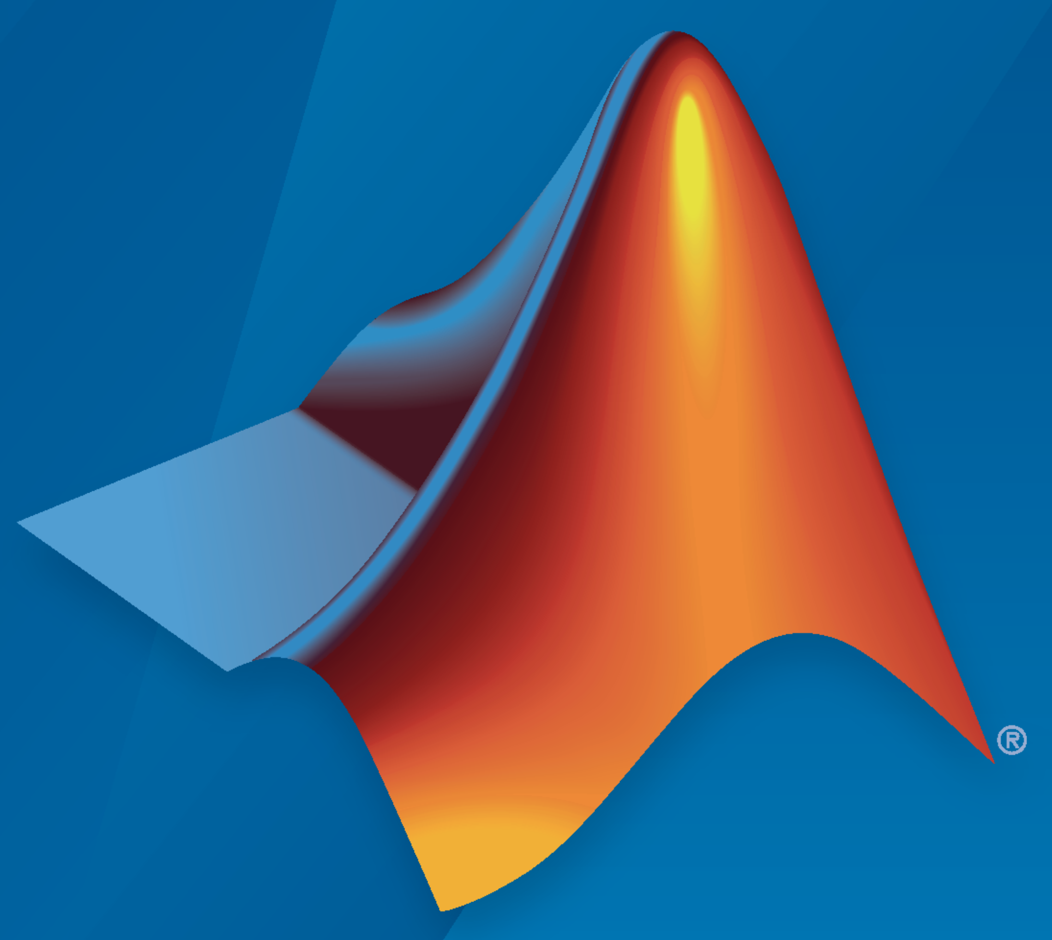
Deep Learning HDL Toolbox™

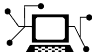
User's Guide



MATLAB®

R2020**b**

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 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)

Contents

1 What is Deep Learning?

Introduction to Deep Learning **1-2**

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

Convolutional Neural Networks **1-5**

2 Deep Learning Processor

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

3 Applications and Examples

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

4 Deep Learning on FPGA Overview

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

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

5 Workflow and APIS

Prototype Deep Learning Networks on FPGA and SoCs Workflow **5-2**

Estimate Performance of Deep Learning Network Running with Bitstream **5-4**

Estimate Performance of Deep Learning Network by Using Custom Processor Configuration **5-5**

Profile Inference Run **5-6**

Multiple Frame Support **5-9**
- Input DDR Format **5-9**
- Output DDR Format **5-9**
- Manually Enable Multiple Frame Mode **5-10**

6 Fast MATLAB to FPGA Connection Using LIBIIO/Ethernet

LIBIIO/Ethernet Connection Based Deployment **6-2**
- Ethernet Interface **6-2**
- Configure your LIBIIO/Ethernet Connection **6-2**
- LIBIIO/Ethernet Performance **6-2**

7 Networks and Layers

Supported Networks, Layers and Boards **7-2**
- Supported Pretrained Networks **7-2**
- Supported Layers **7-6**
- Supported Boards **7-13**

8 Custom Processor Configuration Workflow

Custom Processor Configuration Workflow **8-2**

9 Custom Processor Code Generation Workflow

Generate Custom Bitstream **9-2**
Intel Bitstream Resource Utilization **9-3**
Xilinx Bitstream Resource Utilization **9-3**

Generate Custom Processor IP **9-4**

10 Featured Examples

Get Started with Deep Learning FPGA Deployment on Intel Arria 10 SoC **10-2**

Get Started with Deep Learning FPGA Deployment on Xilinx ZCU102 SoC **10-5**

Logo Recognition Network **10-8**

Deploy Transfer Learning Network for Lane Detection **10-13**

Image Category Classification by Using Deep Learning **10-17**

Defect Detection **10-23**

Profile Network for Performance Improvement **10-32**

Bicyclist and Pedestrian Classification by Using FPGA **10-36**

Visualize Activations of a Deep Learning Network by Using LogoNet . **10-41**

Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core **10-47**

Run a Deep Learning Network on FPGA with Live Camera Input **10-52**

Running Convolution-Only Networks by using FPGA Deployment **10-61**

Accelerate Prototyping Workflow for Large Networks by using Ethernet **10-66**

Create Series Network for Quantization **10-72**

Vehicle Detection Using YOLO v2 Deployed to FPGA **10-76**

Custom Deep Learning Processor Generation to Meet Performance Requirements **10-84**

11 Deep Learning Quantization

Quantization of Deep Neural Networks **11-2**
Precision and Range **11-2**
Histograms of Dynamic Ranges **11-2**

Quantization Workflow Prerequisites **11-9**

Calibration **11-10**
Workflow **11-10**

Validation **11-12**
Workflow **11-12**

Code Generation and Deployment **11-15**

Deploy Quantized Neural Network **11-17**
Prerequisites **11-17**
Create Modified Series Network by Using Transfer Learning **11-17**
Create Quantized Network Object **11-17**
Load Training Data **11-17**
Calibrate Quantized Network **11-17**
Create Target Object **11-18**
Create Workflow Object **11-18**
Compile Quantized Series Network **11-18**
Program Bitstream onto FPGA and Download Network Weights **11-18**
Load the Example Images and Run the Prediction **11-19**

Quantize Neural Network for FPGA Execution Environment **11-22**
Prerequisites **11-22**
Load Pretrained Series Network **11-22**
Define Calibration and Validation Data Sets **11-22**
Create Quantized Network Object **11-23**
Calibrate Quantized Network **11-23**
Create Target Object **11-23**
Define Metric Function **11-23**
Create dlQuantizationOptions Object **11-24**
Validate Quantized Neural Network **11-24**
View Performance of Quantized Neural Network **11-27**

12 Deep Learning Processor IP Core User Guide

Deep Learning Processor IP Core **12-2**

Compiler Output **12-3**
External Memory Address Map **12-3**

External Memory Data Format **12-4**
Key Terminology **12-4**

Convolution Module External Memory Data Format **12-4**
Fully Connected Module External Memory Data Format **12-5**

Deep Learning Processor Register Map **12-7**

What is Deep Learning?

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

Introduction to Deep Learning

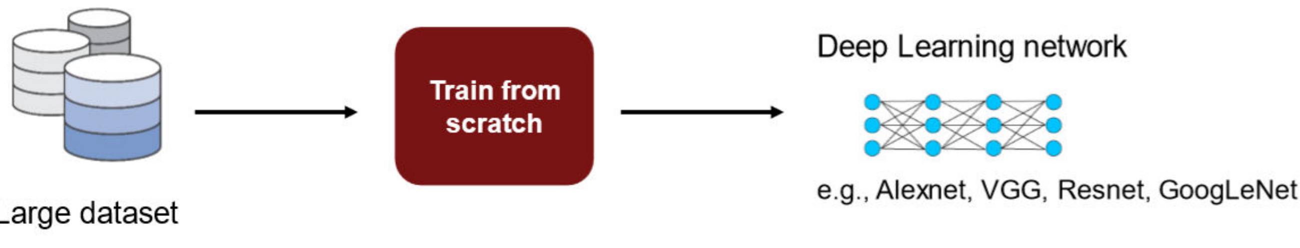
Deep learning is a branch of machine learning that teaches computers to do what comes naturally to humans: learn from experience. The learning algorithms use computational methods to “learn” information directly from data without relying on a predetermined equation as model. Deep learning uses neural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

Training Process

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

Training from Scratch

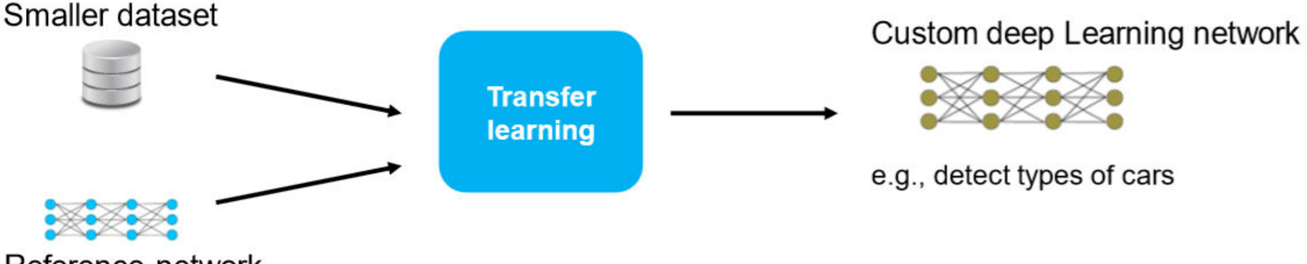
Training a deep learning neural network from scratch requires a large amount of labeled data.To create the network architecture by using Neural Network Toolbox™, you can use the built-in layers, define your own layers, or import layers from Caffe models. The neural network is then trained by using the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



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

Transfer Learning

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



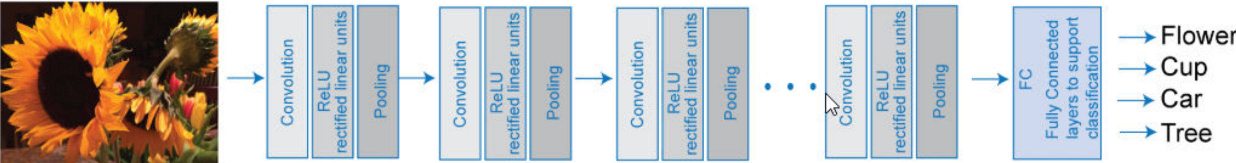
For more information, see “Get Started with Transfer Learning”

Feature Extraction

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

Convolutional Neural Networks

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



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

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

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

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

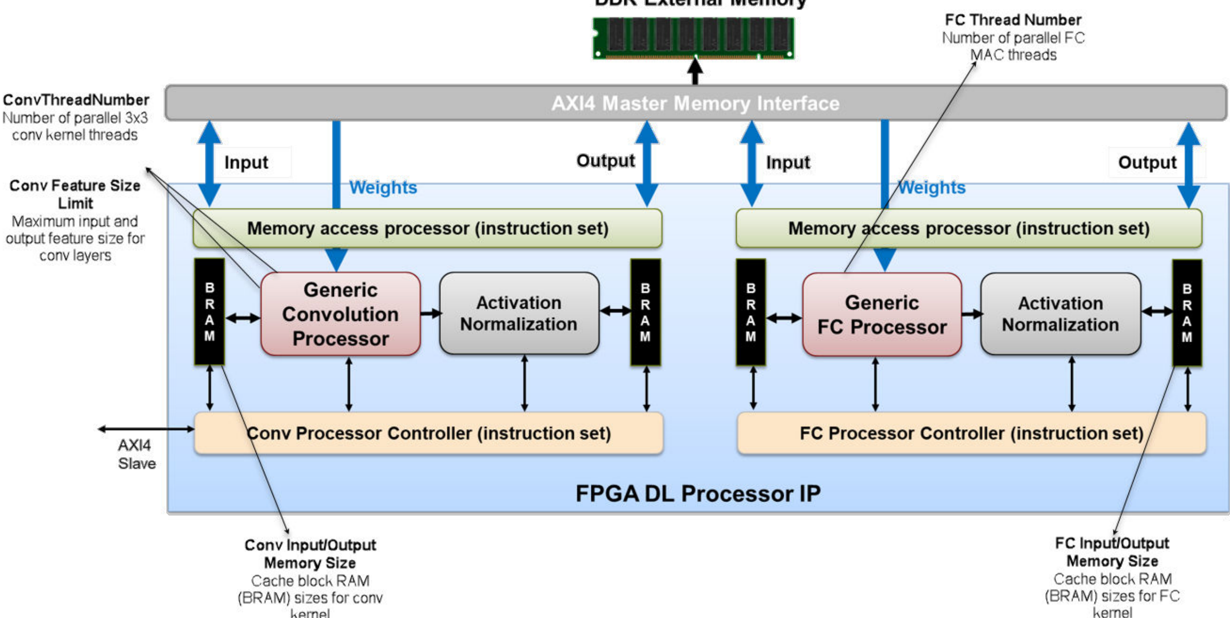
2

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.



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.

3

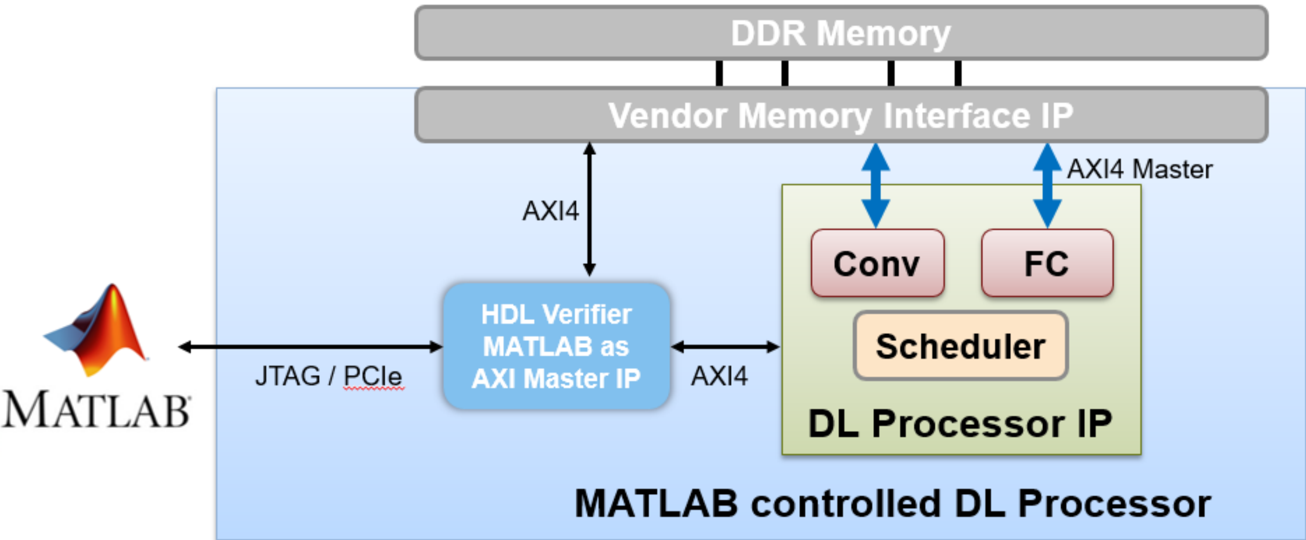
Applications and Examples

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.



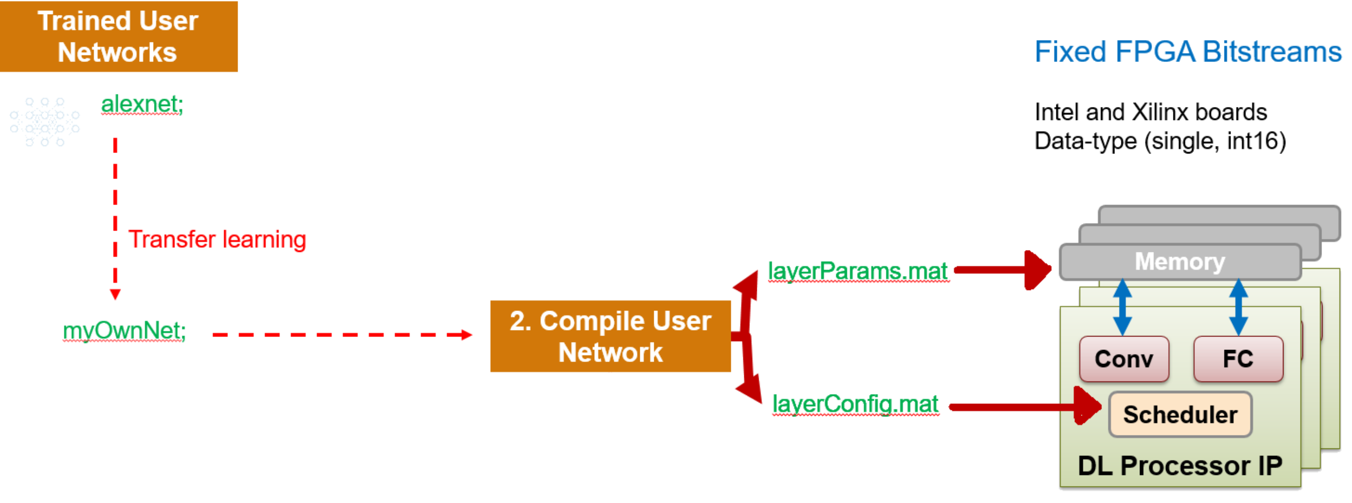
To integrate the generic deep learning processor IP with the MATLAB as AXI Master, use the AXI4 Slave interface of the deep learning processor IP core. By using a JTAG or PCI express interface, the IP responds to read or write commands from MATLAB. Therefore, you can use the MATLAB controlled deep learning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

Deep Learning on FPGA Overview

- "Deep Learning on FPGA Workflow" on page 4-2
- "Deep Learning on FPGA Solution and Workflows" on page 4-4

Deep Learning on FPGA Workflow

This figure illustrates deep learning on FPGA workflow.



To use the workflow:

1. **Load deep learning neural network**

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

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

   Compilation of the user network usually begins with validating the architecture, types of layers present , data type of input and output parameters, and maximum number of activations. This FPGA solution supports series network architecture with data types of single and int16. 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**

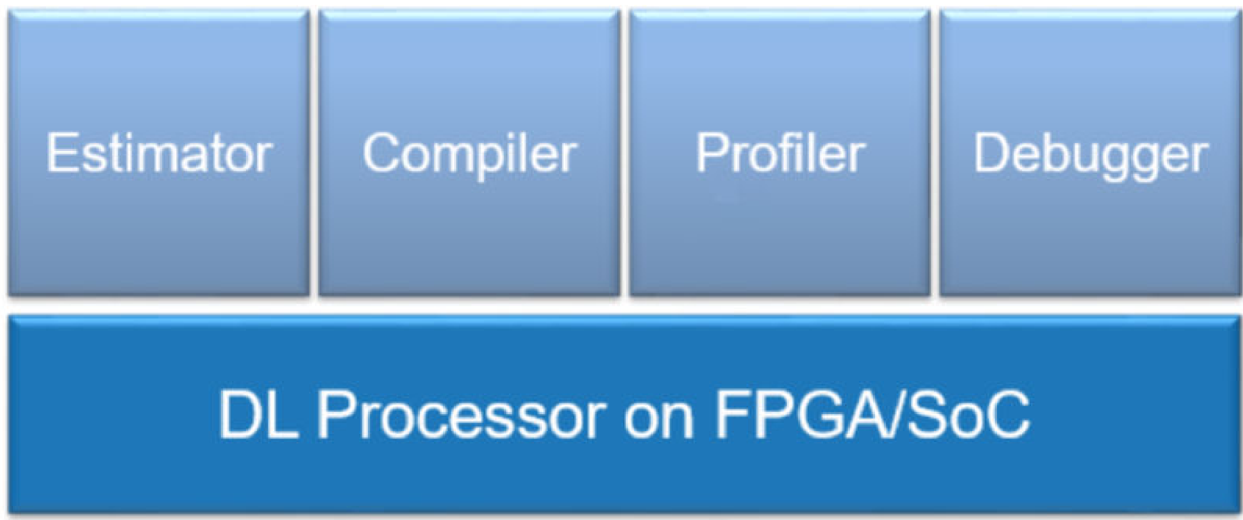
To classify objects in the input image, use the deployed framework on the FPGA board.

See Also

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

Deep Learning on FPGA Solution and Workflows

The figure illustrates the MATLAB solution for implementing deep learning on FPGA.



The FPGA deep learning solution provides an end to end solution that allows you to estimate, compile, profile and debug your custom pretrained series network. You can also generate a custom deep learning processor IP. The estimator is used for estimating the performance of the deep learning framework in terms of speed. The compiler converts the pretrained deep learning network for the current application for deploying it on the intended target FPGA boards.

To learn more about the deep learning processor IP, see “Deep Learning Processor IP Core” on page 12-2 .

FPGA Advantages

FPGAs provide advantages, such as :

- High performance
- Flexible interfacing
- Data parallelism
- Model parallelism
- Pipeline parallelism

Deep Learning on FPGA Workflows

To run certain Deep Learning on FPGA tasks, see the information listed in this table.

Task	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 Running with Bitstream” on page 5-4
Customize the deep learning processor to meet your area or performance constraints.	“Estimate Performance of Deep Learning Network by Using Custom Processor Configuration” on page 5-5
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 11-12
Run a quantized pretrained series network on your target FPGA board.	“Code Generation and Deployment” on page 11-15

5

Workflow and APIS

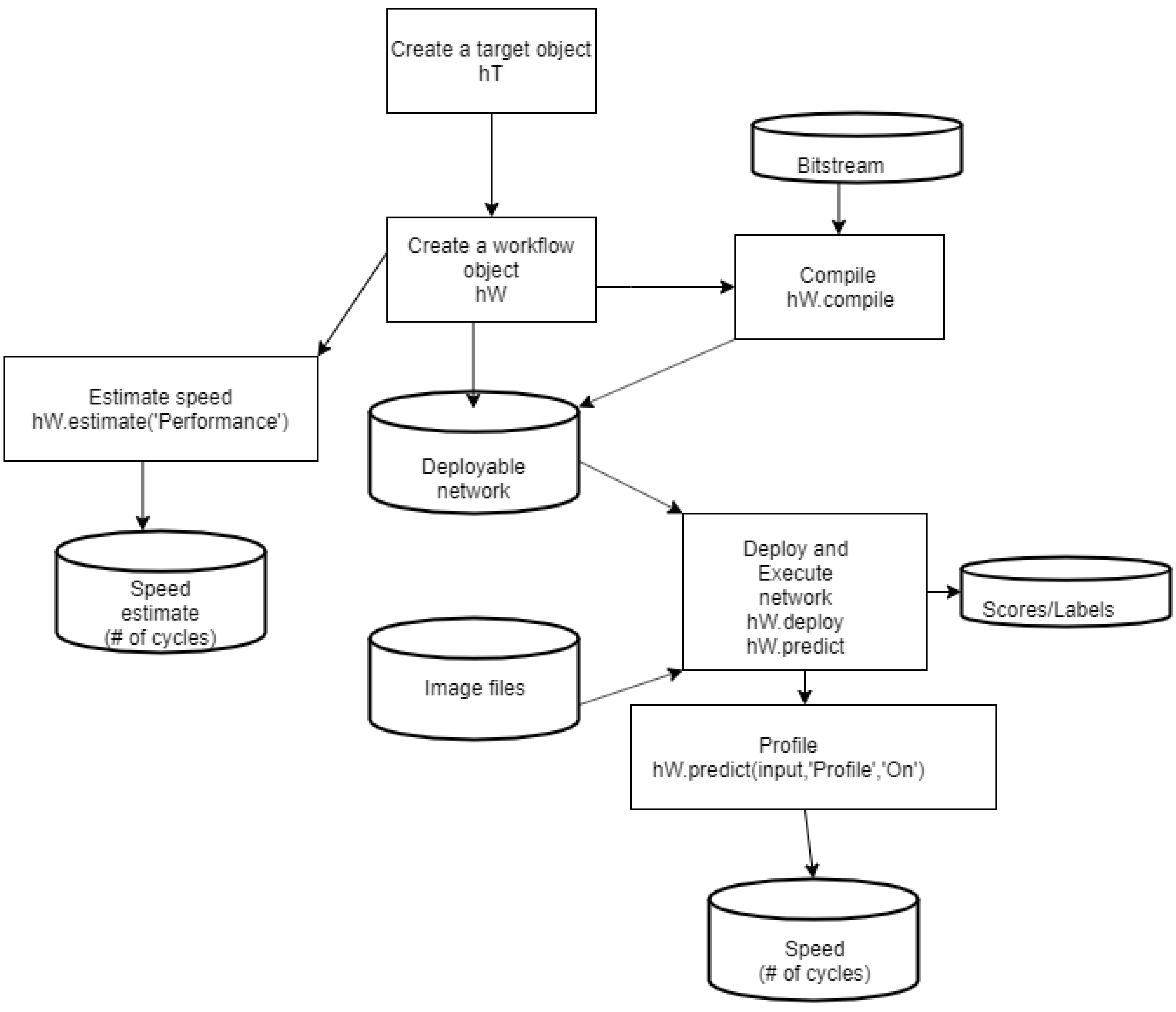
- “Prototype Deep Learning Networks on FPGA and SoCs Workflow” on page 5-2
- “Estimate Performance of Deep Learning Network Running with Bitstream” on page 5-4
- “Estimate Performance of Deep Learning Network by Using Custom Processor Configuration” on page 5-5
- “Profile Inference Run” on page 5-6
- “Multiple Frame Support” on page 5-9

Prototype Deep Learning Networks on FPGA and SoCs Workflow

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

- Compile and deploy the deep learning network on specified target FPGA or SoC board by using the `deploy` function.
- Estimate the speed of the deep learning network in terms of number of cycles by using the `estimate` 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. Set the `Profile` parameter to `on`.

