Conv2'           Convolution               256 3×3×256 convolutions with stride
    56   'yolov2Batch2'          Batch Normalization       Batch normalization with 256 channel
    57   'yolov2Relu2'           ReLU                      ReLU
    58   'yolov2ClassConv'       Convolution               42 1×1×256 convolutions with stride
    59   'yolov2Transform'       YOLO v2 Transform Layer.  YOLO v2 Transform Layer with 7 ancho
    60   'yolov2OutputLayer'     YOLO v2 Output            YOLO v2 Output with 7 anchors.

### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
5 Memory Regions created.

Skipping: data
Compiling leg: conv1>>pool1 ...
Compiling leg: conv1>>pool1 ... complete.
Compiling leg: res2a_branch2a>>res2a_branch2b ...
Compiling leg: res2a_branch2a>>res2a_branch2b ... complete.
Compiling leg: res2b_branch2a>>res2b_branch2b ...
Compiling leg: res2b_branch2a>>res2b_branch2b ... complete.
Compiling leg: res3a_branch1 ...
Compiling leg: res3a_branch1 ... complete.
Compiling leg: res3a_branch2a>>res3a_branch2b ...
Compiling leg: res3a_branch2a>>res3a_branch2b ... complete.
Compiling leg: res3b_branch2a>>res3b_branch2b ...
Compiling leg: res3b_branch2a>>res3b_branch2b ... complete.
Compiling leg: res4a_branch1 ...
Compiling leg: res4a_branch1 ... complete.
Compiling leg: res4a_branch2a>>res4a_branch2b ...
Compiling leg: res4a_branch2a>>res4a_branch2b ... complete.
Compiling leg: res4b_branch2a>>res4b_branch2b ...
Compiling leg: res4b_branch2a>>res4b_branch2b ... complete.
Compiling leg: yolov2Conv1>>yolov2ClassConv ...
Compiling leg: yolov2Conv1>>yolov2ClassConv ... complete.
Skipping: yolov2Transform
Skipping: yolov2OutputLayer
Creating Schedule...
.....................
Creating Schedule...complete.
Creating Status Table...
....................
Creating Status Table...complete.
Emitting Schedule...
....................
Emitting Schedule...complete.
Emitting Status Table...
......................
Emitting Status Table...complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    ________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x02000000"     "28.0 MB"
    "InstructionDataOffset"     "0x03c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04000000"     "20.0 MB"
    "EndOffset"                 "0x05400000"     "Total: 84.0 MB"

### Network compilation complete.

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

Program the Bitstream onto FPGA and Download Network Weights

To deploy the network on the Zynq® UltraScale+™ MPSoC ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object . This function uses the output of the compile function to program the FPGA board by using the programming file.The function also downloads the network weights and biases. The deploy function checks for the Xilinx Vivado tool and the supported tool version. It then starts programming the FPGA device by using the bitstream, displays progress messages and the time it takes to deploy the network.

```
hW.deploy
```

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

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


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 04-Jan-2021 13:59:03
```

Load the Example Image and Run The Prediction

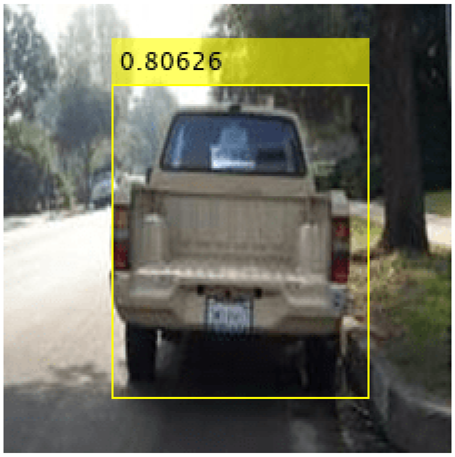
Execute the predict function on the `dlhdl.Workflow` object and display the result:

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

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


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        -
Network                   16974672                  0.07716                       1           169
    conv1                  2224187                  0.01011
    pool1                   573166                  0.00261
    res2a_branch2a          972763                  0.00442
    res2a_branch2b          972632                  0.00442
    res2a                   209363                  0.00095
    res2b_branch2a          972674                  0.00442
    res2b_branch2b          973107                  0.00442
    res2b                   209914                  0.00095
    res3a_branch1           538478                  0.00245
    res3a_branch2a          747078                  0.00340
    res3a_branch2b          904530                  0.00411
    res3a                   104830                  0.00048
    res3b_branch2a          904540                  0.00411
    res3b_branch2b          904278                  0.00411
    res3b                   104900                  0.00048
    res4a_branch1           485804                  0.00221
    res4a_branch2a          485923                  0.00221
    res4a_branch2b          880309                  0.00400
    res4a                    52446                  0.00024
    res4b_branch2a          880071                  0.00400
    res4b_branch2b          880065                  0.00400
    res4b                    52456                  0.00024
    yolov2Conv1             880210                  0.00400
    yolov2Conv2             880375                  0.00400
    yolov2ClassConv         179300                  0.00081
 * The clock frequency of the DL processor is: 220MHz
```

Display the prediction results.

```
[bboxesn, scoresn, labelsn] = yolo_post_proc(prediction,I_pre,anchorBoxes,{'Vehicle'});
I_new3 = insertObjectAnnotation(I,'rectangle',bboxesn,scoresn);
figure
imshow(I_new3)
```



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=21×5 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.laye
### 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 imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                      58101                  0.00023                       1
    ____conv_1                4391                  0.00002
    ____maxpool_1             2877                  0.00001
    ____conv_2                2351                  0.00001
    ____maxpool_2             2265                  0.00001
    ____conv_3                2651                  0.00001
    ____fc                   43566                  0.00017
 * The clock frequency of the DL processor is: 250MHz
```

```
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.laye
### 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 imple


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------          -------------              ---------        --
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 227-by-227-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: [71×1 nnet.cnn.layer.Layer]
    Connections: [78×2 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.

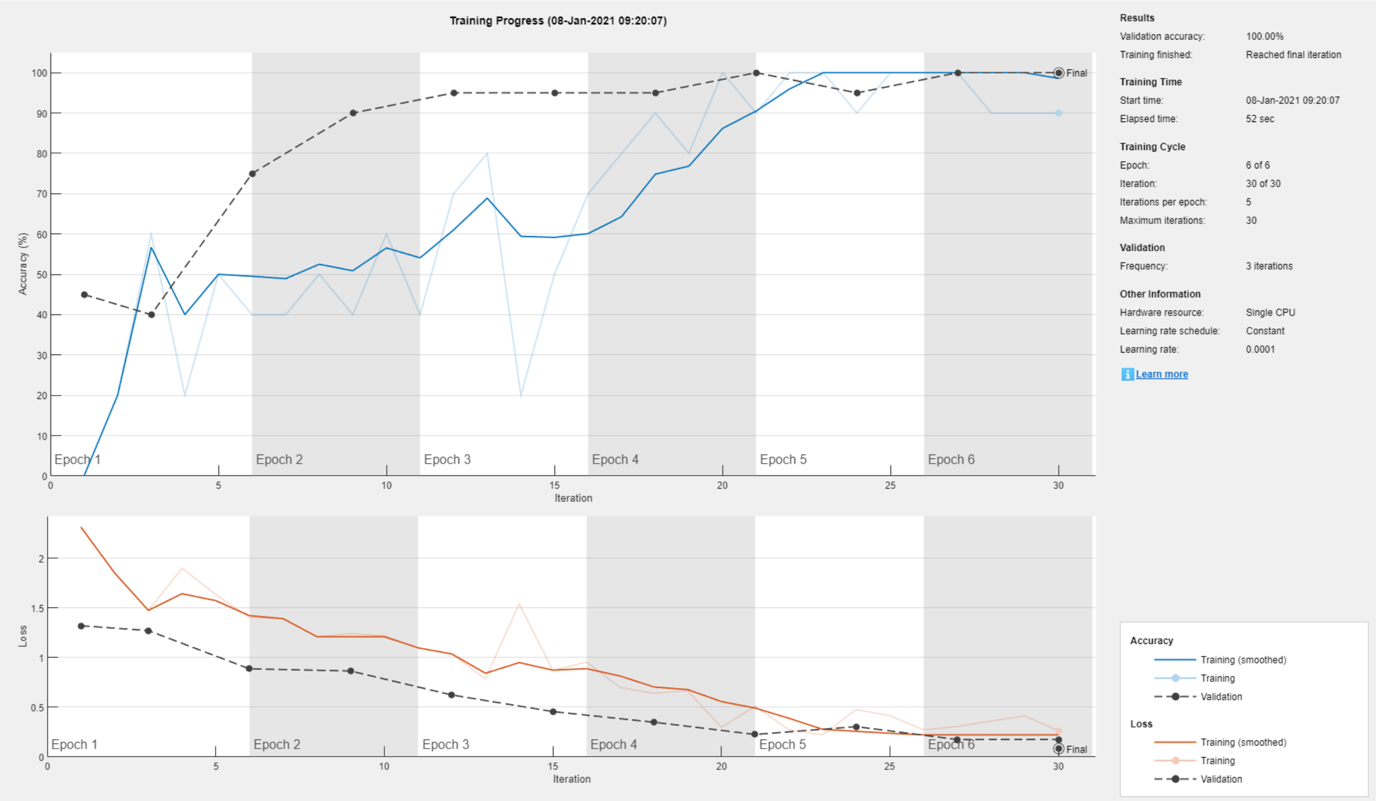
```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
'DataAugmentation',imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

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);
```



Results

Validation accuracy:	100.00%
Training finished:	Reached final iteration

Training Time

Start time:	08-Jan-2021 09:20:07
Elapsed time:	52 sec

Training Cycle

Epoch:	6 of 6
Iteration:	30 of 30
Iterations per epoch:	5
Maximum iterations:	30

Validation

Frequency:	3 iterations

Other Information

Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.0001

ℹ Learn more

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.
```

```
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', 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' normali
     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] and
     6   'res2a_branch2a'        Convolution             64 3×3×64 convolutions with stride [1
     7   'bn2a_branch2a'         Batch Normalization     Batch normalization with 64 channels
     8   'res2a_branch2a_relu'   ReLU                    ReLU
     9   'res2a_branch2b'        Convolution             64 3×3×64 convolutions with stride [1
    10   'bn2a_branch2b'         Batch Normalization     Batch normalization with 64 channels
    11   'res2a'                 Addition                Element-wise addition of 2 inputs
    12   'res2a_relu'            ReLU                    ReLU
    13   'res2b_branch2a'        Convolution             64 3×3×64 convolutions with stride [1
    14   'bn2b_branch2a'         Batch Normalization     Batch normalization with 64 channels
    15   'res2b_branch2a_relu'   ReLU                    ReLU
    16   'res2b_branch2b'        Convolution             64 3×3×64 convolutions with stride [1
    17   'bn2b_branch2b'         Batch Normalization     Batch normalization with 64 channels
    18   'res2b'                 Addition                Element-wise addition of 2 inputs
    19   'res2b_relu'            ReLU                    ReLU
    20   'res3a_branch2a'        Convolution             128 3×3×64 convolutions with stride [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 [
    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
    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 [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 [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 [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 [2
    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'                 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.laye
5 Memory Regions created.

Skipping: data
Compiling leg: conv1>>pool1 ...
Compiling leg: conv1>>pool1 ... complete.
Compiling leg: res2a_branch2a>>res2a_branch2b ...
Compiling leg: res2a_branch2a>>res2a_branch2b ... complete.
Compiling leg: res2b_branch2a>>res2b_branch2b ...
Compiling leg: res2b_branch2a>>res2b_branch2b ... complete.
Compiling leg: res3a_branch1 ...
Compiling leg: res3a_branch1 ... complete.
Compiling leg: res3a_branch2a>>res3a_branch2b ...
Compiling leg: res3a_branch2a>>res3a_branch2b ... complete.
Compiling leg: res3b_branch2a>>res3b_branch2b ...
Compiling leg: res3b_branch2a>>res3b_branch2b ... complete.
Compiling leg: res4a_branch1 ...
Compiling leg: res4a_branch1 ... complete.
Compiling leg: res4a_branch2a>>res4a_branch2b ...
Compiling leg: res4a_branch2a>>res4a_branch2b ... complete.
Compiling leg: res4b_branch2a>>res4b_branch2b ...
Compiling leg: res4b_branch2a>>res4b_branch2b ... complete.
Compiling leg: 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: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

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

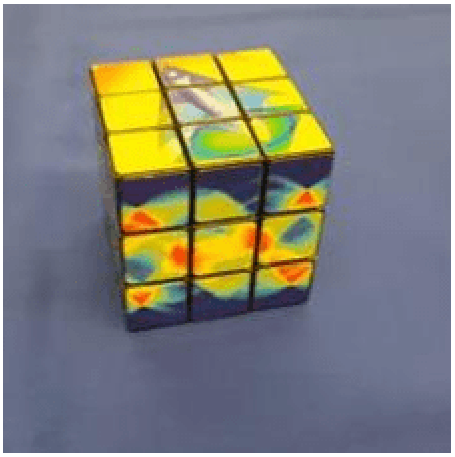
```
hW.deploy
```

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

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      Tota
                         -------------             -------------              ---------        --
Network                   23470681                  0.10668                       1           234
    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 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 227-by-227-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: [71×1 nnet.cnn.layer.Layer]
    Connections: [78×2 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.

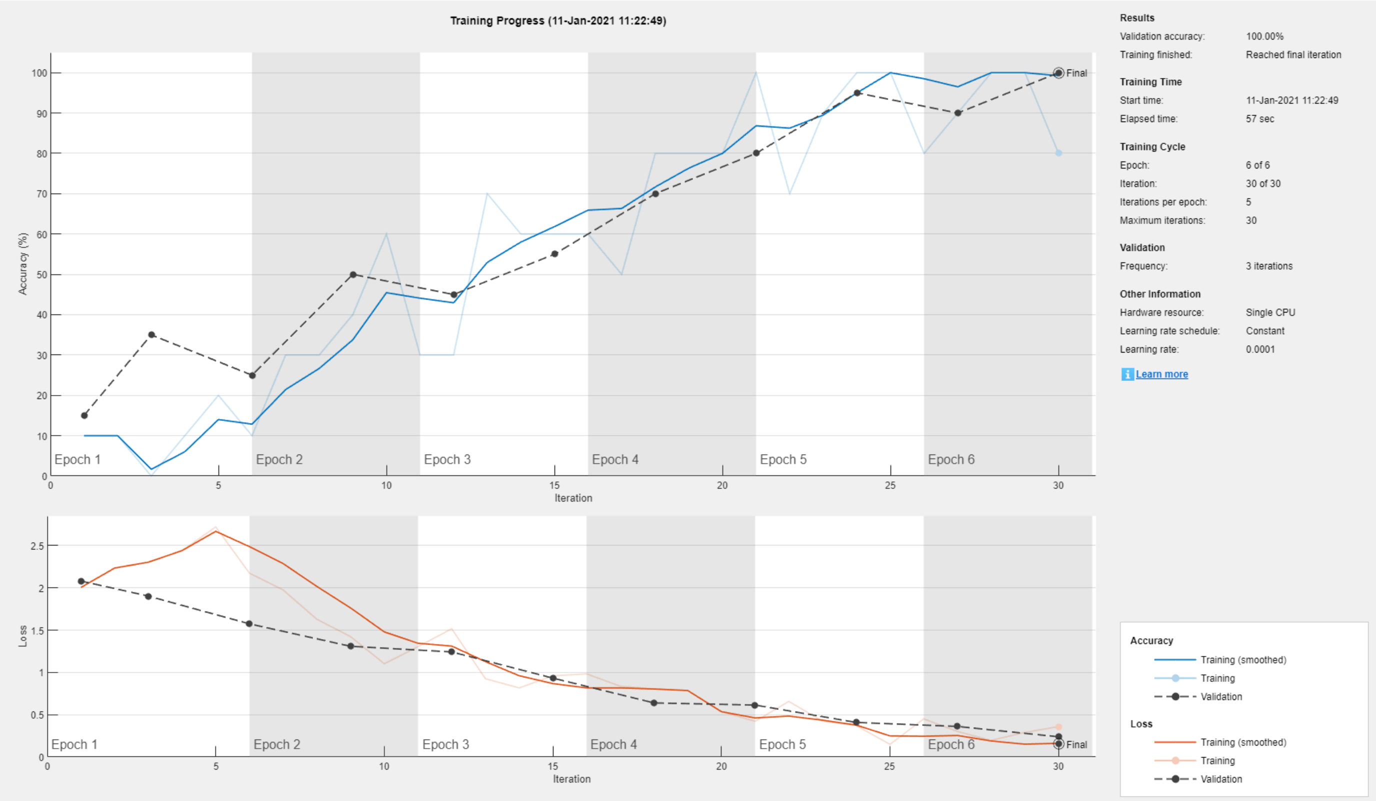
```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
'DataAugmentation',imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

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);
```



Results

Validation accuracy: 100.00%

Training finished: Reached final iteration

Training Time

Start time: 11-Jan-2021 11:22:49

Elapsed time: 57 sec

Training Cycle

Epoch: 6 of 6

Iteration: 30 of 30

Iterations per epoch: 5

Maximum iterations: 30

Validation

Frequency: 3 iterations

Other Information

Hardware resource: Single CPU

Learning rate schedule: Constant

Learning rate: 0.0001

ℹ Learn more

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=95×5 table
       Optimized Layer Name          Network Layer Name    Learnables / Activations    MinValue    M
    __________________________    __________________    ________________________    ________    _

    {'conv1_Weights'          }    {'bn_conv1'     }            "Weights"           -0.86045
    {'conv1_Bias'             }    {'bn_conv1'     }            "Bias"              -0.66706    (
    {'res2a_branch2a_Weights'}    {'bn2a_branch2a'}            "Weights"           -0.40354    (
    {'res2a_branch2a_Bias'   }    {'bn2a_branch2a'}            "Bias"               -0.7954
    {'res2a_branch2b_Weights'}    {'bn2a_branch2b'}            "Weights"           -0.75855
    {'res2a_branch2b_Bias'   }    {'bn2a_branch2b'}            "Bias"               -1.3406
    {'res2b_branch2a_Weights'}    {'bn2b_branch2a'}            "Weights"           -0.32464    (
    {'res2b_branch2a_Bias'   }    {'bn2b_branch2a'}            "Bias"               -1.1606
    {'res2b_branch2b_Weights'}    {'bn2b_branch2b'}            "Weights"            -1.1713    (
    {'res2b_branch2b_Bias'   }    {'bn2b_branch2b'}            "Bias"              -0.73906
    {'res3a_branch2a_Weights'}    {'bn3a_branch2a'}            "Weights"           -0.19423
    {'res3a_branch2a_Bias'   }    {'bn3a_branch2a'}            "Bias"              -0.53868    (
    {'res3a_branch2b_Weights'}    {'bn3a_branch2b'}            "Weights"           -0.53801    (
    {'res3a_branch2b_Bias'   }    {'bn3a_branch2b'}            "Bias"               -0.6457
    {'res3a_branch1_Weights' }    {'bn3a_branch1' }            "Weights"           -0.64085    (
    {'res3a_branch1_Bias'    }    {'bn3a_branch1' }            "Bias"               -0.9258    (
      ⋮
```

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.
```

```
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             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
     7   'bn2a_branch2a'          Batch Normalization     Batch normalization with 64 channels
     8   'res2a_branch2a_relu'    ReLU                    ReLU
     9   'res2a_branch2b'         Convolution             64 3×3×64 convolutions with stride [1
    10   'bn2a_branch2b'          Batch Normalization     Batch normalization with 64 channels
    11   'res2a'                  Addition                Element-wise addition of 2 inputs
    12   'res2a_relu'             ReLU                    ReLU
    13   'res2b_branch2a'         Convolution             64 3×3×64 convolutions with stride [1
    14   'bn2b_branch2a'          Batch Normalization     Batch normalization with 64 channels
    15   'res2b_branch2a_relu'    ReLU                    ReLU
    16   'res2b_branch2b'         Convolution             64 3×3×64 convolutions with stride [1
    17   'bn2b_branch2b'          Batch Normalization     Batch normalization with 64 channels
    18   'res2b'                  Addition                Element-wise addition of 2 inputs
    19   'res2b_relu'             ReLU                    ReLU
    20   'res3a_branch2a'         Convolution             128 3×3×64 convolutions with stride [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
    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
    28   'bn3a_branch1'           Batch Normalization     Batch normalization with 128 channels
    29   'res3b_branch2a'         Convolution             128 3×3×128 convolutions with stride [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
    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
    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 [2
    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 [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 [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 [2
    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'                 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.laye
5 Memory Regions created.

Skipping: data
Compiling leg: conv1>>pool1 ...
Compiling leg: conv1>>pool1 ... complete.
Compiling leg: res2a_branch2a>>res2a_branch2b ...
Compiling leg: res2a_branch2a>>res2a_branch2b ... complete.
Compiling leg: res2b_branch2a>>res2b_branch2b ...
Compiling leg: res2b_branch2a>>res2b_branch2b ... complete.
Compiling leg: res3a_branch1 ...
Compiling leg: res3a_branch1 ... complete.
Compiling leg: res3a_branch2a>>res3a_branch2b ...
Compiling leg: res3a_branch2a>>res3a_branch2b ... complete.
Compiling leg: res3b_branch2a>>res3b_branch2b ...
Compiling leg: res3b_branch2a>>res3b_branch2b ... complete.
Compiling leg: res4a_branch1 ...
Compiling leg: res4a_branch1 ... complete.
Compiling leg: res4a_branch2a>>res4a_branch2b ...
Compiling leg: res4a_branch2a>>res4a_branch2b ... complete.
Compiling leg: res4b_branch2a>>res4b_branch2b ...
Compiling leg: res4b_branch2a>>res4b_branch2b ... complete.
Compiling leg: 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"     "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"     "16.0 MB"       
    "FCWeightDataOffset"        "0x05000000"     "4.0 MB"        
    "EndOffset"                 "0x05400000"     "Total: 84.0 MB"

### Network compilation complete.

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

Program Bitstream onto FPGA and Download Network Weights

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

