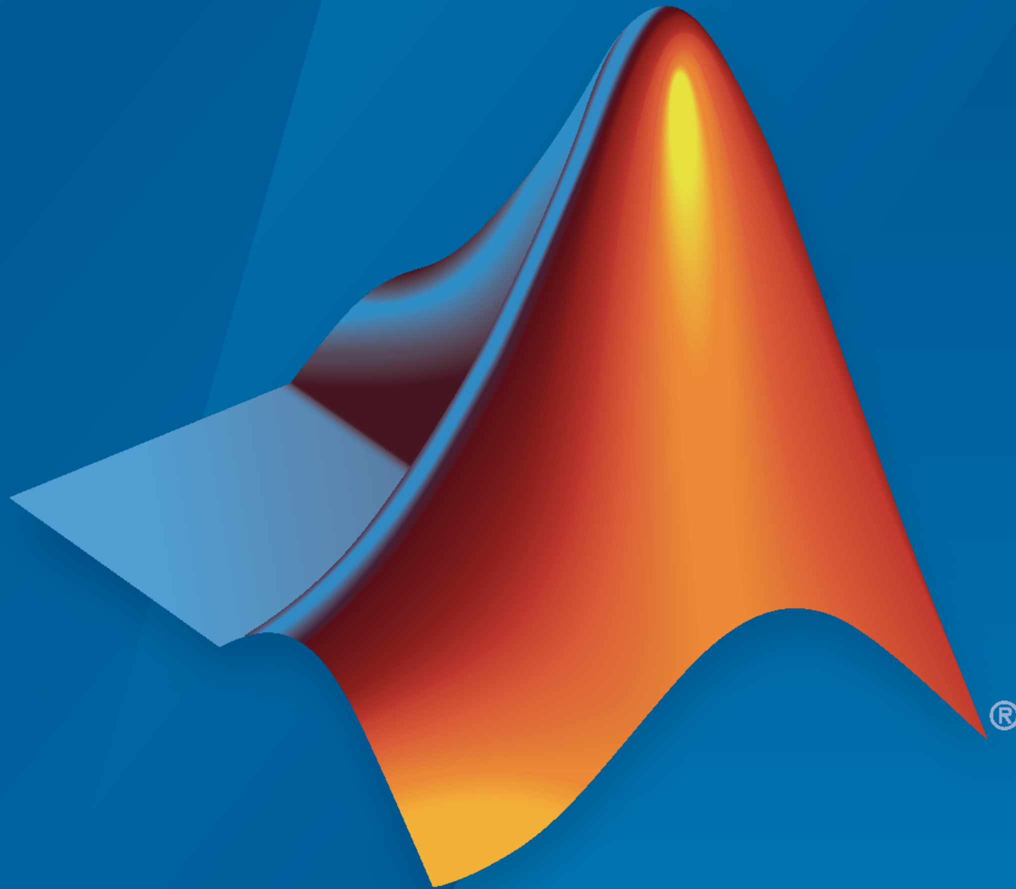
# Deep Learning HDL Toolbox™

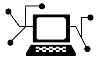
Reference



MATLAB®

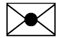
R2022a

# How to Contact MathWorks




| | |
|---|---|
| Latest news: | www.mathworks.com |
| Sales and services: | www.mathworks.com/sales_and_services |
| User community: | www.mathworks.com/matlabcentral |
| Technical support: | www.mathworks.com/support/contact_us |

Phone: 508-647-7000



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


*Deep Learning HDL Toolbox™ Reference*


© COPYRIGHT 2020—2022 by The MathWorks, Inc.

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

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

**Trademarks**

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

**Patents**

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



**Revision History**

| | | |
|---|---|---|
| September 2020 | Online only | New for Version 1.0 (R2020b) |
| March 2021 | Online only | Revised for Version 1.1 (R2021a) |
| September 2021 | Online only | Revised for Version 1.2 (R2021b) |
| March 2022 | Online only | Revised for Version 1.3 (Release R2022a) |

# Contents

1 Functions

# Functions

# dlhdl.Workflow class

**Package:** `dlhdl`

Configure deployment workflow for deep learning neural network

## Description

Use the `dlhdl.Workflow` object to set options for compiling and deploying your deep learning network to a target FPGA. You create an object of the `dlhdl.Workflow` class for the specified deep learning network and FPGA bitstream. Use the object to:

- Compile the deep learning network.
- Estimate the speed and throughput of your network on the specified FPGA device.
- Compile and deploy the neural network onto the FPGA.
- Predict the class of input images.
- Profile the results for the specified network and the FPGA.

# Creation

`hW = dlhdl.Workflow('Network',Network,'Bitstream',Bitstream)` creates a workflow configuration object with a network object and bitstream to deploy your custom pretrained deep learning network object.

`hW = dlhdl.Workflow('Ntwork',Network,'Bitstream',Bitstream,Name,Value)` creates a workflow configuration object with a network object and bitstream to deploy your custom pretrained deep learning network object, with additional options specified by one or more name-value pair arguments.

### Input Arguments

#### `Bitstream` — Name of the FPGA bitstream

'' (default) | character vector

Name of the FPGA bitstream, specified as a character vector. Make sure that the bitstream name matches the data type and the FPGA board that you are targeting. For a list of provided bitstream names, see “Use Deep Learning on FPGA Bitstreams”.

**Example:** `'Bitstream'`, `'arria10soc_single'` specifies that you want to deploy the trained network with `single` data types to an Arria10 SoC board.

#### `Network` — Network object

`SeriesNetwork` object | `DAGNetwork` object | `dlquantizer` object

Name of the deep learning network object.

**Example:** `'network'`, `net` creates a workflow object for the saved pretrained network `net`. To specify `net`, you can import any of the existing supported pretrained networks or use transfer learning to adapt the network to your issue. See “Supported Pretrained Networks”.

```
net = resnet18;
hW = dlhdl.Workflow('Network',net,'Bitstream','zcu102_single');
```

**Example:** `'network'`, `dlquantizeObj` creates a workflow object for the quantized network object `dlquantizeObj`. To specify `dlquantizeObj`, you can import any of the supported existing pretrained networks and create an object by using the `dlquantizer` class. For information on supported networks, see “Supported Pretrained Networks”.

```
net = resnet18;
dlquantObj = dlquantizer(net,'ExecutionEnvironment','FPGA');
dlquantObj.calibrate(imdsTrain);
hW = dlhdl.Workflow('Network',dlquantObj,'Bitstream','zcu102_int8');
```

## Properties

### `'Target'` — `dlhdl.Target` object to deploy network and bitstream to the target device

`hTarget`

Target object specified as `dlhdl.Target` object

Example: `'Target',hTarget`

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

## Methods

### Public Methods

| | |
|---|---|
| activations | Retrieve intermediate layer results for deployed deep learning network |
| compile | Compile workflow object |
| deploy | Deploy the specified neural network to the target FPGA board |
| getBuildInfo | Retrieve bitstream resource utilization |
| predict | Run inference on deployed network and profile speed of neural network deployed on specified target device |