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



See Also

`dlhdl.Target` | `dlhdl.Workflow`

More About

- "Get Started with Deep Learning FPGA Deployment on Xilinx ZCU102 SoC" on page 10-5

Estimate Performance of Deep Learning Network Running with Bitstream

1. Create an object of class `workflow` by using the `dlhdl.Workflow` class.
2. Set the deep learning and bitstream for the workflow object.
3. Call the `estimate` function for the workflow object.

   The speed and latency are stored in a structure `struct` and displayed on the screen.

For example:

```
snet = vgg19;
hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'arria10soc_single');
result = hW.estimate('Performance');
```

The result of the estimation is:

```
              Deep Learning Processor Estimator Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------              -------------             ---------        --
Network                    172441964                  1.14961                     1          1724
    conv_module            162622207                  1.08415
        conv1_1              4528942                  0.03019
        conv1_2             17788981                  0.11859
        pool1                2360417                  0.01574
        conv2_1              8510437                  0.05674
        conv2_2             15432208                  0.10288
        pool2                1242064                  0.00828
        conv3_1              7660645                  0.05107
        conv3_2             14177125                  0.09451
        conv3_3             14177125                  0.09451
        conv3_4             14177125                  0.09451
        pool3                 671713                  0.00448
        conv4_1              6957812                  0.04639
        conv4_2             13621492                  0.09081
        conv4_3             13621492                  0.09081
        conv4_4             13621492                  0.09081
        pool4                 391652                  0.00261
        conv5_1              3396733                  0.02264
        conv5_2              3396733                  0.02264
        conv5_3              3396733                  0.02264
        conv5_4              3396733                  0.02264
        pool5                  94553                  0.00063
    fc_module                9819757                  0.06547
        fc6                  8160258                  0.05440
        fc7                  1331586                  0.00888
        fc8                   327913                  0.00219
 * The clock frequency of the DL processor is: 150MHz
```

Estimate Performance of Deep Learning Network by Using Custom Processor Configuration

1. Create a custom processor configuration object of class `dlhdl.ProcessorConfig`.
2. Create an object of class `workflow` by using the `dlhdl.Workflow` class.
3. Set the deep learning network and processor configuration for the workflow object.
4. Call the `estimate` function for the workflow object.

   The speed and latency is stored in a structure `struct` and displayed on the screen.

For example:

```
hPC = dlhdl.ProcessorConfig;
snet = vgg19;
hW = dlhdl.Workflow('Network', snet, 'ProcessorConfig',hPC);
result = hW.estimate('Performance');
```

The result of the estimation is:

```
              Deep Learning Processor Estimator Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                  202770372                  1.01385                       1          202
    conv_module          158812469                  0.79406
        conv1_1            2022004                  0.01011
        conv1_2           15855549                  0.07928
        pool1              2334753                  0.01167
        conv2_1            7536365                  0.03768
        conv2_2           14837392                  0.07419
        pool2              1446960                  0.00723
        conv3_1            7950445                  0.03975
        conv3_2           14365933                  0.07183
        conv3_3           14365933                  0.07183
        conv3_4           14365933                  0.07183
        pool3               930145                  0.00465
        conv4_1            7073684                  0.03537
        conv4_2           13761300                  0.06881
        conv4_3           13761300                  0.06881
        conv4_4           13761300                  0.06881
        pool4               572644                  0.00286
        conv5_1            3432645                  0.01716
        conv5_2            3432645                  0.01716
        conv5_3            3432645                  0.01716
        conv5_4            3432645                  0.01716
        pool5               140249                  0.00070
    fc_module             43957903                  0.21979
        fc6               36535923                  0.18268
        fc7                5965299                  0.02983
        fc8                1456681                  0.00728
 * The clock frequency of the DL processor is: 200MHz
```

Profile Inference Run

View the network prediction and performance data for the layers, convolution module and fully connected modules in your pretrained series network. The example shows how to retrieve the prediction and profiler results for the VGG-19 network.

1. Create an object of class `Workflow` by using the `dlhdl.Workflow` class.
2. Set a pretrained deep learning network and bitstream for the workflow object.
3. Create an object of class `dlhdl.Target` and specify the target vendor and interface.
4. To deploy the network on a specified target FPGA board, call the `deploy` method for the workflow object.
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.

   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 the code:

```
snet = vgg19;
hT = dlhdl.Target('Intel');
hW = dlhdl.Workflow('Net', snet, 'Bitstream', 'arria10soc_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

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------            -------------               ---------        --
Network                  166206640                  1.10804                       1          1662
```

```
    conv_module         156100737                  1.04067
        conv1_1           2174602                  0.01450
        conv1_2          15580687                  0.10387
        pool1             1976185                  0.01317
        conv2_1           7534356                  0.05023
        conv2_2          14623885                  0.09749
        pool2             1171628                  0.00781
        conv3_1           7540868                  0.05027
        conv3_2          14093791                  0.09396
        conv3_3          14093717                  0.09396
        conv3_4          14094381                  0.09396
        pool3              766669                  0.00511
        conv4_1           6999620                  0.04666
        conv4_2          13725380                  0.09150
        conv4_3          13724671                  0.09150
        conv4_4          13725125                  0.09150
        pool4              465360                  0.00310
        conv5_1           3424060                  0.02283
        conv5_2           3423759                  0.02283
        conv5_3           3424758                  0.02283
        conv5_4           3424461                  0.02283
        pool5              113010                  0.00075
    fc_module            10105903                  0.06737
        fc6               8397997                  0.05599
        fc7               1370215                  0.00913
        fc8                337689                  0.00225
 * The clock frequency of the DL processor is: 150MHz


ans =

    'zebra'
```

The profiler data returns these parameters and their values:

- `LastLayerLatency(cycles)`- Total number of clock cycles for layer or module 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: 150MHz`. The clock frequency of 150 MHz is retrieved from the `arria10soc_single` bitstream.
- `LastLayerLatency(seconds)`- Total number of seconds for layer or module execution. The total time is calculated as `LastLayerLatency(cycles)/Clock Frequency`. For example the `conv_module` `LastLayerLatency(seconds)` is calculated as `156100737/(150*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*150*10^6)/166206873`.

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 10-32

Multiple Frame Support

Deep Learning HDL Toolbox supports multiple frame mode that enables 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.

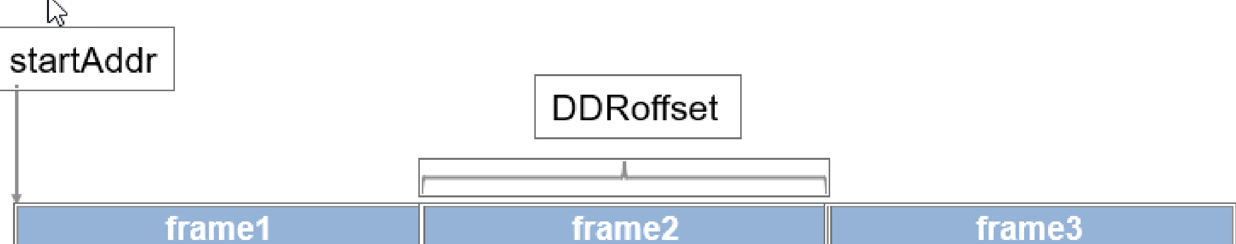
Input DDR Format

Formatting the input images to meet the multiple frame input DDR format requires:

- 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 “Compiler Output” 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”.

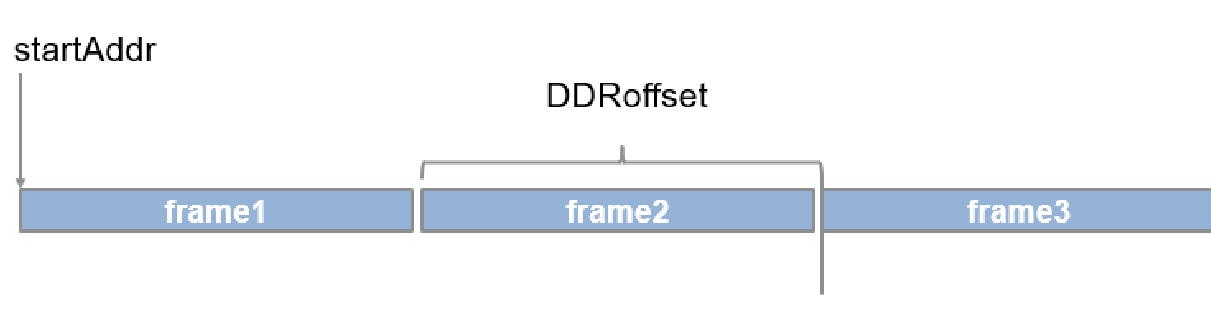


Output DDR Format

Retrieving the results for multiple image inputs from the output area of the DDR requires:

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

The output results have to be formatted to be a multiple of the FC output feature size. The information and formatting are automatically generated by the `compile` method. For more information on the generated DDR address offsets, see “Compiler Output” on page 12-3.



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('nc_op_image_count')),15,hT);
```

The function `addrMap('nc_op_image_count')` returns the AXI register address for `nc_op_image_count`, 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 12-7.
`compile` | `dlhdl.Target` | `dlhdl.Workflow`

More About

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

Fast MATLAB to FPGA Connection Using LIBIIO/Ethernet

LIBIIO/Ethernet Connection Based Deployment

In this section...

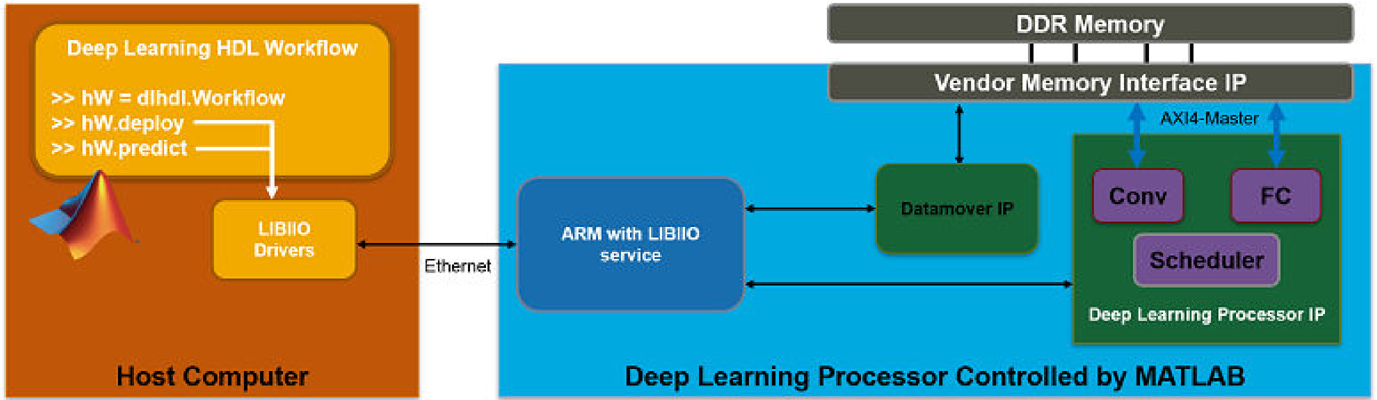
"Ethernet Interface" on page 6-2
"Configure your LIBIIO/Ethernet Connection" on page 6-2
"LIBIIO/Ethernet Performance" on page 6-2

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.

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



Configure your LIBIIO/Ethernet Connection

You can configure your `dlhdl.Workflow` object hardware interface to Ethernet 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 10-66

7

Networks and Layers

Supported Networks, Layers and Boards

In this section...

"Supported Pretrained Networks" on page 7-2
"Supported Layers" on page 7-6
"Supported Boards" on page 7-13

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 convolutional neural network whose computational layers are supported for code generation. See "Supported Layers" on page 7-6. You can use one of the pretrained networks listed in the table and 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

MNIST	MNIST Digit Classification.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Regression
Lane detection	LaneNet convolutional neural network. For more information, see “Deploy Transfer Learning Network for Lane Detection” on page 10-13	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	No. the network contains a `globalAveragePooling` layer that is not supported for INT8 quantization.	No. the network contains a `globalAveragePooling` layer that is not supported for INT8 quantization.	No. the network contains a `globalAveragePooling` layer that is not supported for INT8 quantization.	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 10-36.	Series Network	Yes	Yes	Yes	No. the network contains a `AveragePooling` layer that is not supported for INT8 quantization.	No. the network contains a `AveragePooling` layer that is not supported for INT8 quantization.	No. the network contains a `AveragePooling` layer that is not supported for INT8 quantization.	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 10-23.	Series Network	Yes	Yes	Yes	Yes	Yes	Yes	Classification
Defect Detectionsnet_blemdetnet	snet_blemdetnet is a custom convolutional neural network used to identify and classify defects. For more information, see "Defect Detection" on page 10-23.	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 10-76	Series Network based	Yes	Yes	Yes	Yes	Yes	Yes	Object detection

Supported Layers

The following layers are supported by Deep Learning HDL Toolbox.

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)	Description and Limitations	INT8 Compatible
`convolution2dLayer`	HW	A 2-D convolutional layer applies sliding convolutional filters to the input. These limitations apply when generating code for a network using this layer: • Filter size must be 1-12 and square. For example [1 1] or [12 12]. • Stride size must be 1,2 or 4 and square. • Padding size must be in the range 0-8. • Dilation factor must be [1 1].	Yes

groupedConvolution2dLayer	HW	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. These limitations apply when generating code for a network using this layer: • Filter size must be 1-12 and square. For example [1 1] or [12 12]. • Stride size must be 1,2 or 4 and square. • Padding size must be in the range 0-8. • Dilation factor must be [1 1]. • Number of groups must be 1 or 2.	Yes
fullyConnectedLayer	HW	A fully connected layer multiplies the input by a weight matrix, and then adds a bias vector. These limitations apply when generating code for a network using this layer: • 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)	Description	INT8 Compatible
`reluLayer`	HW	A ReLU layer performs a threshold operation to each element of the input where any value less than zero is set to zero. A clipped ReLU layer is supported only when it is preceded by a convolution layer.	Yes
`leakyReluLayer`	HW	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 a convolution layer.	No
`clippedReluLayer`	HW	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 a convolution layer.	No

Normalization, Dropout, and Cropping Layers

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

batchNormalizationLayer	HW	A batch normalization layer normalizes each input channel across a mini-batch. A batch normalization layer is only supported only when it is preceded by a convolution layer.	Yes
crossChannelNormalizationLayer	HW	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	A dropout layer randomly sets input elements to zero with a given probability.	Yes

Pooling and Unpooling Layers

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

maxPooling2dLayer	HW	A max pooling layer performs down sampling by dividing the input into rectangular pooling regions and computing the maximum of each region. These limitations apply when generating code for a network using this layer: • Pool size must be 1-12and square. For example [1 1] or [12 12]. • Stride size must be 1-7 and square. • Padding size must be in the range 0-2. Padding size can only be used when the pool size is 3-by-3.	Yes
averagePooling2dLayer	HW	An average pooling layer performs down sampling by dividing the input into rectangular pooling regions and computing the average values of each region. These limitations apply when generating code for a network using this layer: • Pool size must be 1-12 and square. For example [3 3] • Stride size must be 1-7 and square. • Padding size must be in the range 0-2. Padding size can only be used when the pool size is 3-by-3.	No

globalAveragePooling2dLayer	HW	A global average pooling layer performs down sampling by computing the mean of the height and width dimensions of the input. These limitations apply when generating code for a network using this layer: • Pool size value must be in the range 1-12 and be square. For example, [1 1] or [12 12]. • Total activation pixel size must be smaller than the deep learning processor convolution module input memory size. For more information, see “InputMemorySize”	No

Output Layer

Layer	Layer Type Hardware (HW) or Software(SW)	Description	INT8 Compatible
softmax	SW	A softmax layer applies a softmax function to the input.	Yes. Runs as single datatype in SW.
classificationLayer	SW	A classification layer computes the cross-entropy loss for multi class classification issues with 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)	Description	INT8 Compatible
`nnet.keras.layer.FlattenCStyleLayer`	HW	Flatten activations into 1-D layers assuming C-style (row-major) order. A `nnet.keras.layer.FlattenCStyleLayer` is only supported only when it is followed by a fully connected layer.	Yes
`nnet.keras.layer.ZeroPadding2dLayer`	HW	Zero padding layer for 2-D input. A `nnet.keras.layer.ZeroPadding2dLayer` is only 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.

See Also

More About

- “Configure Board-Specific Setup Information”

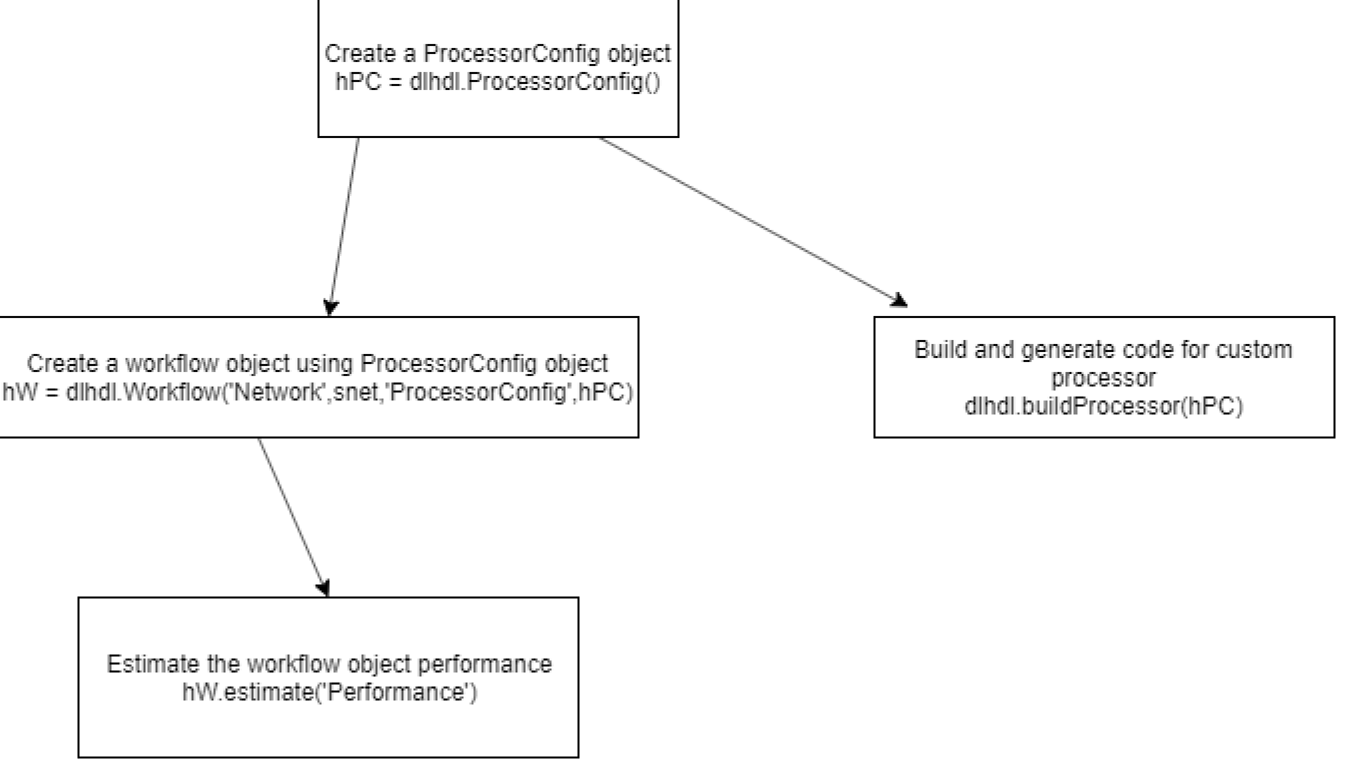
Custom Processor Configuration Workflow

Custom Processor Configuration Workflow

Estimate the performance 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 and for information about the convolution and fully connected module parameters, see “Module 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 build and generate your custom deep learning processor IP core.



See Also

`dlhdl.ProcessorConfig` | `getModuleProperty` | `setModuleProperty`

More About

- “Deep Learning Processor Architecture” on page 2-2

Custom Processor Code Generation Workflow

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

Generate Custom Bitstream

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

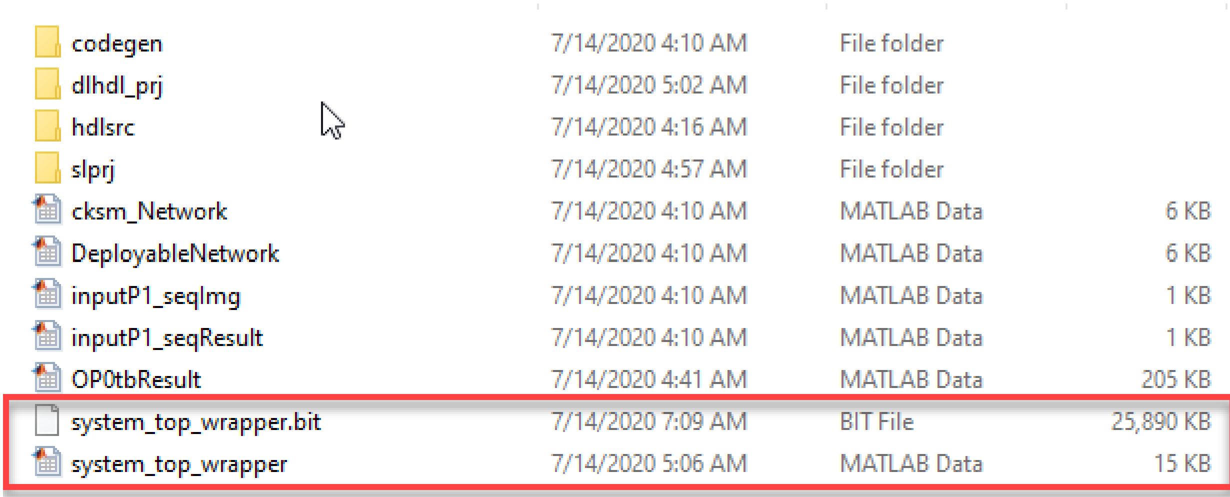
1. Create a `dlhdl.ProcessorConfig` object.

   ```
   hPC = dlhdl.ProcessorConfig;
   ```
2. Setup the tool path to your design tool. For example, to setup 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. After the bitstream generation is completed, you can locate the bitstream file at `cwd\dlhdl_prj\vivado_ip_prj\vivado_prj.runs\impl_1`, where `cwd` is your current working directory. The name of the bitstream file is `system_top_wrapper.bit`. The associated `system_top_wrapper.mat` file is located in the top level of the `cwd`.

   To use the generated bitstream for the supported Xilinx boards, you should copy the `system_top_wrapper.bit` and `system_top_wrapper.mat` files to the same folder.

   To use the generated bitstream for the supported Intel boards, you should copy the `system_core.rbf`, `system.mat`, `system_periph.rbf`, and `system.sof` files to the same folder.

codegen	7/14/2020 4:10 AM	File folder	
dlhdl_prj	7/14/2020 5:04 AM	File folder	
hdlsrc	7/14/2020 4:16 AM	File folder	
slprj	7/14/2020 5:01 AM	File folder	
cksm_Network	7/14/2020 4:10 AM	MATLAB Data	7 KB
DeployableNetwork	7/14/2020 4:10 AM	MATLAB Data	6 KB
inputP1_seqImg	7/14/2020 4:10 AM	MATLAB Data	1 KB
inputP1_seqResult	7/14/2020 4:10 AM	MATLAB Data	1 KB
OP0tbResult	7/14/2020 4:41 AM	MATLAB Data	205 KB
system.core.rbf	7/14/2020 6:30 AM	RBF File	14,657 KB
system	7/14/2020 5:07 AM	MATLAB Data	16 KB
system.periph.rbf	7/14/2020 6:30 AM	RBF File	353 KB
system.sof	7/14/2020 6:25 AM	SOF File	25,989 KB

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

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

Intel Bitstream Resource Utilization

"Bitstream Resource Utilization" (Deep Learning HDL Toolbox Support Package for Intel FPGA and SoC Devices)

Xilinx Bitstream Resource Utilization

"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

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. Setup the tool path to your design tool. For example, to setup the path to the Vivado design tool, enter:

   ```
   hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.bat');
   ```
3. Generate the custom processor IP.

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

See Also

`dlhdl.ProcessorConfig` | `dlhdl.buildProcessor`

More About

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

Featured Examples

- "Get Started with Deep Learning FPGA Deployment on Intel Arria 10 SoC" on page 10-2
- "Get Started with Deep Learning FPGA Deployment on Xilinx ZCU102 SoC" on page 10-5
- "Logo Recognition Network" on page 10-8
- "Deploy Transfer Learning Network for Lane Detection" on page 10-13
- "Image Category Classification by Using Deep Learning" on page 10-17
- "Defect Detection" on page 10-23
- "Profile Network for Performance Improvement" on page 10-32
- "Bicyclist and Pedestrian Classification by Using FPGA " on page 10-36
- "Visualize Activations of a Deep Learning Network by Using LogoNet" on page 10-41
- "Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core" on page 10-47
- "Run a Deep Learning Network on FPGA with Live Camera Input" on page 10-52
- "Running Convolution-Only Networks by using FPGA Deployment" on page 10-61
- "Accelerate Prototyping Workflow for Large Networks by using Ethernet" on page 10-66
- "Create Series Network for Quantization" on page 10-72
- "Vehicle Detection Using YOLO v2 Deployed to FPGA" on page 10-76
- "Custom Deep Learning Processor Generation to Meet Performance Requirements" on page 10-84

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™

Create a Folder and Copy Relevant Files

Create a new folder in your current working folder where you have write permission and copy all the files into this folder.