```
hW.deploy
```

```
### Programming FPGA Bitstream using Ethernet...
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 11-Jan-2021 11:26:16
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 11-Jan-2021 11:26:16
```

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      Tota
                         -------------                -------------           ---------        -
Network                    7323615                  0.02929                       1           73
    conv1                  1111619                  0.00445
    pool1                   235563                  0.00094
    res2a_branch2a          268736                  0.00107
    res2a_branch2b          269031                  0.00108
    res2a                    94319                  0.00038
    res2b_branch2a          268677                  0.00107
    res2b_branch2b          268863                  0.00108
    res2b                    94255                  0.00038
    res3a_branch1             155156                  0.00062
    res3a_branch2a            226445                  0.00091
    res3a_branch2b            243593                  0.00097
    res3a                      47248                  0.00019
    res3b_branch2a            243461                  0.00097
    res3b_branch2b            243581                  0.00097
    res3b                      47232                  0.00019
    res4a_branch1             133899                  0.00054
    res4a_branch2a            134402                  0.00054
    res4a_branch2b            234184                  0.00094
    res4a                      23628                  0.00009
    res4b_branch2a            234058                  0.00094
    res4b_branch2b            234648                  0.00094
    res4b                      23756                  0.00010
    res5a_branch1             310730                  0.00124
    res5a_branch2a            310810                  0.00124
    res5a_branch2b            595374                  0.00238
    res5a                      11827                  0.00005
    res5b_branch2a            595150                  0.00238
    res5b_branch2b            595904                  0.00238
    res5b                      12012                  0.00005
    pool5                      35870                  0.00014
    new_fc                     17811                  0.00007
 * The clock frequency of the DL processor is: 250MHz
```

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

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

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'))
```

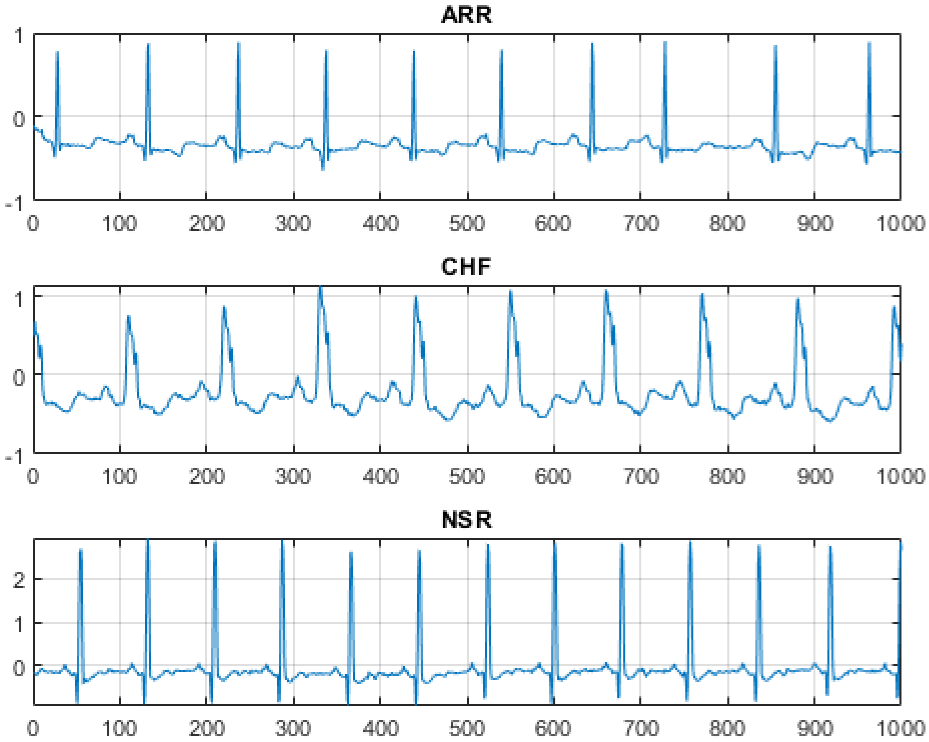
ECGData is a structure array with two fields: Data and Labels. The Data field is a 162-by-65536 matrix where each row is an ECG recording sampled at 128 hertz. Labels is a 162-by-1 cell array of diagnostic labels, one for each row of Data. The three diagnostic categories are: 'ARR', 'CHF', and 'NSR'.

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)
```



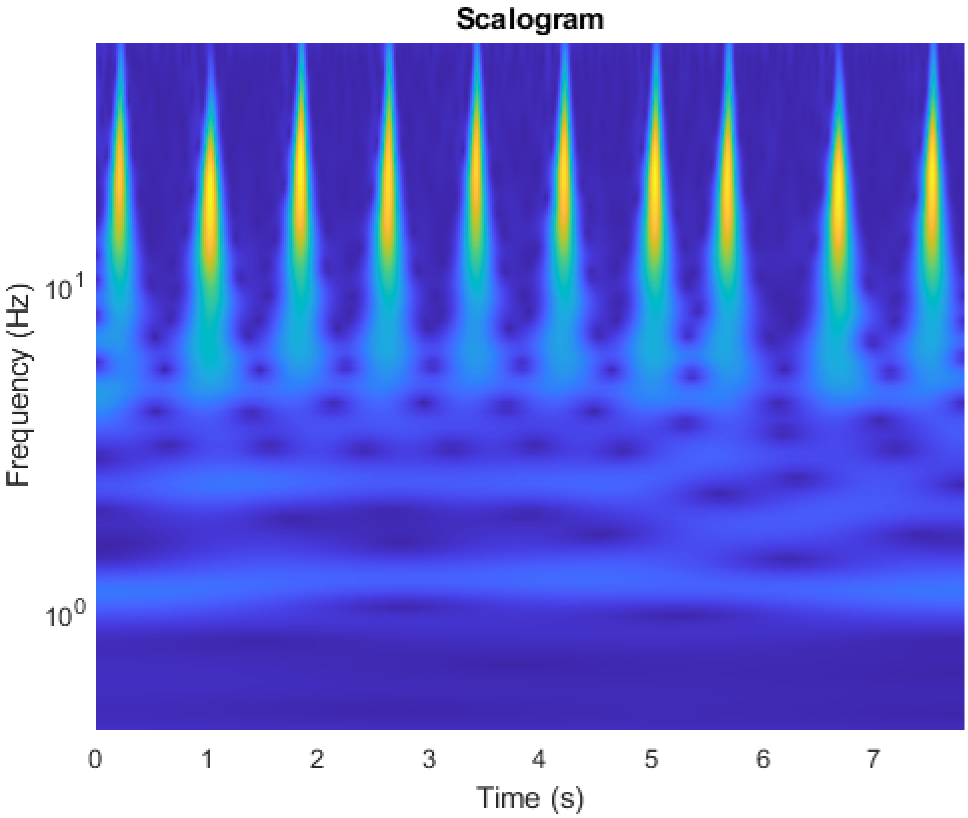
Create Time-Frequency Representations

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
     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 p
     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`.

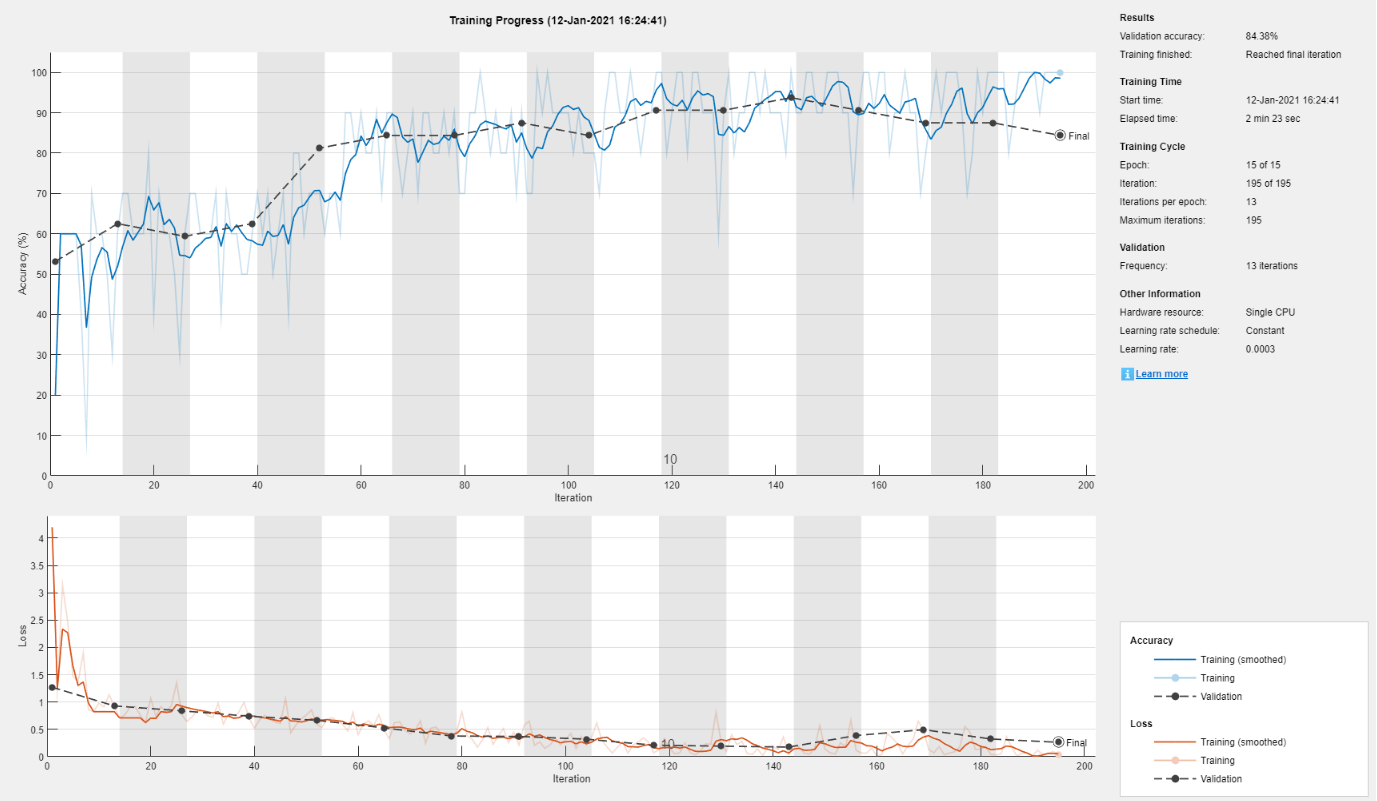
```
augimgsTrain = augmentedImageDatastore([227 227],imgsTrain);
augimgsValidation = augmentedImageDatastore([227 227],imgsValidation);
```

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  |  Mini-batch  |  Validation  |  Mini-batch  |  Validat
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|======================================================================================================
|       1 |           1 |       00:00:06 |       20.00% |       53.12% |       4.2000 |       1.2
|       1 |          13 |       00:00:16 |       60.00% |       62.50% |       0.9170 |       0.9
|       2 |          26 |       00:00:25 |       60.00% |       59.38% |       0.7670 |       0.8
|       3 |          39 |       00:00:35 |       60.00% |       62.50% |       0.7033 |       0.7
|       4 |          50 |       00:00:42 |       70.00% |              |       0.7629 |
|       4 |          52 |       00:00:44 |       70.00% |       81.25% |       0.5941 |       0.6
|       5 |          65 |       00:00:53 |       90.00% |       84.38% |       0.4883 |       0.5
|       6 |          78 |       00:01:02 |       90.00% |       84.38% |       0.3627 |       0.3
|       7 |          91 |       00:01:11 |       90.00% |       87.50% |       0.2145 |       0.3
|       8 |         100 |       00:01:17 |       90.00% |              |       0.3157 |
|       8 |         104 |       00:01:20 |       80.00% |       84.38% |       0.2166 |       0.3
|       9 |         117 |       00:01:29 |      100.00% |       90.62% |       0.0720 |       0.2
|      10 |         130 |       00:01:38 |       90.00% |       90.62% |       0.2510 |       0.1
|      11 |         143 |       00:01:47 |      100.00% |       93.75% |       0.0443 |       0.1
|      12 |         150 |       00:01:51 |      100.00% |              |       0.1377 |
|      12 |         156 |       00:01:56 |       90.00% |       90.62% |       0.1190 |       0.3
|      13 |         169 |       00:02:04 |       80.00% |       87.50% |       0.4859 |       0.4
|      14 |         182 |       00:02:13 |      100.00% |       87.50% |       0.0395 |       0.3
|      15 |         195 |       00:02:22 |      100.00% |       84.38% |       0.0399 |       0.2
|======================================================================================================
```



Results

Validation accuracy: 84.38%

Training finished: Reached final iteration

Training Time

Start time: 12-Jan-2021 16:24:41

Elapsed time: 2 min 23 sec

Training Cycle

Epoch: 15 of 15

Iteration: 195 of 195

Iterations per epoch: 13

Maximum iterations: 195

Validation

Frequency: 13 iterations

Other Information

Hardware resource: Single CPU

Learning rate schedule: Constant

Learning rate: 0.0003

Learn more

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.b
```

```
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: [1×1 DAGNetwork]
          Bitstream: 'zcu102_single'
    ProcessorConfig: []
             Target: [1×1 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] an
     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] an
    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] an
    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 1×1×512 convolutions with stride
    57   'fire9-relu_squeeze1x1'   ReLU                     ReLU
    58   'fire9-expand1x1'         Convolution              256 1×1×64 convolutions with stride
    59   'fire9-relu_expand1x1'    ReLU                     ReLU
    60   'fire9-expand3x3'         Convolution              256 3×3×64 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 1×1×512 convolutions with stride [1
    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: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
```

Program Bitstream onto FPGA and Download Network Weights

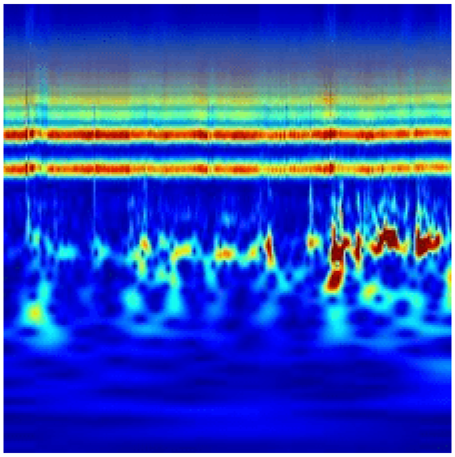
To deploy the network on the Xilinx ZCU102 hardware, run the deploy function of the `dlhdl.Workflow` object. This function uses the output of the compile function to program the FPGA board by using the programming file. It also downloads the network weights and biases. The deploy function starts programming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

```
hW.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Loading weights to 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      Tota
                         -------------              -------------              ---------        --
Network                    8847610                  0.04022                       1           88
    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: [71×1 nnet.cnn.layer.Layer]
    Connections: [78×2 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.

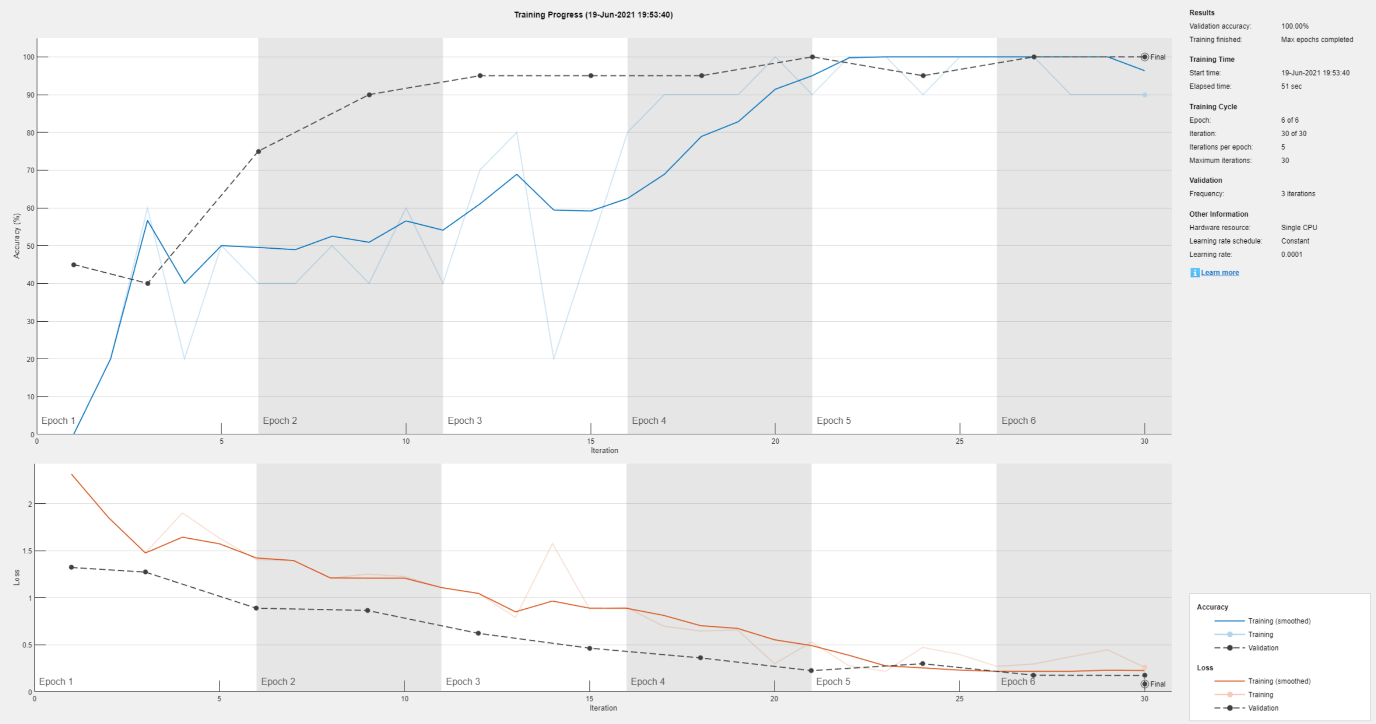
```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
'DataAugmentation',imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

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);
```



Results
Validation accuracy: 100.00%
Training finished: Max epochs completed

Training Time
Start time: 19-Jun-2021 19:53:40
Elapsed time: 51 sec

Training Cycle
Epoch: 6 of 6
Iteration: 30 of 30
Iterations per epoch: 5
Maximum iterations: 30

Validation
Frequency: 3 iterations

Other Information
Hardware resource: Single CPU
Learning rate schedule: Constant
Learning rate: 0.0001

Learn more

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.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### 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 softwar
### 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 i
```

Load Image for Prediction and Intermediate Layer Activation Results

Load the example image. Save it's size for future use.

```
imgFile = fullfile(pwd,'MerchData','MathWorks Cube','Mathworks cube_0.jpg');
inputImg = imresize(imread(imgFile),inputSize(1:2));
imshow(inputImg)
```

Show Activations of First Maxpool Layer

Investigate features by observing which areas in the convolution layers activate on an image. Compare that image to the corresponding areas in the original images. Each layer of a convolutional neural network consists of many 2-D arrays called *channels*. Pass the image through the network and examine the output activations of the `pool1` layer.