## Examples

### Create Workflow Object by using Property Name Value Pairs

```
snet = vgg19;
hW = dlhdl.Workflow('Network',snet,'Bitstream','arria10soc_single', 'Target', hTarget);
```

### Create Workflow Object with Quantized Network Object

```
snet = getLogoNetwork;
dlquantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
Image = imageDatastore('heineken.png','Labels','Heineken');
dlquantObj.calibrate(Image);
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');
hW = dlhdl.Workflow('Network',dlquantObj,'Bitstream','zcu102_int8','Target',hTarget);
```

## See Also

### Objects

`dlhdl.Target` | `dlquantizer` | `dlquantizationOptions`

**Topics**
"Prototype Deep Learning Networks on FPGA and SoCs Workflow"
"Quantization of Deep Neural Networks"

**Introduced in R2020b**

# activations

**Class:** `dlhdl.Workflow`
**Package:** `dlhdl`

Retrieve intermediate layer results for deployed deep learning network

## Syntax

```
act = activations(workflowObject,image,layername)
act = activations(workflowObject,image,layername, Name,Value)
```

## Description

`act = activations(workflowObject,image,layername)` returns intermediate layer activation data results for the image data in `image`, and the name of the layer specified in `layername`. The result size depends on the output size of the layer. The layer output size can be retrieved by using `analyzeNetwork`.

`act = activations(workflowObject,image,layername, Name,Value)`returns intermediate layer activation data results for the image data in `image`, and the name of the layer specified in `layername`, with additional options specified by one or more `Name,Value` pair arguments. The result size depends on the output size of the layer. The layer output size can be retrieved by using `analyzeNetwork`.

## Input Arguments

### `workflowObject` — Instance of workflow object

`dlhdl.Workflow` object

Instance of workflow object, specified as an `dlhld.Workflow` object.

### `image` — Input image

*m*-by-*n*-by-*k* numeric array

Input image, specified as a *m*-by-*n*-by-*k* numeric array. *m*, *n*, and *k* must match the dimensions of the deep learning network input image layer. For example, for the LogoNet network, resize the input images to a 227-by-227-by-3 array.

Data Types: `single`

### `layername` — Layer to extract activations from

'' (default) | character vector

Layer to extract activations from, specified as a character vector.

To compute the activations of a `SeriesNetwork` object, specify the layer as a character vector corresponding to the layer name.

To compute the activations of a `DAGNetwork` object, specify the layer as a character vector corresponding to the layer name. If the layer has multiple outputs, specify the layer and output as the

layer name, followed by the character "/", followed by the name of the layer output. That is, `layername` of the form `'layerName/outputName'`.

These limitations apply when extracting activations from layers:

- A `maxPooling2DLayer` which has the `HasUnpoolingIndices` argument set to `true` has three outputs `out`, `indices`, and `size`. Activations are supported only for the output `out`.

Example: `'conv1'`

Example: `'mpool/out'`

#### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

#### `Profiler` — Flag that returns profiling results

'off' (default) | 'on'

Flag to return profiling results for the deep learning network deployed to the target board.

Example: `'Profiler','on'`

## Output Arguments

#### `act` — Intermediate layer activation data

`array of single`

Intermediate layer activation data, returned as an array of singles. The array size depends on the layer output size. For example, for the ResNet-18 network `pool1` layer, the size of the returned result array is 56-by-56-by-64.

## Examples

### 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"
```

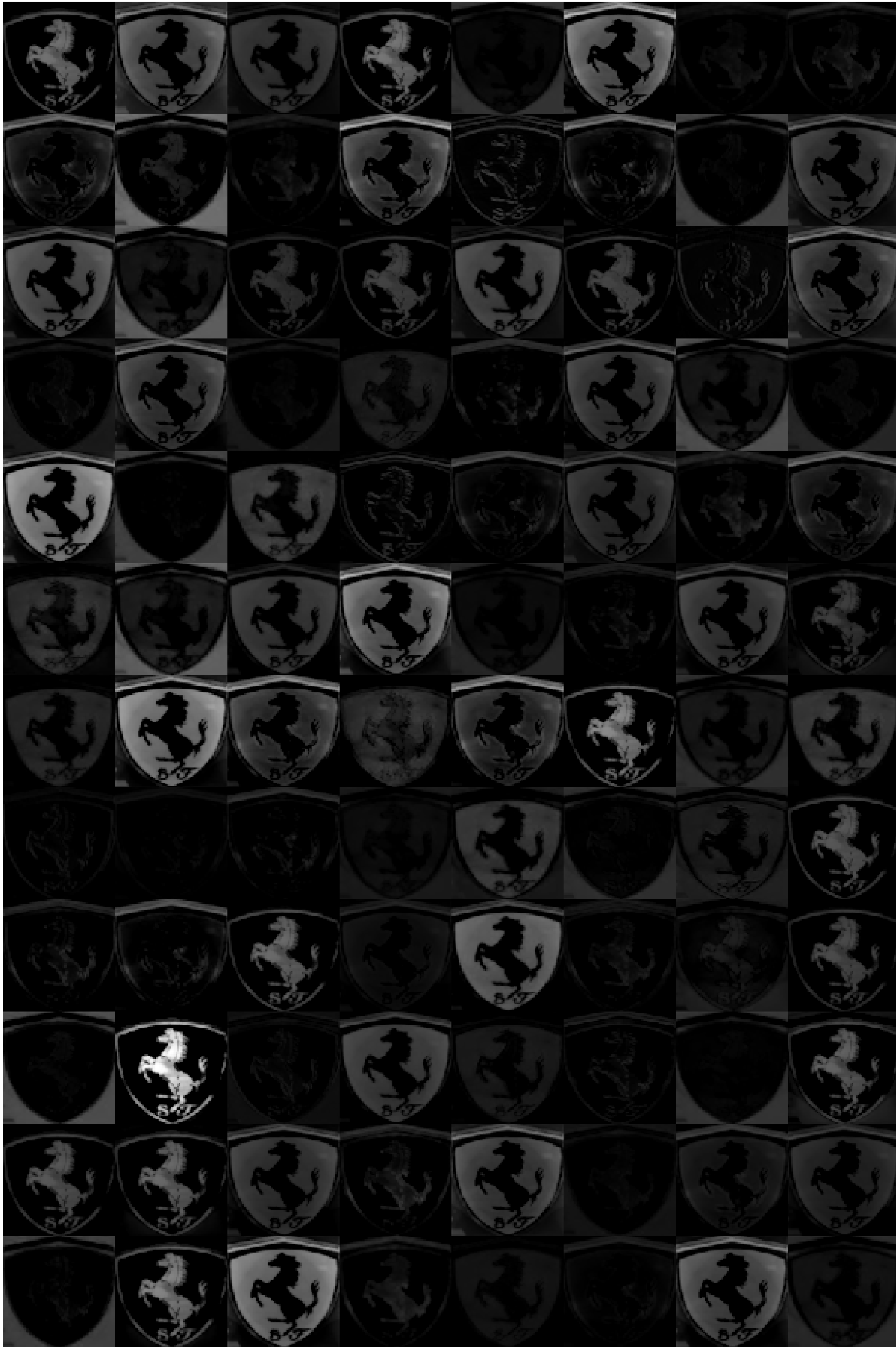
```
### Programming FPGA Bitstream using JTAG...
### Programming the FPGA bitstream has been completed successfully.