```
unzip('dnnfpga_digits.zip');
[newDir, origDir] = cloneSetupDir('dnnfpga_digits');
cd(newDir);
```

Load the Pretrained SeriesNetwork

To load the pretrained series network, that has been trained on the Modified National Institue Standards of Technolofy (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
```

```
### 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 15:19:17
```

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                      49438                  0.00033                       1
    conv_module              26288                  0.00018
        conv_1                6741                  0.00004
        maxpool_1             4680                  0.00003
        conv_2                5231                  0.00003
        maxpool_2             3879                  0.00003
        conv_3                5817                  0.00004
    fc_module                23150                  0.00015
        fc                   23150                  0.00015
 * 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
```

```
cd(origDir);
```

See Also

More About

- “Check Host Computer Connection to FPGA Boards”
- “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 Institue Standards of Technolofy (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: '10.10.10.15'
     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;
```

```
### 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. 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 12:37:32
```

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                     73717                  0.00034                       1
    conv_module             27207                  0.00012
        conv_1               6673                  0.00003
        maxpool_1            4891                  0.00002
        conv_2               4999                  0.00002
        maxpool_2            3569                  0.00002
        conv_3               7135                  0.00003
    fc_module               46510                  0.00021
        fc                  46510                  0.00021
 * 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

- "Check Host Computer Connection to FPGA Boards"
- "Create Simple Deep Learning Network for Classification"

Logo Recognition Network

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

The Logo Recognition Network

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

Prerequisites

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

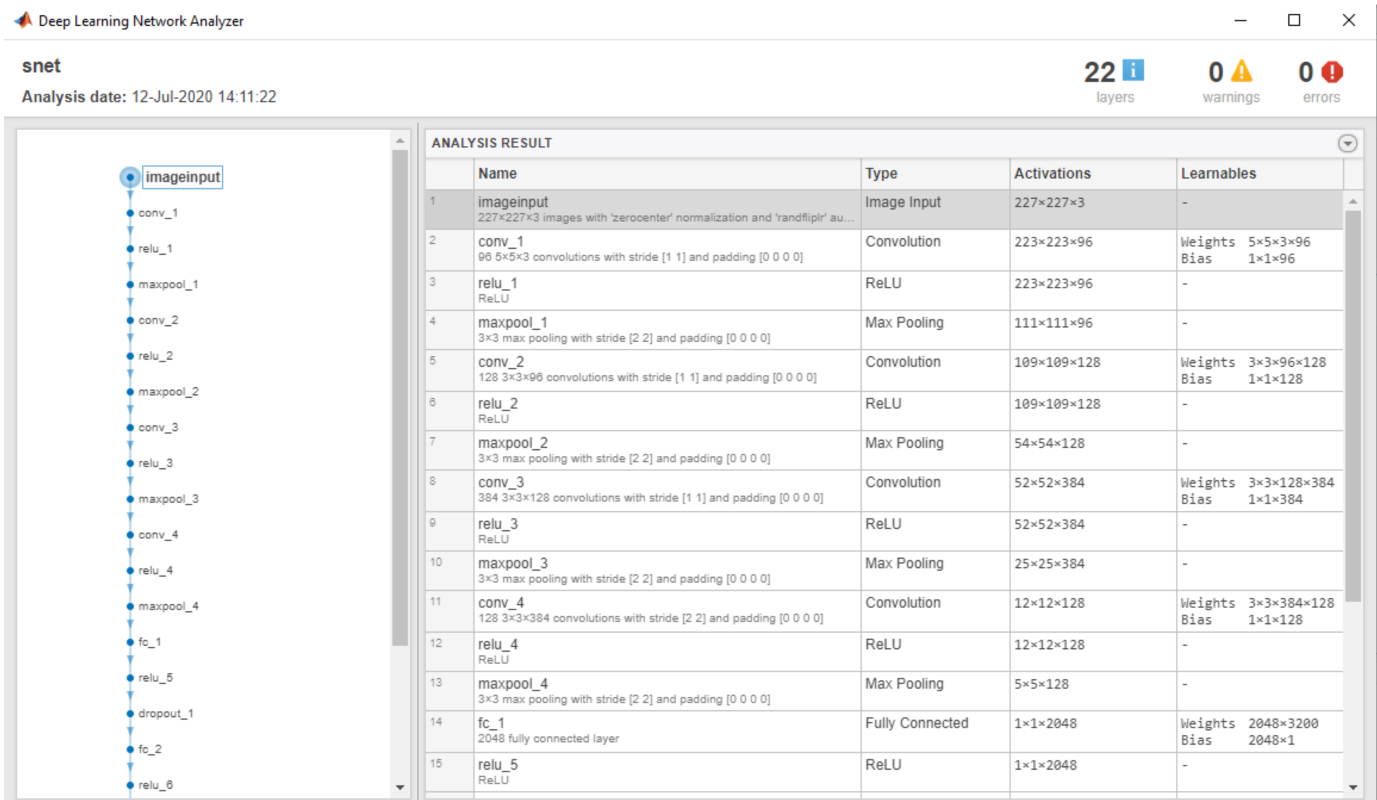
Load the Pretrained Series Network

To load the pretrained series network logonet, enter:

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

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

```
analyzeNetwork(snet)
```



Create Target Object

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

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

To create the target object, enter:

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

Create WorkFlow Object

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

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

Compile the Logo Recognition Network

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

```
dn = hW.compile
```

```
          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

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

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

Program Bitstream onto FPGA and Download Network Weights

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

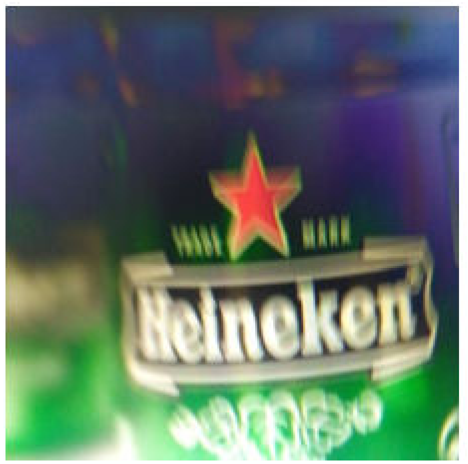
```
 hW.deploy
```

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

Load the Example Image

Load the example image.

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



Run the Prediction

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

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

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

              Deep Learning Processor Profiler Performance Results

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

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

```
ans =
'heineken'
```

See Also

More About

- “Check Host Computer Connection to FPGA Boards”

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();
```

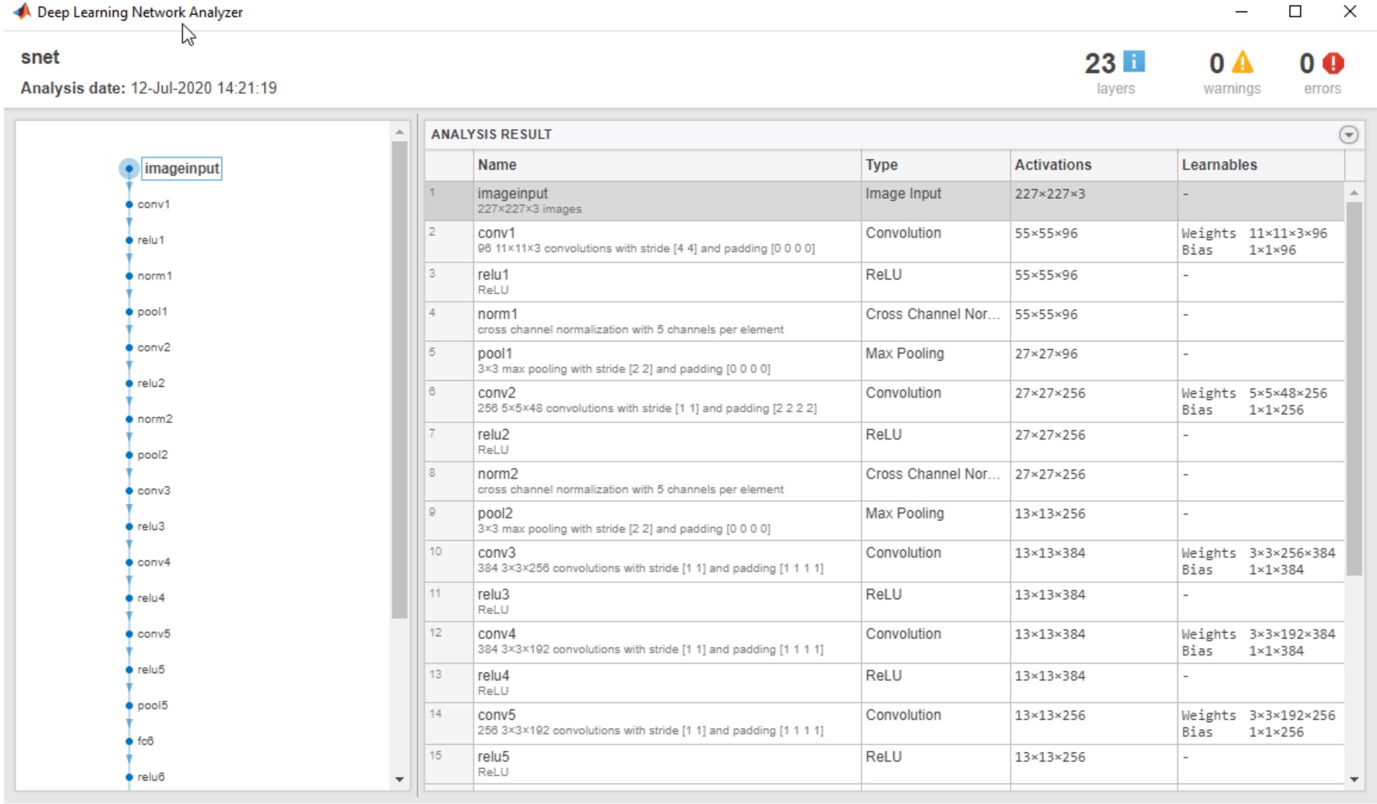
Normalize the Input Layer

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.
```



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 Intel 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:

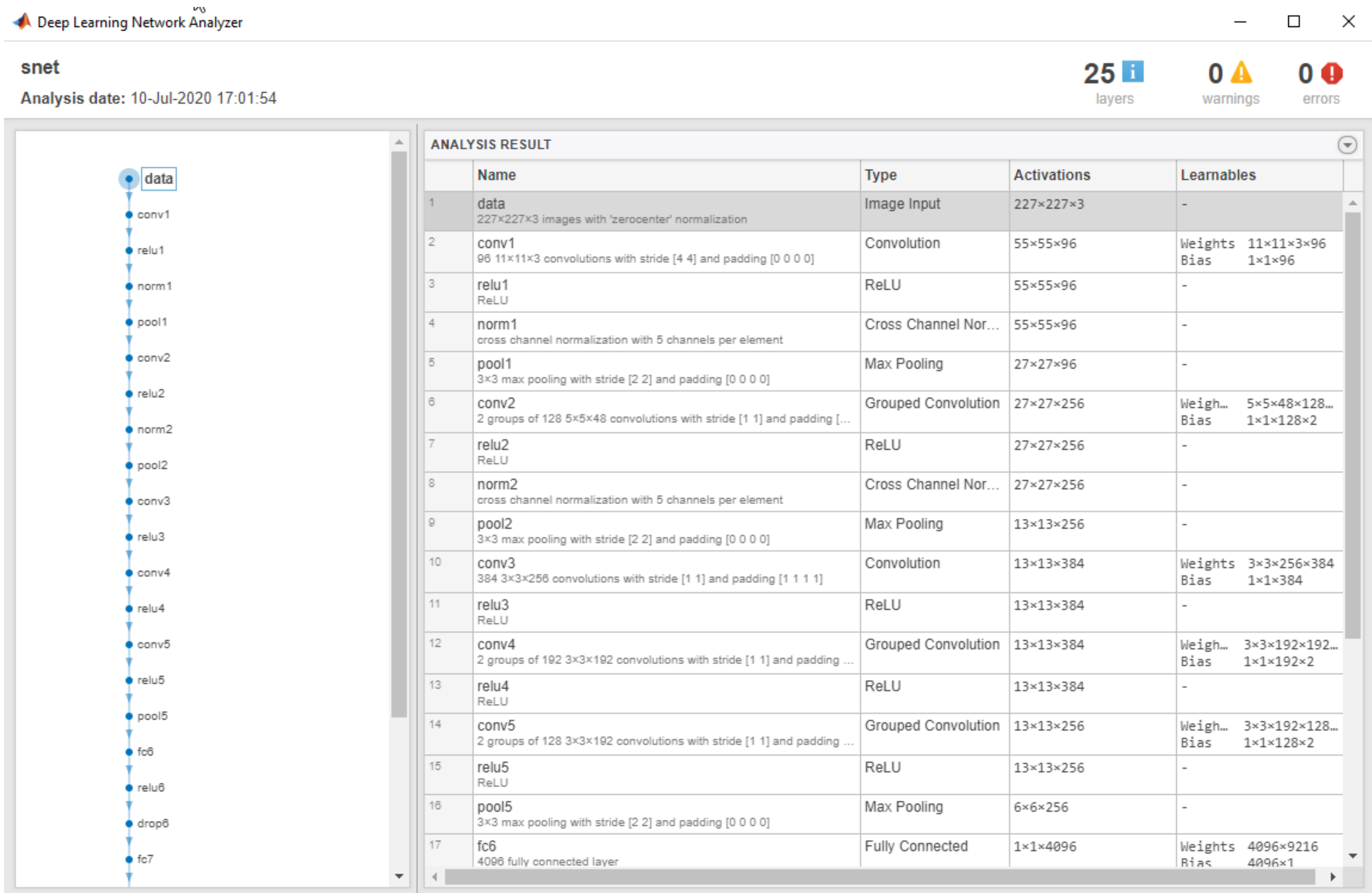
```
% snet = vgg19;
```

To load the pretrained series network darkent 19, 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.
```



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', 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 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.

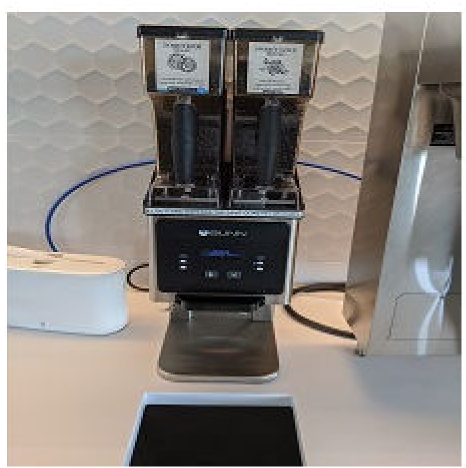
```
hW.deploy
```

```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### 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}
```

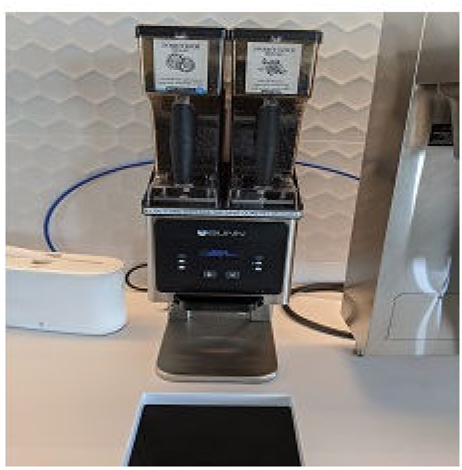
```
ans =
'espresso maker'
```

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 5-9.

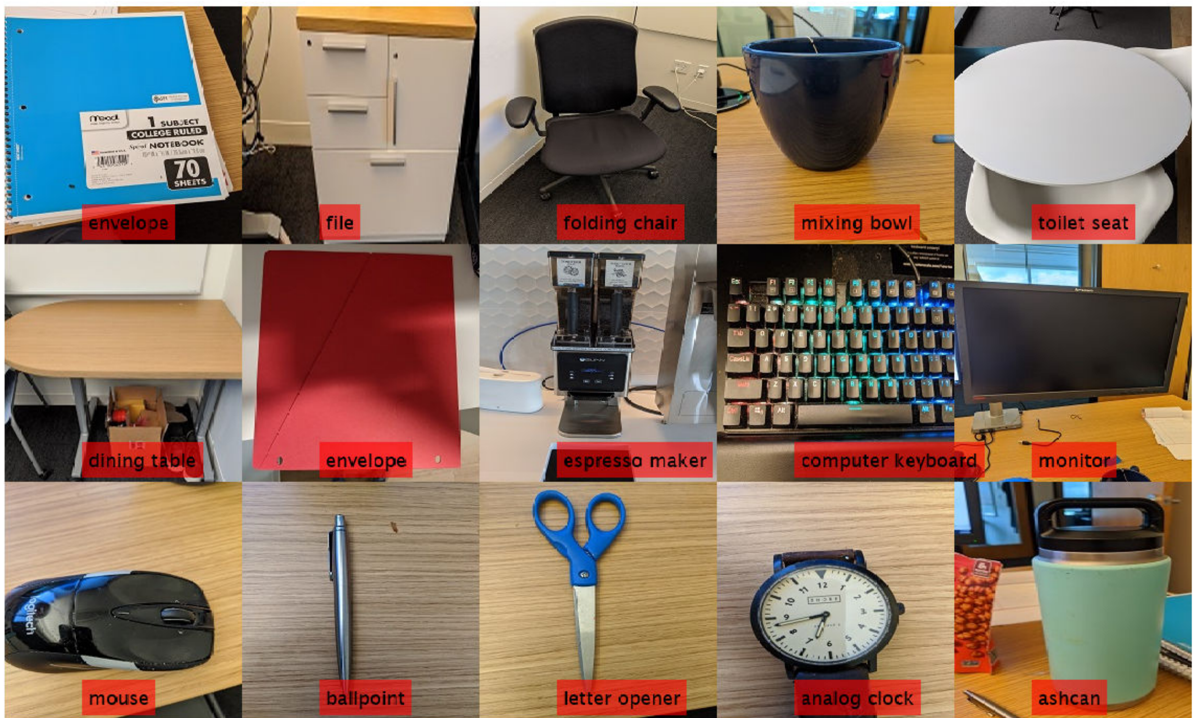
The `demoOnImage` function loads multiple images and retrieves their prediction results. The `annotateresults` function displays the image prediction result on top of the images which are assembed into a 3-by-5 array.

```
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

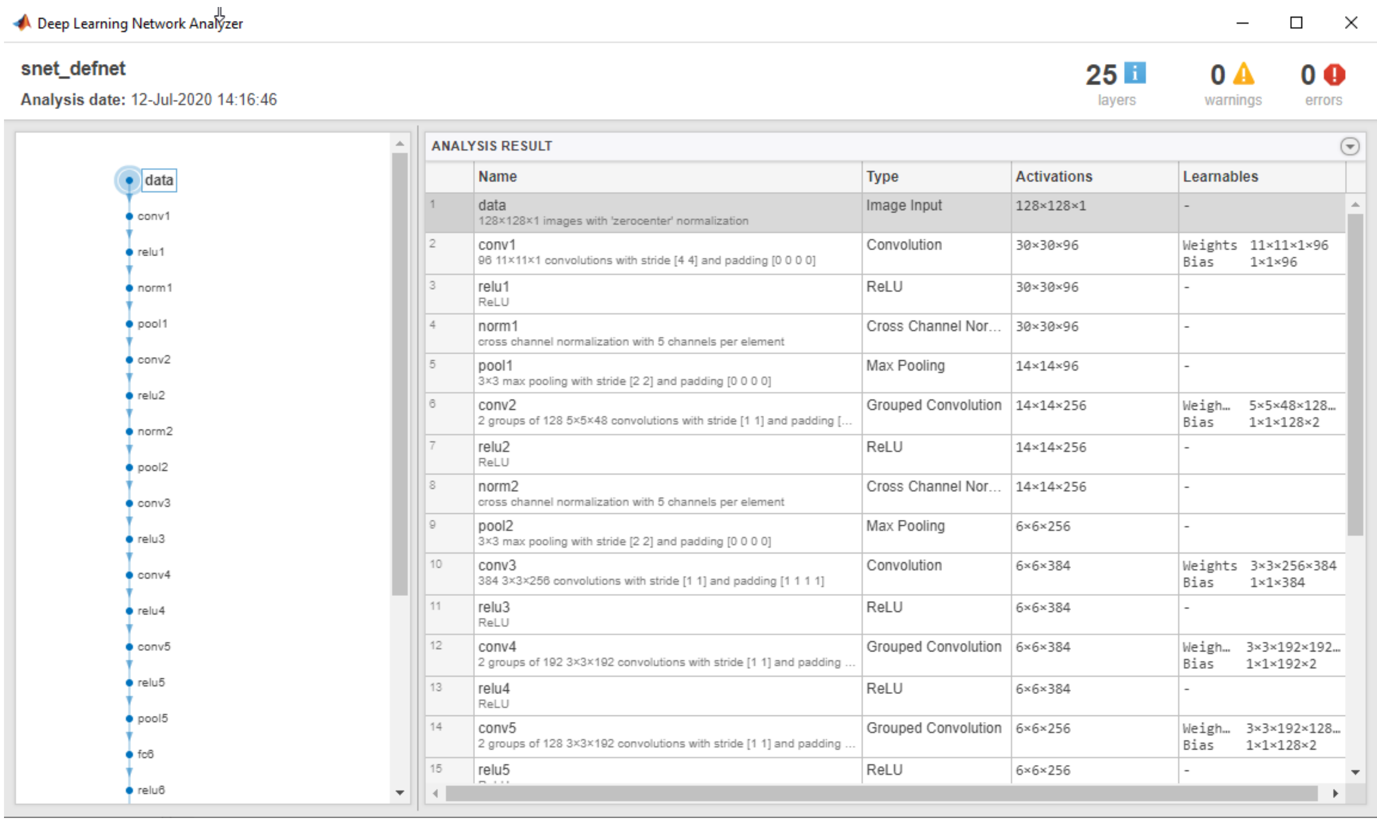
To download and load the custom pretrained series networks `trainedDefNet` and `trainedBlemDetNet`, enter:

```
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 `snet_defnet` layers.