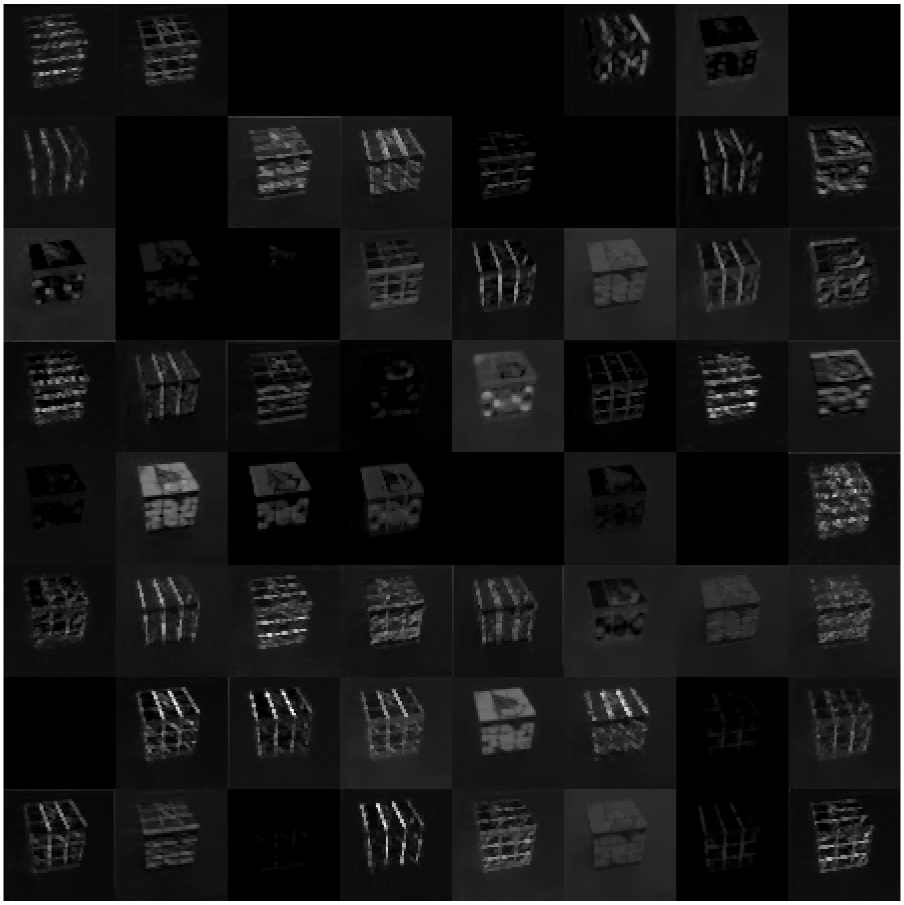
```
act1 = simObj.activations(single(inputImg),'pool1');
```

The activations are returned as a 3-D array, with the third dimension indexing the channel on the `pool1` layer. To show these activations by using the `imtile` function, reshape the array to 4-D. The third dimension in the input to `imtile` represents the image color. Set the third dimension to have size 1 because the activations do not have color. The fourth dimension indexes the channel.

```
sz = size(act1);
act1 = reshape(act1,[sz(1) sz(2) 1 sz(3)]);
```

Display the activations. Each activation can take any value, so normalize the output by using the `mat2gray`. All activations are scaled so that the minimum activation is 0 and the maximum activation is 1. Display the 64 images on an 8-by-8 grid, one for each channel in the layer.

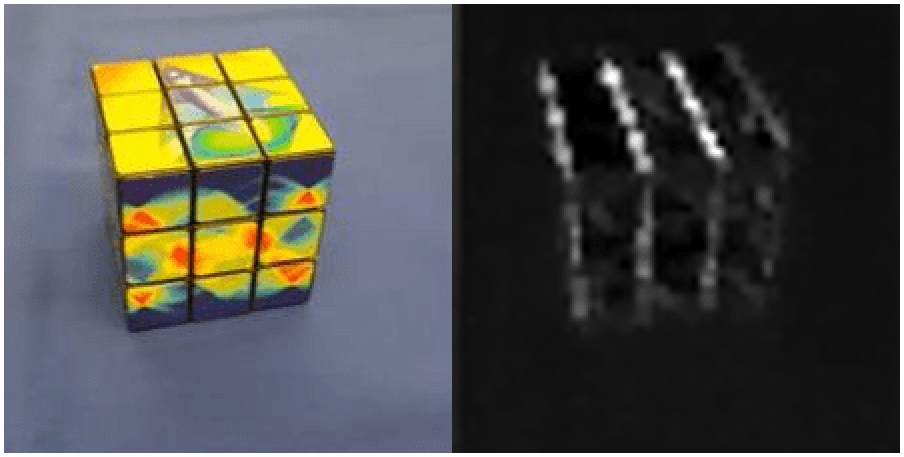
```
I = imtile(mat2gray(act1),'GridSize',[8 8]);
imshow(I)
```



Find Strongest Activation Channel

Find the strongest channels by programmatically investigating channels with large activations. Find the channel that has the largest activation by using the `max` function, resize the channel output, and display the activations.

```
[maxValue,maxValueIndex] = max(max(max(act1)));
act1chMax = act1(:,:,:,maxValueIndex);
act1chMax = mat2gray(act1chMax);
act1chMax = imresize(act1chMax,inputSize(1:2));

I = imtile({inputImg,act1chMax});
imshow(I)
```



Compare the strongest activation channel image to the original image. This channel activates on edges. It activates positively on light left/dark right edges and negatively on dark left/light right edges.

Verify Prediction Results

Verify and display the prediction results of the `dlhdl.Simulator` object by using the `predict` function.

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

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

See Also

`dlhdl.Simulator` | `activations` | `predict` | `dlhdl.ProcessorConfig`

Classify Images on FPGA by Using Quantized GoogLeNet Network

This example show how to use the Deep Learning HDL Toolbox™ to deploy a quantized GoogleNet network to classify an image. The example uses the pretrained GoogLeNet network to demonstrate transfer learning, quantization, and deployment for the quantized network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results.

Deploy the quantized GoogLeNet network by creating a `dlhdl.Workflow` object. Use the `dlhdl.Workflow` object to:

- Generate a list of instructions, weights and biases by using the `compile` method.
- Generate a programming file for the FPGA by using the `deploy` method.
- Retrieve the network prediction results and performance by using the `predict` method.

GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories.

Prerequisites

- Deep Learning Toolbox™
- Deep Learning HDL Toolbox™
- Deep Learning Toolbox Model for GoogLeNet Network
- Deep Learning HDL Toolbox™ Support Package for Intel FPGA and SoC
- Image Processing Toolbox™
- Intel Arria10 SoC development kit
- Deep Learning Toolbox™ Model Quantization Library support package.
- MATLAB Coder Interface for Deep learning Libraries

Transfer Learning Using GoogLeNet

To perform classification on a new set of images, you fine-tune a pretrained GoogLeNet convolutional neural network by transfer learning. In transfer learning, you can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training images.

Load Pretrained DAG Network

Load the pretrained DAG network, GoogLeNet.

```
net = googlenet;
```

Use the `analyzeNetwork` function to obtain information about the network layers.

```
analyzeNetwork(net);
```

The first layer, the image input layer, requires input images of size 224-by-224-by-3, where 3 is the number of color channels.

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

```
inputSize = 1×3

   224   224     3
```

Define Training and Validation Data Sets

This example uses the MathWorks MerchData data set. This is a small data set containing 75 images of MathWorks merchandise, belonging to five different classes (*cap*, *cube*, *playing cards*, *screwdriver*, and *torch*).

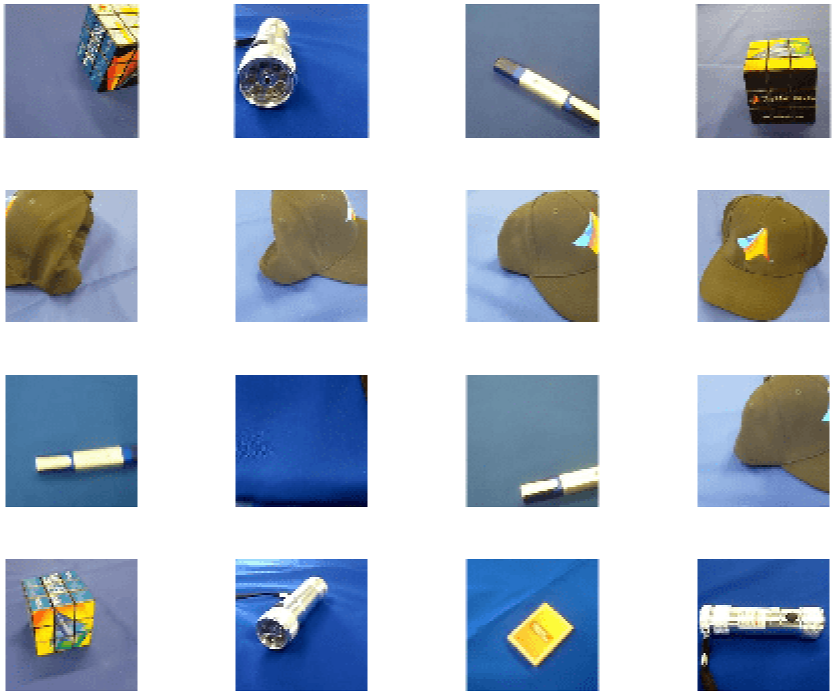
```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. `splitEachLabel` splits the `images` datastore into two new datastores.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

This data set now contains 55 training images and 20 validation images. Display some sample images.

```
numTrainImages = numel(imdsTrain.Labels);
idx = randperm(numTrainImages,16);
figure
for i = 1:16
    subplot(4,4,i)
    I = readimage(imdsTrain,idx(i));
    imshow(I)
end
```



Replace Final Layers

The fully connected layer and classification layer of the pretrained network `net` are configured for 1000 classes. These two layers, `loss3-classifier` and `output` in GoogLeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set.

Extract the layer graph from the trained network.

```
lgraph = layerGraph(net)
```

```
lgraph = 
  LayerGraph with properties:

         Layers: [144×1 nnet.cnn.layer.Layer]
    Connections: [170×2 table]
     InputNames: {'data'}
    OutputNames: {'output'}
```

Replace the fully connected layer with a new fully connected layer that has number of outputs equal to the number of classes. To make learning faster in the new layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the fully connected layer.

```
numClasses = numel(categories(imdsTrain.Labels))
numClasses = 5
```

Remove 'loss3-classifier', 'prob' and 'output' layers from the lgraph.

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

Create three new layers and add them to the lgraph. Ensure the transferred and new layers are properly connected together in the lgraph.

```
newLayers = [
    fullyConnectedLayer(numClasses,'WeightLearnRateFactor',20,'BiasLearnRateFactor',20,'Name','ne
    softmaxLayer('Name','newProb')
    classificationLayer('Name','newClassOutput',"Classes","auto")];

lgraph = addLayers(lgraph,newLayers);
lgraph = connectLayers(lgraph,layers(end-3).Name,'newFC');
```

Train Network

The network requires input images of size 224-by-224-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from over-fitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

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

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, the learning rate factors were increased for the fully conencted 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.e
```

Create Target Object

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

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

Alternatively, you can use an Ethernet interface.