### 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

`compile` | `deploy` | `getBuildInfo` | `predict`

**Introduced in R2020b**

# compile

**Class:** `dlhdl.Workflow`
**Package:** `dlhdl`

Compile workflow object

## Syntax

```
compile(workflowObject)
compile(workflowObject,Name,Value)
```

## Description

`compile(workflowObject)` compiles the `dlhdl.Workflow` object and generates the parameters for deploying the network on the target device.

`compile(workflowObject,Name,Value)` compiles the `dlhdl.Workflow` object and generates the parameters for deploying the network on the target device, with additional options specified by one or more `Name,Value` pair arguments.

The function returns two matrices. One matrix describes the layers of the network. The `Conv Controller (Scheduling)` and the `FC Controller (Scheduling)` modules in the deep learning processor IP use this matrix to schedule the convolution and fully connected layer operations. The second matrix contains the weights, biases, and inputs of the neural network. This information is loaded onto the DDR memory and used by the `Generic Convolution Processor` and the `Generic FC Processor` in the deep learning processor.

## Input Arguments

**`workflowObject` — Instance of workflow object**
`dlhdl.Workflow` object

Instance of workflow object, specified as an `dlhld.Workflow` object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

**`InputFrameNumberLimit` — Maximum input frame number limit**
integer

Parameter to specify maximum input frame number limit to calculate DDR memory access allocation.

Example: `'InputFrameNumberLimit',30`

**`HardwareNormalization` — Flag to enable hardware implementation of image input layer normalization function**
'auto' (default) | 'on | 'off'

Flag to enable hardware implementation of image input layer normalization function , specified as a string or character vector.

Example: `HardwareNormalization = "auto"`

## Examples

### Compile the `dlhdl.Workflow` object

Compile the `dlhdl.Workflow` object, for deployment to the Intel® Arria® 10 SoC development kit that has `single` data types.

Create a `dlhdl.Workflow` object and then use the `compile` function to deploy the pretrained network to the target hardware.

```
snet = vgg19;
hT = dlhdl.Target('Intel');
hW = dlhdl.Workflow('network', snet, 'Bitstream', 'arria10soc_single','Target',hT);
hW.compile
```

Once the code is executed the result is:

```
  hW.compile
          offset_name          offset_address     allocated_space 
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"        
    "OutputResultOffset"        "0x01800000"     "4.0 MB"         
    "SystemBufferOffset"        "0x01c00000"     "52.0 MB"        
    "InstructionDataOffset"     "0x05000000"     "20.0 MB"        
    "ConvWeightDataOffset"      "0x06400000"     "276.0 MB"       
    "FCWeightDataOffset"        "0x17800000"     "472.0 MB"       
    "EndOffset"                 "0x35000000"     "Total: 848.0 MB"


ans = 

  struct with fields:

       Operators: [1×1 struct]
    LayerConfigs: [1×1 struct]
      NetConfigs: [1×1 struct]
```

### Generate DDR Memory Offsets Based On Number of Input Frames

1. Create a `dlhdl.Workflow` object and then use the `compile` function with optional argument of `InputFrameNumberLimit` to deploy the pretrained network to the target hardware.

   ```
   net = resnet18;
   hT = dlhdl.Target('Xilinx');
   hW = dlhdl.Workflow('Network', net, 'Bitstream', 'zcu102_single','Target',hT);
   hW.compile('InputFrameNumberLimit',30);
   ```
2. The result of the code execution is:

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### The network includes the following layers:
     1   'data'                              Image Input                 224×224×3 images with 'zscore' normalization
     2   'conv1'                             Convolution                 64 7×7×3 convolutions with stride [2  2] and padding [3
     3   'bn_conv1'                          Batch Normalization         Batch normalization with 64 channels
     4   'conv1_relu'                        ReLU                        ReLU
     5   'pool1'                             Max Pooling                 3×3 max pooling with stride [2  2] and padding [1  1  1
     6   'res2a_branch2a'                    Convolution                 64 3×3×64 convolutions with stride [1  1] and padding [1
     7   'bn2a_branch2a'                     Batch Normalization         Batch normalization with 64 channels
     8   'res2a_branch2a_relu'               ReLU                        ReLU
     9   'res2a_branch2b'                    Convolution                 64 3×3×64 convolutions with stride [1  1] and padding [1
    10   'bn2a_branch2b'                     Batch Normalization         Batch normalization with 64 channels
    11   'res2a'                             Addition                    Element-wise addition of 2 inputs
    12   'res2a_relu'                        ReLU                        ReLU
    13   'res2b_branch2a'                    Convolution                 64 3×3×64 convolutions with stride [1  1] and padding [1
    14   'bn2b_branch2a'                     Batch Normalization         Batch normalization with 64 channels
    15   'res2b_branch2a_relu'               ReLU                        ReLU
    16   'res2b_branch2b'                    Convolution                 64 3×3×64 convolutions with stride [1  1] and padding [1
    17   'bn2b_branch2b'                     Batch Normalization         Batch normalization with 64 channels
    18   'res2b'                             Addition                    Element-wise addition of 2 inputs
    19   'res2b_relu'                        ReLU                        ReLU
    20   'res3a_branch2a'                    Convolution                 128 3×3×64 convolutions with stride [2  2] and padding [1
    21   'bn3a_branch2a'                     Batch Normalization         Batch normalization with 128 channels
    22   'res3a_branch2a_relu'               ReLU                        ReLU
    23   'res3a_branch2b'                    Convolution                 128 3×3×128 convolutions with stride [1  1] and padding
    24   'bn3a_branch2b'                     Batch Normalization         Batch normalization with 128 channels
    25   'res3a'                             Addition                    Element-wise addition of 2 inputs
    26   'res3a_relu'                        ReLU                        ReLU
    27   'res3a_branch1'                     Convolution                 128 1×1×64 convolutions with stride [2  2] and padding [0
    28   'bn3a_branch1'                      Batch Normalization         Batch normalization with 128 channels
    29   'res3b_branch2a'                    Convolution                 128 3×3×128 convolutions with stride [1  1] and padding
    30   'bn3b_branch2a'                     Batch Normalization         Batch normalization with 128 channels
    31   'res3b_branch2a_relu'               ReLU                        ReLU
    32   'res3b_branch2b'                    Convolution                 128 3×3×128 convolutions with stride [1  1] and padding
    33   'bn3b_branch2b'                     Batch Normalization         Batch normalization with 128 channels
    34   'res3b'                             Addition                    Element-wise addition of 2 inputs
    35   'res3b_relu'                        ReLU                        ReLU
    36   'res4a_branch2a'                    Convolution                 256 3×3×128 convolutions with stride [2  2] and padding
    37   'bn4a_branch2a'                     Batch Normalization         Batch normalization with 256 channels
    38   'res4a_branch2a_relu'               ReLU                        ReLU
    39   'res4a_branch2b'                    Convolution                 256 3×3×256 convolutions with stride [1  1] and padding
    40   'bn4a_branch2b'                     Batch Normalization         Batch normalization with 256 channels
    41   'res4a'                             Addition                    Element-wise addition of 2 inputs
    42   'res4a_relu'                        ReLU                        ReLU
    43   'res4a_branch1'                     Convolution                 256 1×1×128 convolutions with stride [2  2] and padding
    44   'bn4a_branch1'                      Batch Normalization         Batch normalization with 256 channels
    45   'res4b_branch2a'                    Convolution                 256 3×3×256 convolutions with stride [1  1] and padding
    46   'bn4b_branch2a'                     Batch Normalization         Batch normalization with 256 channels
    47   'res4b_branch2a_relu'               ReLU                        ReLU
    48   'res4b_branch2b'                    Convolution                 256 3×3×256 convolutions with stride [1  1] and padding
    49   'bn4b_branch2b'                     Batch Normalization         Batch normalization with 256 channels
    50   'res4b'                             Addition                    Element-wise addition of 2 inputs
    51   'res4b_relu'                        ReLU                        ReLU
    52   'res5a_branch2a'                    Convolution                 512 3×3×256 convolutions with stride [2  2] and padding
    53   'bn5a_branch2a'                     Batch Normalization         Batch normalization with 512 channels
    54   'res5a_branch2a_relu'               ReLU                        ReLU
    55   'res5a_branch2b'                    Convolution                 512 3×3×512 convolutions with stride [1  1] and padding
    56   'bn5a_branch2b'                     Batch Normalization         Batch normalization with 512 channels
    57   'res5a'                             Addition                    Element-wise addition of 2 inputs
    58   'res5a_relu'                        ReLU                        ReLU
    59   'res5a_branch1'                     Convolution                 512 1×1×256 convolutions with stride [2  2] and padding
    60   'bn5a_branch1'                      Batch Normalization         Batch normalization with 512 channels
    61   'res5b_branch2a'                    Convolution                 512 3×3×512 convolutions with stride [1  1] and padding
    62   'bn5b_branch2a'                     Batch Normalization         Batch normalization with 512 channels
    63   'res5b_branch2a_relu'               ReLU                        ReLU
    64   'res5b_branch2b'                    Convolution                 512 3×3×512 convolutions with stride [1  1] and padding
    65   'bn5b_branch2b'                     Batch Normalization         Batch normalization with 512 channels
    66   'res5b'                             Addition                    Element-wise addition of 2 inputs
    67   'res5b_relu'                        ReLU                        ReLU
    68   'pool5'                             2-D Global Average Pooling  2-D global average pooling
    69   'fc1000'                            Fully Connected             1000 fully connected layer
    70   'prob'                              Softmax                     softmax
    71   'ClassificationLayer_predictions'   Classification Output       crossentropyex with 'tench' and 999 other classes