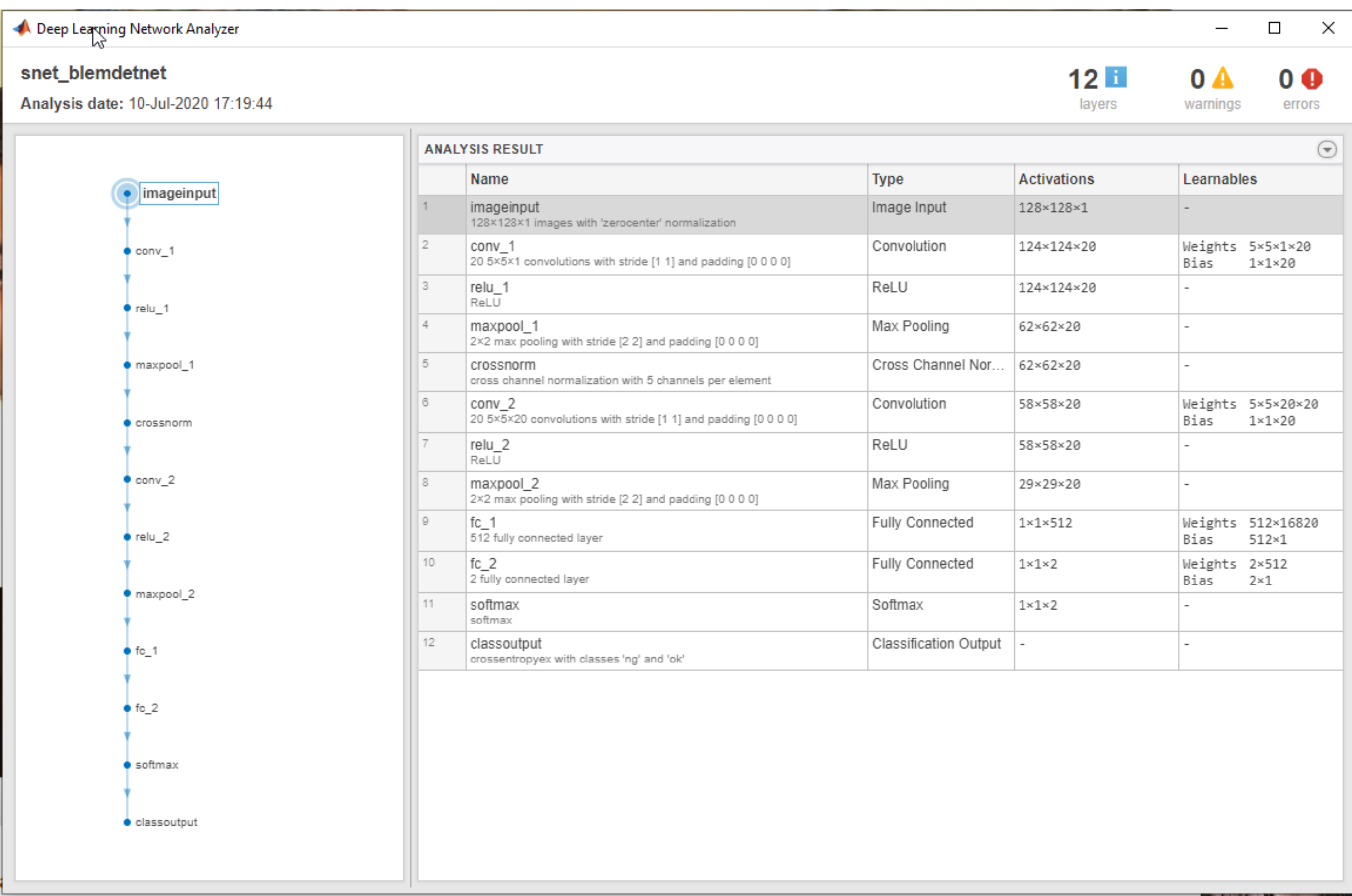
```
    analyzeNetwork(snet_defnet)
```




```
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'}
```

```
    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(TM) Vivado(TM) Design Suite 2019.2.

To set the Xilinx Vivado toolpath, enter:

```
% hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.b
    hT = dlhdl.Target('Xilinx','Interface','Ethernet')
```

```
hT = 
  Target with properties:

       Vendor: 'Xilinx'
    Interface: Ethernet
    IPAddress: '10.10.10.15'
     Username: 'root'
         Port: 22
```

Create Workflow Object for trainedDefNet Network

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pretrained `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 a 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

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

```
hW.compile
```

```
          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "8.0 MB"         
    "OutputResultOffset"        "0x00800000"     "4.0 MB"         
    "SystemBufferOffset"        "0x00c00000"     "28.0 MB"        
    "InstructionDataOffset"     "0x02800000"     "4.0 MB"         
    "ConvWeightDataOffset"      "0x02c00000"     "12.0 MB"        
    "FCWeightDataOffset"        "0x03800000"     "84.0 MB"        
    "EndOffset"                 "0x08c00000"     "Total: 140.0 MB"

ans = 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
```

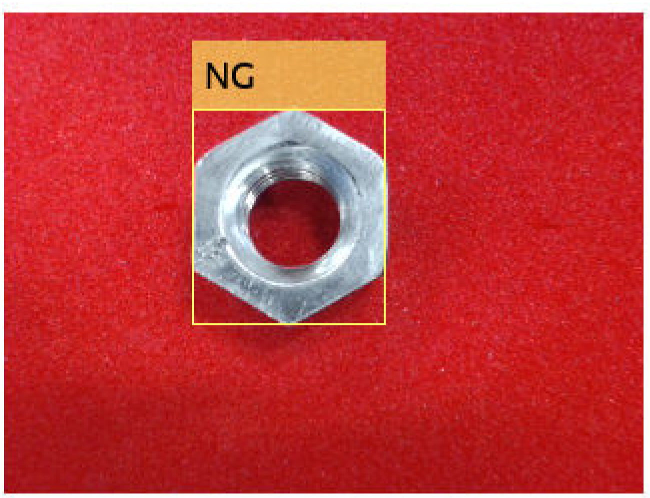
```
### FPGA bitstream programming has been skipped as the same bitstream is already loaded on the ta
### Deep learning network programming has been skipped as the same network is already loaded on t
```

Run Prediction for One Image

Load an image from the attached `testImages` folder, 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=[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 > 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

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    12199544                  0.05545                       1          121
    conv_module             3292478                  0.01497
        conv1                412777                  0.00188
        norm1                173433                  0.00079
        pool1                 58705                  0.00027
        conv2                656607                  0.00298
        norm2                128094                  0.00058
        pool2                 53221                  0.00024
        conv3                780491                  0.00355
        conv4                600179                  0.00273
        conv5                409095                  0.00186
        pool5                 19991                  0.00009
    fc_module               8907066                  0.04049
        fc6                 1759795                  0.00800
        fc7                 7030223                  0.03196
        fc8                  117046                  0.00053
 * The clock frequency of the DL processor is: 220MHz

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

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

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



Create Workflow Object for trainedBlemDetNet Network

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. Specify the saved pretrained `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 a 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

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

```
hW.compile
```

```
          offset_name           offset_address    allocated_space
    _______________________     ______________    _______________

    "InputDataOffset"            "0x00000000"     "8.0 MB"
    "OutputResultOffset"         "0x00800000"     "4.0 MB"
    "SystemBufferOffset"         "0x00c00000"     "28.0 MB"
    "InstructionDataOffset"      "0x02800000"     "4.0 MB"
    "ConvWeightDataOffset"       "0x02c00000"     "4.0 MB"
    "FCWeightDataOffset"         "0x03000000"     "36.0 MB"
    "EndOffset"                  "0x05400000"     "Total: 84.0 MB"

ans = 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.
### 50% finished, current time is 28-Jun-2020 12:33:36.
### FC Weights loaded. Current time is 28-Jun-2020 12:33:37
```

Run Prediction for One Image

Load an image from the attached `testImages` folder, 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=[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

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                     4886257                  0.02221                      1           48
    conv_module             1256664                  0.00571
        conv_1               467349                  0.00212
        maxpool_1            191204                  0.00087
        crossnorm            159553                  0.00073
        conv_2               397552                  0.00181
        maxpool_2             41066                  0.00019
    fc_module               3629593                  0.01650
        fc_1                3614829                  0.01643
        fc_2                  14763                  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 annotation for postprocessing
    out = myNDNet_Postprocess(Iori, num, bbox, scores, wi, he, ch);

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

OK

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 r


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 28-Jun-2020 12:24:21
```

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`.

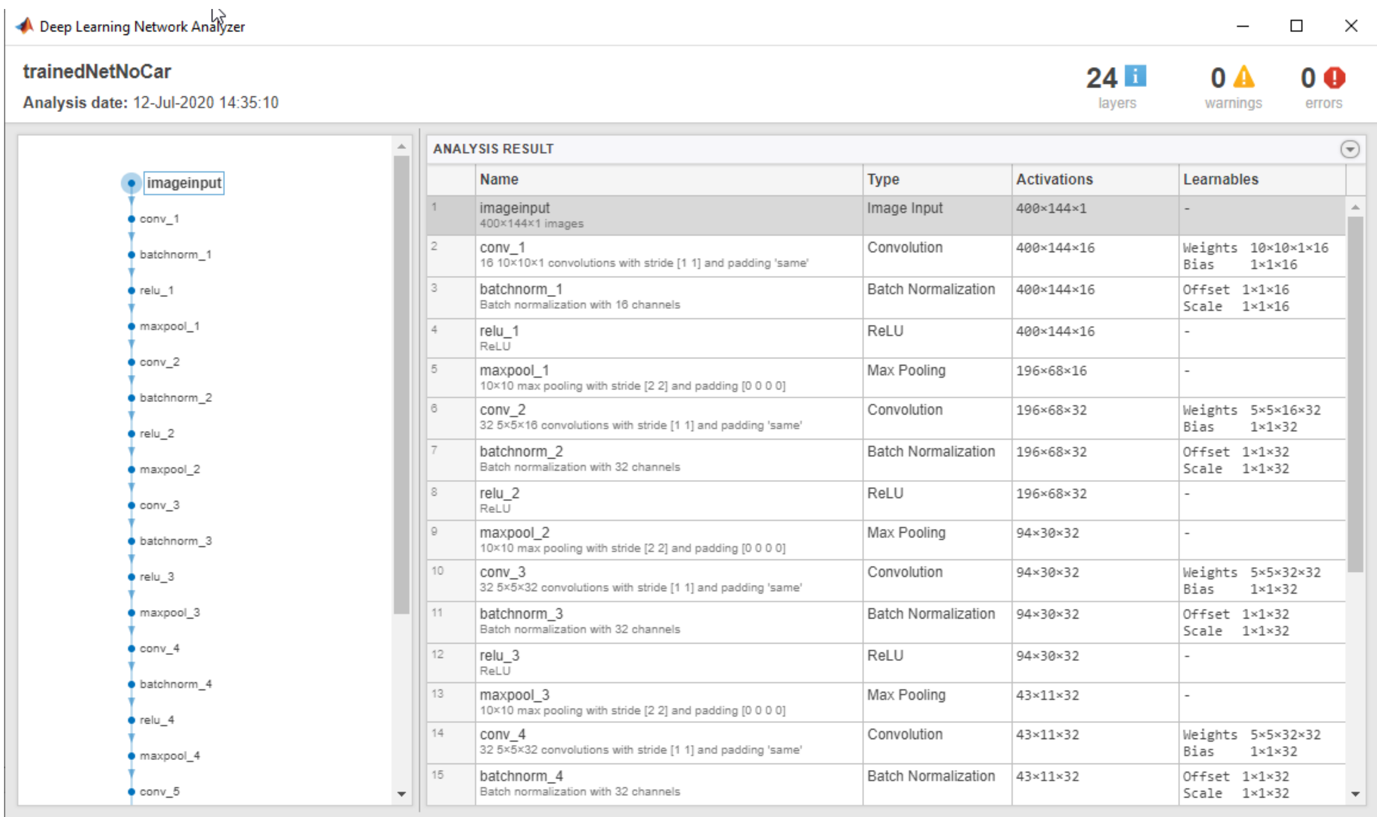
Load Data and Network

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);
```



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
```

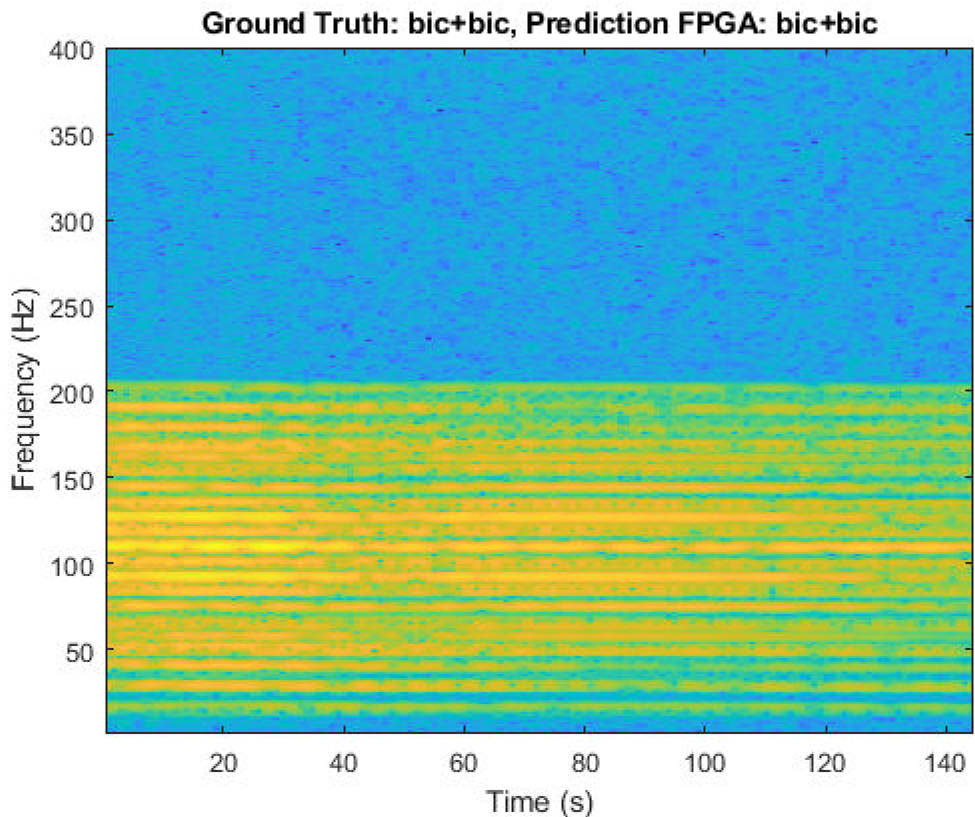
Load five random images from the sample test data set and execute the predict function of the `dlhdl.Workflow` object to display the labels alongside the signatures. The predictions will happen at once since the input is concatenated along the fourth dimension.

```
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           471
    conv_module           9427488                  0.04285
        conv_1            4195175                  0.01907
        maxpool_1         1394705                  0.00634
        conv_2            1975204                  0.00898
        maxpool_2          706332                  0.00321
        conv_3             813499                  0.00370
        maxpool_3          121869                  0.00055
        conv_4             148063                  0.00067
        maxpool_4           22019                  0.00010
        conv_5              42053                  0.00019
        avgpool2d            8684                  0.00004
    fc_module               19441                  0.00009
```

```
        fc                      19441                   0.00009
 * The clock frequency of the DL processor is: 220MHz
```

```
[~, idx2] = max(scores, [], 2);
predTestLabelBatch = classnames(idx2);

% Display the micro-doppler signatures along with the ground truth and
% predictions.
for k = 1:numView
    index = listIndex(k);
    imagesc(testDataNoCar(:, :, :, index));
    axis xy
    xlabel('Time (s)')
    ylabel('Frequency (Hz)')
    title('Ground Truth: '+string(testLabelNoCar(index))+', Prediction FPGA: '+string(predTestLal
    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.

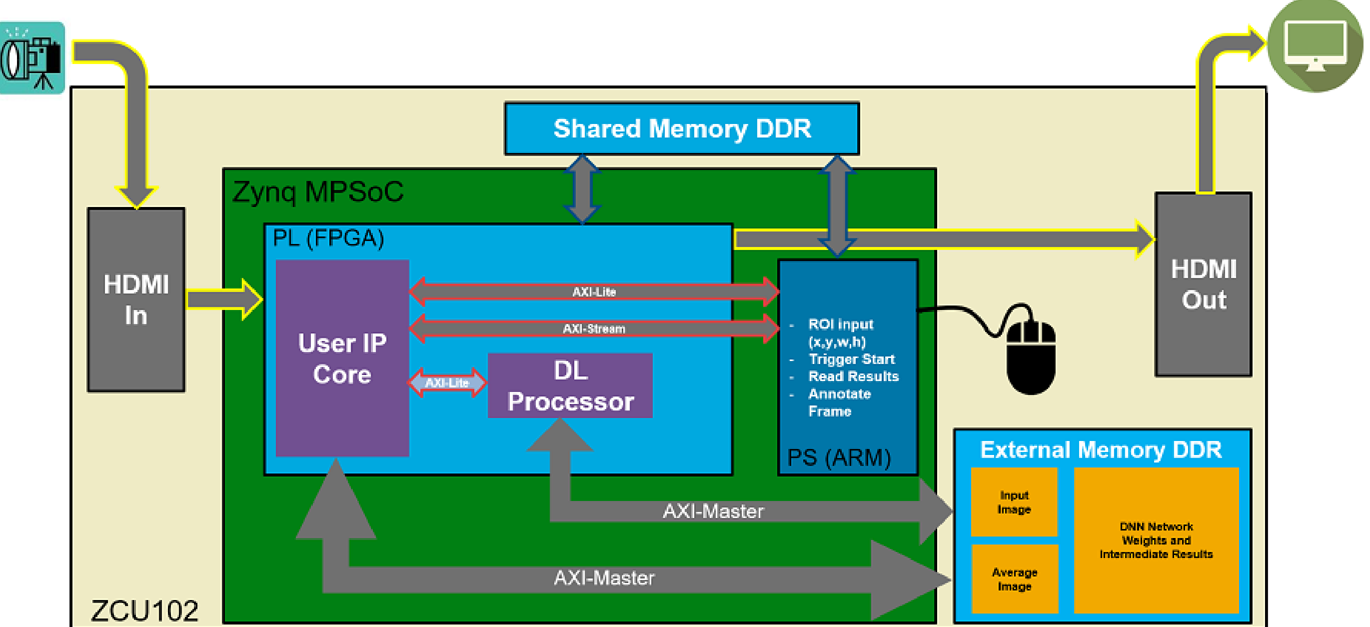
See Also

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 10-52.

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"	

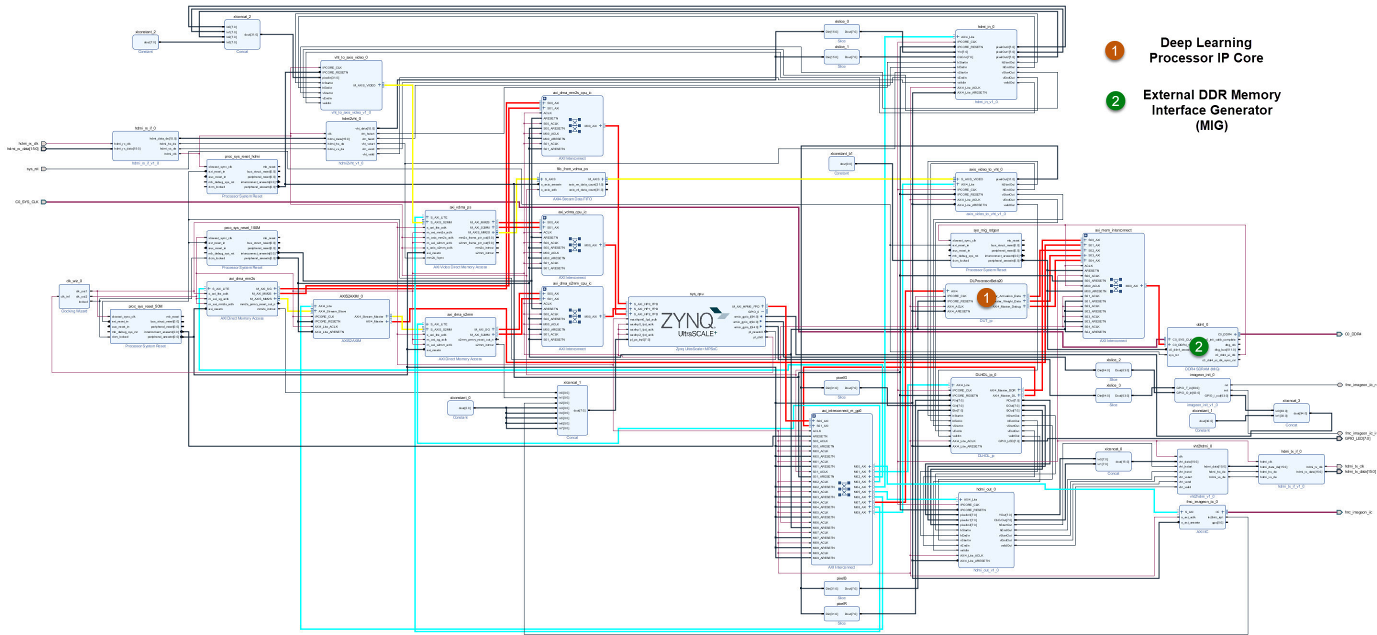
The HTML report contains a description of the deep learning processor IP core, instructions for using the core and integrating the core into your Vivado™ reference design, and a lsit of AXI4 registers. You will need the AXI4 register list to enter addresses into the Vivado™ Address Mapping tool. For more information about the AXI4 registers, see "Deep Learning Processor Register Map" on page 12-7.

Integrate the Generated Deep Learning Processor IP Core into the Reference Design

Insert the generated deep learning processor IP core into your reference design. After inserting the generated deep learning processor IP core into the reference design, you must:

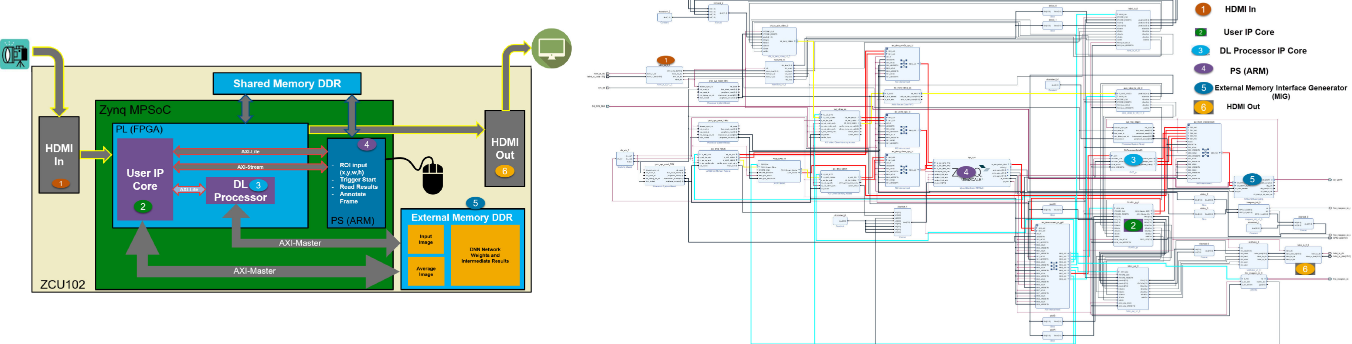
- Connect the generated deep learning processor IP core AXI4 slave interface to an AXI4 master device such as a JTAG AXI master IP core or a Zynq™ processing system (PS). Use the AXI4 master device to communicate with the deep learning processor IP core.
- Connect the vendor provided external memory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

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



Connect the External Memory Interface Generator

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



Create the Reference Design Definition File

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

```
function hRD = plugin_rd(varargin)

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

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

Verify the Reference Design

After creating the reference design, use the HDL Coder™ IP core generation workflow to generate the bitstream and program the ZCU102 board. You can then use MATLAB® and a `dlhdl.Workflow` object to verify the deep learning processor IP core or you can use the HDL Coder™ workflow to prototype the entire system. To verify the reference design, see “Run a Deep Learning Network on FPGA with Live Camera Input” on page 10-52.

Run a Deep Learning Network on FPGA with Live Camera Input

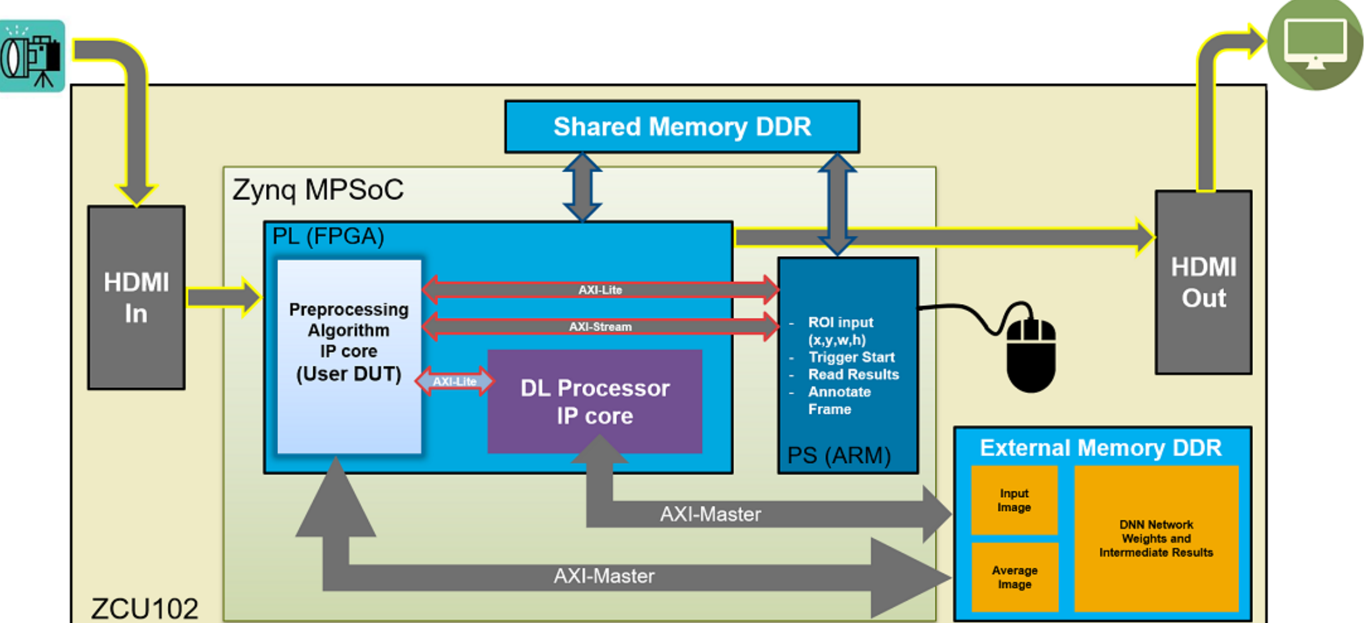
This example shows how to model preprocessing logic that receives a live camera input. You implement it on a Zynq® Ultrascale+(TM) MPSoC ZCU102 board by using a custom video reference design that has an integrated deep learning processor IP core for object classification. This example uses the HDL Coder™ HW/SW co-design workflow.