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

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. 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 Intel Arria10 SOC board and the bitstream uses the int8 data type.

```
hW = dlhdl.Workflow('network', dlQuantObj, 'Bitstream', 'arria10soc_int8','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 stri
     5   'pool1-norm1'                    Cross Channel Normalization   cross channel normalizati
     6   'conv2-3x3_reduce'               Convolution                   64 1×1×64 convolutions wi
     7   'conv2-relu_3x3_reduce'          ReLU                          ReLU
     8   'conv2-3x3'                      Convolution                   192 3×3×64 convolutions w
     9   'conv2-relu_3x3'                 ReLU                          ReLU
    10   'conv2-norm2'                    Cross Channel Normalization   cross channel normalizati
    11   'pool2-3x3_s2'                   Max Pooling                   3×3 max pooling with stri
    12   'inception_3a-1x1'               Convolution                   64 1×1×192 convolutions w
    13   'inception_3a-relu_1x1'          ReLU                          ReLU
    14   'inception_3a-3x3_reduce'        Convolution                   96 1×1×192 convolutions w
    15   'inception_3a-relu_3x3_reduce'   ReLU                          ReLU
    16   'inception_3a-3x3'               Convolution                   128 3×3×96 convolutions w
    17   'inception_3a-relu_3x3'          ReLU                          ReLU
    18   'inception_3a-5x5_reduce'        Convolution                   16 1×1×192 convolutions w
    19   'inception_3a-relu_5x5_reduce'   ReLU                          ReLU
    20   'inception_3a-5x5'               Convolution                   32 5×5×16 convolutions wi
    21   'inception_3a-relu_5x5'          ReLU                          ReLU
    22   'inception_3a-pool'              Max Pooling                   3×3 max pooling with stri
    23   'inception_3a-pool_proj'         Convolution                   32 1×1×192 convolutions w
    24   'inception_3a-relu_pool_proj'    ReLU                          ReLU
    25   'inception_3a-output'            Depth concatenation           Depth concatenation of 4
    26   'inception_3b-1x1'               Convolution                   128 1×1×256 convolutions
    27   'inception_3b-relu_1x1'          ReLU                          ReLU
    28   'inception_3b-3x3_reduce'        Convolution                   128 1×1×256 convolutions
    29   'inception_3b-relu_3x3_reduce'   ReLU                          ReLU
    30   'inception_3b-3x3'               Convolution                   192 3×3×128 convolutions
    31   'inception_3b-relu_3x3'          ReLU                          ReLU
    32   'inception_3b-5x5_reduce'        Convolution                   32 1×1×256 convolutions w
    33   'inception_3b-relu_5x5_reduce'   ReLU                          ReLU
    34   'inception_3b-5x5'               Convolution                   96 5×5×32 convolutions wi
    35   'inception_3b-relu_5x5'          ReLU                          ReLU
    36   'inception_3b-pool'              Max Pooling                   3×3 max pooling with stri
    37   'inception_3b-pool_proj'         Convolution                   64 1×1×256 convolutions w
    38   'inception_3b-relu_pool_proj'    ReLU                          ReLU
    39   'inception_3b-output'            Depth concatenation           Depth concatenation of 4
    40   'pool3-3x3_s2'                   Max Pooling                   3×3 max pooling with stri
    41   'inception_4a-1x1'               Convolution                   192 1×1×480 convolutions
    42   'inception_4a-relu_1x1'          ReLU                          ReLU
    43   'inception_4a-3x3_reduce'        Convolution                   96 1×1×480 convolutions w
    44   'inception_4a-relu_3x3_reduce'   ReLU                          ReLU
    45   'inception_4a-3x3'               Convolution                   208 3×3×96 convolutions w
    46   'inception_4a-relu_3x3'          ReLU                          ReLU
    47   'inception_4a-5x5_reduce'        Convolution                   16 1×1×480 convolutions w
    48   'inception_4a-relu_5x5_reduce'   ReLU                          ReLU
    49   'inception_4a-5x5'               Convolution                   48 5×5×16 convolutions wi
```

```
50   'inception_4a-relu_5x5'          ReLU                          ReLU
51   'inception_4a-pool'              Max Pooling                   3×3 max pooling with stri
52   'inception_4a-pool_proj'         Convolution                   64 1×1×480 convolutions w
53   'inception_4a-relu_pool_proj'    ReLU                          ReLU
54   'inception_4a-output'            Depth concatenation           Depth concatenation of 4
55   'inception_4b-1x1'               Convolution                   160 1×1×512 convolutions
56   'inception_4b-relu_1x1'          ReLU                          ReLU
57   'inception_4b-3x3_reduce'        Convolution                   112 1×1×512 convolutions
58   'inception_4b-relu_3x3_reduce'   ReLU                          ReLU
59   'inception_4b-3x3'               Convolution                   224 3×3×112 convolutions
60   'inception_4b-relu_3x3'          ReLU                          ReLU
61   'inception_4b-5x5_reduce'        Convolution                   24 1×1×512 convolutions w
62   'inception_4b-relu_5x5_reduce'   ReLU                          ReLU
63   'inception_4b-5x5'               Convolution                   64 5×5×24 convolutions wi
64   'inception_4b-relu_5x5'          ReLU                          ReLU
65   'inception_4b-pool'              Max Pooling                   3×3 max pooling with stri
66   'inception_4b-pool_proj'         Convolution                   64 1×1×512 convolutions w
67   'inception_4b-relu_pool_proj'    ReLU                          ReLU
68   'inception_4b-output'            Depth concatenation           Depth concatenation of 4
69   'inception_4c-1x1'               Convolution                   128 1×1×512 convolutions
70   'inception_4c-relu_1x1'          ReLU                          ReLU
71   'inception_4c-3x3_reduce'        Convolution                   128 1×1×512 convolutions
72   'inception_4c-relu_3x3_reduce'   ReLU                          ReLU
73   'inception_4c-3x3'               Convolution                   256 3×3×128 convolutions
74   'inception_4c-relu_3x3'          ReLU                          ReLU
75   'inception_4c-5x5_reduce'        Convolution                   24 1×1×512 convolutions w
76   'inception_4c-relu_5x5_reduce'   ReLU                          ReLU
77   'inception_4c-5x5'               Convolution                   64 5×5×24 convolutions wi
78   'inception_4c-relu_5x5'          ReLU                          ReLU
79   'inception_4c-pool'              Max Pooling                   3×3 max pooling with stri
80   'inception_4c-pool_proj'         Convolution                   64 1×1×512 convolutions w
81   'inception_4c-relu_pool_proj'    ReLU                          ReLU
82   'inception_4c-output'            Depth concatenation           Depth concatenation of 4
83   'inception_4d-1x1'               Convolution                   112 1×1×512 convolutions
84   'inception_4d-relu_1x1'          ReLU                          ReLU
85   'inception_4d-3x3_reduce'        Convolution                   144 1×1×512 convolutions
86   'inception_4d-relu_3x3_reduce'   ReLU                          ReLU
87   'inception_4d-3x3'               Convolution                   288 3×3×144 convolutions
88   'inception_4d-relu_3x3'          ReLU                          ReLU
89   'inception_4d-5x5_reduce'        Convolution                   32 1×1×512 convolutions w
90   'inception_4d-relu_5x5_reduce'   ReLU                          ReLU
91   'inception_4d-5x5'               Convolution                   64 5×5×32 convolutions wi
92   'inception_4d-relu_5x5'          ReLU                          ReLU
93   'inception_4d-pool'              Max Pooling                   3×3 max pooling with stri
94   'inception_4d-pool_proj'         Convolution                   64 1×1×512 convolutions w
95   'inception_4d-relu_pool_proj'    ReLU                          ReLU
96   'inception_4d-output'            Depth concatenation           Depth concatenation of 4
97   'inception_4e-1x1'               Convolution                   256 1×1×528 convolutions
98   'inception_4e-relu_1x1'          ReLU                          ReLU
99   'inception_4e-3x3_reduce'        Convolution                   160 1×1×528 convolutions
100   'inception_4e-relu_3x3_reduce'   ReLU                          ReLU
101   'inception_4e-3x3'               Convolution                   320 3×3×160 convolutions
102   'inception_4e-relu_3x3'          ReLU                          ReLU
103   'inception_4e-5x5_reduce'        Convolution                   32 1×1×528 convolutions
104   'inception_4e-relu_5x5_reduce'   ReLU                          ReLU
105   'inception_4e-5x5'               Convolution                   128 5×5×32 convolutions
106   'inception_4e-relu_5x5'          ReLU                          ReLU
107   'inception_4e-pool'              Max Pooling                   3×3 max pooling with str
```

```
    108   'inception_4e-pool_proj'         Convolution                   128 1×1×528 convolutions
    109   'inception_4e-relu_pool_proj'    ReLU                          ReLU
    110   'inception_4e-output'            Depth concatenation           Depth concatenation of 4
    111   'pool4-3x3_s2'                   Max Pooling                   3×3 max pooling with str
    112   'inception_5a-1x1'               Convolution                   256 1×1×832 convolutions
    113   'inception_5a-relu_1x1'          ReLU                          ReLU
    114   'inception_5a-3x3_reduce'        Convolution                   160 1×1×832 convolutions
    115   'inception_5a-relu_3x3_reduce'   ReLU                          ReLU
    116   'inception_5a-3x3'               Convolution                   320 3×3×160 convolutions
    117   'inception_5a-relu_3x3'          ReLU                          ReLU
    118   'inception_5a-5x5_reduce'        Convolution                   32 1×1×832 convolutions
    119   'inception_5a-relu_5x5_reduce'   ReLU                          ReLU
    120   'inception_5a-5x5'               Convolution                   128 5×5×32 convolutions
    121   'inception_5a-relu_5x5'          ReLU                          ReLU
    122   'inception_5a-pool'              Max Pooling                   3×3 max pooling with str
    123   'inception_5a-pool_proj'         Convolution                   128 1×1×832 convolutions
    124   'inception_5a-relu_pool_proj'    ReLU                          ReLU
    125   'inception_5a-output'            Depth concatenation           Depth concatenation of 4
    126   'inception_5b-1x1'               Convolution                   384 1×1×832 convolutions
    127   'inception_5b-relu_1x1'          ReLU                          ReLU
    128   'inception_5b-3x3_reduce'        Convolution                   192 1×1×832 convolutions
    129   'inception_5b-relu_3x3_reduce'   ReLU                          ReLU
    130   'inception_5b-3x3'               Convolution                   384 3×3×192 convolutions
    131   'inception_5b-relu_3x3'          ReLU                          ReLU
    132   'inception_5b-5x5_reduce'        Convolution                   48 1×1×832 convolutions
    133   'inception_5b-relu_5x5_reduce'   ReLU                          ReLU
    134   'inception_5b-5x5'               Convolution                   128 5×5×48 convolutions
    135   'inception_5b-relu_5x5'          ReLU                          ReLU
    136   'inception_5b-pool'              Max Pooling                   3×3 max pooling with str
    137   'inception_5b-pool_proj'         Convolution                   128 1×1×832 convolutions
    138   'inception_5b-relu_pool_proj'    ReLU                          ReLU
    139   'inception_5b-output'            Depth concatenation           Depth concatenation of 4
    140   'pool5-7x7_s1'                   2-D Global Average Pooling    2-D global average pooli
    141   'pool5-drop_7x7_s1'              Dropout                       40% dropout
    142   'newFC'                          Fully Connected               5 fully connected layer
    143   'newProb'                        Softmax                       softmax
    144   'newClassOutput'                 Classification Output         crossentropyex with 'Ma

### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwar
### Notice: The layer 'newClassOutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is i
### Compiling layer group: conv1-7x7_s2>>pool2-3x3_s2 ...
### Compiling layer group: conv1-7x7_s2>>pool2-3x3_s2 ... complete.
### Compiling layer group: inception_3a-1x1>>inception_3a-relu_1x1 ...
### Compiling layer group: inception_3a-1x1>>inception_3a-relu_1x1 ... complete.
### Compiling layer group: inception_3a-3x3_reduce>>inception_3a-relu_3x3 ...
### Compiling layer group: inception_3a-3x3_reduce>>inception_3a-relu_3x3 ... complete.
### Compiling layer group: inception_3a-5x5_reduce>>inception_3a-relu_5x5 ...
### Compiling layer group: inception_3a-5x5_reduce>>inception_3a-relu_5x5 ... complete.
### Compiling layer group: inception_3a-pool>>inception_3a-relu_pool_proj ...
### Compiling layer group: inception_3a-pool>>inception_3a-relu_pool_proj ... complete.
### Compiling layer group: inception_3b-1x1>>inception_3b-relu_1x1 ...
### Compiling layer group: inception_3b-1x1>>inception_3b-relu_1x1 ... complete.
### Compiling layer group: inception_3b-3x3_reduce>>inception_3b-relu_3x3 ...
### Compiling layer group: inception_3b-3x3_reduce>>inception_3b-relu_3x3 ... complete.
### Compiling layer group: inception_3b-5x5_reduce>>inception_3b-relu_5x5 ...
### Compiling layer group: inception_3b-5x5_reduce>>inception_3b-relu_5x5 ... complete.
### Compiling layer group: inception_3b-pool>>inception_3b-relu_pool_proj ...
### Compiling layer group: inception_3b-pool>>inception_3b-relu_pool_proj ... complete.
### Compiling layer group: pool3-3x3_s2 ...
### Compiling layer group: pool3-3x3_s2 ... complete.
### Compiling layer group: inception_4a-1x1>>inception_4a-relu_1x1 ...
### Compiling layer group: inception_4a-1x1>>inception_4a-relu_1x1 ... complete.
### Compiling layer group: inception_4a-3x3_reduce>>inception_4a-relu_3x3 ...
### Compiling layer group: inception_4a-3x3_reduce>>inception_4a-relu_3x3 ... complete.
### Compiling layer group: inception_4a-5x5_reduce>>inception_4a-relu_5x5 ...
### Compiling layer group: inception_4a-5x5_reduce>>inception_4a-relu_5x5 ... complete.
### Compiling layer group: inception_4a-pool>>inception_4a-relu_pool_proj ...
### Compiling layer group: inception_4a-pool>>inception_4a-relu_pool_proj ... complete.
### Compiling layer group: inception_4b-1x1>>inception_4b-relu_1x1 ...
### Compiling layer group: inception_4b-1x1>>inception_4b-relu_1x1 ... complete.
### Compiling layer group: inception_4b-3x3_reduce>>inception_4b-relu_3x3 ...
### Compiling layer group: inception_4b-3x3_reduce>>inception_4b-relu_3x3 ... complete.
### Compiling layer group: inception_4b-5x5_reduce>>inception_4b-relu_5x5 ...
### Compiling layer group: inception_4b-5x5_reduce>>inception_4b-relu_5x5 ... complete.
### Compiling layer group: inception_4b-pool>>inception_4b-relu_pool_proj ...
### Compiling layer group: inception_4b-pool>>inception_4b-relu_pool_proj ... complete.
### Compiling layer group: inception_4c-1x1>>inception_4c-relu_1x1 ...
### Compiling layer group: inception_4c-1x1>>inception_4c-relu_1x1 ... complete.
### Compiling layer group: inception_4c-3x3_reduce>>inception_4c-relu_3x3 ...
### Compiling layer group: inception_4c-3x3_reduce>>inception_4c-relu_3x3 ... complete.
### Compiling layer group: inception_4c-5x5_reduce>>inception_4c-relu_5x5 ...
### Compiling layer group: inception_4c-5x5_reduce>>inception_4c-relu_5x5 ... complete.
### Compiling layer group: inception_4c-pool>>inception_4c-relu_pool_proj ...
### Compiling layer group: inception_4c-pool>>inception_4c-relu_pool_proj ... complete.
### Compiling layer group: inception_4d-1x1>>inception_4d-relu_1x1 ...
### Compiling layer group: inception_4d-1x1>>inception_4d-relu_1x1 ... complete.
### Compiling layer group: inception_4d-3x3_reduce>>inception_4d-relu_3x3 ...
### Compiling layer group: inception_4d-3x3_reduce>>inception_4d-relu_3x3 ... complete.
### Compiling layer group: inception_4d-5x5_reduce>>inception_4d-relu_5x5 ...
### Compiling layer group: inception_4d-5x5_reduce>>inception_4d-relu_5x5 ... complete.
### Compiling layer group: inception_4d-pool>>inception_4d-relu_pool_proj ...
### Compiling layer group: inception_4d-pool>>inception_4d-relu_pool_proj ... complete.
### Compiling layer group: inception_4e-1x1>>inception_4e-relu_1x1 ...
### Compiling layer group: inception_4e-1x1>>inception_4e-relu_1x1 ... complete.
### Compiling layer group: inception_4e-3x3_reduce>>inception_4e-relu_3x3 ...
### Compiling layer group: inception_4e-3x3_reduce>>inception_4e-relu_3x3 ... complete.
### Compiling layer group: inception_4e-5x5_reduce>>inception_4e-relu_5x5 ...
### Compiling layer group: inception_4e-5x5_reduce>>inception_4e-relu_5x5 ... complete.
### Compiling layer group: inception_4e-pool>>inception_4e-relu_pool_proj ...
### Compiling layer group: inception_4e-pool>>inception_4e-relu_pool_proj ... complete.
### Compiling layer group: pool4-3x3_s2 ...
### Compiling layer group: pool4-3x3_s2 ... complete.
### Compiling layer group: inception_5a-1x1>>inception_5a-relu_1x1 ...
### Compiling layer group: inception_5a-1x1>>inception_5a-relu_1x1 ... complete.
### Compiling layer group: inception_5a-3x3_reduce>>inception_5a-relu_3x3 ...
### Compiling layer group: inception_5a-3x3_reduce>>inception_5a-relu_3x3 ... complete.
### Compiling layer group: inception_5a-5x5_reduce>>inception_5a-relu_5x5 ...
### Compiling layer group: inception_5a-5x5_reduce>>inception_5a-relu_5x5 ... complete.
### Compiling layer group: inception_5a-pool>>inception_5a-relu_pool_proj ...
### Compiling layer group: inception_5a-pool>>inception_5a-relu_pool_proj ... complete.
### Compiling layer group: inception_5b-1x1>>inception_5b-relu_1x1 ...
### Compiling layer group: inception_5b-1x1>>inception_5b-relu_1x1 ... complete.
### Compiling layer group: inception_5b-3x3_reduce>>inception_5b-relu_3x3 ...
### Compiling layer group: inception_5b-3x3_reduce>>inception_5b-relu_3x3 ... complete.
### Compiling layer group: inception_5b-5x5_reduce>>inception_5b-relu_5x5 ...
### Compiling layer group: inception_5b-5x5_reduce>>inception_5b-relu_5x5 ... complete.
### Compiling layer group: inception_5b-pool>>inception_5b-relu_pool_proj ...
### Compiling layer group: inception_5b-pool>>inception_5b-relu_pool_proj ... complete.
### Compiling layer group: pool5-7x7_s1 ...
### Compiling layer group: pool5-7x7_s1 ... complete.
### Compiling layer group: newFC ...
### Compiling layer group: newFC ... complete.

### Allocating external memory buffers:

          offset_name          offset_address    allocated_space
    _______________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "12.0 MB"
    "OutputResultOffset"        "0x00c00000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01000000"     "4.0 MB"
    "SystemBufferOffset"        "0x01400000"     "28.0 MB"
    "InstructionDataOffset"     "0x03000000"     "8.0 MB"
    "ConvWeightDataOffset"      "0x03800000"     "32.0 MB"
    "FCWeightDataOffset"        "0x05800000"     "4.0 MB"
    "EndOffset"                 "0x05c00000"     "Total: 92.0 MB"

### Network compilation complete.

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

Program Bitstream onto FPGA and Download Network Weights

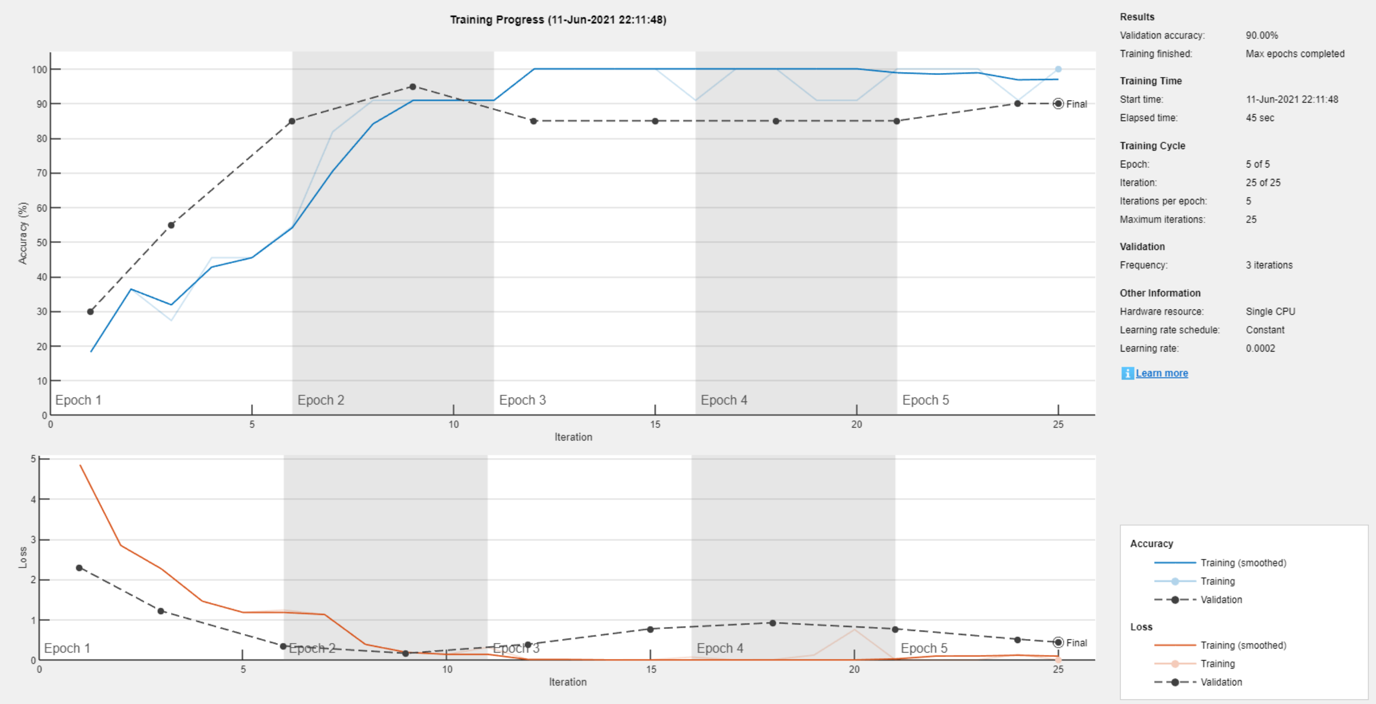
To deploy the network on the Intel Arria10 SoC hardware, run the deploy function of the `dlhdl.Workflow` object . This function uses the output of the compile function to program the FPGA board by using the programming file. The function also downloads the network weights and biases. The deploy function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

```
hW.deploy
```

```
### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 11-Jun-2021 22:20:12
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 11-Jun-2021 22:20:12
```

Load Example Image

```
I = imresize(readimage(imdsValidation,1),[224 224]);
figure
imshow(I)
```



Results	
Validation accuracy:	90.00%
Training finished:	Max epochs completed
Training Time	
Start time:	11-Jun-2021 22:11:48
Elapsed time:	45 sec
Training Cycle	
Epoch:	5 of 5
Iteration:	25 of 25
Iterations per epoch:	5
Maximum iterations:	25
Validation	
Frequency:	3 iterations
Other Information	
Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.0002

Retrieve Image Prediction

Execute the predict function of the `dlhdl.Workflow object` and display the prediction results.

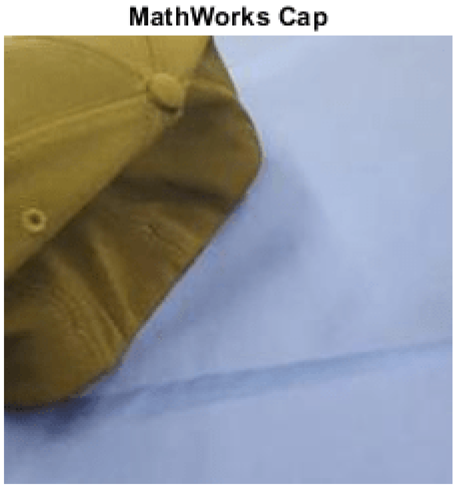
```
[prediction, speed] = hW.predict(single(I),'Profile','off');
```

```
### 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 ofd the deployed network by using the `predict` method with the `Profile` argument set to `on`.

```
[~, speed] = hW.predict(single(I),'Profile','on')
```

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


              Deep Learning Processor Profiler Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        -
Network                   15836394                  0.10558                       1           158
    conv1-7x7_s2           1139964                  0.00760
    pool1-3x3_s2            268928                  0.00179
    pool1-norm1             310985                  0.00207
    conv2-3x3_reduce        278740                  0.00186
    conv2-3x3               823735                  0.00549
    conv2-norm2             952105                  0.00635
    pool2-3x3_s2            273479                  0.00182
    inception_3a-1x1        198078                  0.00132
    inception_3a-3x3_reduce    280845                  0.00187
    inception_3a-3x3        196410                  0.00131
    inception_3a-5x5_reduce     73846                  0.00049
    inception_3a-5x5         35295                  0.00024
    inception_3a-pool        94554                  0.00063
    inception_3a-pool_proj    115223                  0.00077
    inception_3b-1x1        619945                  0.00413
    inception_3b-3x3_reduce    620509                  0.00414
    inception_3b-3x3        367297                  0.00245
    inception_3b-5x5_reduce    207909                  0.00139
    inception_3b-5x5        178552                  0.00119
```

```
   inception_3b-pool       179959                  0.00120
   inception_3b-pool_proj    344959                  0.00230
   pool3-3x3_s2            293640                  0.00196
   inception_4a-1x1        332992                  0.00222
   inception_4a-3x3_reduce    181829                  0.00121
   inception_4a-3x3         83777                  0.00056
   inception_4a-5x5_reduce     55639                  0.00037
   inception_4a-5x5         14500                  0.00010
   inception_4a-pool        77187                  0.00051
   inception_4a-pool_proj    130965                  0.00087
   inception_4b-1x1        300254                  0.00200
   inception_4b-3x3_reduce    220515                  0.00147
   inception_4b-3x3        101764                  0.00068
   inception_4b-5x5_reduce     73096                  0.00049
   inception_4b-5x5         25720                  0.00017
   inception_4b-pool        82277                  0.00055
   inception_4b-pool_proj    139530                  0.00093
   inception_4c-1x1        246715                  0.00164
   inception_4c-3x3_reduce    246987                  0.00165
   inception_4c-3x3        129291                  0.00086
   inception_4c-5x5_reduce     72855                  0.00049
   inception_4c-5x5         25444                  0.00017
   inception_4c-pool        82661                  0.00055
   inception_4c-pool_proj    139761                  0.00093
   inception_4d-1x1        220154                  0.00147
   inception_4d-3x3_reduce    273136                  0.00182
   inception_4d-3x3        159811                  0.00107
   inception_4d-5x5_reduce     86719                  0.00058
   inception_4d-5x5         32485                  0.00022
   inception_4d-pool        82309                  0.00055
   inception_4d-pool_proj    139464                  0.00093
   inception_4e-1x1        474515                  0.00316
   inception_4e-3x3_reduce    309661                  0.00206
   inception_4e-3x3        193442                  0.00129
   inception_4e-5x5_reduce     88661                  0.00059
   inception_4e-5x5         62881                  0.00042
   inception_4e-pool        85098                  0.00057
   inception_4e-pool_proj    254234                  0.00169
   pool4-3x3_s2            164072                  0.00109
   inception_5a-1x1        385821                  0.00257
   inception_5a-3x3_reduce    250827                  0.00167
   inception_5a-3x3         99439                  0.00066
   inception_5a-5x5_reduce     69697                  0.00046
   inception_5a-5x5         32465                  0.00022
   inception_5a-pool        53624                  0.00036
   inception_5a-pool_proj    205084                  0.00137
   inception_5b-1x1        567107                  0.00378
   inception_5b-3x3_reduce    295819                  0.00197
   inception_5b-3x3        139308                  0.00093
   inception_5b-5x5_reduce     92415                  0.00062
   inception_5b-5x5         46311                  0.00031
   inception_5b-pool        53882                  0.00036
   inception_5b-pool_proj    205632                  0.00137
   pool5-7x7_s1             69837                  0.00047
   newFC                    23215                  0.00015
* The clock frequency of the DL processor is: 150MHz
```

```
speed=75×5 table
                                   Latency(cycles)    Latency(seconds)    NumFrames    Total Late
                                   _______________    ________________    _________    __________

    Network                          1.5836e+07           0.10558            "1"          "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 = 'xcku11p-ffva1156-1-e';
hPC.estimateResources
```

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

              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2928             600            298560
                        -------------    -------------    -------------
DL_Processor                377( 13%)        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                     380( 16%)        586( 65%)       269176( 99%)
ReferenceDesign             3(  1%)         78(  9%)        35000( 13%)
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 estimateResources function returns the resource utilization for the reference design and for the deep learning processor configuration.

Supporting Files

Device Registration File

Use the dlhdl_device_registration.m file to register a custom device family. Estimate the resource utilization of the custom device by using the estimateResources function.