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convolution2DLayer'
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data', an addition layer 'data_norm_add
### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in sof
### Compiling layer group: conv1>>pool1 ...
### Compiling layer group: conv1>>pool1 ... complete.
```

```
### Compiling layer group: res2a_branch2a>>res2a_branch2b ...
### Compiling layer group: res2a_branch2a>>res2a_branch2b ... complete.
### Compiling layer group: res2b_branch2a>>res2b_branch2b ...
### Compiling layer group: res2b_branch2a>>res2b_branch2b ... complete.
### Compiling layer group: res3a_branch1 ...
### Compiling layer group: res3a_branch1 ... complete.
### Compiling layer group: res3a_branch2a>>res3a_branch2b ...
### Compiling layer group: res3a_branch2a>>res3a_branch2b ... complete.
### Compiling layer group: res3b_branch2a>>res3b_branch2b ...
### Compiling layer group: res3b_branch2a>>res3b_branch2b ... complete.
### Compiling layer group: res4a_branch1 ...
### Compiling layer group: res4a_branch1 ... complete.
### Compiling layer group: res4a_branch2a>>res4a_branch2b ...
### Compiling layer group: res4a_branch2a>>res4a_branch2b ... complete.
### Compiling layer group: res4b_branch2a>>res4b_branch2b ...
### Compiling layer group: res4b_branch2a>>res4b_branch2b ... complete.
### Compiling layer group: res5a_branch1 ...
### Compiling layer group: res5a_branch1 ... complete.
### Compiling layer group: res5a_branch2a>>res5a_branch2b ...
### Compiling layer group: res5a_branch2a>>res5a_branch2b ... complete.
### Compiling layer group: res5b_branch2a>>res5b_branch2b ...
### Compiling layer group: res5b_branch2a>>res5b_branch2b ... complete.
### Compiling layer group: pool5 ...
### Compiling layer group: pool5 ... complete.
### Compiling layer group: fc1000 ...
### Compiling layer group: fc1000 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01c00000"     "8.0 MB"
    "SystemBufferOffset"        "0x02400000"     "28.0 MB"
    "InstructionDataOffset"     "0x04000000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04400000"     "52.0 MB"
    "FCWeightDataOffset"        "0x07800000"     "4.0 MB"
    "EndOffset"                 "0x07c00000"     "Total: 124.0 MB"

### Network compilation complete.
```

### Compile dagnet network object

1. Create a `dlhdl.Workflow` object with `resnet18` as the network for deployment to a Xilinx® Zynq® UltraScale+™ MPSoC ZCU102 board which uses `single` data types.

```
net = resnet18;
hTarget = dlhdl.Target('Xilinx');
hW = dlhdl.Workflow('Network',snet,'Bitstream','zcu102_single','Target',hTarget);
```

2. Call the `compile` function on `hW`