Introduction

In this example, you:

1. Model the preprocesing 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.



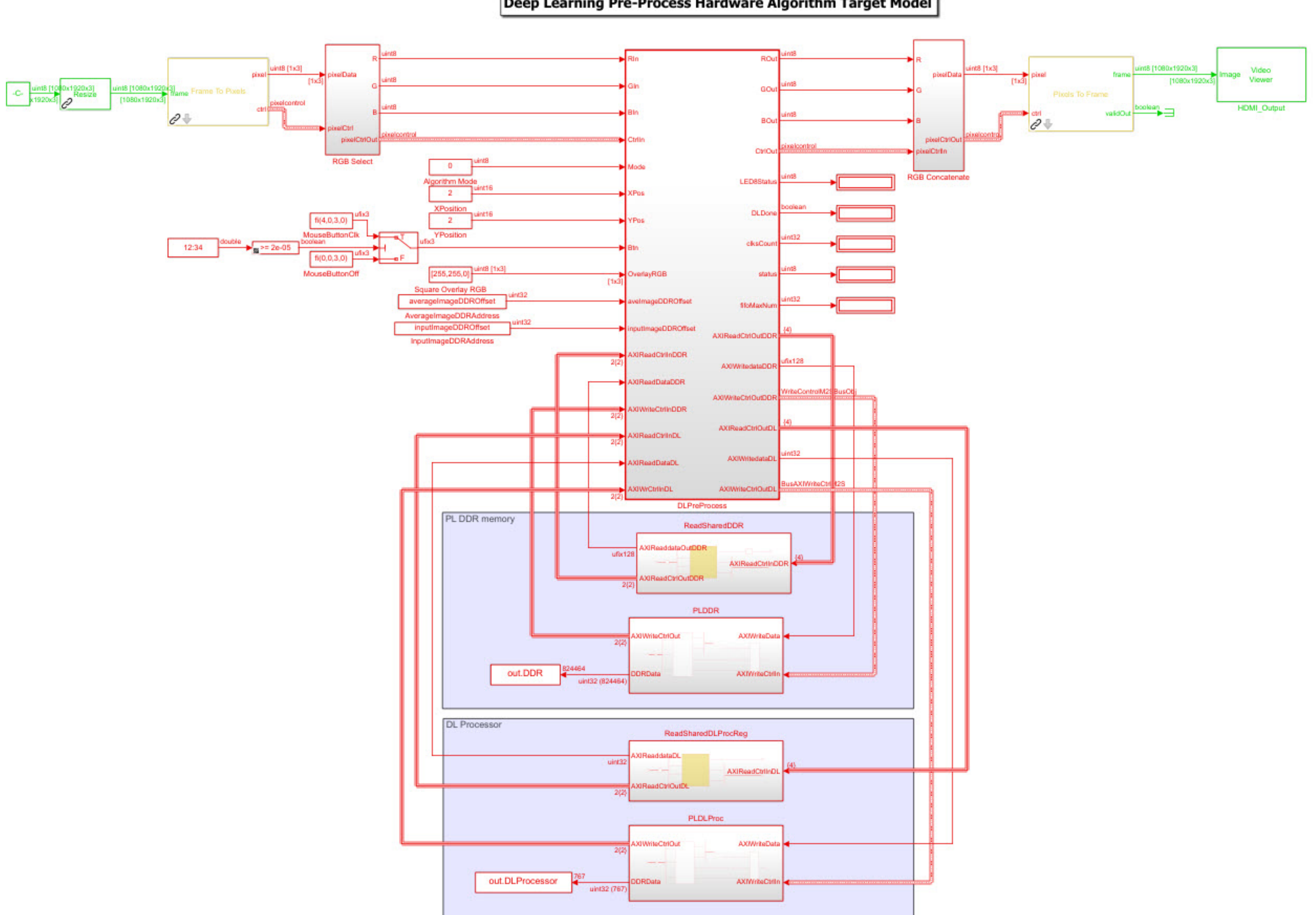
The objective of this system is to receive the live camera input through the HDMI input of the FMC daughter card on the ZCU102 board. You design the preprocessing logic in Simulink® to select and resize the region of interest (ROI). You then transmit the processed image frame to the deep learning processor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

Select and Resize the Region of Interest

Model the preprocesing 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 Preprocesing Logic HDL IP Core

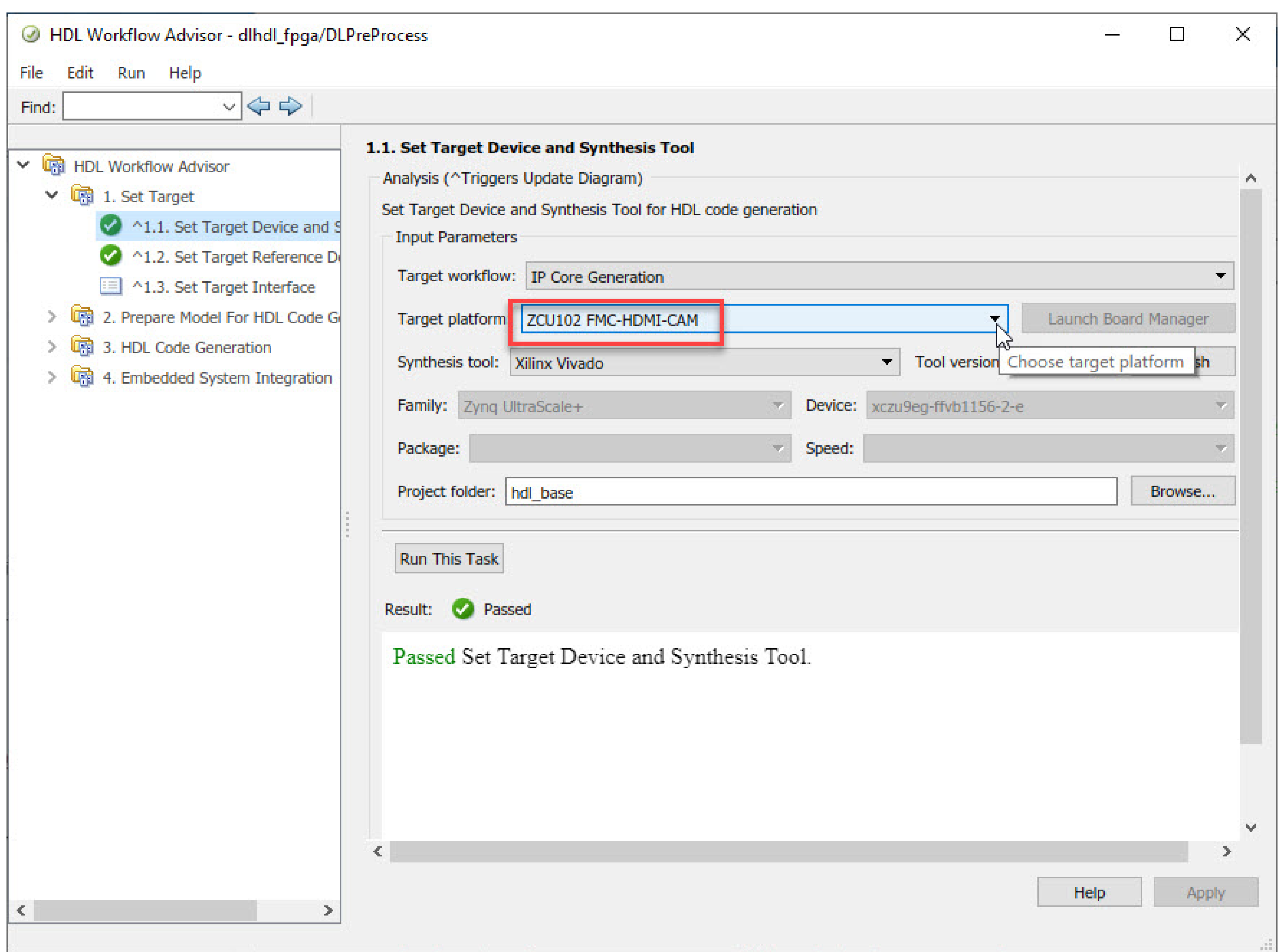
To implement the preprocessing logic model on a ZCU102 SoC board, create an HDL Coder™ reference design in Vivado™ which receives the live camera input and transmits the processed video data to the deep learning processor IP core. To create a custom video reference design that

integrates the deep learning processor IP core, see "Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core" on page 10-47.

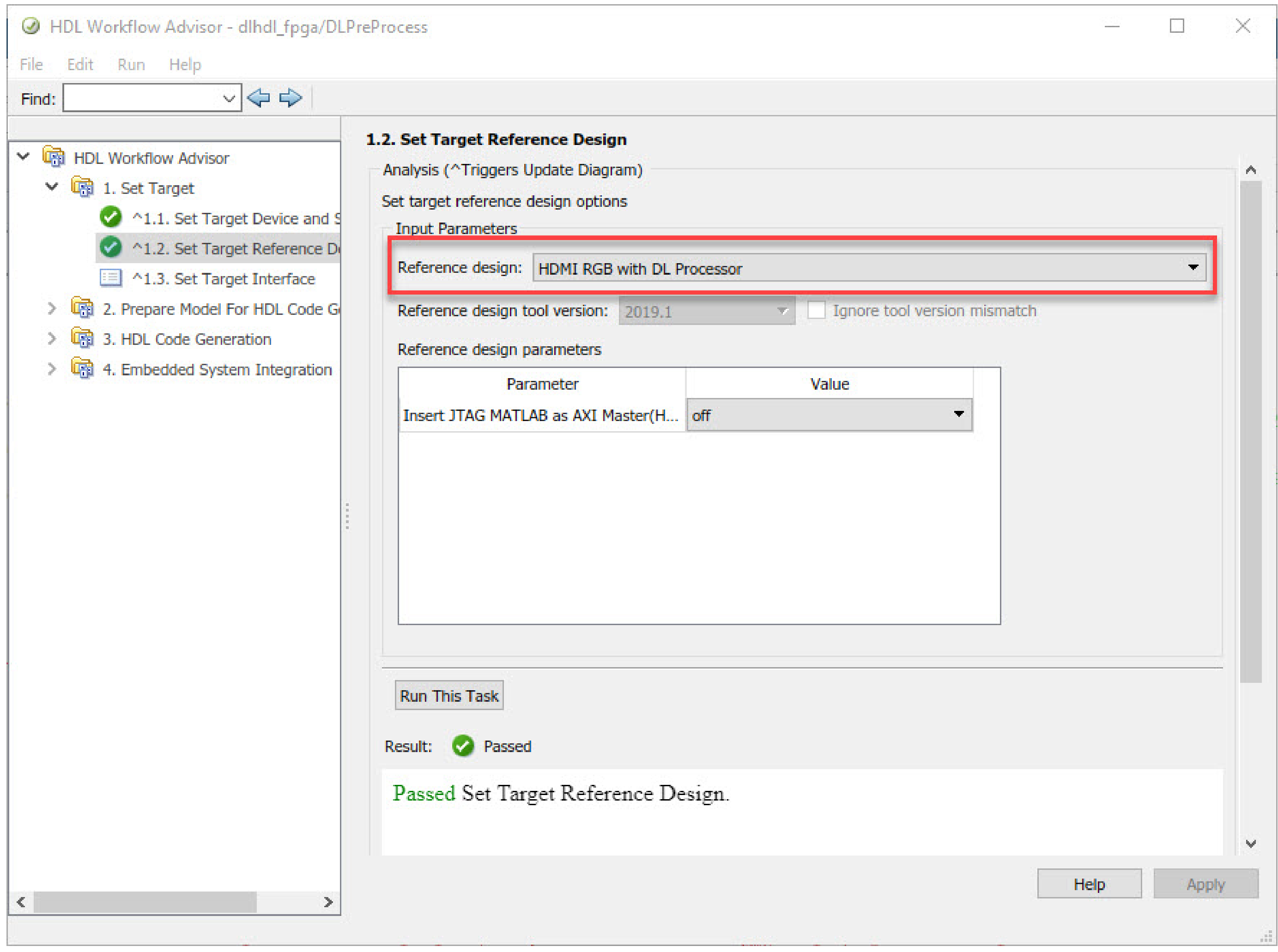
Start the HDL Coder HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy the preprocessing logic model on Zynq hardware. This workflow is the standard HDL Coder workflow. In this example the only difference is that this reference design contains the generated deep learning processor IP core. For more detais refer to the "Getting Started with Targeting Xilinx Zynq Platform" (HDL Coder) example.

1. Start the HDL Workflow Advisor from the model by right-clicking the `DLPreProcess` DUT subsystem and selecting **HDL Advisor Workflow**.

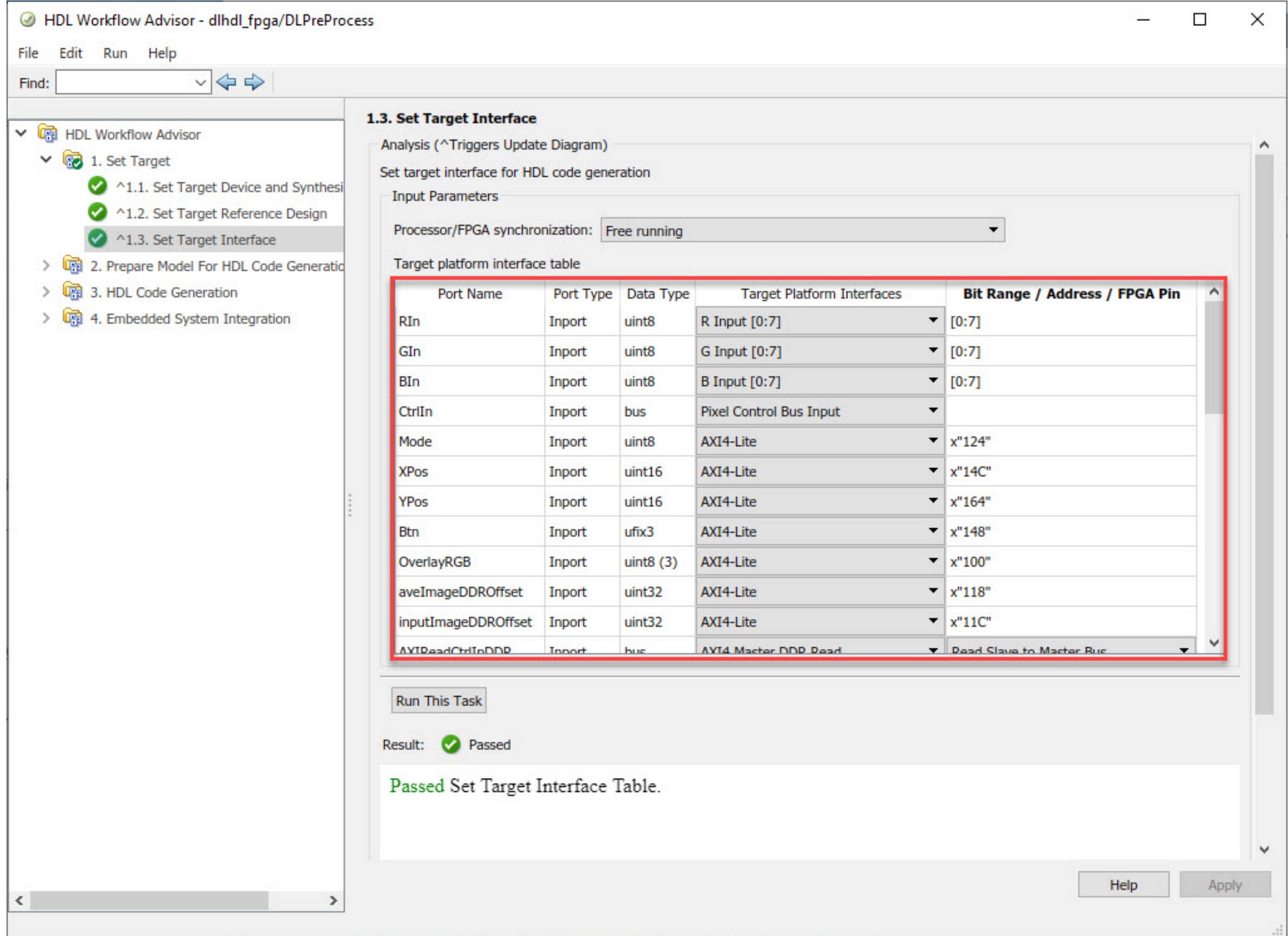
In Task 1.1, **IP Core Generation** is selected for **Target workflow** and **ZCU102-FMC-HDMI-CAM** is selected for **Target platform**.



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



In Task 1.3, the **Target platform interface table** is loaded as shown in the following screenshot. Here you can map the ports of the DUT subsystem to the interfaces in the reference design.



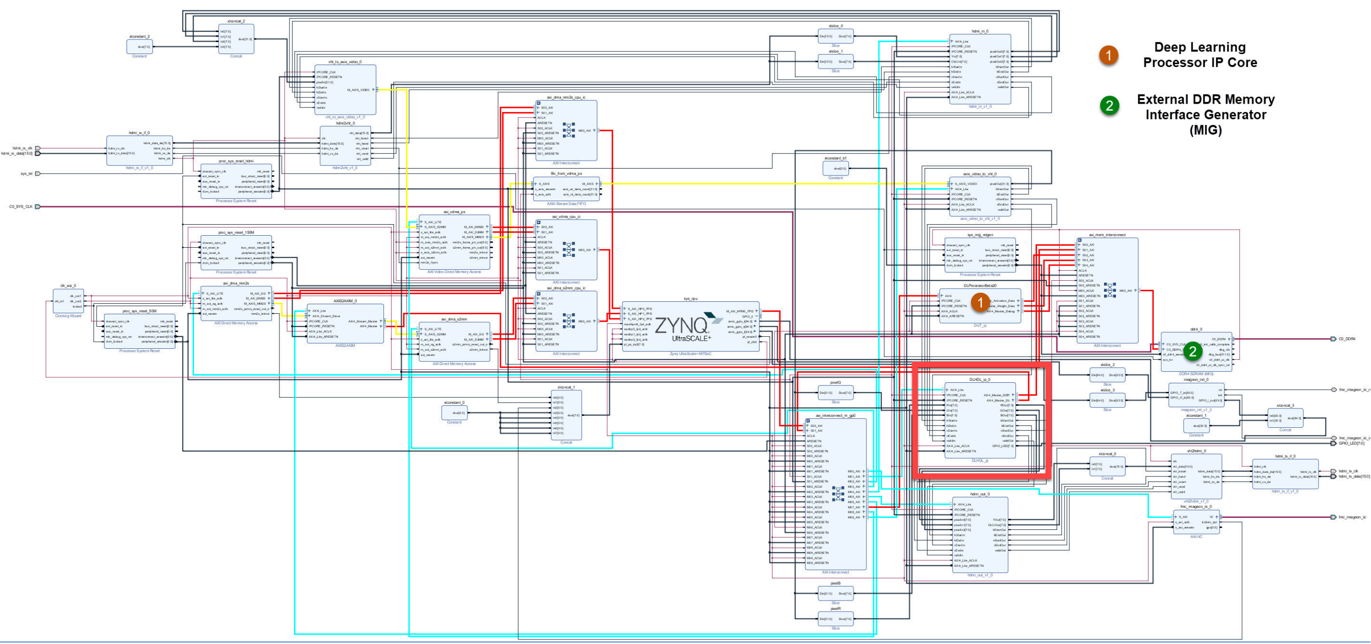
2. Right-click Task 3.2, **Generate RTL Code and IP Core**, and then select **Run to Selected Task**. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

Integrate IP into the Custom Video Reference Design

In the HDL Workflow Advisor, run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Zynq hardware.

1. Run Task 4.1, **Create Project**. This task inserts the generated IP core into the **HDMI RGB with DL Processor** reference design. To create a reference design that integrates the deep learning processor IP core, see “Authoring a Reference Design for Live Camera Integration with Deep Learning Processor IP Core” on page 10-47.

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 procesor IP core and the Zynq processor.



3. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model and build and download the FPGA bitstream.

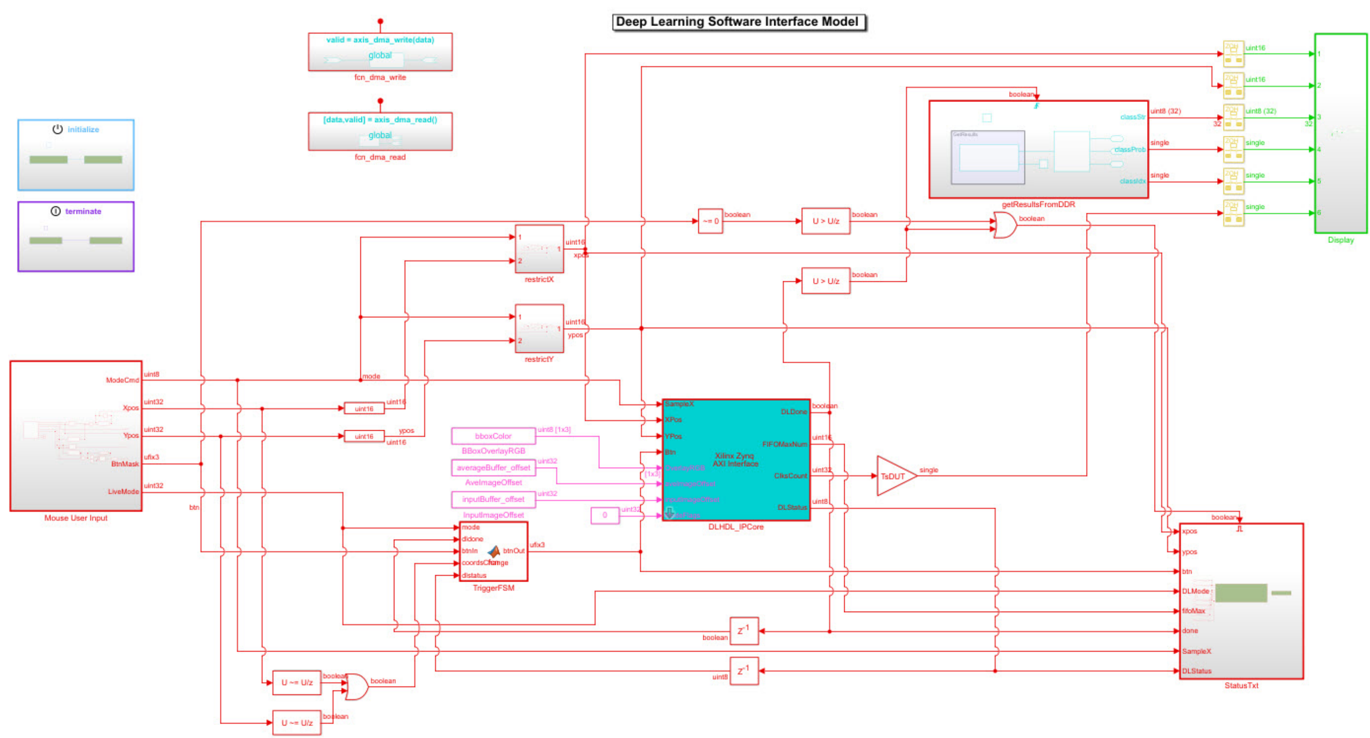
Deploy and Validate the Integrated Reference Design

To validate the integrated reference design that includes the generated preprocessing logic IP core, deep learning processor IP core, and the Zynq processor:

1. Individually validate the preprocessing logic on the FPGA board.
2. Individually validate the deep learning processor IP core functionality by using the Deep Learning HDL Toolbox™ prototyping workflow.
3. Deploy and validate the entire system on a ZCU102 board.
4. Deploy the entire system as an executable file on the SD card on the ZCU102 board.

1. Using the standard HDL Coder hardware/software co-design workflow, you can validate that the preprocessing logic works as expected on the FPGA. The HDL Workflow Advisor generates a software interface subsystem during Task 4.2 **Generate Software Interface Model**, which you can use in your software model for interfacing with the FPGA logic. From the software model, you can tune and probe the FPGA design on the hardware by using Simulink External Mode. Instruct the FPGA preprocessing logic to capture an input frame and send it to the external DDR memory.

You can then use `fpga` object to create a connection from MATLAB to the ZCU102 board and read the contents of the external DDR memory into MATLAB for validation. to use the `fpga` object, see “Create 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.