```
type dlhdl_device_registration.m
```

```
function hFPGADeviceFamily = dlhdl_device_registration
% Register a new device family by providing the following details:
% 1. Device Family Name
% 2. Vendor(Intel/Xilinx)
% 3. DSP Width
% 4. RAM Width
% 5. RAM Depth
% 6. SplitDSP Width(Optional) - alternative DSP Width supported by the DSP macro
% 7. SplitRAM Width(Optional) - alternative RAM Width supported by the RAM macro

hFPGADeviceFamily = { ...
    kintex_ultrascale();...
    };
end

function hFPGADeviceFamily = kintex_ultrascale()
    % Datasheets :
    % https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf
    % https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources
    hFPGADeviceFamily = hdlcoder.FPGADeviceInfo('Name', 'Kintex Ultrascale+');
    hFPGADeviceFamily.Vendor = 'Xilinx';
    hFPGADeviceFamily.DSPWidth = [27, 18];
    hFPGADeviceFamily.RAMWidth = 36;
    hFPGADeviceFamily.SplitRAMWidth = 18;
    hFPGADeviceFamily.RAMDepth = 1024;
end
```

Deep Learning Quantization

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:

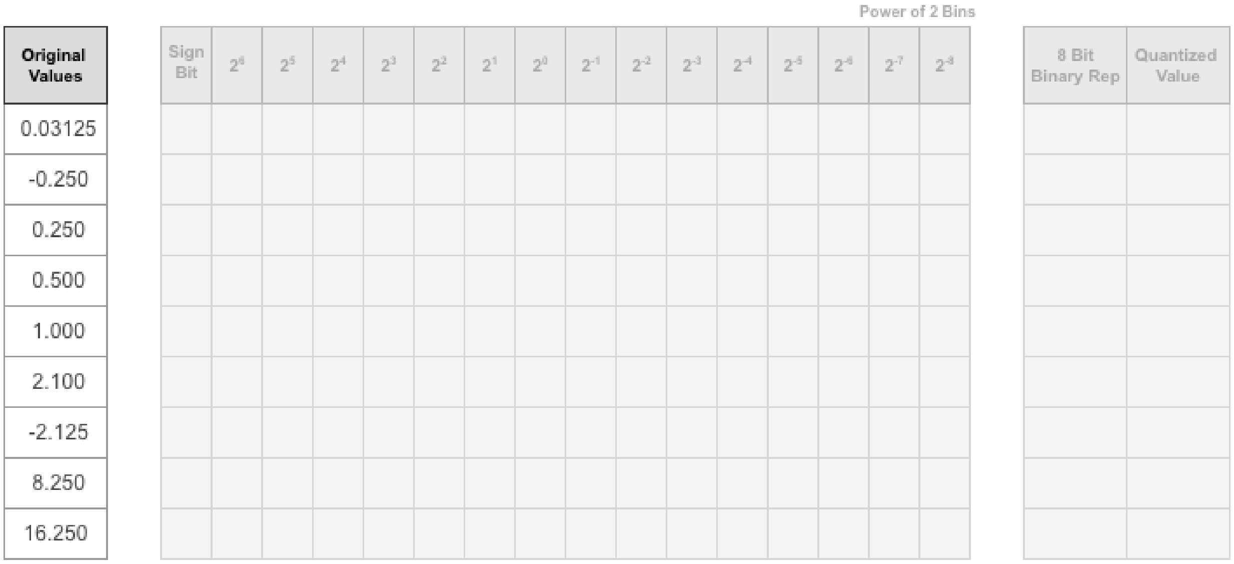
  - Positive numbers to the closest representable value in the direction of positive infinity.
  - Negative numbers to the closest representable value in the direction of negative infinity.

  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.

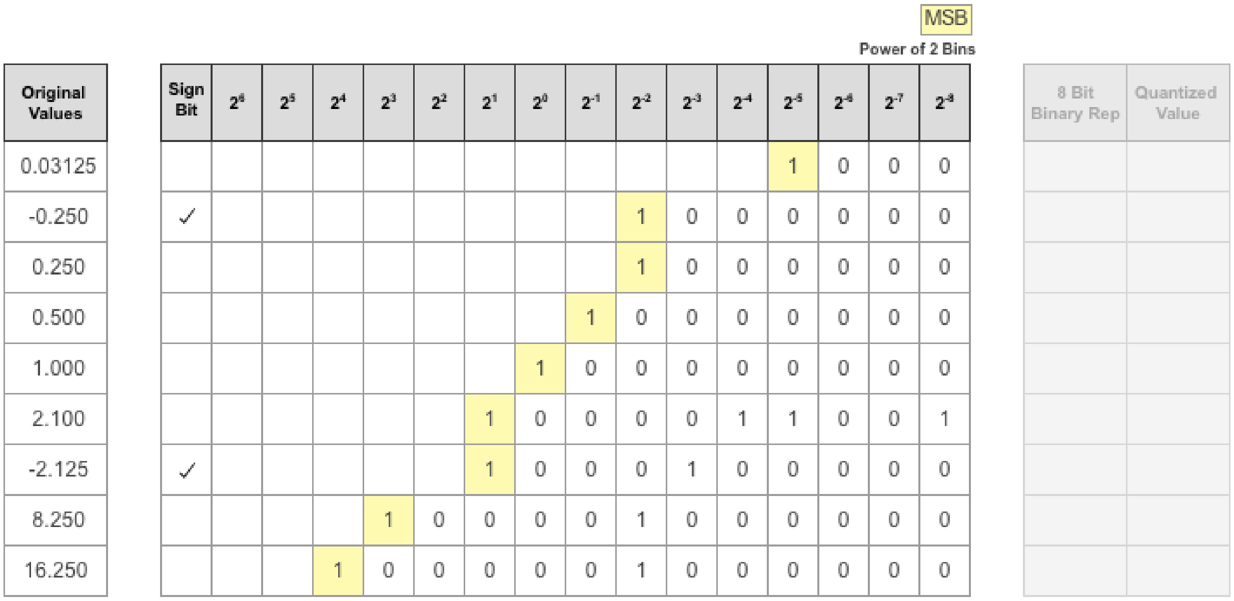


Power of 2 Bins

Original Values	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}	8 Bit Binary Rep	Quantized Value
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.

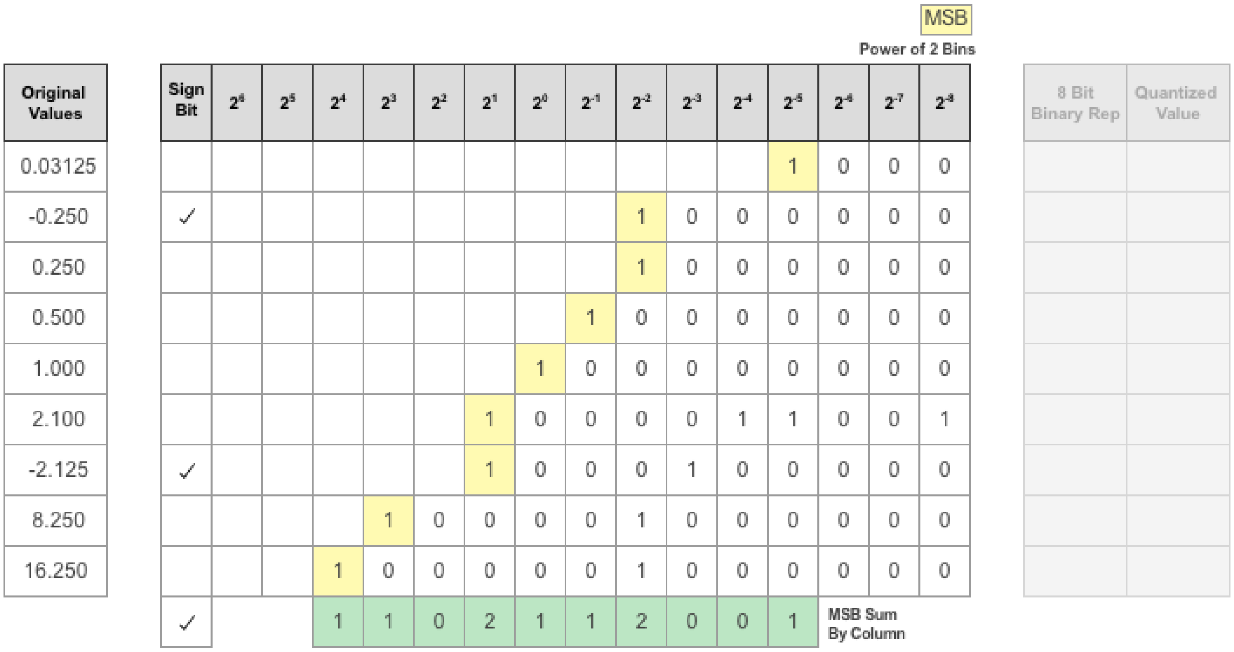


MSB

Power of 2 Bins

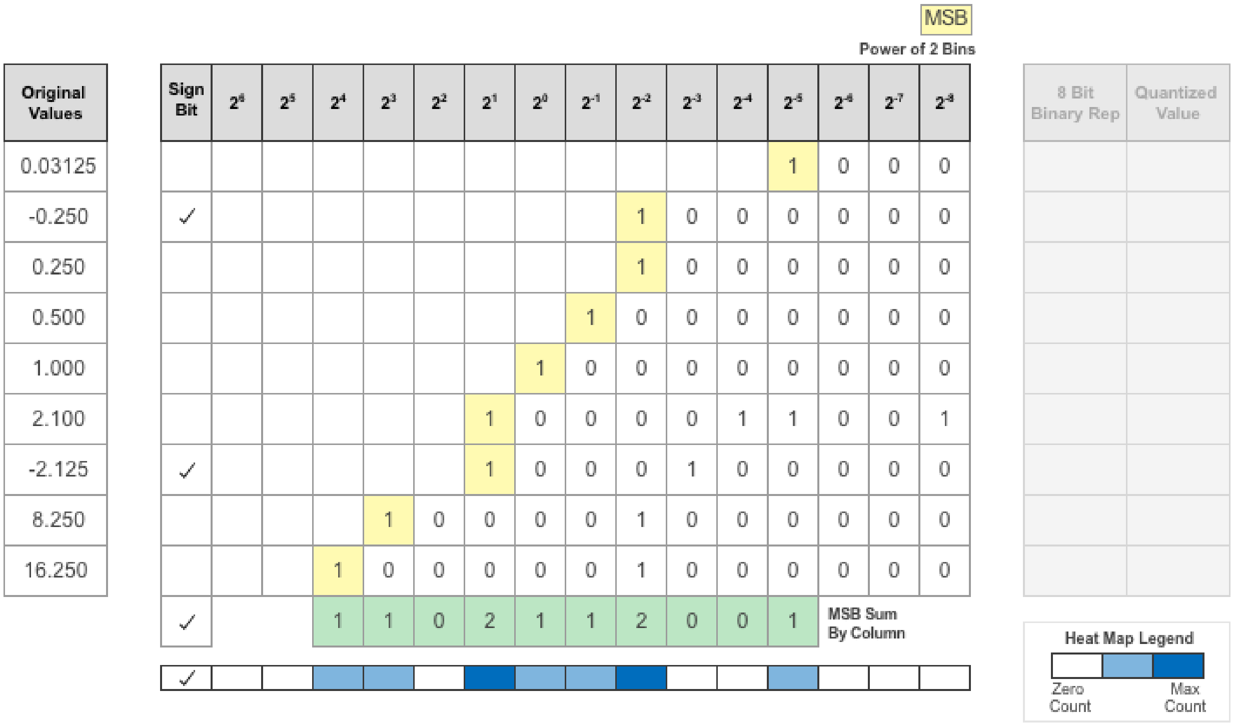
Original Values	Sign Bit	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	8 Bit Binary Rep	Quantized Value
0.03125													1	0	0	0		
-0.250	✓									1	0	0	0	0	0	0		
0.250										1	0	0	0	0	0	0		
0.500									1	0	0	0	0	0	0	0		
1.000								1	0	0	0	0	0	0	0	0		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

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.


MSB
Power of 2 Bins

Original Values	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}	8 Bit Binary Rep	Quantized Value
0.03125													1	0	0	0		
-0.250	✓									1	0	0	0	0	0	0		
0.250										1	0	0	0	0	0	0		
0.500									1	0	0	0	0	0	0	0		
1.000								1	0	0	0	0	0	0	0	0		
2.100							1	0	0	0	0	1	1	0	0	1		
-2.125	✓						1	0	0	0	1	0	0	0	0	0		
8.250					1	0	0	0	0	1	0	0	0	0	0	0		
16.250				1	0	0	0	0	0	1	0	0	0	0	0	0		
	✓			1	1	0	2	1	1	2	0	0	1	MSB Sum By Column				

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.


MSB
Power of 2 Bins

Original Values	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}	8 Bit Binary Rep	Quantized Value
0.03125													1	0	0	0		
-0.250	✓									1	0	0	0	0	0	0		
0.250										1	0	0	0	0	0	0		
0.500									1	0	0	0	0	0	0	0		
1.000								1	0	0	0	0	0	0	0	0		
2.100							1	0	0	0	0	1	1	0	0	1		
-2.125	✓						1	0	0	0	1	0	0	0	0	0		
8.250					1	0	0	0	0	1	0	0	0	0	0	0		
16.250				1	0	0	0	0	0	1	0	0	0	0	0	0		
	✓			1	1	0	2	1	1	2	0	0	1	MSB Sum By Column				
	✓	□	□	▨	▨	□	■	▨	▨	■	□	□	▨	□	□	□		

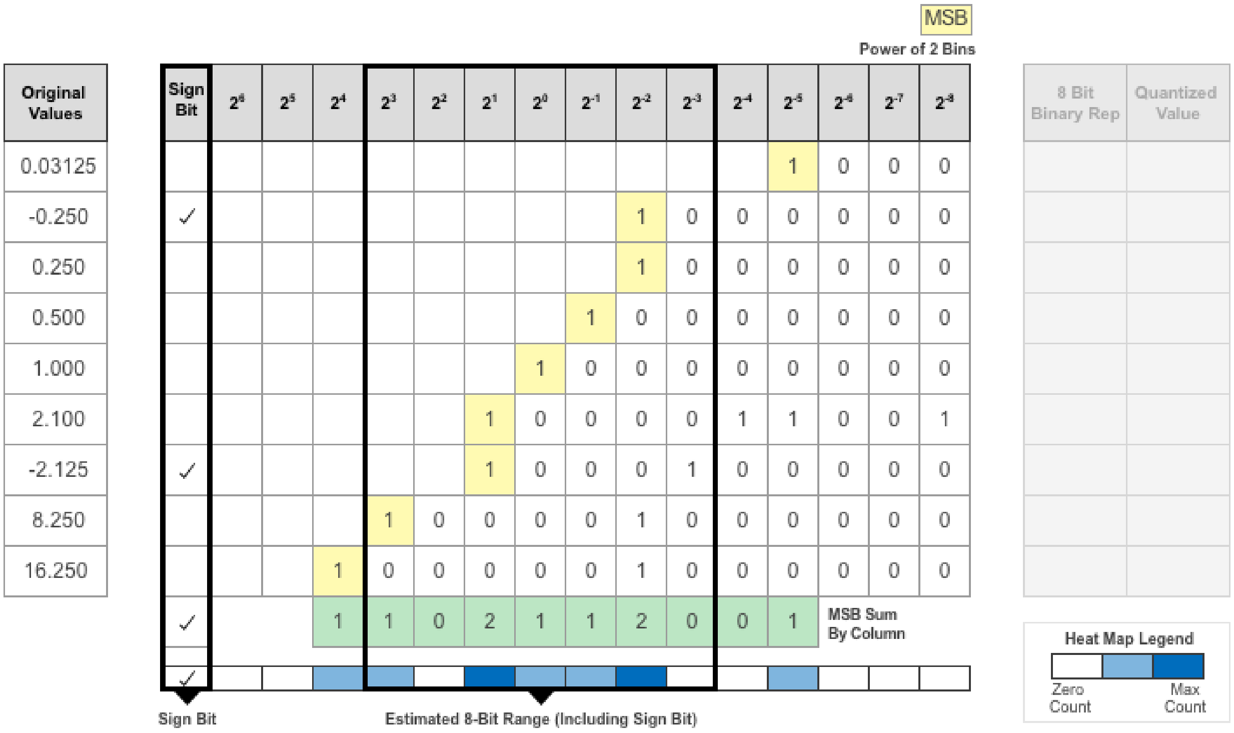
Heat Map Legend

□ ▨ ■

Zero Count — Max Count

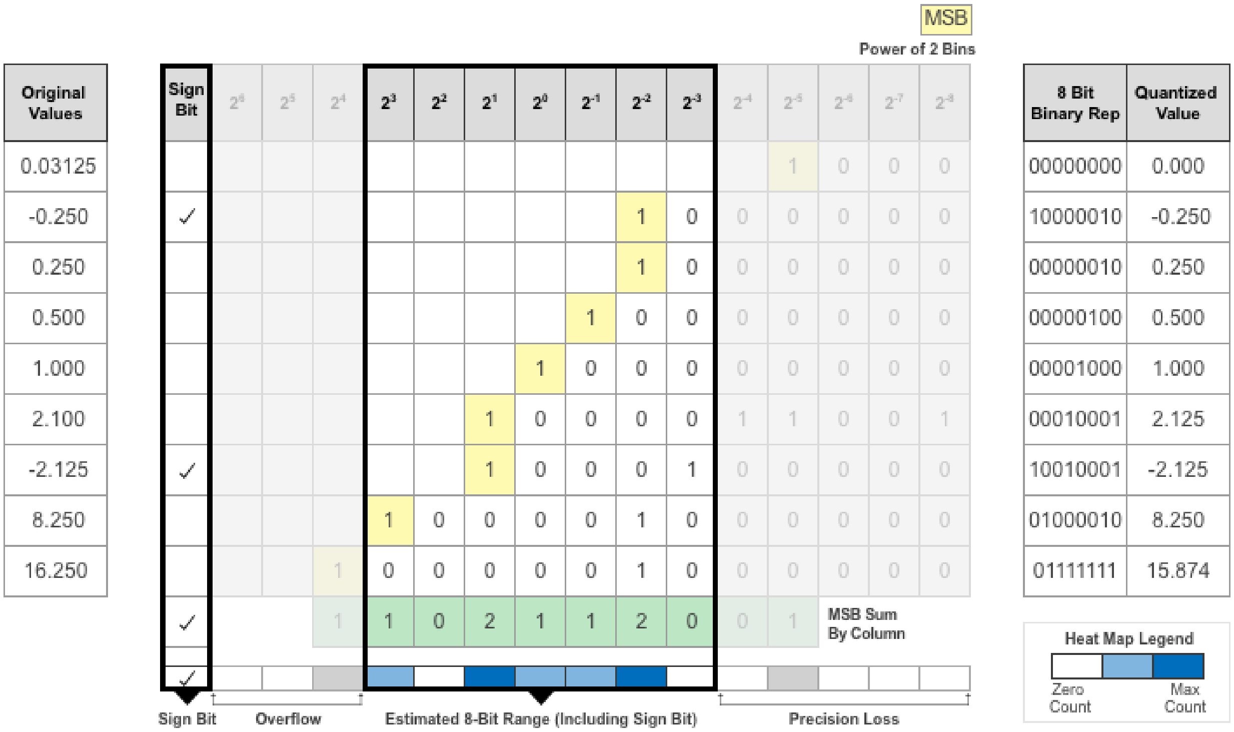
5 The **Deep Network Quantizer** app assigns a data type that can avoid overflow, cover the range, and allow underflow. An additional sign bit is required to represent the signedness of the value.

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



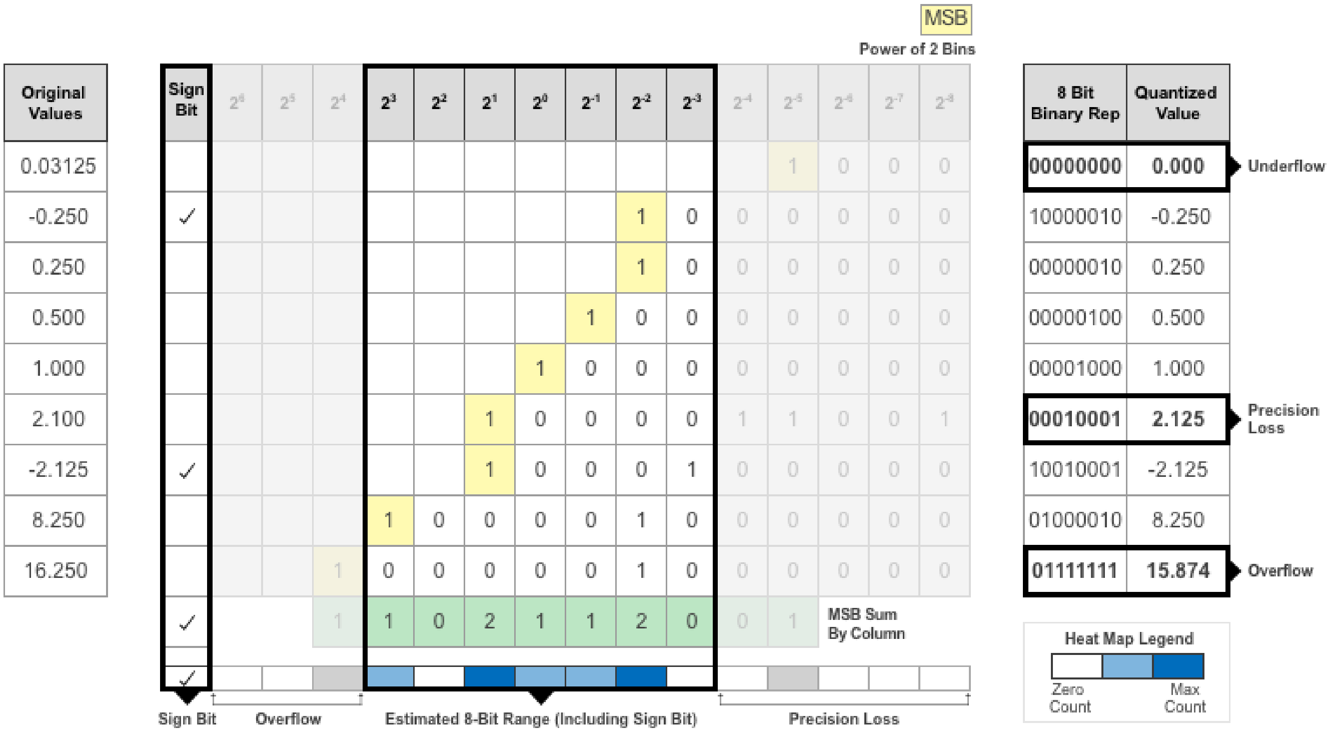
Original Values	Sign Bit	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	8 Bit Binary Rep	Quantized Value
0.03125													1	0	0	0		
-0.250	✓									1	0	0	0	0	0	0		
0.250										1	0	0	0	0	0	0		
0.500									1	0	0	0	0	0	0	0		
1.000								1	0	0	0	0	0	0	0	0		
2.100							1	0	0	0	0	1	1	0	0	1		
-2.125	✓						1	0	0	0	1	0	0	0	0	0		
8.250					1	0	0	0	0	1	0	0	0	0	0	0		
16.250				1	0	0	0	0	0	1	0	0	0	0	0	0		
	✓			1	1	0	2	1	1	2	0	0	1	MSB Sum By Column				
	✓	□	□	▨	▨	□	■	▨	▨	■	□	□	▨	□	□	□		

6 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.



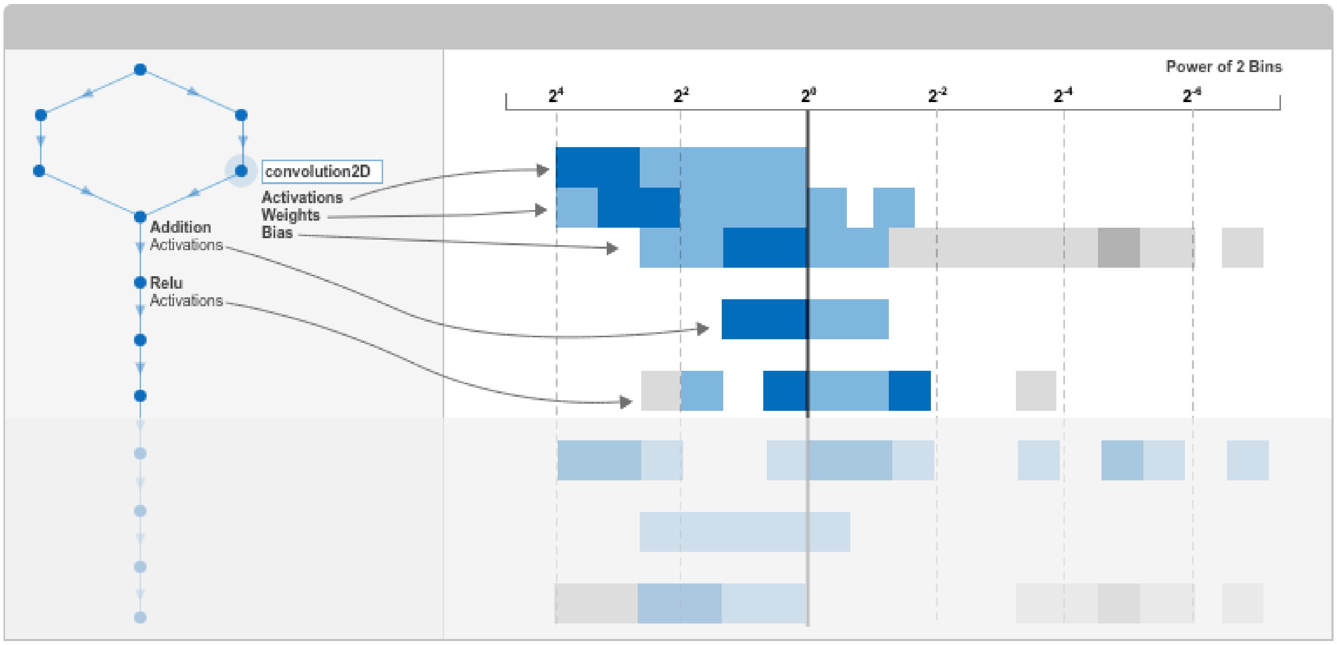
Original Values	Sign Bit	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	8 Bit Binary Rep	Quantized Value
0.03125													1	0	0	0	00000000	0.000
-0.250	✓									1	0	0	0	0	0	0	10000010	-0.250
0.250										1	0	0	0	0	0	0	00000010	0.250
0.500									1	0	0	0	0	0	0	0	00000100	0.500
1.000								1	0	0	0	0	0	0	0	0	00001000	1.000
2.100							1	0	0	0	0	1	1	0	0	1	00010001	2.125
-2.125	✓						1	0	0	0	1	0	0	0	0	0	10010001	-2.125
8.250					1	0	0	0	0	1	0	0	0	0	0	0	01000010	8.250
16.250				1	0	0	0	0	0	1	0	0	0	0	0	0	01111111	15.874
	✓			1	1	0	2	1	1	2	0	0	1	MSB Sum By Column				
	✓																	

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.



Original Values	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}	8 Bit Binary Rep	Quantized Value	
0.03125													1	0	0	0	00000000	0.000	Underflow
-0.250	✓									1	0	0	0	0	0	0	10000010	-0.250	
0.250										1	0	0	0	0	0	0	00000010	0.250	
0.500									1	0	0	0	0	0	0	0	00000100	0.500	
1.000								1	0	0	0	0	0	0	0	0	00001000	1.000	
2.100							1	0	0	0	0	1	1	0	0	1	00010001	2.125	Precision Loss
-2.125	✓						1	0	0	0	1	0	0	0	0	0	10010001	-2.125	
8.250					1	0	0	0	0	1	0	0	0	0	0	0	01000010	8.250	
16.250				1	0	0	0	0	0	1	0	0	0	0	0	0	01111111	15.874	Overflow
	✓			1	1	0	2	1	1	2	0	0	1	MSB Sum By Column					

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**
Setup Toolkit Environment	`hdlsetuptoolpath` (HDL Coder) The quantization workflow does not support the `MinGW C/C++` compiler. Use `Microsoft Visual C++ 2017` or `Microsoft Visual C++ 2015`. For a list of supported compilers, see Supported and Compatible Compilers.	“Setting Up the Prerequisite Products” (GPU Coder) The quantization workflow does not support the `MinGW C/C++` compiler. Use `Microsoft Visual C++ 2017` or `Microsoft Visual C++ 2015`. For a list of supported compilers, see Supported and Compatible Compilers.	“Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder) The quantization workflow does not support the `MinGW C/C++` compiler. Use `Microsoft Visual C++ 2017` or `Microsoft Visual C++ 2015`. For a list of supported compilers, see Supported and Compatible Compilers.