```
hW.compile
```

Calling the `compile` function, returns:

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

     1   'data'                              Image Input              224×224×3 images with 'zscore' normalization
     2   'conv1'                             Convolution              64 7×7×3 convolutions with stride [2  2] and p
     3   'bn_conv1'                          Batch Normalization      Batch normalization with 64 channels
     4   'conv1_relu'                        ReLU                     ReLU
     5   'pool1'                             Max Pooling              3×3 max pooling with stride [2  2] and padding
     6   'res2a_branch2a'                    Convolution              64 3×3×64 convolutions with stride [1  1] and
     7   'bn2a_branch2a'                     Batch Normalization      Batch normalization with 64 channels
     8   'res2a_branch2a_relu'               ReLU                     ReLU
     9   'res2a_branch2b'                    Convolution              64 3×3×64 convolutions with stride [1  1] and
    10   'bn2a_branch2b'                     Batch Normalization      Batch normalization with 64 channels
    11   'res2a'                             Addition                 Element-wise addition of 2 inputs
```

```
    12   'res2a_relu'                        ReLU                     ReLU
    13   'res2b_branch2a'                    Convolution              64 3×3×64 convolutions with stride [1  1] and
    14   'bn2b_branch2a'                     Batch Normalization      Batch normalization with 64 channels
    15   'res2b_branch2a_relu'               ReLU                     ReLU
    16   'res2b_branch2b'                    Convolution              64 3×3×64 convolutions with stride [1  1] and
    17   'bn2b_branch2b'                     Batch Normalization      Batch normalization with 64 channels
    18   'res2b'                             Addition                 Element-wise addition of 2 inputs
    19   'res2b_relu'                        ReLU                     ReLU
    20   'res3a_branch2a'                    Convolution              128 3×3×64 convolutions with stride [2  2] and
    21   'bn3a_branch2a'                     Batch Normalization      Batch normalization with 128 channels
    22   'res3a_branch2a_relu'               ReLU                     ReLU
    23   'res3a_branch2b'                    Convolution              128 3×3×128 convolutions with stride [1  1] an
    24   'bn3a_branch2b'                     Batch Normalization      Batch normalization with 128 channels
    25   'res3a'                             Addition                 Element-wise addition of 2 inputs
    26   'res3a_relu'                        ReLU                     ReLU
    27   'res3a_branch1'                     Convolution              128 1×1×64 convolutions with stride [2  2] and
    28   'bn3a_branch1'                      Batch Normalization      Batch normalization with 128 channels
    29   'res3b_branch2a'                    Convolution              128 3×3×128 convolutions with stride [1  1] an
    30   'bn3b_branch2a'                     Batch Normalization      Batch normalization with 128 channels
    31   'res3b_branch2a_relu'               ReLU                     ReLU
    32   'res3b_branch2b'                    Convolution              128 3×3×128 convolutions with stride [1  1] an
    33   'bn3b_branch2b'                     Batch Normalization      Batch normalization with 128 channels
    34   'res3b'                             Addition                 Element-wise addition of 2 inputs
    35   'res3b_relu'                        ReLU                     ReLU
    36   'res4a_branch2a'                    Convolution              256 3×3×128 convolutions with stride [2  2] an
    37   'bn4a_branch2a'                     Batch Normalization      Batch normalization with 256 channels
    38   'res4a_branch2a_relu'               ReLU                     ReLU
    39   'res4a_branch2b'                    Convolution              256 3×3×256 convolutions with stride [1  1] an
    40   'bn4a_branch2b'                     Batch Normalization      Batch normalization with 256 channels
    41   'res4a'                             Addition                 Element-wise addition of 2 inputs
    42   'res4a_relu'                        ReLU                     ReLU
    43   'res4a_branch1'                     Convolution              256 1×1×128 convolutions with stride [2  2] an
    44   'bn4a_branch1'                      Batch Normalization      Batch normalization with 256 channels
    45   'res4b_branch2a'                    Convolution              256 3×3×256 convolutions with stride [1  1] an
    46   'bn4b_branch2a'                     Batch Normalization      Batch normalization with 256 channels
    47   'res4b_branch2a_relu'               ReLU                     ReLU
    48   'res4b_branch2b'                    Convolution              256 3×3×256 convolutions with stride [1  1] an
    49   'bn4b_branch2b'                     Batch Normalization      Batch normalization with 256 channels
    50   'res4b'                             Addition                 Element-wise addition of 2 inputs
    51   'res4b_relu'                        ReLU                     ReLU
    52   'res5a_branch2a'                    Convolution              512 3×3×256 convolutions with stride [2  2] an
    53   'bn5a_branch2a'                     Batch Normalization      Batch normalization with 512 channels
    54   'res5a_branch2a_relu'               ReLU                     ReLU
    55   'res5a_branch2b'                    Convolution              512 3×3×512 convolutions with stride [1  1] an
    56   'bn5a_branch2b'                     Batch Normalization      Batch normalization with 512 channels
    57   'res5a'                             Addition                 Element-wise addition of 2 inputs
    58   'res5a_relu'                        ReLU                     ReLU
    59   'res5a_branch1'                     Convolution              512 1×1×256 convolutions with stride [2  2] an
    60   'bn5a_branch1'                      Batch Normalization      Batch normalization with 512 channels
    61   'res5b_branch2a'                    Convolution              512 3×3×512 convolutions with stride [1  1] an
    62   'bn5b_branch2a'                     Batch Normalization      Batch normalization with 512 channels
    63   'res5b_branch2a_relu'               ReLU                     ReLU
    64   'res5b_branch2b'                    Convolution              512 3×3×512 convolutions with stride [1  1] an
    65   'bn5b_branch2b'                     Batch Normalization      Batch normalization with 512 channels
    66   'res5b'                             Addition                 Element-wise addition of 2 inputs
    67   'res5b_relu'                        ReLU                     ReLU
    68   'pool5'                             Global Average Pooling   Global average pooling
    69   'fc1000'                            Fully Connected          1000 fully connected layer
    70   'prob'                              Softmax                  softmax
    71   'ClassificationLayer_predictions'   Classification Output    crossentropyex with 'tench' and 999 other clas

### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convolution2DLaye
5 Memory Regions created.

Skipping: data
Compiling leg: conv1>>pool1 ...
Compiling leg: conv1>>pool1 ... complete.
Compiling leg: res2a_branch2a>>res2a_branch2b ...
Compiling leg: res2a_branch2a>>res2a_branch2b ... complete.
Compiling leg: res2b_branch2a>>res2b_branch2b ...
Compiling leg: res2b_branch2a>>res2b_branch2b ... complete.
Compiling leg: res3a_branch2a>>res3a_branch2b ...
```

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

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x02000000"     "28.0 MB"
    "InstructionDataOffset"     "0x03c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04000000"     "52.0 MB"
    "FCWeightDataOffset"        "0x07400000"     "4.0 MB"
    "EndOffset"                 "0x07800000"     "Total: 120.0 MB"

### Network compilation complete.


ans =

  struct with fields:

             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
```

#### Enable Hardware Implementation of Input Image Layer Normalization Function

1 Create a `dlhdl.Workflow` object with `resnet18` as the network for deployment to a Xilinx Zynq UltraScale+ MPSoC ZCU102 board which uses `single` data types.

```
net = resnet18;
hTarget = dlhdl.Target('Xilinx',Interface = 'Ethernet');
hW = dlhdl.Workflow(Network = net,Bitstream ='zcu102_single',Target = hTarget);
```

**2** Call the `compile` function on `hW`. . Enable hardware implementation of the input image layer normalization function by setting the`HardwareNormalization` argument to `auto`.

```
hW.compile(HardwareNormalization = 'auto')
```

Calling the `compile` function, returns:

```
### Compiling network for Deep Learning FPGA prototyping ...
### Targeting FPGA bitstream zcu102_single.
### The network includes the following layers:
     1   'data'                         Image Input             224×224×3 images with 'zscore' normalization
     2   'conv1'                        Convolution             64 7×7×3 convolutions with stride [2  2] and padding [3
     3   'bn_conv1'                     Batch Normalization     Batch normalization with 64 channels
     4   'conv1_relu'                   ReLU                    ReLU
     5   'pool1'                        Max Pooling             3×3 max pooling with stride [2  2] and padding [1  1  1
     6   'res2a_branch2a'               Convolution             64 3×3×64 convolutions with stride [1  1] and padding [1
     7   'bn2a_branch2a'                Batch Normalization     Batch normalization with 64 channels
     8   'res2a_branch2a_relu'          ReLU                    ReLU
     9   'res2a_branch2b'               Convolution             64 3×3×64 convolutions with stride [1  1] and padding [1
    10   'bn2a_branch2b'                Batch Normalization     Batch normalization with 64 channels
    11   'res2a'                        Addition                Element-wise addition of 2 inputs
    12   'res2a_relu'                   ReLU                    ReLU
    13   'res2b_branch2a'               Convolution             64 3×3×64 convolutions with stride [1  1] and padding [1
    14   'bn2b_branch2a'                Batch Normalization     Batch normalization with 64 channels
    15   'res2b_branch2a_relu'          ReLU                    ReLU
    16   'res2b_branch2b'               Convolution             64 3×3×64 convolutions with stride [1  1] and padding [1
    17   'bn2b_branch2b'                Batch Normalization     Batch normalization with 64 channels
    18   'res2b'                        Addition                Element-wise addition of 2 inputs
    19   'res2b_relu'                   ReLU                    ReLU
    20   'res3a_branch2a'               Convolution             128 3×3×64 convolutions with stride [2  2] and padding [
    21   'bn3a_branch2a'                Batch Normalization     Batch normalization with 128 channels
    22   'res3a_branch2a_relu'          ReLU                    ReLU
    23   'res3a_branch2b'               Convolution             128 3×3×128 convolutions with stride [1  1] and padding
    24   'bn3a_branch2b'                Batch Normalization     Batch normalization with 128 channels
    25   'res3a'                        Addition                Element-wise addition of 2 inputs
    26   'res3a_relu'                   ReLU                    ReLU
    27   'res3a_branch1'                Convolution             128 1×1×64 convolutions with stride [2  2] and padding [
    28   'bn3a_branch1'                 Batch Normalization     Batch normalization with 128 channels
    29   'res3b_branch2a'               Convolution             128 3×3×128 convolutions with stride [1  1] and padding
    30   'bn3b_branch2a'                Batch Normalization     Batch normalization with 128 channels
    31   'res3b_branch2a_relu'          ReLU                    ReLU
    32   'res3b_branch2b'               Convolution             128 3×3×128 convolutions with stride [1  1] and padding
    33   'bn3b_branch2b'                Batch Normalization     Batch normalization with 128 channels
    34   'res3b'                        Addition                Element-wise addition of 2 inputs
    35   'res3b_relu'                   ReLU                    ReLU
    36   'res4a_branch2a'               Convolution             256 3×3×128 convolutions with stride [2  2] and padding
    37   'bn4a_branch2a'                Batch Normalization     Batch normalization with 256 channels
    38   'res4a_branch2a_relu'          ReLU                    ReLU
    39   'res4a_branch2b'               Convolution             256 3×3×256 convolutions with stride [1  1] and padding
    40   'bn4a_branch2b'                Batch Normalization     Batch normalization with 256 channels
    41   'res4a'                        Addition                Element-wise addition of 2 inputs
    42   'res4a_relu'                   ReLU                    ReLU
    43   'res4a_branch1'                Convolution             256 1×1×128 convolutions with stride [2  2] and padding
    44   'bn4a_branch1'                 Batch Normalization     Batch normalization with 256 channels
    45   'res4b_branch2a'               Convolution             256 3×3×256 convolutions with stride [1  1] and padding
    46   'bn4b_branch2a'                Batch Normalization     Batch normalization with 256 channels
    47   'res4b_branch2a_relu'          ReLU                    ReLU
    48   'res4b_branch2b'               Convolution             256 3×3×256 convolutions with stride [1  1] and padding
    49   'bn4b_branch2b'                Batch Normalization     Batch normalization with 256 channels
    50   'res4b'                        Addition                Element-wise addition of 2 inputs
    51   'res4b_relu'                   ReLU                    ReLU
    52   'res5a_branch2a'               Convolution             512 3×3×256 convolutions with stride [2  2] and padding
    53   'bn5a_branch2a'                Batch Normalization     Batch normalization with 512 channels
    54   'res5a_branch2a_relu'          ReLU                    ReLU
    55   'res5a_branch2b'               Convolution             512 3×3×512 convolutions with stride [1  1] and padding
    56   'bn5a_branch2b'                Batch Normalization     Batch normalization with 512 channels
    57   'res5a'                        Addition                Element-wise addition of 2 inputs
    58   'res5a_relu'                   ReLU                    ReLU
    59   'res5a_branch1'                Convolution             512 1×1×256 convolutions with stride [2  2] and padding
    60   'bn5a_branch1'                 Batch Normalization     Batch normalization with 512 channels
    61   'res5b_branch2a'               Convolution             512 3×3×512 convolutions with stride [1  1] and padding
    62   'bn5b_branch2a'                Batch Normalization     Batch normalization with 512 channels
    63   'res5b_branch2a_relu'          ReLU                    ReLU
    64   'res5b_branch2b'               Convolution             512 3×3×512 convolutions with stride [1  1] and padding
    65   'bn5b_branch2b'                Batch Normalization     Batch normalization with 512 channels
    66   'res5b'                        Addition                Element-wise addition of 2 inputs
```

```
    67   'res5b_relu'                        ReLU                          ReLU
    68   'pool5'                             2-D Global Average Pooling   2-D global average pooling
    69   'fc1000'                            Fully Connected               1000 fully connected layer
    70   'prob'                              Softmax                       softmax
    71   'ClassificationLayer_predictions'   Classification Output         crossentropyex with 'tench' and 999 other classes

### Optimizing network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convolution2DLayer'
### Notice: The layer 'data' of type 'ImageInputLayer' is split into an image input layer 'data', an addition layer 'data_norm_add
### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in sof
### Compiling layer group: conv1>>pool1 ...
### Compiling layer group: conv1>>pool1 ... complete.
### Compiling layer group: res2a_branch2a>>res2a_branch2b ...
### Compiling layer group: res2a_branch2a>>res2a_branch2b ... complete.
### Compiling layer group: res2b_branch2a>>res2b_branch2b ...
### Compiling layer group: res2b_branch2a>>res2b_branch2b ... complete.
### Compiling layer group: res3a_branch1 ...
### Compiling layer group: res3a_branch1 ... complete.
### Compiling layer group: res3a_branch2a>>res3a_branch2b ...
### Compiling layer group: res3a_branch2a>>res3a_branch2b ... complete.
### Compiling layer group: res3b_branch2a>>res3b_branch2b ...
### Compiling layer group: res3b_branch2a>>res3b_branch2b ... complete.
### Compiling layer group: res4a_branch1 ...
### Compiling layer group: res4a_branch1 ... complete.
### Compiling layer group: res4a_branch2a>>res4a_branch2b ...
### Compiling layer group: res4a_branch2a>>res4a_branch2b ... complete.
### Compiling layer group: res4b_branch2a>>res4b_branch2b ...
### Compiling layer group: res4b_branch2a>>res4b_branch2b ... complete.
### Compiling layer group: res5a_branch1 ...
### Compiling layer group: res5a_branch1 ... complete.
### Compiling layer group: res5a_branch2a>>res5a_branch2b ...
### Compiling layer group: res5a_branch2a>>res5a_branch2b ... complete.
### Compiling layer group: res5b_branch2a>>res5b_branch2b ...
### Compiling layer group: res5b_branch2a>>res5b_branch2b ... complete.
### Compiling layer group: pool5 ...
### Compiling layer group: pool5 ... complete.
### Compiling layer group: fc1000 ...
### Compiling layer group: fc1000 ... complete.