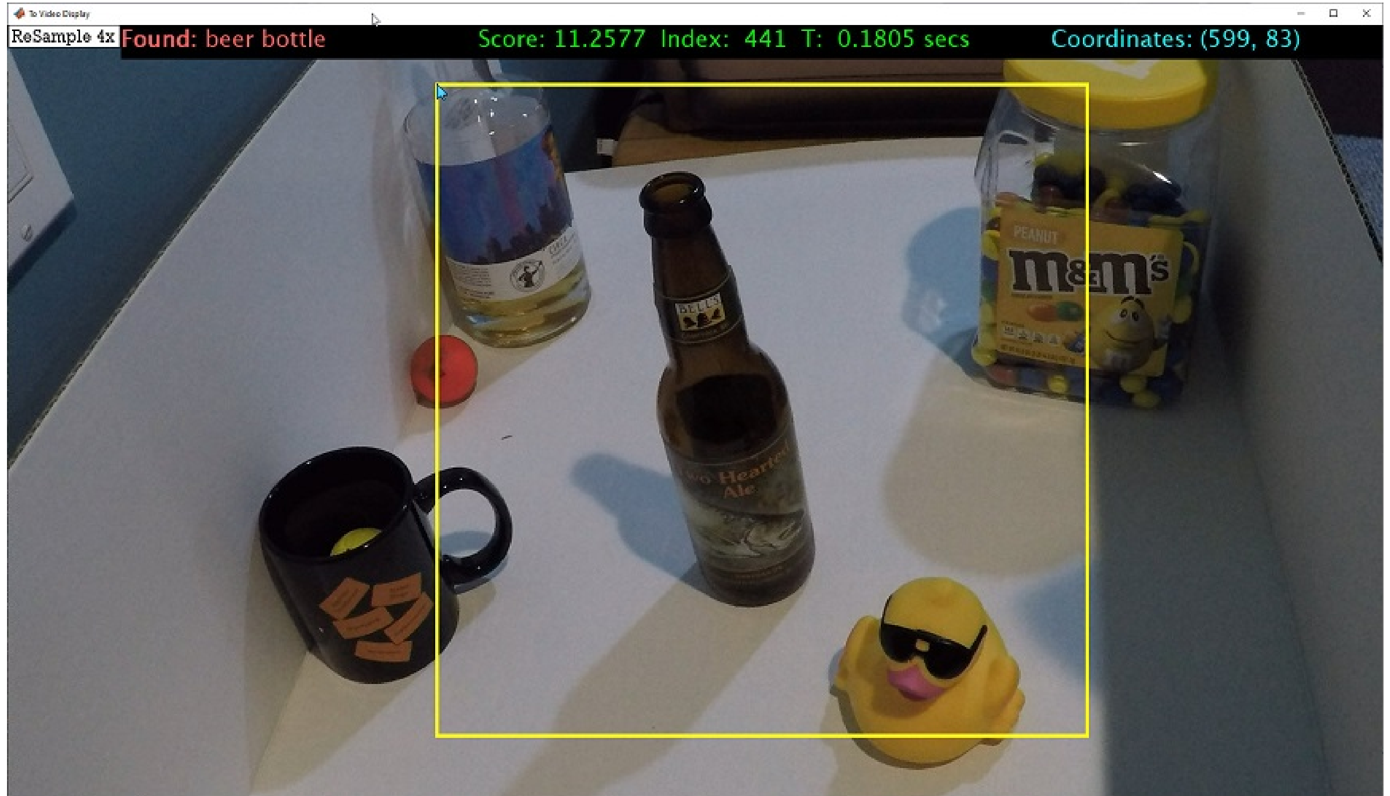
3. After you individually validate the preprocessing logic IP core and the deep learning processor IP core, you can prototype the entire integrated system on the FPGA board. Using Simulink External mode, instruct the FPGA preprocessing logic to send a processed input image frame to the DDR buffer, instruct 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 screenshots shows that you can read the ARM processor prediction result by using a serial connection.

COM5 - PuTTY

```
 1) envelope  8.8149 550
 2) laptop  7.6177 621
 3) binder  7.4577 447
 4) notebook  7.4564 682
 5) rule  7.4436 770
Class: envelope  Prob: 8.814860  Idx: 550.000000
SampleX: 2 XY:[117, 22]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7D
Top 5  Run 280
---------------------------------
 1) velvet 10.0684 886
 2) envelope  9.0011 550
 3) rule  8.8459 770
 4) wool  8.8402 912
 5) jean  8.4882 609
Class: velvet  Prob: 10.068416  Idx: 886.000000
SampleX: 2 XY:[970, 175]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7F
Top 5  Run 281
---------------------------------
 1) velvet 10.6247 886
 2) envelope  9.8796 550
 3) wool  9.2945 912
 4) rule  9.0598 770
 5) bath towel  8.8611 435
Class: velvet  Prob: 10.624667  Idx: 886.000000
SampleX: 2 XY:[993, 154]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7C
Top 5  Run 282
---------------------------------
 1) lipstick 10.4688 630
 2) pill bottle  8.7858 721
 3) beer bottle  8.5406 441
 4) thimble  8.4648 856
 5) saltshaker  8.3658 774
Class: lipstick  Prob: 10.468786  Idx: 630.000000
SampleX: 2 XY:[1084, 230]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7F
Top 5  Run 283
---------------------------------
 1) lipstick 10.1775 630
 2) pill bottle  9.0086 721
 3) loupe  8.8113 634
 4) hair spray  8.7907 586
 5) beer bottle  8.4889 441
Class: lipstick  Prob: 10.177537  Idx: 630.000000
SampleX: 2 XY:[1151, 233]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7D
Top 5  Run 284
---------------------------------
 1) beer bottle 15.1420 441
 2) whiskey jug 12.0200 902
 3) wine bottle 11.8346 908
 4) vase 11.3211 884
 5) pop bottle 11.2343 738
Class: beer bottle  Prob: 15.141971  Idx: 441.000000
SampleX: 2 XY:[1155, 233]  Btn: 0 Mode: 1 FIFOMax: 194 DLDone: 1 Status: 0x7E
Top 5  Run 285
---------------------------------
 1) beer bottle 15.5207 441
 2) pop bottle 11.7170 738
 3) wine bottle 11.7112 908
 4) whiskey jug 10.4509 902
 5) vase  9.9780 884
Class: beer bottle  Prob: 15.520669  Idx: 441.000000
```

This screenshot shows the frame captured from the output video stream which includes the ROI selection and the annotated prediction result.



4. After completing all your verification steps, manually deploy the entire reference design as an executable on the SD card on the ZCU102 board by using the ARM processor. Once the manual deployment is completed a MATLAB connection to the FPGA board is not required to operate the reference design.

Running Convolution-Only Networks by using FPGA Deployment

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.

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 Subset of Resnet-50 Network

To examine the outputs of the `max_pooling2d_1` layer, create this network which is a subset of the ResNet-50 network:

```
layers = rnet.Layers(1:5);
outLayer = regressionLayer('Name','output');
layers(end+1) = outLayer;

snet = assembleNetwork(layers);
```

Create Target Object

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 ResNet-50 subset 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);
```

Compile Modified Resnet-50 Series Network

To compile the modified ResNet-50 series 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. It 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.

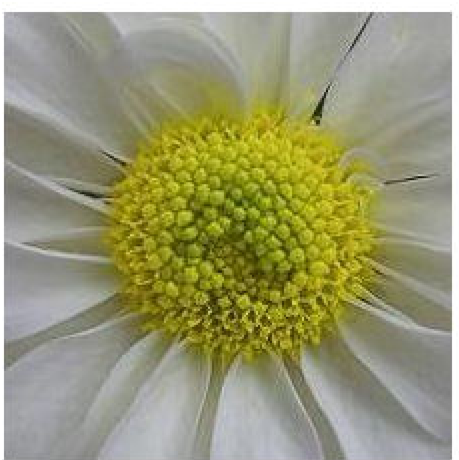
```
hW.deploy
```

```
### 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 4 dimensions, which is appropriate input to the `imtile` function

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

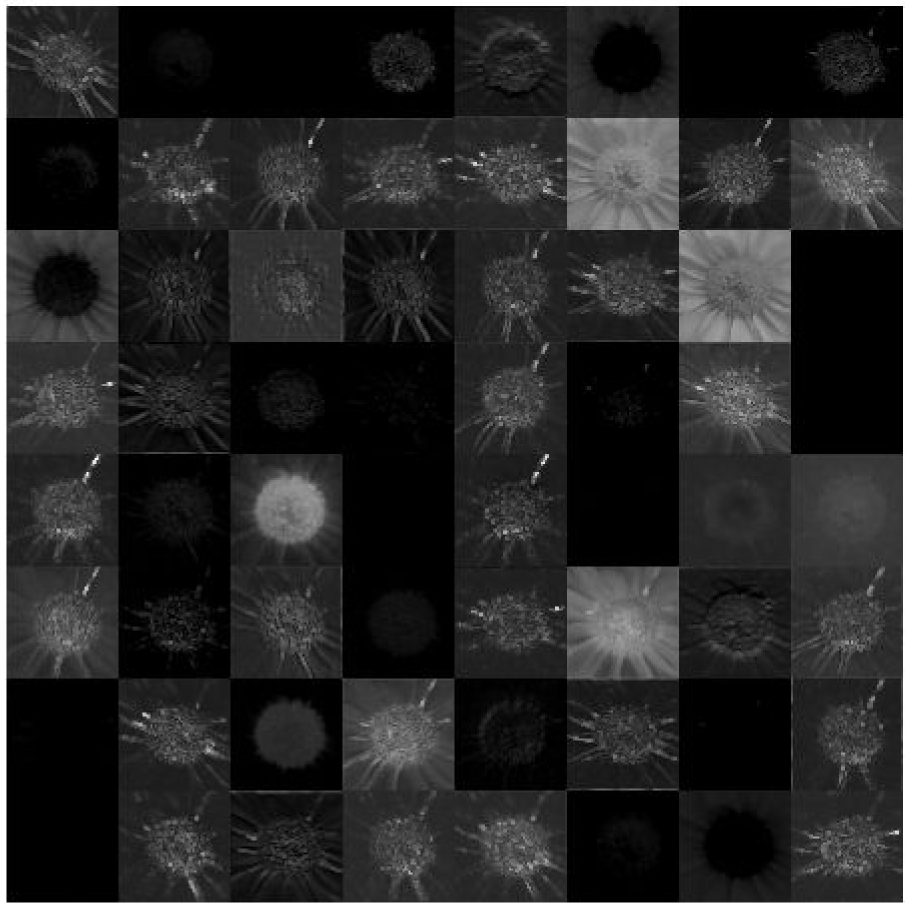
```
sz = 1×4

    56    56     1    64
```

The 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]);
```

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 size 1 because the activations do not have color. The fourth dimension indexes the channel. A gride size of 8x8 is selected because there are 64 features to display.

```
imshow(J)
```



Bright features indicate a strong activation. To understand and debug convolutional networks, running and visualizing data is a useful tool.

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 and Boards” on page 7-2.

Prerequisites

- Xilinx ZCU102 SoC development kit. For help with board setup, see “Guided SD Card Setup” (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

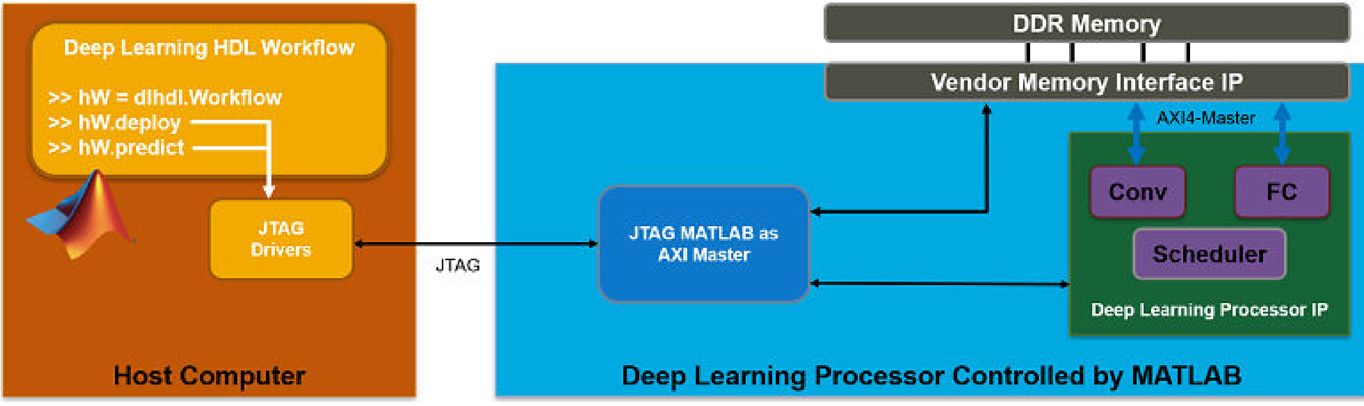
There are two hardware interfaces for establishing a connection between the host computer and FPGA board: JTAG and Ethernet.

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.

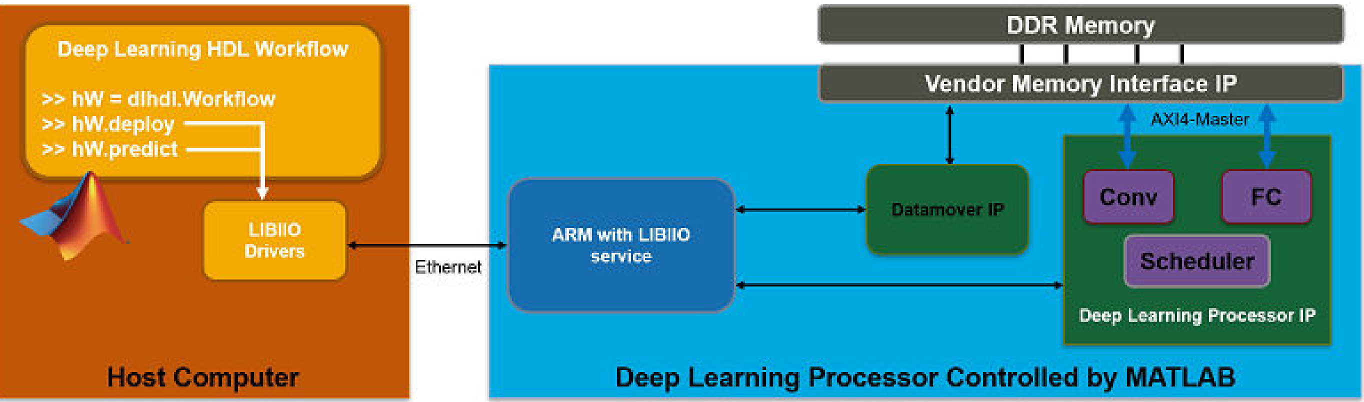


Ethernet Interface

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.\



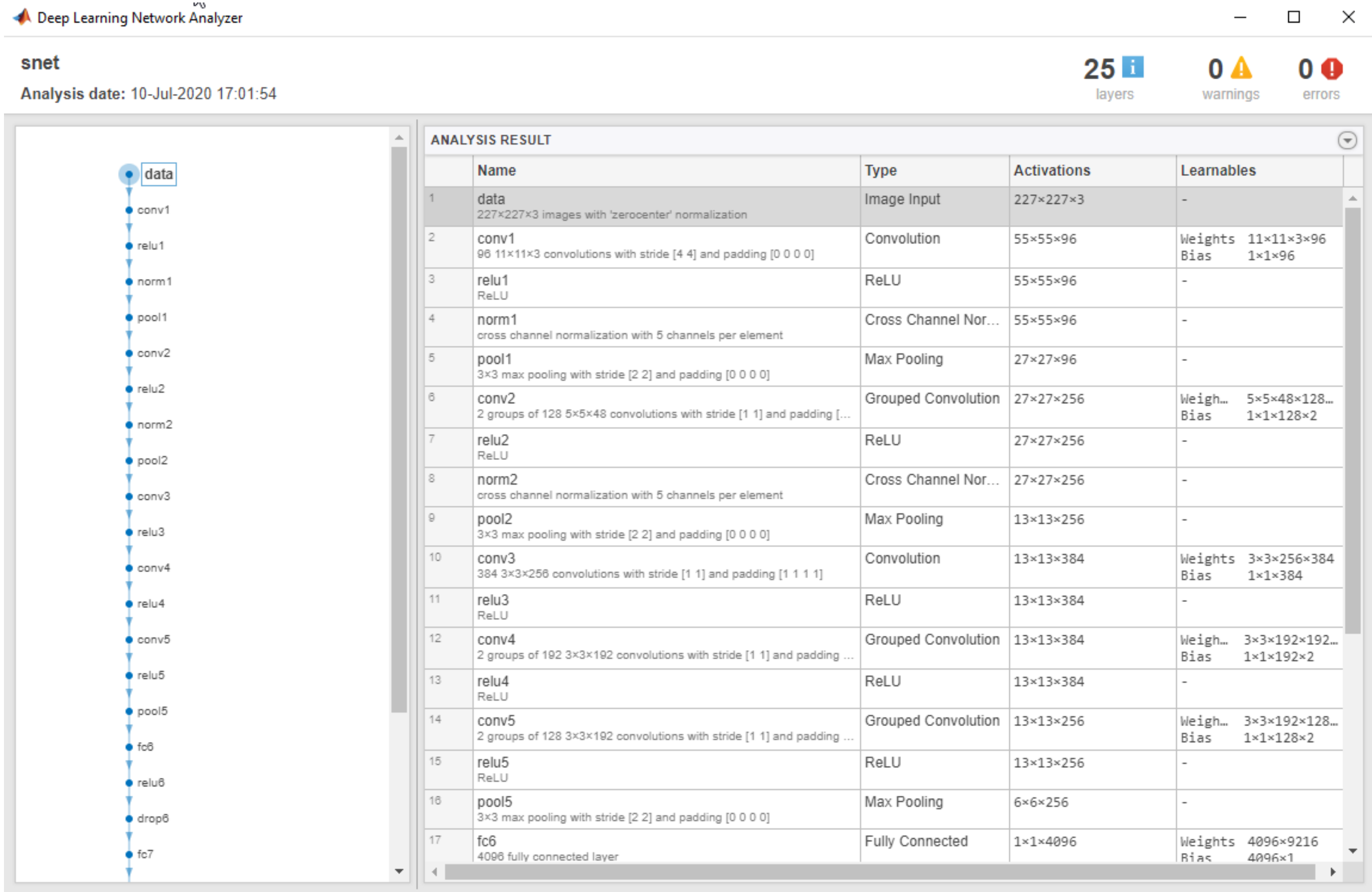
Load and Compile Deep Learning Network

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 normaliz
% max pool, fully connected, and the softmax output layers.
```



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.b
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 netwok 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 r


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
```

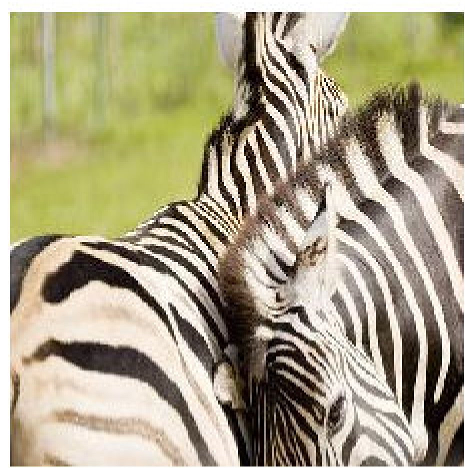
```
elapsedTimeEthernet = 47.5854
```

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');
```

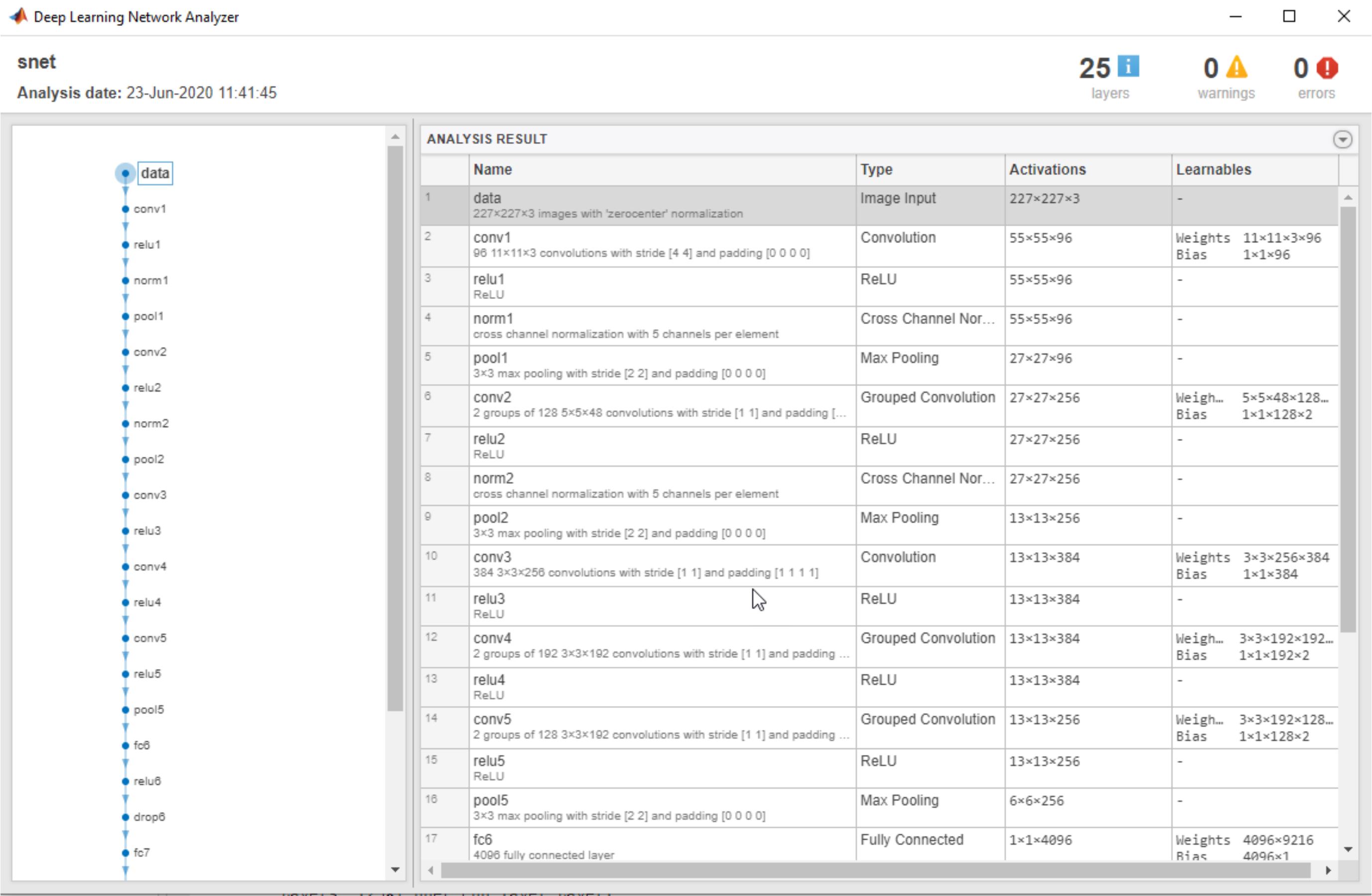
Load Pretrained Network

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)
```



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.

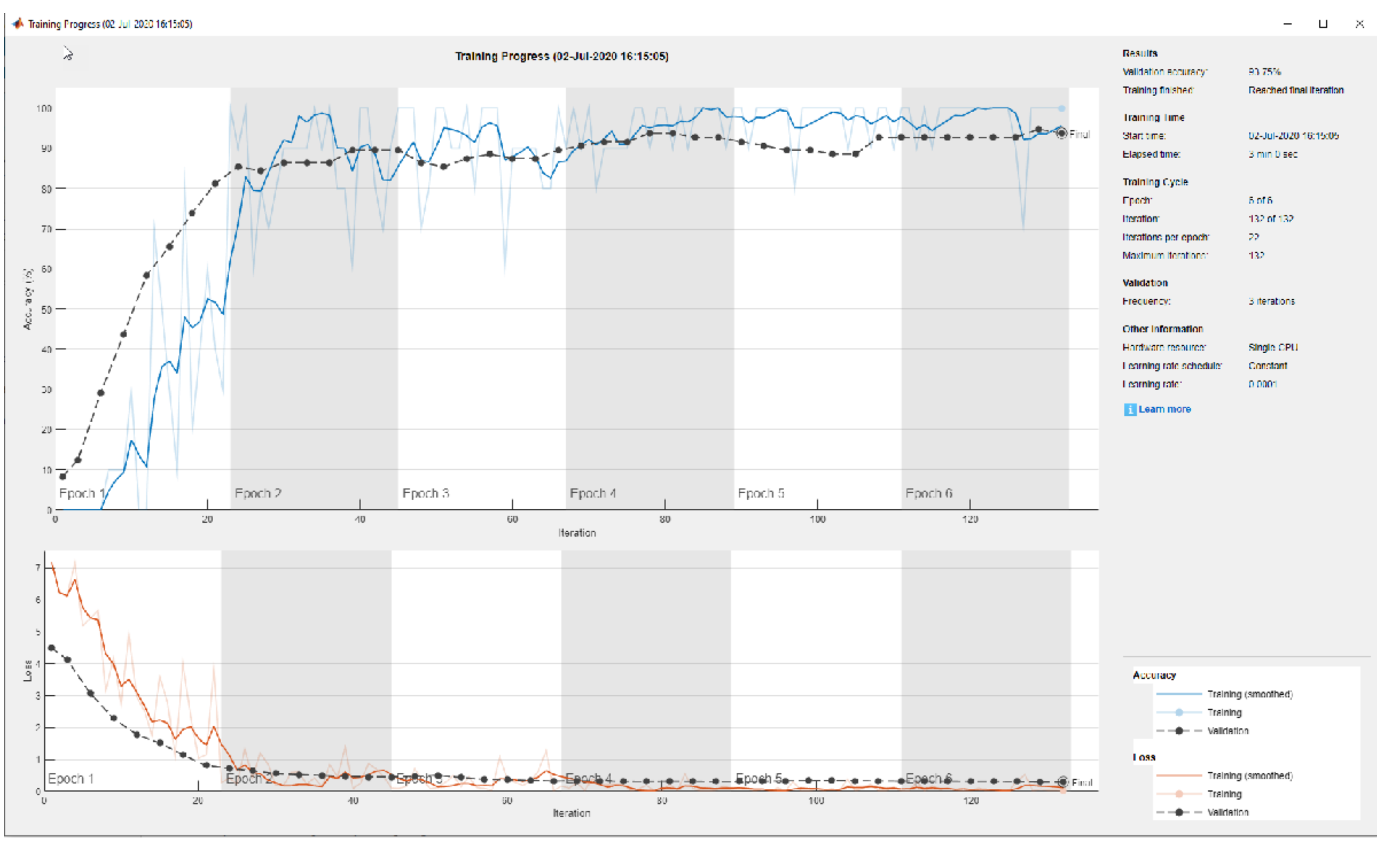
```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

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 3.0 or higher). Otherwise, it uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

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


Training Progress (02-Jul-2020 16:15:05)
Results
Validation accuracy: 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
Accuracy
Training (smoothed)
Training
Validation
Loss
Training (smoothed)
Training
Validation
Accuracy (%)
Loss
Iteration
Epoch 1
Epoch 2
Epoch 3
Epoch 4
Epoch 5
Epoch 6
Final

Vehicle Detection Using 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. 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 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 automatically given a pretrained ResNet-50 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 `alexnet` to load a pretrained AlexNet 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 further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis.