Required Products	• Deep Learning Toolbox • Deep Learning HDL Toolbox	Deep Learning Toolbox	Deep Learning Toolbox
Required Support Packages	• 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
Required Add Ons	MATLAB Coder™ Interface for Deep Learning Libraries	• 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

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)
Deployment	Deep Learning HDL Toolbox	GPU Coder	MATLAB Coder

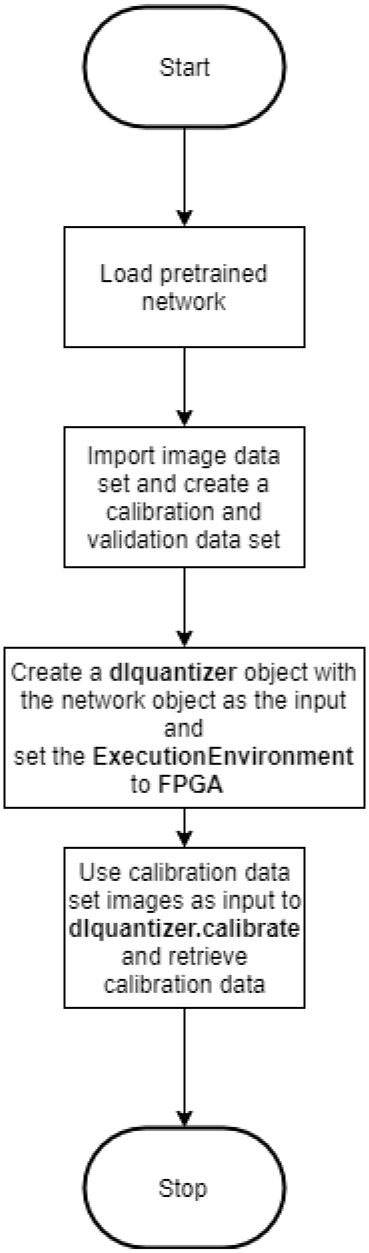
Calibration

Workflow

Collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the quantized network and the dynamic ranges of the activations in all layers.

The `calibrate` method uses the collected dynamic ranges to generate an exponents file. The `dlhdl.Workflow` class `compile` method uses the exponents file to generate a configuration file that contains the weights and biases of the quantized network.

This workflow is the workflow to calibrate your quantized series deep learning network.


Start
Load pretrained network
Import image data set and create a calibration and validation data set
Create a **dlquantizer** object with the network object as the input and set the **ExecutionEnvironment** to **FPGA**
Use calibration data set images as input to **dlquantizer.calibrate** and retrieve calibration data
Stop

See Also

`calibrate` | `dlquantizationOptions` | `dlquantizer` | `validate`

More About

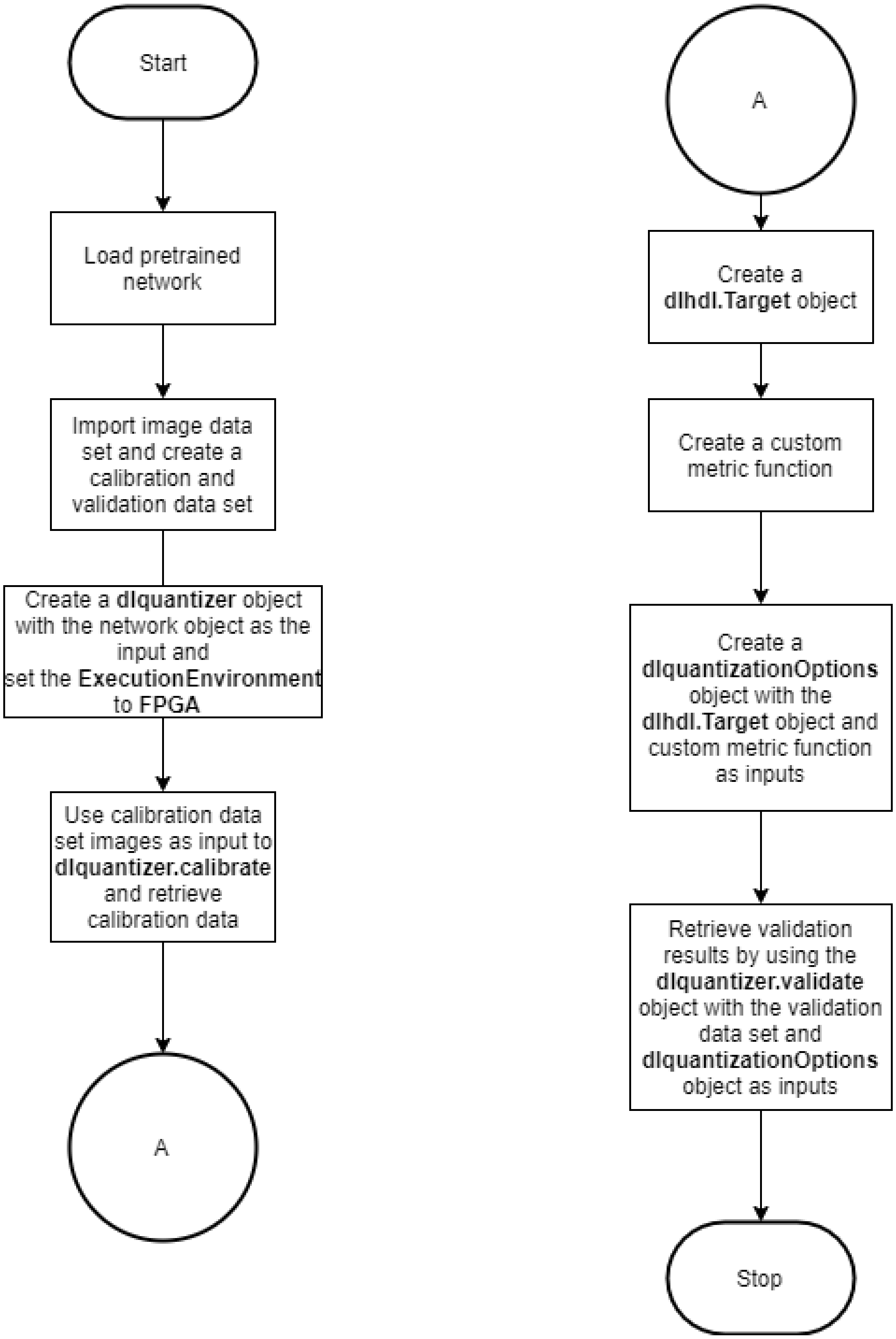
- “Quantization of Deep Neural Networks” on page 11-2
- “Validation” on page 11-14
- “Code Generation and Deployment” on page 11-17

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.


Start
Load pretrained network
Import image data set and create a calibration and validation data set
Create a **dlquantizer** object with the network object as the input and set the **ExecutionEnvironment** to **FPGA**
Use calibration data set images as input to **dlquantizer.calibrate** and retrieve calibration data
A
A
Create a **dlhdl.Target** object
Create a custom metric function
Create a **dlquantizationOptions** object with the **dlhdl.Target** object and custom metric function as inputs
Retrieve validation results by using the **dlquantizer.validate** object with the validation data set and **dlquantizationOptions** object as inputs
Stop

See Also

`validate` | `dlquantizationOptions` | `dlquantizer`

More About

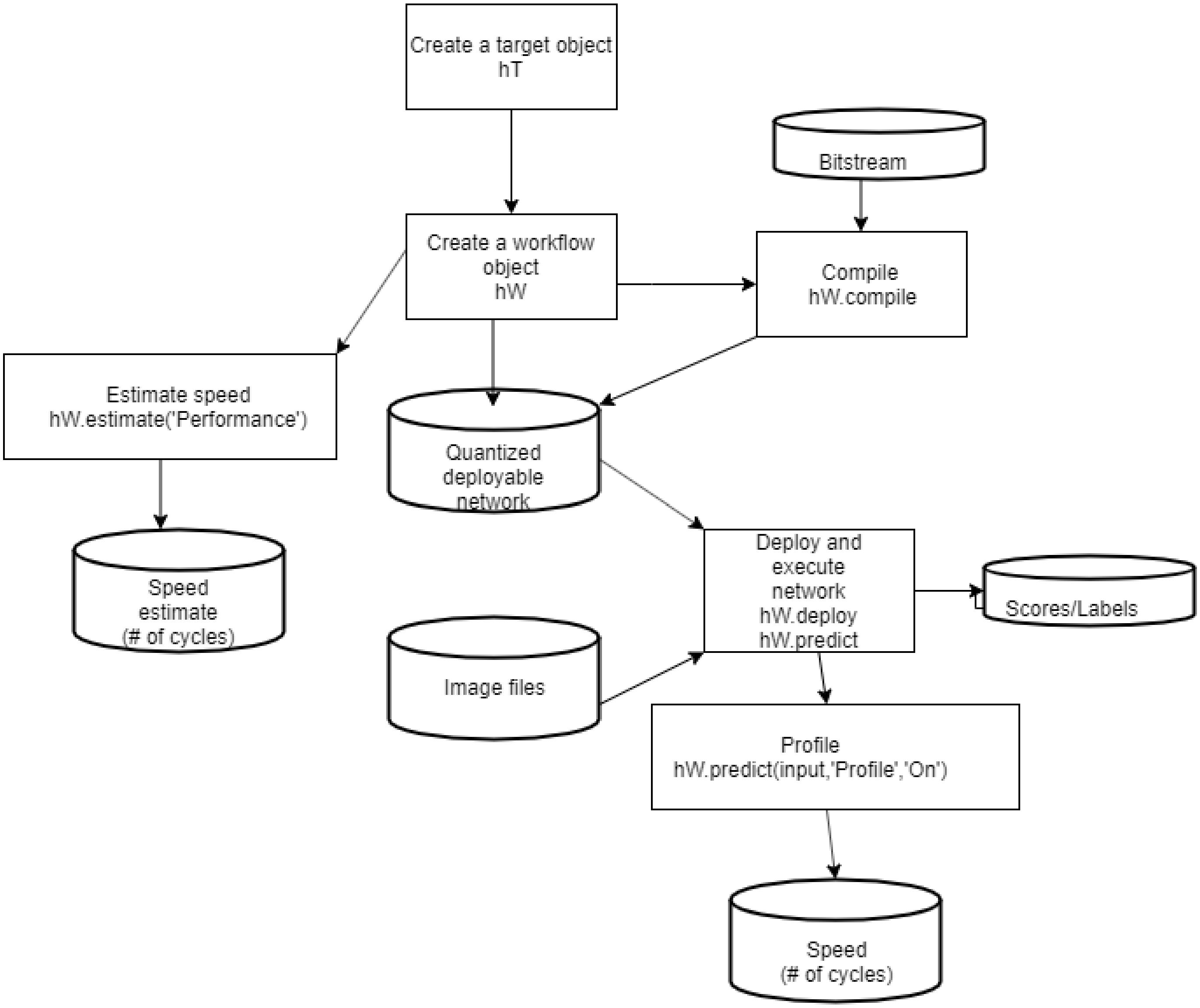
- “Quantization of Deep Neural Networks” on page 11-2
- “Calibration” on page 11-12
- “Code Generation and Deployment” on page 11-17

Code Generation and Deployment

To generated code for and deploy your quantized deep learning network, create an object of class `dlhdl.Workflow`. Use this object to accomplish tasks such as:

- Compile and deploy the quantized deep learning network on a target FPGA or SoC board by using the `deploy` function.
- Estimate the speed of the quantized deep learning network in terms of number of frames per second by using the `estimate` function.
- Execute the deployed quantized deep learning network and predict the classification of input images by using the `predict` function.
- Calculate the speed and profile of the deployed quantized deep learning network by using the `predict` function. Set the `Profile` parameter to `on`.

This figure illustrates the workflow to deploy your quantized deep learning network to the FPGA boards.



See Also

`dlhdl.Workflow` | `dlhdl.Target` | `dlquantizer`

More About

- "Quantization of Deep Neural Networks" on page 11-2
- "Calibration" on page 11-12
- "Validation" on page 11-14

Deep Learning Processor IP Core User Guide

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`).

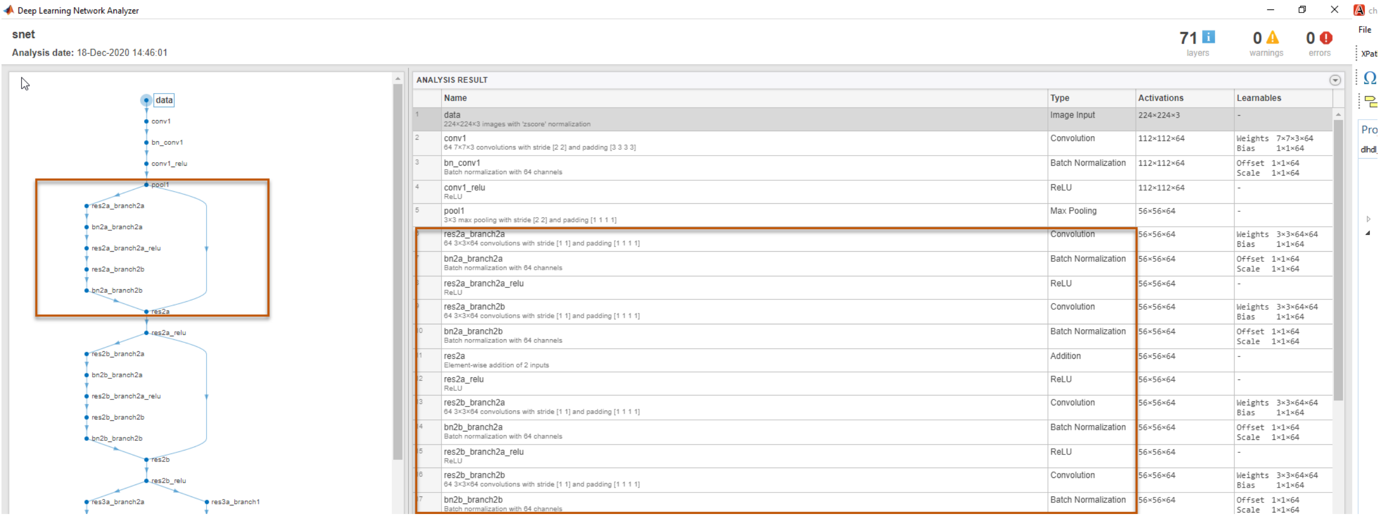
- 2-D zero padding layer (`nnet.keras.layer.ZeroPadding2dLayer`) and 2-D convolution layer (`convolution2dLayer`).
- 2-D zero padding layer (`nnet.keras.layer.ZeroPadding2dLayer`) and 2-D max polling layer (`maxPooling2dLayer`).

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.



Deep Learning Network Analyzer

snet

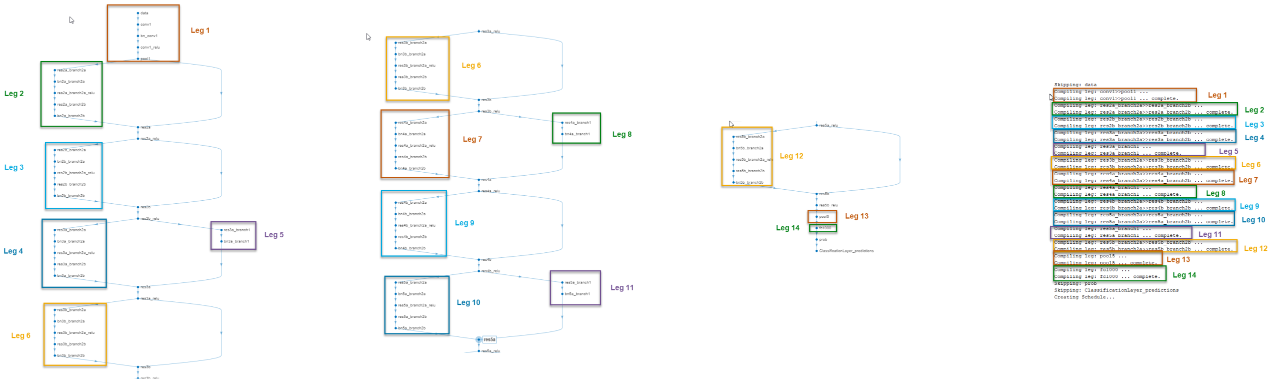
Analysis date: 18-Dec-2020 14:46:01

71 layers 0 warnings 0 errors

ANALYSIS RESULT

	Name	Type	Activations	Learnables
1	data 224×224×3 images with 'zscore' normalization	Image Input	224×224×3	-
2	conv1 64 7×7×3 convolutions with stride [2 2] and padding [3 3 3 3]	Convolution	112×112×64	Weights 7×7×3×64 Bias 1×1×64
3	bn_conv1 Batch normalization with 64 channels	Batch Normalization	112×112×64	Offset 1×1×64 Scale 1×1×64
4	conv1_relu ReLU	ReLU	112×112×64	-
5	pool1 3×3 max pooling with stride [2 2] and padding [1 1 1 1]	Max Pooling	56×56×64	-
6	res2a_branch2a 64 3×3×64 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	56×56×64	Weights 3×3×64×64 Bias 1×1×64
7	bn2a_branch2a Batch normalization with 64 channels	Batch Normalization	56×56×64	Offset 1×1×64 Scale 1×1×64
8	res2a_branch2a_relu ReLU	ReLU	56×56×64	-
9	res2a_branch2b 64 3×3×64 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	56×56×64	Weights 3×3×64×64 Bias 1×1×64
10	bn2a_branch2b Batch normalization with 64 channels	Batch Normalization	56×56×64	Offset 1×1×64 Scale 1×1×64
11	res2a Element-wise addition of 2 inputs	Addition	56×56×64	-
12	res2a_relu ReLU	ReLU	56×56×64	-
13	res2b_branch2a 64 3×3×64 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	56×56×64	Weights 3×3×64×64 Bias 1×1×64
14	bn2b_branch2a Batch normalization with 64 channels	Batch Normalization	56×56×64	Offset 1×1×64 Scale 1×1×64
15	res2b_branch2a_relu ReLU	ReLU	56×56×64	-
16	res2b_branch2b 64 3×3×64 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	56×56×64	Weights 3×3×64×64 Bias 1×1×64
17	bn2b_branch2b Batch normalization with 64 channels	Batch Normalization	56×56×64	Offset 1×1×64 Scale 1×1×64

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.