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01c00000"     "8.0 MB"
    "SystemBufferOffset"        "0x02400000"     "28.0 MB"
    "InstructionDataOffset"     "0x04000000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04400000"     "52.0 MB"
    "FCWeightDataOffset"        "0x07800000"     "4.0 MB"
    "EndOffset"                 "0x07c00000"     "Total: 124.0 MB"

### Network compilation complete.


ans =

  struct with fields:

             weights: [1×1 struct]
        instructions: [1×1 struct]
           registers: [1×1 struct]
    syncInstructions: [1×1 struct]
        constantData: {{1×2 cell}  [0.0171 0.0175 0.0174 0 0.0171 0.0175 0.0174 0 0.0171 0.0175 0.0174 0 0.0171 0.0175 0.0174 0 …
```

During compilation the compiler splits the image input layer into an image input layer, addition layer, and multiplication layer for hardware implementation.

## See Also

deploy | getBuildInfo | predict

### Topics

"Use the Compiler Output for System Integration"
"Image Input Layer Normalization Hardware Implementation"

**Introduced in R2020b**

# deploy

**Class:** dlhdl.Workflow
**Package:** dlhdl

Deploy the specified neural network to the target FPGA board

## Syntax

`deploy(workflowObject)`

## Description

`deploy(workflowObject)` programs the specified target board with the bitstream and deploys the deep learning network on it.

## Input Arguments

**workflowObject — Instance of workflow object**
dlhdl.Workflow object

Instance of workflow object, specified as an `dlhld.Workflow` object.

## Examples

### Deploy LogoNet to Intel Arria 10 SoC Development Kit

Deploy VGG-19 to the Intel Arria 10 SoC development kit that has `single` data types.

The `deploy` function starts programming the FPGA device, displays progress messages, and the time it takes to deploy the network.

1 
```
snet = vgg19;
hTarget = dlhdl.Target('Intel');
hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'arria10soc_single','Target',hTarget);
hW.deploy
```
2 
```
### Programming FPGA bitstream using JTAG ...
### Programming FPGA bitstream has completed successfully.

tableOut =
          offset_name          offset_address     allocated_space
    _______________________    ______________    _________________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SystemBufferOffset"        "0x01c00000"     "52.0 MB"
    "InstructionDataOffset"     "0x05000000"     "20.0 MB"
    "ConvWeightDataOffset"      "0x06400000"     "276.0 MB"
    "FCWeightDataOffset"        "0x17800000"     "472.0 MB"
    "EndOffset"                 "0x35000000"     "Total: 848.0 MB"

### Loading weights to FC Processor.
### 4% finished, current time is 14-Jun-2020 18:31:07.
### 8% finished, current time is 14-Jun-2020 18:31:32.
### 12% finished, current time is 14-Jun-2020 18:31:58.
```

```
### 16% finished, current time is 14-Jun-2020 18:32:23.
### 20% finished, current time is 14-Jun-2020 18:32:48.
### 24% finished, current time is 14-Jun-2020 18:33:13.
### 28% finished, current time is 14-Jun-2020 18:33:39.
### 32% finished, current time is 14-Jun-2020 18:34:04.
### 36% finished, current time is 14-Jun-2020 18:34:30.
### 40% finished, current time is 14-Jun-2020 18:34:56.
### 44% finished, current time is 14-Jun-2020 18:35:21.
### 48% finished, current time is 14-Jun-2020 18:35:46.
### 52% finished, current time is 14-Jun-2020 18:36:11.
### 56% finished, current time is 14-Jun-2020 18:36:36.
### 60% finished, current time is 14-Jun-2020 18:37:02.
### 64% finished, current time is 14-Jun-2020 18:37:27.
### 68% finished, current time is 14-Jun-2020 18:37:52.
### 72% finished, current time is 14-Jun-2020 18:38:17.
### 76% finished, current time is 14-Jun-2020 18:38:43.
### 80% finished, current time is 14-Jun-2020 18:39:08.
### 84% finished, current time is 14-Jun-2020 18:39:33.
### 88% finished, current time is 14-Jun-2020 18:39:58.
### 92% finished, current time is 14-Jun-2020 18:40:23.
### 96% finished, current time is 14-Jun-2020 18:40:48.
### FC Weights loaded. Current time is 14-Jun-2020 18:41:06
```

#### Deploy Quantized LogoNet to Xilinx ZCU102 Development Kit

1. 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
```

2. Create an image datastore and split 70 percent of the images into a training data set and 30 percent of the images into a validation data set.

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

3. Create a `dlhdl.Workflow` object which has quantized LogoNet as the network argument, `zcu102_int8` as the bitstream, and `hTarget` as the target argument.

   To quantize the network, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

```
% Save the pretrained SeriesNetwork object
snet = getLogoNetwork;

% Create a Target object and define the interface to the target board
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');

% Create a Quantized Network Object

dlquantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
dlquantObj.calibrate(imdsTrain);
```

```
% Create a workflow object for the SeriesNetwork and using the FPFA bitstream
hW = dlhdl.Workflow('Network', dlquantObj, 'Bitstream', 'zcu102_int8','Target',hTarget);
```

4. Deploy the `dlhdl.Workflow` object to your target FPGA board by using the `deploy` method.

```
% Deploy the workflow object
hW.deploy;
```

5. When you call the `deploy` method, the method returns:

```
### 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 reboot for persisten


System is rebooting . . . . . .
### Programming the FPGA bitstream has been completed successfully.
          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"

### Loading weights to FC Processor.
### FC Weights loaded. Current time is 12-Jun-2020 13:17:56
```

## See Also

`compile` | `getBuildInfo` | `predict` | `dlquantizer` | `dlquantizationOptions` | `calibrate` | `validate`

**Introduced in R2020b**

# getBuildInfo

**Class:** dlhdl.Workflow
**Package:** dlhdl

Retrieve bitstream resource utilization

## Syntax

area = getBuildInfo(workflowObject)

## Description

area = getBuildInfo(workflowObject) returns a structure containing the bitstream resource utilization.

## Input Arguments

### workflowObject — Instance of workflow object

dlhdl.Workflow object

Instance of workflow object, specified as an dlhld.Workflow object.

## Output Arguments

### area — Bitstream resource utilization

structure

Bitstream resource utilization returned as a structure.

- The Block Memory Bits utilization is available for Intel bitstreams only.
- The resource utilization results of Intel bitstreams show the Block RAM utilization as 100%. To analyze bitstream resource utilization, refer to the Block Memory Bits utilization instead.