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

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);
```

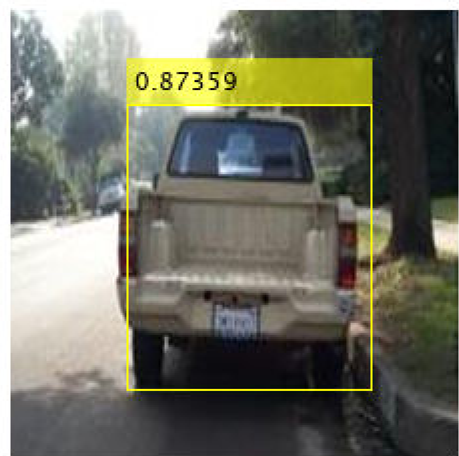
```
*************************************************************************
Training a YOLO v2 Object Detector for the following object classes:

* 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 |       00:00:01 |         7.23 |         52.3 |          0.0010 |
|       5 |          50 |       00:00:35 |         0.98 |          1.0 |          0.0010 |
|      10 |         100 |       00:01:13 |         0.78 |          0.6 |          0.0010 |
|      14 |         150 |       00:01:51 |         0.64 |          0.4 |          0.0010 |
|      19 |         200 |       00:02:29 |         0.59 |          0.3 |          0.0010 |
|      20 |         220 |       00:02:43 |         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)
```



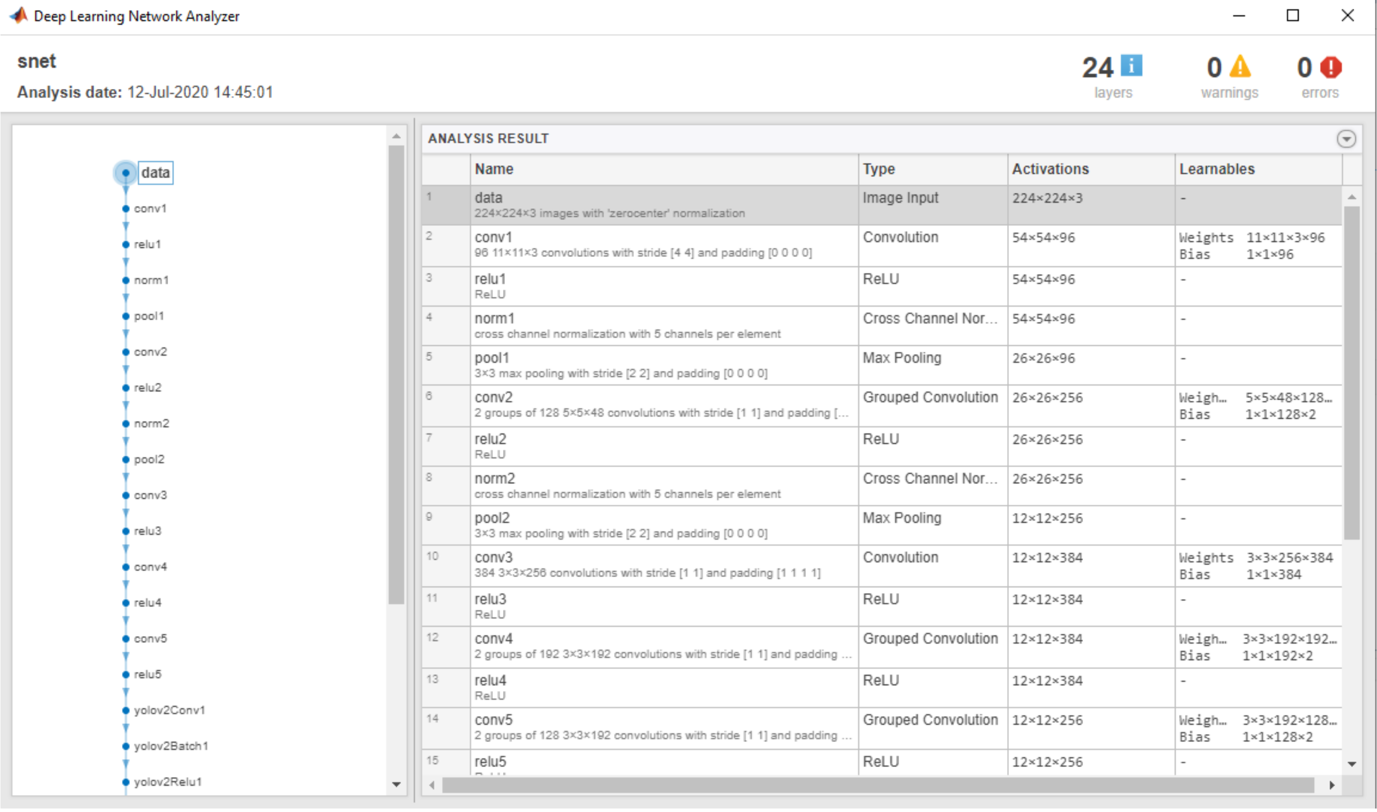
Load Pretrained Network

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
```

```
### 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"     "4.0 MB"
    "SystemBufferOffset"        "0x01c00000"     "28.0 MB"
    "InstructionDataOffset"     "0x03800000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x03c00000"     "16.0 MB"
    "EndOffset"                 "0x04c00000"     "Total: 76.0 MB"

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

```
### 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 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

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Tota
                         -------------             -------------              ---------        --
Network                    8724510                  0.03966                       1          87
    conv_module            8724510                  0.03966
        conv1              1355434                  0.00616
        norm1               581412                  0.00264
        pool1               219416                  0.00100
        conv2              2208308                  0.01004
        norm2               368019                  0.00167
        pool2               221821                  0.00101
        conv3               982880                  0.00447
```

```
        conv4              772573                  0.00351
        conv5              533396                  0.00242
        yolov2Conv1        667481                  0.00303
        yolov2Conv2        668607                  0.00304
        yolov2ClassConv    145300                  0.00066
 * 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.

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 = 

                    Processing Module "conv"
                            ConvThreadNumber: 16
                             InputMemorySize: [227  227    3]
                            OutputMemorySize: [227  227    3]
                            FeatureSizeLimit: 1024

                      Processing Module "fc"
                              FCThreadNumber: 4
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

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

Create Workflow Object

Create a `dlhdl.Workflow` object. Specify `snet` as the network and `hPC` as the `ProcessorConfig`.

```
hW = dlhdl.Workflow('Network',snet,'ProcessorConfig',hPC)
```

Estimate LogoNet Performance

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

```
hW.estimate('Performance')
```

The output of the estimate function is:

```
              Deep Learning Processor Estimator Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                    39356308                  0.17889                       1           39356308            5.6
    conv_module            35334882                  0.16061
        conv_1              6821351                  0.03101
        maxpool_1           3750444                  0.01705
        conv_2             10433717                  0.04743
        maxpool_2           1445648                  0.00657
        conv_3              9359533                  0.04254
        maxpool_3           1762640                  0.00801
        conv_4              1733588                  0.00788
        maxpool_4             27961                  0.00013
    fc_module               4021426                  0.01828
        fc_1                2420823                  0.01100
        fc_2                1549111                  0.00704
        fc_3                  51492                  0.00023
 * The clock frequency of the DL processor is: 220MHz
```

The estimated frames per second is 5.6 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;
hPC.TargetFrequency = 300;
hPCNew.setModuleProperty('conv', 'KernelDataType',   'int8');
hPCNew.setModuleProperty('conv', 'ConvThreadNumber', 64);
hPCNew.setModuleProperty('fc', 'KernelDataType',   'int8');
hPCNew.setModuleProperty('fc', 'FCThreadNumber',   16);
```

```
hPCNew =

                    Processing Module "conv"
                            ConvThreadNumber: 64
                             InputMemorySize: [227   227     3]
                            OutputMemorySize: [227   227     3]
                            FeatureSizeLimit: 1024

                      Processing Module "fc"
                              FCThreadNumber: 16
                             InputMemorySize: 25088
                            OutputMemorySize: 4096

                     System Level Properties
                              TargetPlatform: 'Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit'
                             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 estimate the performance of the LogoNet series network by using the new custom processor configuration, quantize the LogoNet network. For more information, see “Estimate Performance of Quantized LogoNet Running On ZCU102 Bitstream”. Use the quantized network object `dlquantObj` to estimate performance by using the new custom processor configuration.

Create Workflow Object

Create a `dlhdl.Workflow` object. Specify `dlQuantObj` as the network and `hPC` as the `ProcessorConfig`.

```
hW = dlhdl.Workflow('Network',dlquantObj,'ProcessorConfig',hPCNew)
```

Estimate LogoNet Performance

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

```
hW.estimate('Performance')
```

The output of the `estimate` function is:

```
          Deep Learning Processor Estimator Performance Results

                 LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                       -------------             -------------               ---------        ---------       ---------
Network                  13923758                  0.04641                       1          13923758            21.5
    conv_module          12737303                  0.04246
        conv_1            3327693                  0.01109
        maxpool_1         1876824                  0.00626
        conv_2            2936929                  0.00979
        maxpool_2          723536                  0.00241
        conv_3            2456212                  0.00819
        maxpool_3          882032                  0.00294
        conv_4             520052                  0.00173
        maxpool_4           14025                  0.00005
    fc_module            1186455                  0.00395
        fc_1               708503                  0.00236
        fc_2               453111                  0.00151
        fc_3                24841                  0.00008
 * The clock frequency of the DL processor is: 300MHz
```

The estimated frames per second is 21.5 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.ba
dlhdl.buildProcessor(hPCNew);
```

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

The generated bitstream in this example is similar to the `zcu102_int8` bitstream. To deploy the quantized LogoNet network using the `zcu102_int8` bitstream, see “Obtain Prediction Results for Quantized LogoNet Network”.

Deep Learning Quantization

- “Quantization of Deep Neural Networks” on page 11-2
- “Quantization Workflow Prerequisites” on page 11-9
- “Calibration” on page 11-10
- “Validation” on page 11-12
- “Code Generation and Deployment” on page 11-15
- “Deploy Quantized Neural Network” on page 11-17
- “Quantize Neural Network for FPGA Execution Environment” on page 11-22

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.

Quantization of a neural network requires a GPU, the GPU Coder™ Interface for Deep Learning Libraries support package, and the Deep Learning Toolbox Model Quantization Library support package. Using a GPU requires a CUDA® enabled NVIDIA® GPU with compute capability 6.1, 6.3 or higher.

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. For example, to begin, 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. Sum the number of MSB's in each column for 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 Display the MSB counts of each bit location 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 software assigns a data type that can represent the bit locations that capture the most information. In this example, the software selects a data type that represents bits from 2^3 to 2^{-3}. An additional sign bit is required to represent the signedness of the value.

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				
	✓																	

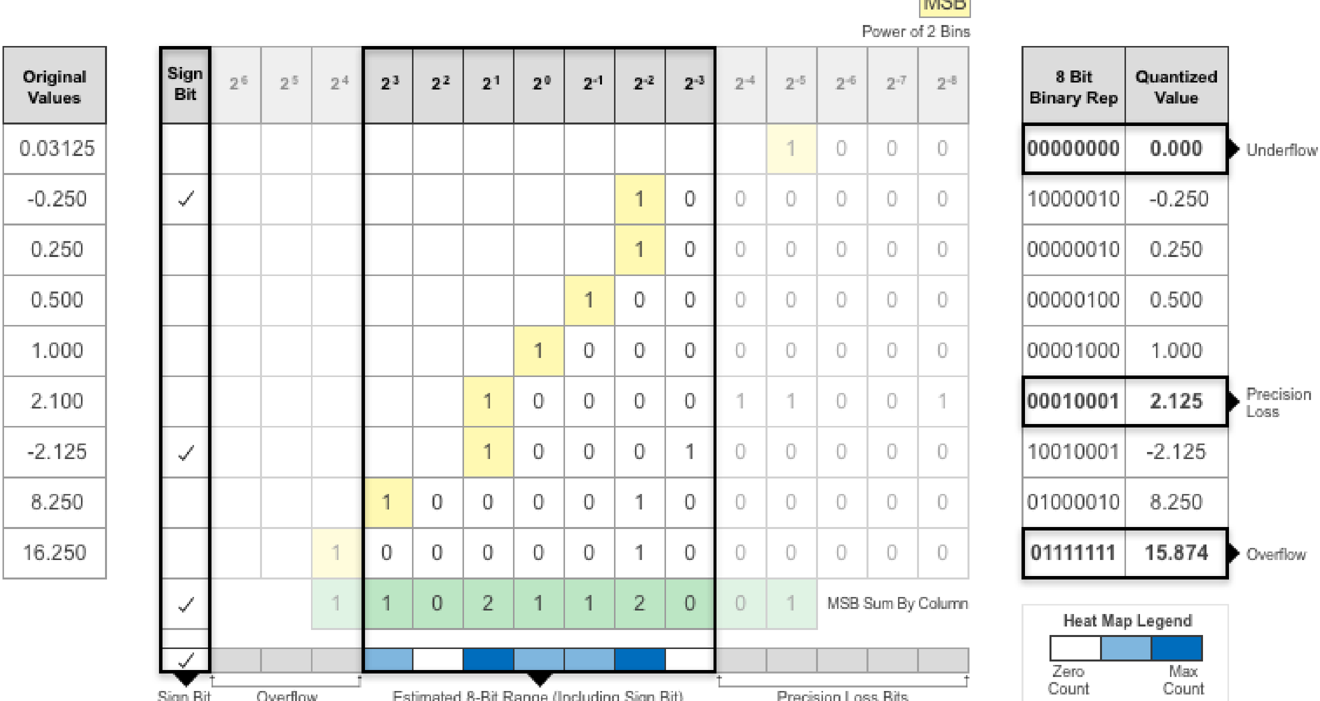
Sign Bit

Estimated 8-Bit Range (Including Sign Bit)

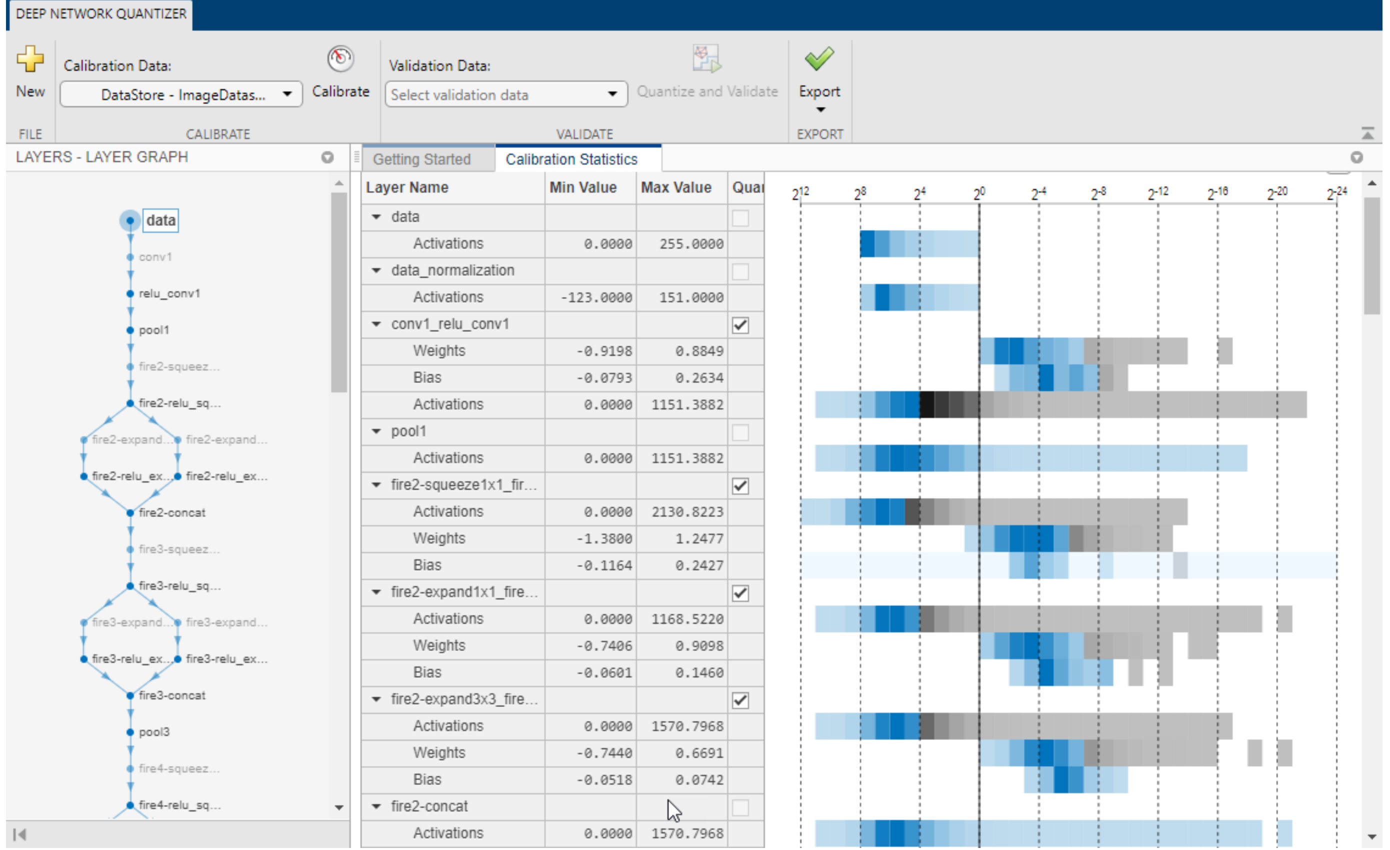
Heat Map Legend

Zero Count — Max Count

6 After assigning the data type, any bits outside of that data type are removed. In this sample, the first value, 0.03125, suffers from an underflow, so the quantized value is 0. The ideal 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. The quantized value saturates to 15.874.



7 The 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` | `dlquantizationOptions` | `dlquantizer` | `validate`

Quantization Workflow Prerequisites

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

	Execution Environment	
Development Host Requirements	**FPGA**	**GPU**
Setup Toolkit Environment	`hdlsetuptoolpath` (HDL Coder)	"Setting Up the Prerequisite Products" (GPU Coder)
Required Products	• Deep Learning Toolbox • Deep Learning HDL 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
Supported Networks and Layers	"Supported Networks, Layers and Boards" on page 7-2	"Supported Networks and Layers" (GPU Coder)
Deployment	Deep Learning HDL Toolbox	GPU Coder
Additional 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.

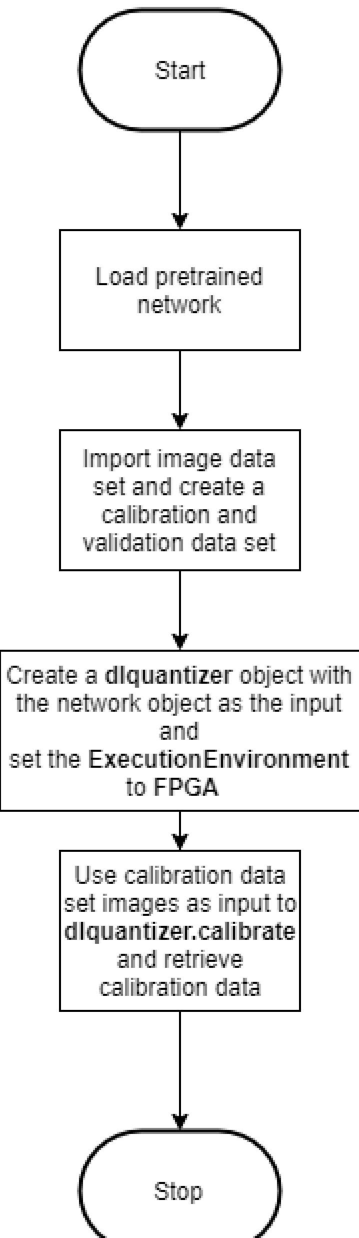
Calibration

Workflow

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

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

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



See Also

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

More About

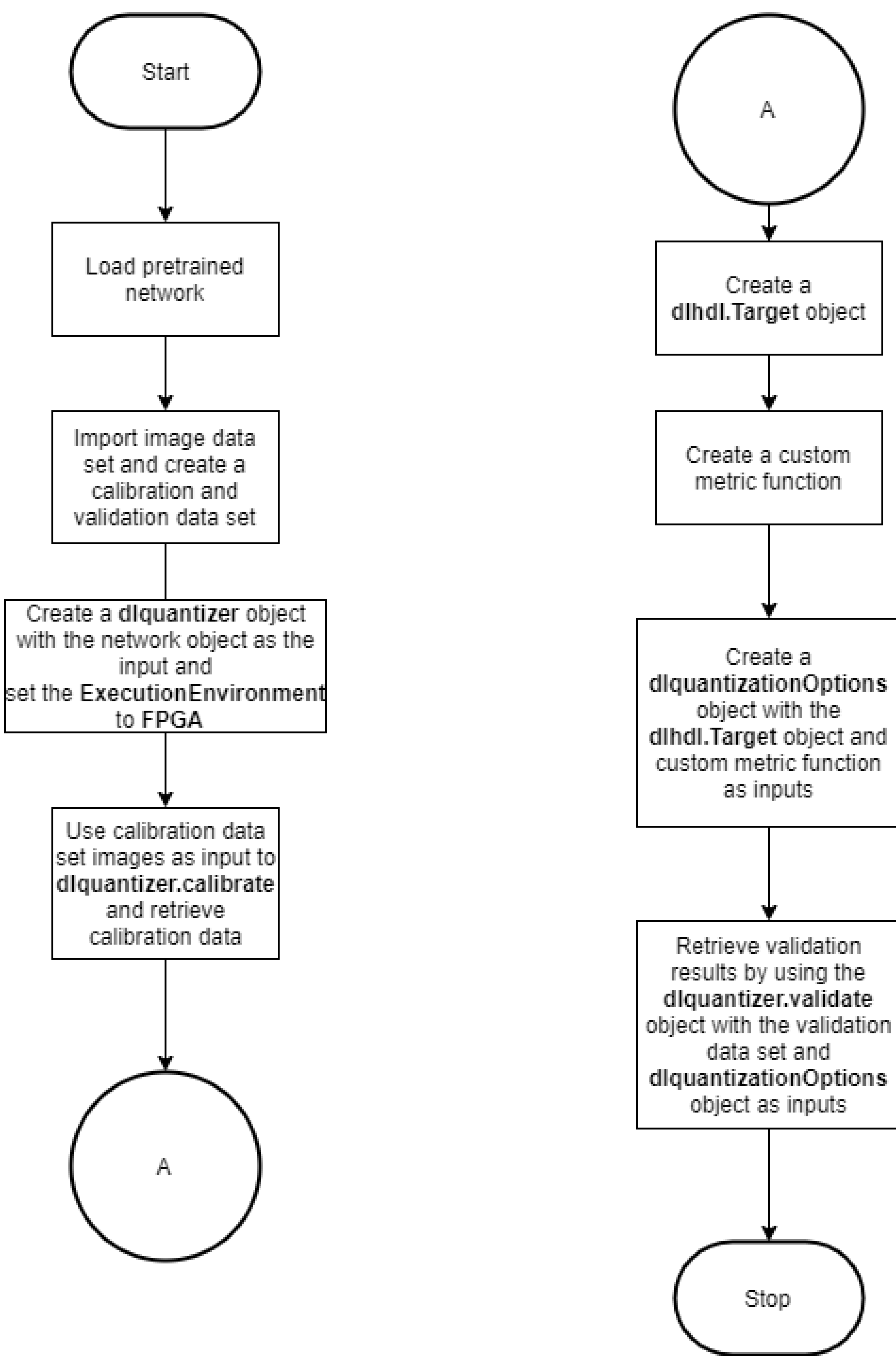
- “Quantization of Deep Neural Networks” on page 11-2
- “Validation” on page 11-12
- “Code Generation and Deployment” on page 11-15

Validation

Workflow

Before deploying the quantized network to your target FPGA or SoC board, to verify the accuracy of your quantized network, use the validation workflow.

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



See Also

`dlquantizationOptions` | `dlquantizer` | `validate`

More About

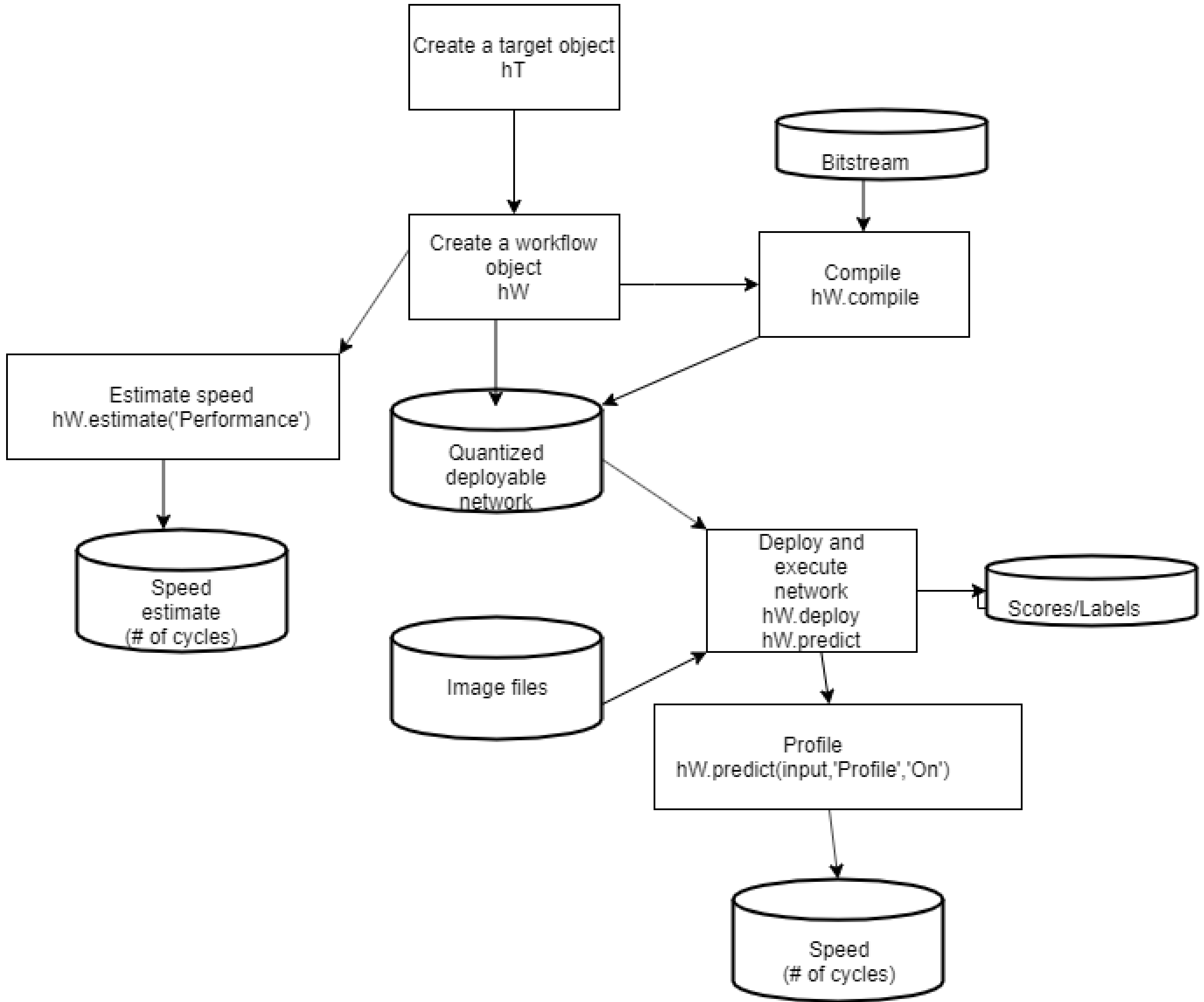
- “Quantization of Deep Neural Networks” on page 11-2
- “Calibration” on page 11-10
- “Code Generation and Deployment” on page 11-15

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 11-10
- "Validation" on page 11-12

Deploy Quantized Neural Network

This example shows how to train, compile, and deploy a modified quantized AlexNet pretrained series network 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.

Prerequisites

To run this example, you need the products listed under FPGA in “Quantization Workflow Prerequisites” on page 11-9.

Create Modified Series Network by Using Transfer Learning

Create a modified series network by using transfer learning. For more information, see “Create Series Network for Quantization” on page 10-72.

Create Quantized Network Object

Create a dlquantizer object and specify the network to quantize and ExecutionEnvironment . The netTransfer network is the output of the modified network created by transfer learning. To create the netTransfer series network, see “Create Series Network for Quantization” on page 10-72.

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

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.

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.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir);
unzip('logos_dataset.zip');
unzip('logos_dataset.zip');

imds = imageDatastore('logos_dataset', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

Calibrate Quantized Network

Use the calibrate function to run the network with sample inputs and collect 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 function returns a table. Each row of the table contains

range information for a learnable parameter of the optimized network. For best quantization results, the calibration data must be a representative of actual inputs that would be predicted by the network.