```
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_branch2a>>res3a_branch2b ...
Compiling leg: res3a_branch2a>>res3a_branch2b ... complete.
Compiling leg: res3a_branch1 ...
Compiling leg: res3a_branch1 ... complete.
Compiling leg: res3b_branch2a>>res3b_branch2b ...
Compiling leg: res3b_branch2a>>res3b_branch2b ... complete.
Compiling leg: res4a_branch2a>>res4a_branch2b ...
Compiling leg: res4a_branch2a>>res4a_branch2b ... complete.
Compiling leg: res4a_branch1 ...
Compiling leg: res4a_branch1 ... complete.
Compiling leg: res4b_branch2a>>res4b_branch2b ...
Compiling leg: res4b_branch2a>>res4b_branch2b ... complete.
Compiling leg: res5a_branch2a>>res5a_branch2b ...
Compiling leg: res5a_branch2a>>res5a_branch2b ... complete.
Compiling leg: res5a_branch1 ...
Compiling leg: res5a_branch1 ... complete.
Compiling leg: res5b_branch2a>>res5b_branch2b ...
Compiling leg: res5b_branch2a>>res5b_branch2b ... complete.
Compiling leg: pool5 ...
Compiling leg: pool5 ... complete.
Compiling leg: fc1000 ...
Compiling leg: fc1000 ... complete.
Skipping: prob
Skipping: ClassificationLayer_predictions
Creating Schedule...
```

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

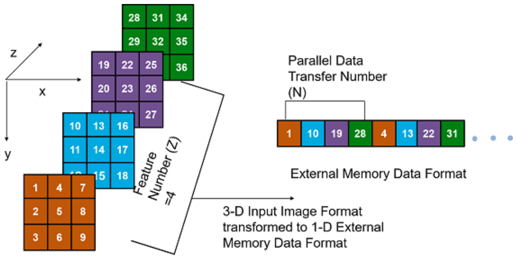
- `Parallel Data Transfer Number` refers to the number of pixels that are transferred every clock cycle through the AXI master interface. Use the letter `N` in place of the `Parallel Data Transfer Number`. Mathematically `N` is the square root of the `ConvThreadNumber`. See “ConvThreadNumber”.
- `Feature 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.

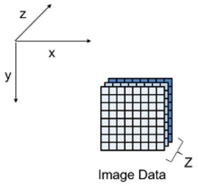
The image demonstrates how the data stored in a 3-by-3-by-4 matrix is translated into a 1-by-36 matrix that is then stored in the external memory.


z
x
y
28 31 34
29 32 35
36
19 22 25
20 23 26
27
10 13 16
11 14 17
15 18
1 4 7
2 5 8
3 6 9
Feature Number (Z) =4
Parallel Data Transfer Number (N)
1 10 19 28 4 13 22 31
External Memory Data Format
3-D Input Image Format transformed to 1-D External Memory Data Format

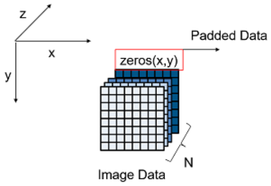
When the image `Feature Number` (Z) is not a multiple of the `Parallel Data Transfer Number` (N), then we must pad a zeroes matrix of size x-by-y along the z dimension of the matrix to make the image `Z` value a multiple of `N`.

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.

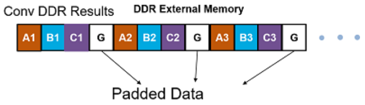
This image is the input image format before padding.



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.

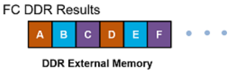


If your deep learning processor consists of only a convolution (conv) processing module, the output external data is using the conv module external data format, which means it possibly contains padded data if your output Z value is not a multiple of the N value. The padded data is removed when you use 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

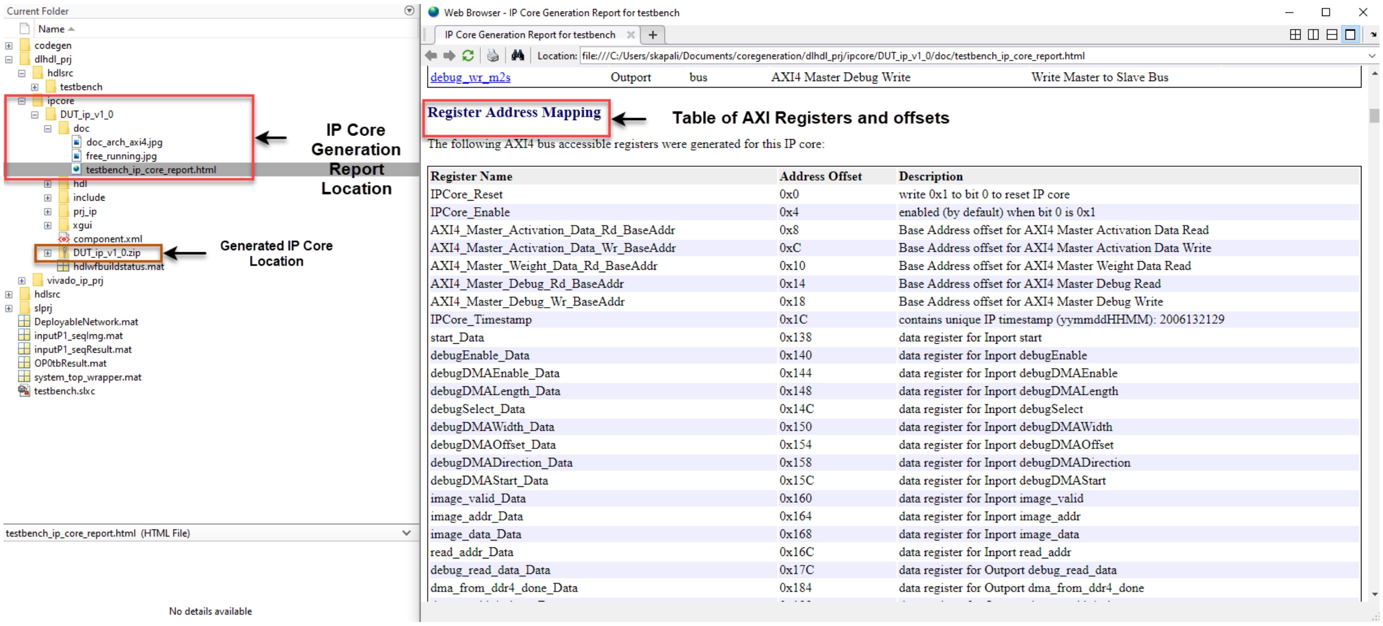
See Also

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.


Current Folder
Name
codegen
dlhdl_prj
hdlsrc
testbench
ipcore
DUT_ip_v1_0
doc
doc_arch_axi4.jpg
free_running.jpg
testbench_ip_core_report.html
hdl
include
prj_ip
xgui
component.xml
DUT_ip_v1_0.zip
hdlwfbuildstatus.mat
vivado_ip_prj
hdlsrc
slprj
DeployableNetwork.mat
inputP1_seqImg.mat
inputP1_seqResult.mat
OP0tbResult.mat
system_top_wrapper.mat
testbench.slxc
testbench_ip_core_report.html (HTML File)
No details available
IP Core Generation Report Location
Generated IP Core Location
Web Browser - IP Core Generation Report for testbench
IP Core Generation Report for testbench
Location: file:///C:/Users/skapali/Documents/coregeneration/dlhdl_prj/ipcore/DUT_ip_v1_0/doc/testbench_ip_core_report.html
Table of AXI Registers and offsets

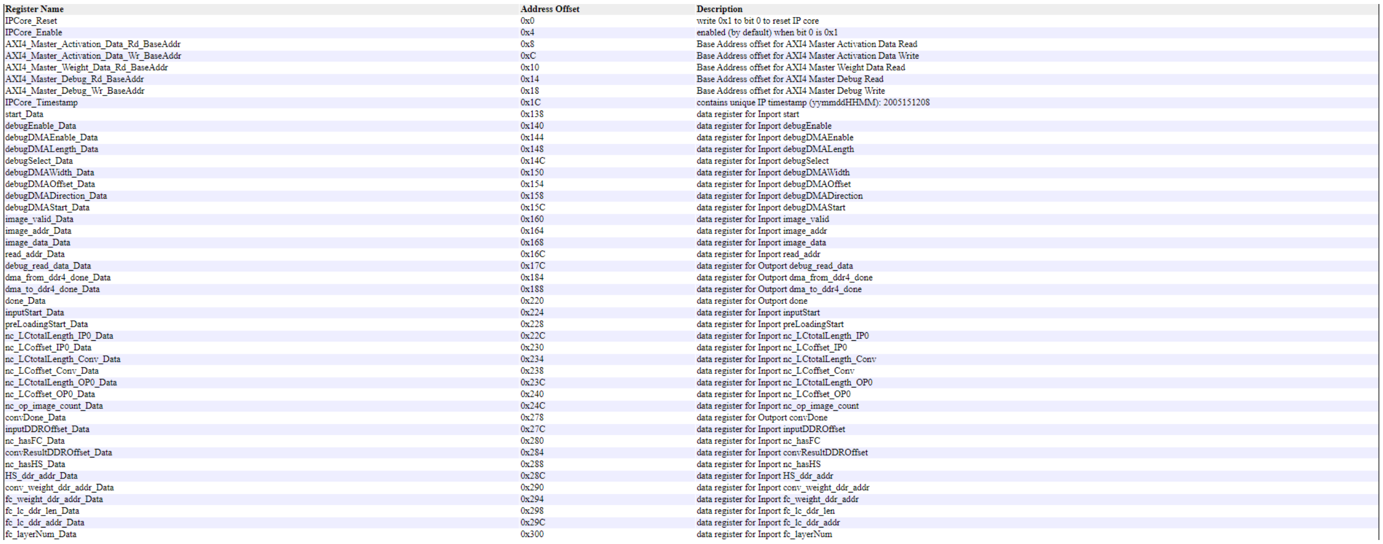
debug_wr_m2s	Outport	bus	AXI4 Master Debug Write	Write Master to Slave Bus

Register Address Mapping

The following AXI4 bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
AXI4_Master_Activation_Data_Rd_BaseAddr	0x8	Base Address offset for AXI4 Master Activation Data Read
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 Inport start
debugEnable_Data	0x140	data register for Inport debugEnable
debugDMAEnable_Data	0x144	data register for Inport debugDMAEnable
debugDMALength_Data	0x148	data register for Inport debugDMALength
debugSelect_Data	0x14C	data register for Inport debugSelect
debugDMAWidth_Data	0x150	data register for Inport debugDMAWidth
debugDMAOffset_Data	0x154	data register for Inport debugDMAOffset
debugDMADirection_Data	0x158	data register for Inport debugDMADirection
debugDMAStart_Data	0x15C	data register for Inport debugDMAStart
image_valid_Data	0x160	data register for Inport image_valid
image_addr_Data	0x164	data register for Inport image_addr
image_data_Data	0x168	data register for Inport image_data
read_addr_Data	0x16C	data register for Inport read_addr
debug_read_data_Data	0x17C	data register for Outport debug_read_data
dma_from_ddr4_done_Data	0x184	data register for Outport 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 Inport start
debugEnable_Data	0x140	data register for Inport debugEnable
debugDMAEnable_Data	0x144	data register for Inport debugDMAEnable
debugDMALength_Data	0x148	data register for Inport debugDMALength
debugSelect_Data	0x14C	data register for Inport debugSelect
debugDMAWidth_Data	0x150	data register for Inport debugDMAWidth
debugDMAOffset_Data	0x154	data register for Inport debugDMAOffset
debugDMADirection_Data	0x158	data register for Inport debugDMADirection
debugDMAStart_Data	0x15C	data register for Inport debugDMAStart
image_valid_Data	0x160	data register for Inport image_valid
image_addr_Data	0x164	data register for Inport image_addr
image_data_Data	0x168	data register for Inport image_data
read_addr_Data	0x16C	data register for Inport read_addr
debug_read_data_Data	0x17C	data register for Outport debug_read_data
dma_from_ddr4_done_Data	0x184	data register for Outport dma_from_ddr4_done
dma_to_ddr4_done_Data	0x188	data register for Outport dma_to_ddr4_done
done_Data	0x220	data register for Outport done
inputStart_Data	0x224	data register for Inport inputStart
preLoadingStart_Data	0x228	data register for Inport preLoadingStart
nc_LCtotalLength_IP0_Data	0x22C	data register for Inport nc_LCtotalLength_IP0
nc_LCoffset_IP0_Data	0x230	data register for Inport nc_LCoffset_IP0
nc_LCtotalLength_Conv_Data	0x234	data register for Inport nc_LCtotalLength_Conv
nc_LCoffset_Conv_Data	0x238	data register for Inport nc_LCoffset_Conv
nc_LCtotalLength_OP0_Data	0x23C	data register for Inport nc_LCtotalLength_OP0
nc_LCoffset_OP0_Data	0x240	data register for Inport nc_LCoffset_OP0
nc_op_image_count_Data	0x24C	data register for Inport nc_op_image_count
convDone_Data	0x278	data register for Outport convDone
inputDDROffset_Data	0x27C	data register for Inport inputDDROffset
nc_hasFC_Data	0x280	data register for Inport nc_hasFC
convResultDDROffset_Data	0x284	data register for Inport convResultDDROffset
nc_hasHS_Data	0x288	data register for Inport nc_hasHS
HS_ddr_addr_Data	0x28C	data register for Inport HS_ddr_addr
conv_weight_ddr_addr_Data	0x290	data register for Inport conv_weight_ddr_addr
fc_weight_ddr_addr_Data	0x294	data register for Inport fc_weight_ddr_addr
fc_lc_ddr_len_Data	0x298	data register for Inport fc_lc_ddr_len
fc_lc_ddr_addr_Data	0x29C	data register for Inport fc_lc_ddr_addr
fc_layerNum_Data	0x300	data register for Inport 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 and handshaking signals. Handshaking signals are used only when the data processing mode is set to streaming mode.

Run-Time Signals

This table lists the run-time signals, data types, interface types, and description.

Signal Name	Data Type	Interface Type (Port or Register)	Description
`Done`	logical	`register`	Signal indicating that the deep learning processor IP core has processed all input data and written the last output to memory.
`InputStart`	logical	`register`	Signal from the user to the deep learning processor IP core to start processing the data.
`FrameCount`	integer	`register`	Signal from the user to the deep learning processor IP core specifying the number of input data frames.
`StreamingMode`	logical	`register`	Signal from the user to the deep learning processor IP core specifying the data processing mode. `false` selects buffer mode and `true` selects streaming mode.
`StreamingDone`	logical	`register`	Signal to test streaming mode. During testing, the signal becomes true when you retrieve the last output.
`InputStop`	logical	`register`	Signal to stop the continuous streaming mode. To stop the continuous streaming mode, set this signal to `true`.