```
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');

dlQuantObj.calibrate(imageData);
```

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 create the target object, enter:

```
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet','IPAddress','192.168.1.101');
```

Create Workflow Object

Create an object of the `dlhdl.Workflow` class. When you create the object, specify the network and the bitstream name. 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. The bitstream uses an int8 data type.

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

Compile Quantized Series Network

Compile the quantized series network.

```
dn = hW.compile
```

```
    offset_name                 offset_address    allocated_space
    "InputDataOffset"           "0x00000000"     "48.0 MB"
    "OutputResultOffset"        "0x03000000"     "4.0 MB"
    "SystemBufferOffset"        "0x03400000"     "28.0 MB"
    "InstructionDataOffset"     "0x05000000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x05400000"     "4.0 MB"
    "FCWeightDataOffset"        "0x05800000"     "56.0 MB"
    "EndOffset"                 "0x09000000"     "Total: 144.0 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

Run the deploy function of the `dlhdl.Workflow` object to deploy the network on the Xilinx ZCU102 SoC hardware. 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
```

Load the Example Images and Run the Prediction

Load the example images and retrieve the prediction results.

```
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 ResultsLastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Late
                          -------------        -------------            ---------        ---------        ---------
Network                      7615557              0.05077                   1            7616123             19.7
    conv_module              3123657              0.02082
        conv1                 733903              0.00489
        norm1                 485953              0.00324
        pool1                 108979              0.00073
        conv2                 631639              0.00421
        norm2                 289646              0.00193
        pool2                 115286              0.00077
        conv3                 307112              0.00205
        conv4                 249627              0.00166
        conv5                 176223              0.00117
        pool5                  25404              0.00017
    fc_module                4491900              0.02995
        fc6                  3083885              0.02056
        fc7                  1370258              0.00914
        fc                     37755              0.00025
 * The clock frequency of the DL processor is: 150MHz
ans = 'carlsberg'

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


              Deep Learning Processor Profiler Performance ResultsLastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum
                          -------------        -------------            ---------        ---------        ---------
Network                      7615364              0.05077                   1            7615905             19.7
    conv_module              3123385              0.02082
        conv1                 733946              0.00489
        norm1                 485695              0.00324
        pool1                 108971              0.00073
        conv2                 631616              0.00421
        norm2                 289612              0.00193
        pool2                 115363              0.00077
        conv3                 307034              0.00205
        conv4                 249683              0.00166
        conv5                 176216              0.00117
        pool5                  25364              0.00017
    fc_module                4491979              0.02995
        fc6                  3083961              0.02056
        fc7                  1370258              0.00914
        fc                     37758              0.00025
 * The clock frequency of the DL processor is: 150MHz
ans = 'pepsi'

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


              Deep Learning Processor Profiler Performance ResultsLastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum
                          -------------        -------------            ---------        ---------        ---------
Network                      7615042              0.05077                   1            7615582             19.7
    conv_module              3123107              0.02082
        conv1                 733949              0.00489
        norm1                 485783              0.00324
```

```
        pool1              108565                 0.00072
        conv2              631567                 0.00421
        norm2              289568                 0.00193
        pool2              115037                 0.00077
        conv3              307355                 0.00205
        conv4              249793                 0.00167
        conv5              176217                 0.00117
        pool5               25388                 0.00017
    fc_module             4491935                 0.02995
        fc6               3083920                 0.02056
        fc7               1370258                 0.00914
        fc                  37755                 0.00025
 * The clock frequency of the DL processor is: 150MHz
ans = 'tsingtao'

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

              Deep Learning Processor Profiler Performance ResultsLastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum
                           -------------          -------------            ---------      ---------       ---------
Network                    7615303                0.05077                      1          7615843            19.7
    conv_module            3123324                0.02082
        conv1              733883                 0.00489
        norm1              485688                 0.00324
        pool1              108995                 0.00073
        conv2              631598                 0.00421
        norm2              289636                 0.00193
        pool2              115351                 0.00077
        conv3              307108                 0.00205
        conv4              249623                 0.00166
        conv5              176193                 0.00117
        pool5               25364                 0.00017
    fc_module             4491979                 0.02995
        fc6               3083961                 0.02056
        fc7               1370258                 0.00914
        fc                  37758                 0.00025
 * The clock frequency of the DL processor is: 150MHz
ans = 'singha'
```

carlsberg

pepsi

tsingtao

singha

See Also

Functions
`calibrate` | `validate` | `compile` | `deploy` | `predict`

Objects
`dlhdl.Target` | `dlhdl.Workflow` | `dlquantizationOptions` | `dlquantizer`

More About

- “Quantization of Deep Neural Networks” on page 11-2
- “Transfer Learning”

Quantize Neural Network for FPGA Execution Environment

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and explore the behavior of the quantized network. In this example, you quantize the LogoNet neural 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 from the target device.

Prerequisites

To run this example, you need the products listed under FPGA in “Quantization Workflow Prerequisites” on page 11-9.

Load Pretrained Series Network

Create a file in your current working directory called getLogoNetwork.m. Enter these lines into the file:

```
function net = getLogoNetwork()
    data = getLogoData();
    net  = data.convnet;
end

function data = getLogoData()
    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');
end
```

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

```
snet = 

  SeriesNetwork with properties:

         Layers: [22×1 nnet.cnn.layer.Layer]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

Define Calibration and Validation Data Sets

The calibration data is used to collect 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. For the best quantization results, the calibration data must be representative of inputs to the network.

The validation data is used to test the network after quantization to understand the effects of the limited range and precision of the quantized convolution layers in the network.

In this example, use the images in the logos_dataset data set. Define an augmentedImageDatastore object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir);
unzip('logos_dataset.zip');
```

```
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
```

Create Quantized Network Object

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

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

Calibrate Quantized Network

Use the `calibrate` function to exercise the network with sample inputs and collect 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 function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```
 dlQuantObj.calibrate(calibrationData)
```

```
ans =
        Optimized Layer Name          Network Layer Name    Learnables / Activations     MinValue       MaxValue
    ____________________________      __________________    ________________________    ___________    __________

    {'conv_1_Weights'          }       {'conv_1'    }            "Weights"                -0.048978      0.039352
    {'conv_1_Bias'             }       {'conv_1'    }            "Bias"                     0.99996        1.0028
    {'conv_2_Weights'          }       {'conv_2'    }            "Weights"                -0.055518      0.061901
    {'conv_2_Bias'             }       {'conv_2'    }            "Bias"                 -0.00061171       0.00227
    {'conv_3_Weights'          }       {'conv_3'    }            "Weights"                -0.045942      0.046927
    {'conv_3_Bias'             }       {'conv_3'    }            "Bias"                  -0.0013998     0.0015218
    {'conv_4_Weights'          }       {'conv_4'    }            "Weights"                -0.045967         0.051
    {'conv_4_Bias'             }       {'conv_4'    }            "Bias"                    -0.00164     0.0037892
    {'fc_1_Weights'            }       {'fc_1'      }            "Weights"                -0.051394      0.054344
    {'fc_1_Bias'               }       {'fc_1'      }            "Bias"                 -0.00052319    0.00084454
    {'fc_2_Weights'            }       {'fc_2'      }            "Weights"                 -0.05016      0.051557
    {'fc_2_Bias'               }       {'fc_2'      }            "Bias"                  -0.0017564     0.0018502
    {'fc_3_Weights'            }       {'fc_3'      }            "Weights"                -0.050706       0.04678
    {'fc_3_Bias'               }       {'fc_3'      }            "Bias"                    -0.02951      0.024855
    {'imageinput'              }       {'imageinput'}            "Activations"                    0           255
    {'imageinput_normalization'}       {'imageinput'}            "Activations"              -139.34        198.72
```

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 create the target object, enter:

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

Define Metric Function

Define a metric function to use to compare the behavior of the network before and after quantization. Save this function in a local file.

```
function accuracy = hComputeAccuracy(predictionScores, net, dataStore)
%% hComputeAccuracy test helper function computes model level accuracy statistics

% Copyright 2020 The MathWorks, Inc.

    % Load ground truth
    groundTruth = dataStore.Labels;

    % Compare with predicted label with actual ground truth
    predictionError = {};
```

```
    for idx=1:numel(groundTruth)
        [~, idy] = max(predictionScores(idx, :));
        yActual = net.Layers(end).Classes(idy);
        predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
    end

    % Sum all prediction errors.
    predictionError = [predictionError{:}];
    accuracy = sum(predictionError)/numel(predictionError);
end
```

Create dlQuantizationOptions Object

Specify the metric function in a `dlquantizationOptions` object.

```
options = dlquantizationOptions('MetricFcn', ...
    {@(x)hComputeModelAccuracy(x, snet, validationData)},'Bitstream','arria10soc_int8',...
'Target',hTarget);
```

Validate Quantized Neural Network

To compile and deploy the quantized network, run the `validate` function of the `dlquantizer` object. Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. 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 checks for the Intel Quartus tool and the supported tool version. It then starts programming the FPGA device by using the sof file, displays progress messages, and the time it takes to deploy the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
prediction = dlQuantObj.validate(validationData,options);
```

```
          offset_name          offset_address    allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "48.0 MB"
    "OutputResultOffset"        "0x03000000"     "4.0 MB"
    "SystemBufferOffset"        "0x03400000"     "60.0 MB"
    "InstructionDataOffset"     "0x07000000"     "8.0 MB"
    "ConvWeightDataOffset"      "0x07800000"     "8.0 MB"
    "FCWeightDataOffset"        "0x08000000"     "12.0 MB"
    "EndOffset"                 "0x08c00000"     "Total: 140.0 MB"

### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.
### Loading weights to Conv Processor.
### Conv Weights loaded. Current time is 16-Jul-2020 12:45:10
### Loading weights to FC Processor.
### FC Weights loaded. Current time is 16-Jul-2020 12:45:26
### Finished writing input activations.
### Running single input activations.


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                   13570959                  0.09047                      30          380609145            11.8
    conv_module           12667786                  0.08445
        conv_1             3938907                  0.02626
        maxpool_1          1544560                  0.01030
        conv_2             2910954                  0.01941
        maxpool_2           577524                  0.00385
        conv_3             2552707                  0.01702
        maxpool_3           676542                  0.00451
        conv_4              455434                  0.00304
        maxpool_4            11251                  0.00008
    fc_module               903173                  0.00602
        fc_1                536164                  0.00357
        fc_2                342643                  0.00228
```

```
        fc_3                 24364                  0.00016
 * The clock frequency of the DL processor is: 150MHz


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


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                   13570364                  0.09047                      30          380612682            11.8
    conv_module           12667103                  0.08445
        conv_1             3939296                  0.02626
        maxpool_1          1544371                  0.01030
        conv_2             2910747                  0.01940
        maxpool_2           577654                  0.00385
        conv_3             2551829                  0.01701
        maxpool_3           676548                  0.00451
        conv_4              455396                  0.00304
        maxpool_4            11355                  0.00008
    fc_module               903261                  0.00602
        fc_1                536206                  0.00357
        fc_2                342688                  0.00228
        fc_3                 24365                  0.00016
 * The clock frequency of the DL processor is: 150MHz


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


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                   13571561                  0.09048                      30          380608338            11.8
    conv_module           12668340                  0.08446
        conv_1             3939070                  0.02626
        maxpool_1          1545327                  0.01030
        conv_2             2911061                  0.01941
        maxpool_2           577557                  0.00385
        conv_3             2552082                  0.01701
        maxpool_3           676506                  0.00451
        conv_4              455582                  0.00304
        maxpool_4            11248                  0.00007
    fc_module               903221                  0.00602
        fc_1                536167                  0.00357
        fc_2                342643                  0.00228
        fc_3                 24409                  0.00016
 * The clock frequency of the DL processor is: 150MHz


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


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                   13569862                  0.09047                      30          380613327            11.8
    conv_module           12666756                  0.08445
        conv_1             3939212                  0.02626
        maxpool_1          1543267                  0.01029
        conv_2             2911184                  0.01941
        maxpool_2           577275                  0.00385
        conv_3             2552868                  0.01702
        maxpool_3           676438                  0.00451
        conv_4              455353                  0.00304
        maxpool_4            11252                  0.00008
    fc_module               903106                  0.00602
        fc_1                536050                  0.00357
        fc_2                342645                  0.00228
        fc_3                 24409                  0.00016
 * The clock frequency of the DL processor is: 150MHz


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

```
              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                   13570823                  0.09047                      30          380619836            11.8
    conv_module           12667607                  0.08445
        conv_1             3939074                  0.02626
        maxpool_1          1544519                  0.01030
        conv_2             2910636                  0.01940
        maxpool_2           577769                  0.00385
        conv_3             2551800                  0.01701
        maxpool_3           676795                  0.00451
        conv_4              455859                  0.00304
        maxpool_4            11248                  0.00007
    fc_module               903216                  0.00602
        fc_1                536165                  0.00357
        fc_2                342643                  0.00228
        fc_3                 24406                  0.00016
 * The clock frequency of the DL processor is: 150MHz

          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "48.0 MB"        
    "OutputResultOffset"        "0x03000000"     "4.0 MB"         
    "SystemBufferOffset"        "0x03400000"     "60.0 MB"        
    "InstructionDataOffset"     "0x07000000"     "8.0 MB"         
    "ConvWeightDataOffset"      "0x07800000"     "8.0 MB"         
    "FCWeightDataOffset"        "0x08000000"     "12.0 MB"        
    "EndOffset"                 "0x08c00000"     "Total: 140.0 MB"

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


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                   13572329                  0.09048                      10          127265075            11.8
    conv_module           12669135                  0.08446
        conv_1             3939559                  0.02626
        maxpool_1          1545378                  0.01030
        conv_2             2911243                  0.01941
        maxpool_2           577422                  0.00385
        conv_3             2552064                  0.01701
        maxpool_3           676678                  0.00451
        conv_4              455657                  0.00304
        maxpool_4            11227                  0.00007
    fc_module               903194                  0.00602
        fc_1                536140                  0.00357
        fc_2                342688                  0.00228
        fc_3                 24364                  0.00016
 * The clock frequency of the DL processor is: 150MHz


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


              Deep Learning Processor Profiler Performance Results

                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                   13572527                  0.09048                      10          127266427            11.8
    conv_module           12669266                  0.08446
        conv_1             3939776                  0.02627
        maxpool_1          1545632                  0.01030
        conv_2             2911169                  0.01941
        maxpool_2           577592                  0.00385
        conv_3             2551613                  0.01701
        maxpool_3           676811                  0.00451
        conv_4              455418                  0.00304
        maxpool_4            11348                  0.00008
    fc_module               903261                  0.00602
        fc_1                536205                  0.00357
```

```
        fc_2              342689                  0.00228
        fc_3               24365                  0.00016
 * The clock frequency of the DL processor is: 150MHz
```

View Performance of Quantized Neural Network

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

```
prediction.MetricResults.Result
```

```
ans =
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}         0.9875
     {'Quantized'     }         0.9875
```

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

```
prediction.QuantizedNetworkFPS
```

```
ans = 11.8126
```

See Also

Functions
`calibrate` | `validate` | `compile` | `deploy` | `predict`

Objects
`dlhdl.Target` | `dlhdl.Workflow` | `dlquantizationOptions` | `dlquantizer`

More About

- “Quantization of Deep Neural Networks” on page 11-2
- “Deploy Quantized Neural Network” on page 11-17

12

Deep Learning Processor IP Core User Guide

- "Deep Learning Processor IP Core" on page 12-2
- "Compiler Output" on page 12-3
- "External Memory Data Format" on page 12-4
- "Deep Learning Processor Register Map" on page 12-7

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 12-7

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 “Compiler Output” 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 12-4.

See Also

More About

- “Custom IP Core Generation” (HDL Coder)
- “Compiler Output” on page 12-3
- “External Memory Data Format” on page 12-4
- “Deep Learning Processor Register Map” on page 12-7

Compiler Output

To manually load the input data, deep learning processor IP core convolution and fully connected module instructions, pretrained series network layer instructions, weights and biases, and retrieve the output results use the compiler generated external memory address map. Or, use the `dlhdl.Workflow` workflow. The workflow generates the external memory address map, loads the inputs, module instructions, layers instructions, weights and biases, and retrieves the output results.

External Memory Address Map

When you create a `dlhdl.Workflow` object and use the `compile` method, an external memory address map is generated.

The `compile` method generates these address offsets based on the deep learning network and target board:

- `InputDataOffset`—Address offset where the input images are loaded.
- `OutputResultOffset`— Output results are written starting at this address offset.
- `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.

The example displays the external memory map generated for the logo recognition network that uses the `arria10soc_single` bitstream. “Compile the dlhdl.Workflow object”.

See Also

More About

- “Deep Learning Processor IP Core” on page 12-2
- “External Memory Data Format” on page 12-4
- “Deep Learning Processor Register Map” on page 12-7

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

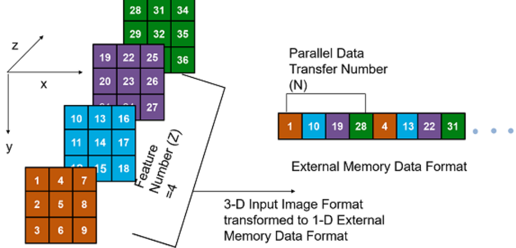
- `Parallel Data Transfer Number` refers to the number of pixels that are transferred every clock cycle through the AXI master interface. Use the letter `N` in place of the `Parallel Data Transfer Number`. Mathematically `N` is the square root of the `ConvThreadNumber`. See “ConvThreadNumber”.
- `Feature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

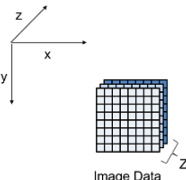
The image demonstrates how the data stored in a 3-by-3-by-4 matrix is translated into a 1-by-36 matrix that is then stored in the external memory.



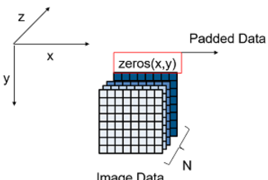
When the image `Feature Number` (Z) is not a multiple of the `Parallel Data Transfer Number` (N), then we must pad a zeroes matrix of size x-by-y along the z dimension of the matrix to make the image `Z` value a multiple of `N`.

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.

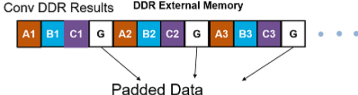
This image is the input image format before padding.



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`.

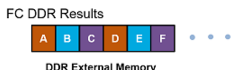


If your deep learning processor consists of only a convolution (conv) processing module, the output external data is using the conv module external data format, which means it possibly contains padded data if your output `Z` value is not a multiple of the `N` value. The padded data is removed when you 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
- “Compiler Output” on page 12-3
- “Deep Learning Processor Register Map” on page 12-7

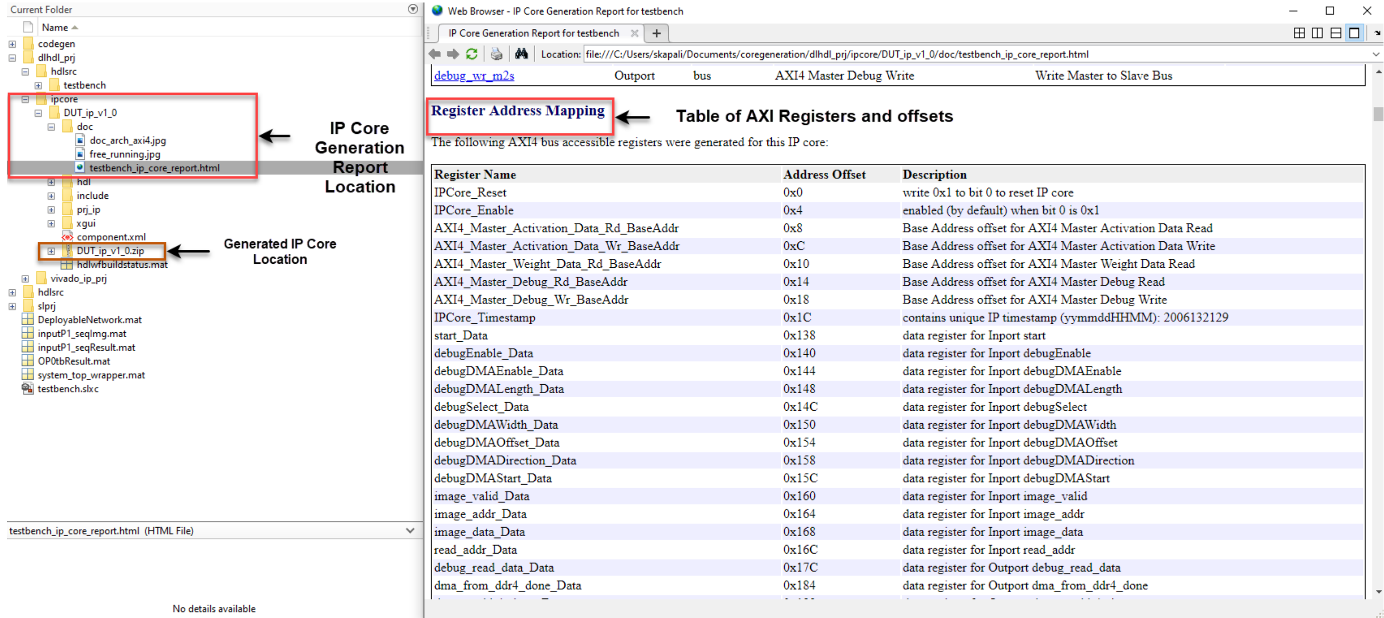
See Also

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.



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
- “Compiler Output” on page 12-3
- “External Memory Data Format” on page 12-4