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.

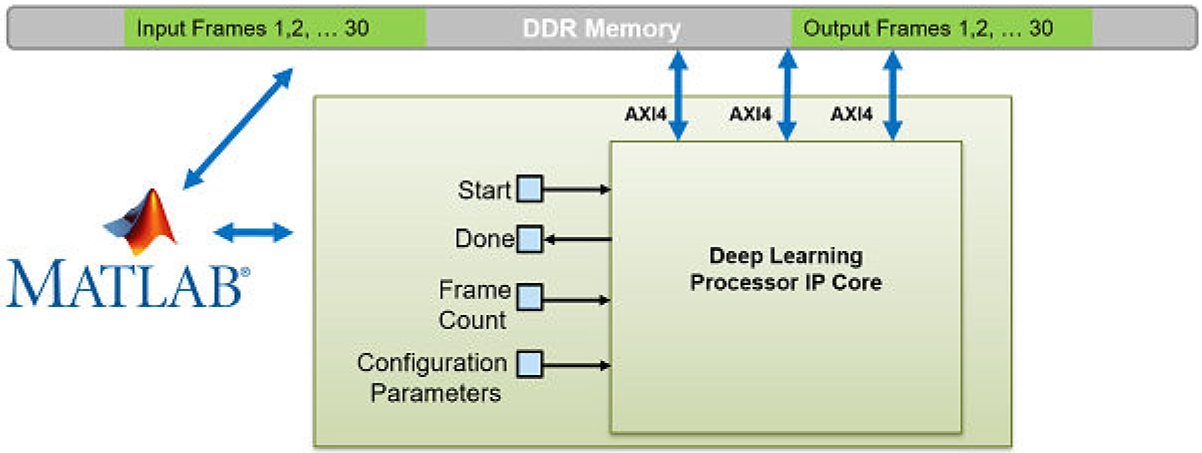
Signal Name	**Data Type**	**InputRunTimeControl or OutputRunTimeControl**	**Interface Type (Port or Register)**	**Description**
InputAddr	uint32	InputRunTimeControl	port/register	Signal indicating the address location in memory for loading the input data. Use this signal when the InputValid signal is high.
InputNext	logical	InputRunTimeControl	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.
InputSize	uint32	InputRunTimeControl	port/register	Signal indicating the size of the next input data frame. Use this signal when the InputValid signal is high.
InputValid	logical	InputRunTimeControl	port/register	Signal from the deep learning processor IP core indicating that the input data is valid.
OutputAddr	uint32	OutputRunTimeControl	port/registerr	Signal indicating the address location in memory from where to retrieve the output data. Use this signal when the OutputValid signal is high.

OutputNext	logical	OutputRunTimeControl	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.
OutputSize	uint32	OutputRunTimeControl	port/register	Signal indicating the size of the next output data frame. Use this signal when the OutputValid signal is high.
OutputValid	logical	OutputRunTimeControl	port/register	Signal from the deep learning processor IP core indicating that the output data is valid.

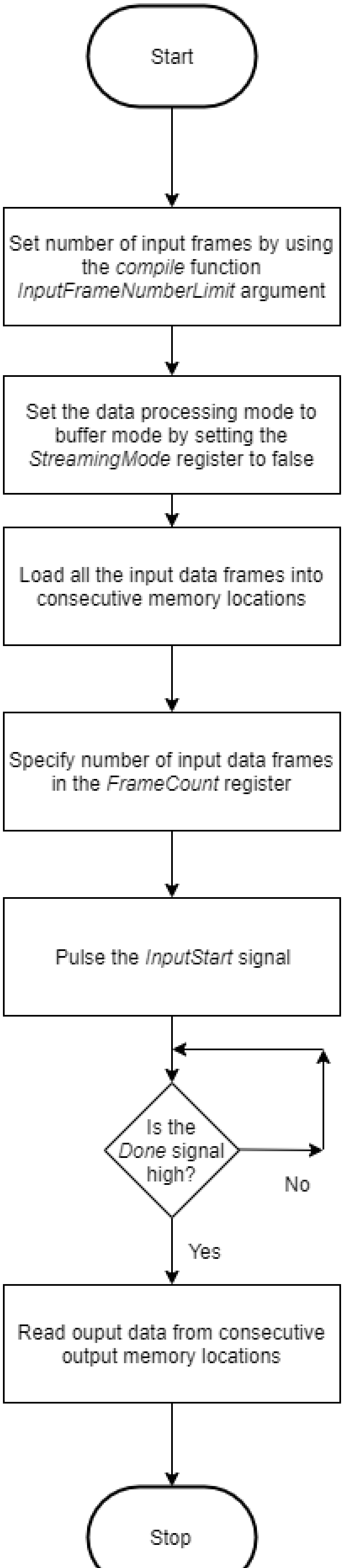
Design Batch Processing Mode Interface

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.

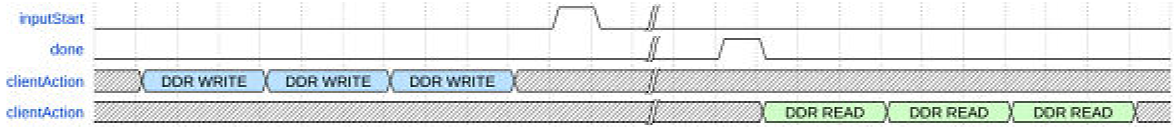
To process a single data frame set the FrameCount register value to one.



This flowchart shows the operation of the batch processing mode.


Start
Set number of input frames by using the *compile* function *InputFrameNumberLimit* argument
Set the data processing mode to buffer mode by setting the *StreamingMode* register to false
Load all the input data frames into consecutive memory locations
Specify number of input data frames in the *FrameCount* register
Pulse the *InputStart* signal
Is the *Done* signal high?
No
Yes
Read ouput data from consecutive output memory locations
Stop

This timing diagram shows the operation of the batch processing mode.



Load all the data frames into consecutive input DDR memory locations, toggle the `inputStart` signal, wait for the `done` signal to go high, and then read the output data from the consecutive output DDR memory locations. The `clientAction` signals represent your actions of loading input data and reading output data into the DDR memory.

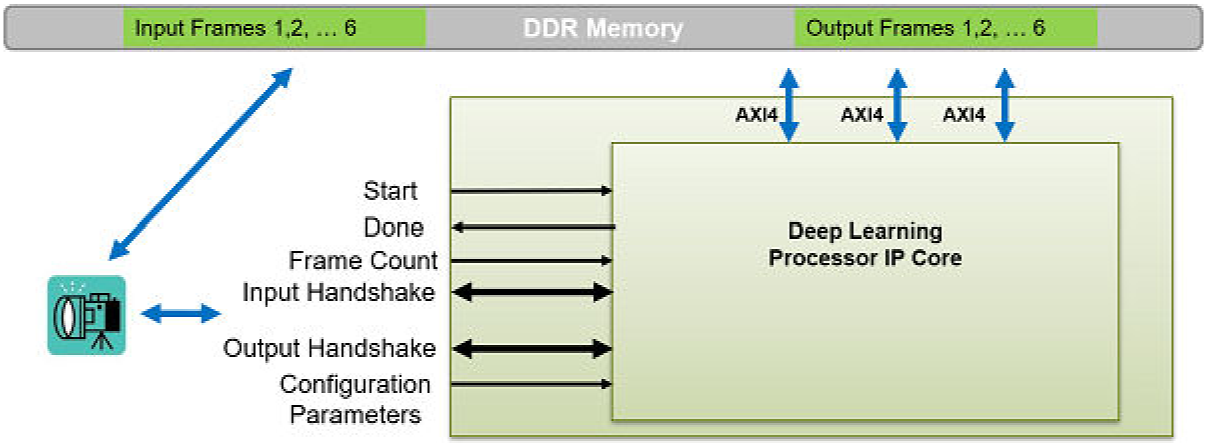
Design Streaming Mode Interface

When your input data is streaming in, when you have access to limited double data rate (DDR) memory space, and when your application requires handshaking protocols, process multiple frames by using the streaming mode. The figure shows the generated deep learning processor IP core with interface signals for the streaming mode of operation. In this figure, the live camera streams data to an image preprocessing design under test (DUT) that implements the streaming mode handshaking protocol to interact with the generated deep learning processor IP core.

Date can be streamed to the deep learning processor IP core in two modes:

- Stream Data up to a frame count value— In this mode the deep learning processor processes data frames up to the value specified in `FrameCount`. After processing all the frames the deep learning processor IP core sets the `Done` signal to `true`.

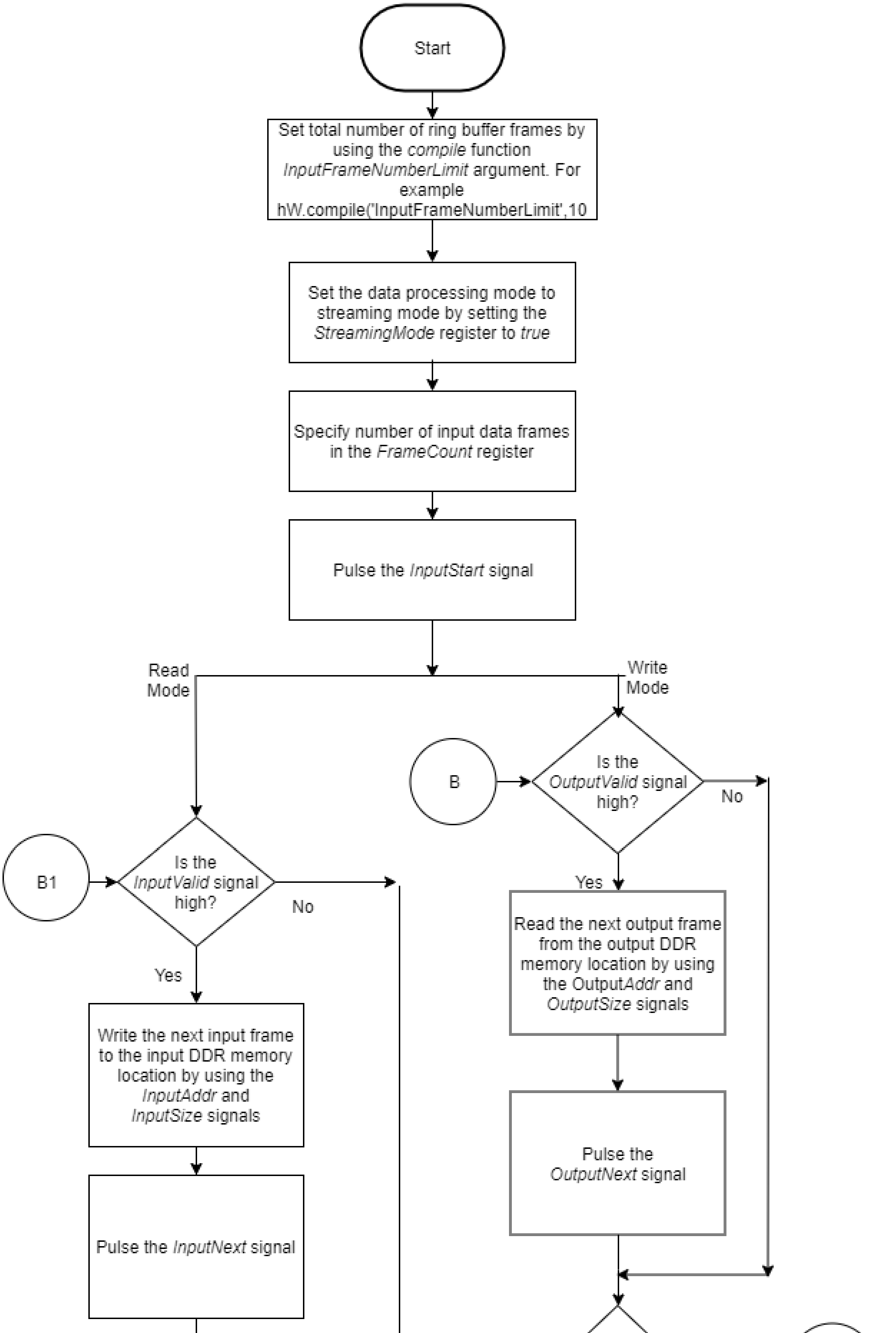
  To process a single data frame set the `FrameCount` register value to one.
- Continuous streaming mode— In this mode the deep learning processor IP core processes data frames until you set the `InputStop` value to `true`.



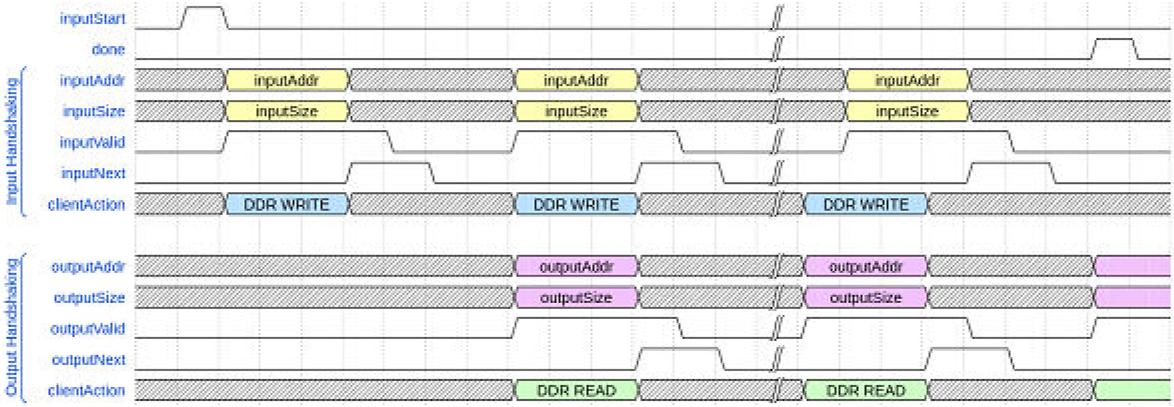
Streaming Mode up to a Frame Count

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.


Start
Set total number of ring buffer frames by using the compile function InputFrameNumberLimit argument. For example hW.compile('InputFrameNumberLimit',10
Set the data processing mode to streaming mode by setting the StreamingMode register to true
Specify number of input data frames in the FrameCount register
Pulse the InputStart signal
Read Mode
Write Mode
B
Is the OutputValid signal high?
No
Yes
Read the next output frame from the output DDR memory location by using the OutputAddr and OutputSize signals
Pulse the OutputNext signal
B1
Is the InputValid signal high?
No
Yes
Write the next input frame to the input DDR memory location by using the InputAddr and InputSize signals
Pulse the InputNext signal

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



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

   a. Wait for the `inputValid` signal to become `true` and then:

      - Use the `inputAddr` and `inputSize` signals to write the next input data frame to DDR memory.
      - Pulse the `inputNext` signal.

   b. Wait for the `outputValid` signal to become `true` and then:

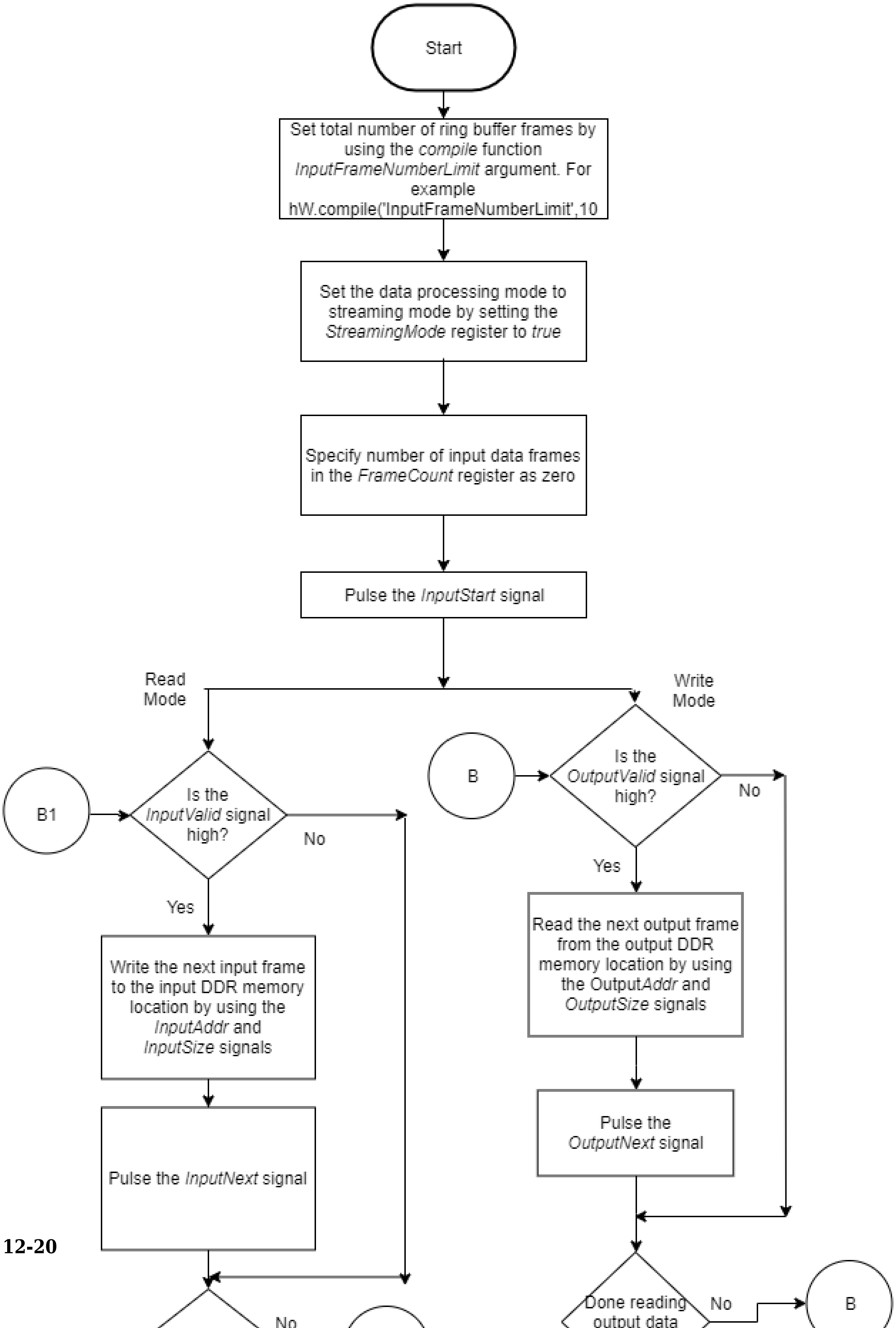
      - Use the `outputAddr` and `outputSize` signals to read the processed output data frame.
      - Pulse the `outputNext` signal.
5. Once the deep learning processor IP core has processed all the frames it sets the `done` signal to `true`.

The `clientAction` signals represent your actions of loading input data and reading output data into the DDR memory.

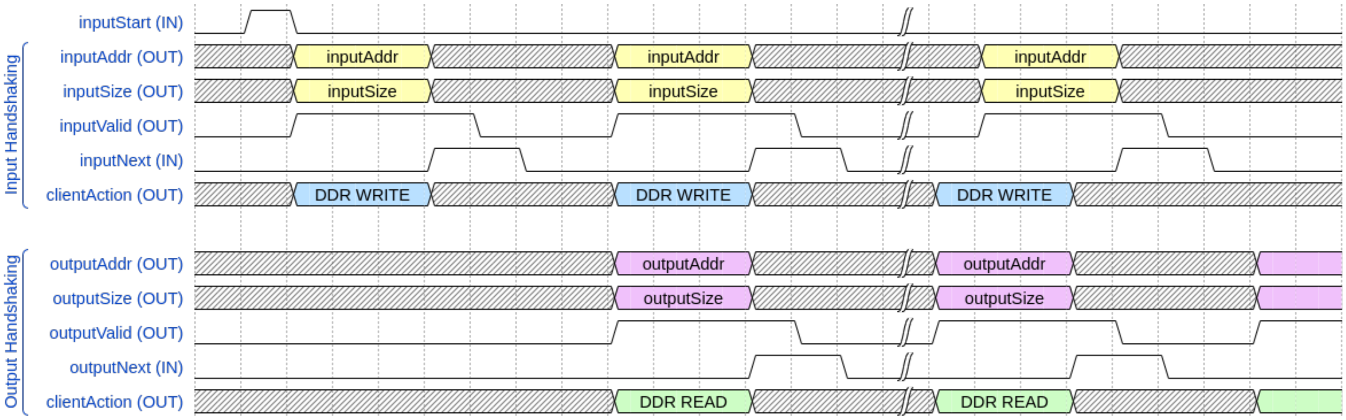
Continuous Streaming Mode

You can continuously stream data to the deep learning processor in continuous streaming mode. To use the continuous streaming mode, set the `FrameCount` to zero. To stop the data processing set the `InputStop` signal to `true`.

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


Start
Set total number of ring buffer frames by using the *compile* function *InputFrameNumberLimit* argument. For example hW.compile('InputFrameNumberLimit',10
Set the data processing mode to streaming mode by setting the *StreamingMode* register to *true*
Specify number of input data frames in the *FrameCount* register as zero
Pulse the *InputStart* signal
Read Mode
Write Mode
B1
Is the *InputValid* signal high?
No
Yes
Write the next input frame to the input DDR memory location by using the *InputAddr* and *InputSize* signals
Pulse the *InputNext* signal
No
B
Is the *OutputValid* signal high?
No
Yes
Read the next output frame from the output DDR memory location by using the Output*Addr* and *OutputSize* signals
Pulse the *OutputNext* signal
Done reading output data
No
B

This timing diagram shows the operation of the continuous streaming mode.



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

   a. Wait for the `inputValid` signal to become `true` and then:

      - Use the `inputAddr` and `inputSize` signals to write the next input data frame to DDR memory.
      - Pulse the `inputNext` signal.

   b. Wait for the `outputValid` signal to become `true` and then:

      - Use the `outputAddr` and `outputSize` signals to read the processed output data frame.
      - Pulse the `outputNext` signal.
5. Once you have written all the input data and read all the output data pulse the `InputStop` signal.

Access Data from DDR

The deep learning IP core uses the three AXI4 Master interfaces to store and process:

- Activation data
- Weight data
- Debug data

The deep learning processor reads and writes data from the DDR based on the data processing mode of operation by using these AXI4 Master interfaces.

See Also

“Deep Learning Processor IP Core” on page 12-2 | “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

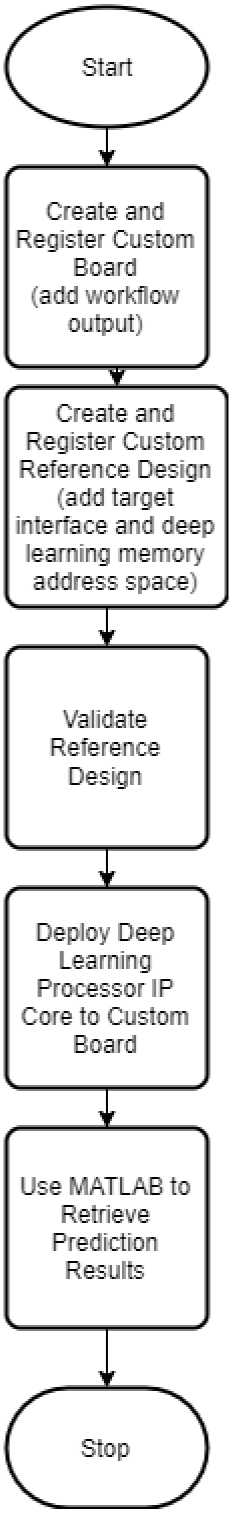
This example shows how to define and register a custom board and reference design for the deep learning processor IP core workflow. If you have a system that has preprocessing and postprocessing design under test (DUT) blocks, you can iterate on those blocks to improve your performance and resource utilization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


Start
Create and Register Custom Board (add workflow output)
Create and Register Custom Reference Design (add target interface and deep learning memory address space)
Validate Reference Design
Deploy Deep Learning Processor IP Core to Custom Board
Use MATLAB to Retrieve Prediction Results
Stop

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.FPGAPackage  = '';
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_PostCreateProje

% 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)