## Examples

### Retrieve arria10soc_singleBitstream Resource Utilization

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

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

2. Create a dlhdl.Workflow object that has LogoNet as the Network argument and arria10soc_single as the Bitstream argument.

```
snet = getLogoNetwork;
hW = dlhdl.Workflow('Network',snet,'Bitstream','arria10soc_single');
```

3. Call `getBuildInfo` argument to retrieve the `arria10soc_single` resource utilization. Store the resource utilization in `area`.

```
area = hW.getBuildInfo
```

```
              Deep Learning Processor Bitstream Build Info

Resource                   Utilized           Total        Percentage
------------------        ----------      ------------    ------------
LUTs (CLB/ALM)*               93578          251680           37.18
DSPs                            278            1687           16.48
Block RAM                      2131            2131          100.00
Block Memory Bits          23211920        43642880           53.19
* LUT count represents Configurable Logic Block(CLB) utilization in Xilinx devices and Adaptive Logic Module (ALM) u


area = 

  struct with fields:

                LUT: [93578 251680]
    BlockMemoryBits: [23211920 43642880]
           BlockRAM: [2131 2131]
                DSP: [278 1687]
```

## See Also

compile | deploy | predict

**Introduced in R2021a**

# predict

**Class:** `dlhdl.Workflow`
**Package:** `dlhdl`

Run inference on deployed network and profile speed of neural network deployed on specified target device

## Syntax

```
predict(workflowObject,image)
predict(workflowObject,image, Name,Value)
```

## Description

`predict(workflowObject,image)` predicts responses for the image data in `imds` by using the deep learning network that you specified in the `dlhdl.Workflow` class for deployment on the specified target board and returns the results.

`predict(workflowObject,image, Name,Value)` predicts responses for the image data in `imds` by using the deep learning network that you specified by using the `dlhdl.Workflow` class for deployment on the specified target boards and returns the results, with one or more arguments specified by optional name-value pair arguments.

## Input Arguments

**`workflowObject` — Instance of workflow object**
`dlhdl.Workflow` object

Instance of workflow object, specified as an `dlhld.Workflow` object.

**`image` — Input image**
*m*-by-*n*-by-*k* numeric array

Input image, specified as a *m*-by-*n*-by-*k* numeric array. *m*, *n*, and *k* must match the dimensions of the deep learning network input image layer. For example, for the LogoNet network, resize the input images to a 227-by-227-by-3 array.

Data Types: `single`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

Example:

**`Profile` — Flag that returns profiling results**
'off' (default) | 'on'

Flag to return profiling results, for the deep learning network deployed to the target board.

Example: `'Profile', 'On'`

## Examples

### Predict Outcome and Profile Results

1. Create a `dlhdl.Workflow` object that has VGG-19 as the network argument, `arria10soc_single` as the bitstream argument, and *hTarget* as the target argument.

```
% Save the pretrained SeriesNetwork object
snet = vgg19;

% Create a Target object and define the interface to the target board
hTarget = dlhdl.Target('Intel');

% Create a workflow object for the SeriesNetwork and using the FPFA bitstream
hW = dlhdl.Workflow('Network', snet, 'Bitstream', 'arria10soc_single','Target',hTarget);
```

2. Load your input image and resize the input image to match the image input layer size for the VGG-19 network.

```
% Load input images and resize them according to the network specifications
image = imread('zebra.jpeg');
inputImg = imresize(image, [224, 224]);
imshow(inputImg);
imIn = single(inputImg);
```

3. Deploy the `dlhdl.Workflow` object to your target FPGA board by using the `deploy` method. Retrieve the VGG-19 network prediction result for your input image from the FPGA board by using the `predict` method.

```
% Deploy the workflow object
hW.deploy;
% Predict the outcome and optionally profile the results to measure performance.
[prediction, speed] = hW.predict(imIn,'Profile','on');
[val, idx] = max(prediction);
snet.Layers(end).ClassNames{idx}
```

### Obtain Prediction Results for Quantized LogoNet Network

This example shows how to use the `predict` method to retrieve the prediction results for an input image from a deployed quantized LogoNet network.

1. Create a file in your current working folder 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
```

2. Create an image datastore and split 70 percent of the images into a training data set and 30 percent of the images into a validation data set.

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

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

3. Create a `dlhdl.Workflow` object which has quantized LogoNet as the network argument, `zcu102_int8` as the bitstream, and `hTarget` as the target argument.

   To quantize the network, you need the products listed under `FPGA` in "Quantization Workflow Prerequisites".

```
% Save the pretrained SeriesNetwork object
snet = getLogoNetwork;

% Create a Target object and define the interface to the target board
hTarget = dlhdl.Target('Xilinx','Interface','Ethernet');

% Create a Quantized Network Object

dlquantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA');
dlquantObj.calibrate(imdsTrain);

% Create a workflow object for the SeriesNetwork and using the FPFA bitstream
hW = dlhdl.Workflow('Network', dlquantObj, 'Bitstream', 'zcu102_int8','Target',hTarget);
```

4. Retrieve a random image from `logos_dataset`.

```
index = randperm(numel(imdsValidation.Files),1)
imIn = readimage(imdsValidation,index)
inputImg = imresize(imIn, [227 227]);
```

5. Deploy the `dlhdl.Workflow` object to your target FPGA board by using the `deploy` method. Retrieve the prediction for the image by using the `predict` method.

```
% Deploy the workflow object
hW.deploy;
% Predict the outcome and optionally profile the results to measure performance.
[prediction, speed] = hW.predict(single(inputImg),'Profile','on');
[val, idx] = max(prediction);
snet.Layers(end).ClassNames{idx}
```

## See Also

`compile` | `deploy` | `getBuildInfo` | `dlquantizer` | `dlquantizationOptions` | `calibrate` | `validate`

### Topics

"Profile Inference Run"
"Profile Network for Performance Improvement"

**Introduced in R2020b**

# dlhdl.Target class

**Package:** dlhdl

Configure interface to target board for workflow deployment

## Description

Use the dlhdl.Target object to create the interface to deploy the dlhdl.Workflow object to your target hardware.

# Creation

hTarget = dlhdl.Target(Vendor) creates a target object that you pass on to dlhdl.Workflow to deploy your deep learning network to your target device.

hTarget = dlhdl.Target(Vendor,Name,Value) creates a target object that you pass on to dlhdl.Workflow, with additional properties specified by one or more Name,Value pair arguments.

### Input Arguments

#### Vendor — Target board vendor name

'Xilinx' (default) | 'Intel' | "Xilinx" | "Intel"

Target device vendor name, specified as a character vector or string.

Example: 'Xilinx' or "Xilinx"

## Properties

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

#### Interface — Interface to connect to the target board

'JTAG' (default) | 'Ethernet' | 'File'

Name of the interface specified as a character vector.

Example: 'Interface','JTAG' creates a target configuration object with 'JTAG' as the interface to the target device.

#### IPAddress — IP address for the target device with Ethernet interface

'' (default)

IP address for the target device with the Ethernet interface specified as a character vector.

Example: 'IPAddress','192.168.1.101' creates a target configuration object with '192.168.1.101' as the target device IP address.

#### Username — SSH user name

'root' (default)

SSH user name specified as a character vector.

Example: `'Username','root'` creates a target configuration object with `'root'` as the SSH user name.

#### `Password` — SSH password
`'root'` | `'cyclonevsoc'`

Password of the root user specified as a character vector. Use `'root'` on the Xilinx SoC boards and `'cyclonevsoc'` on the Intel SoC boards.

Example: `'Password','root'` creates a target configuration object with `'root'` as the SSH password for Xilinx SoC boards.

Example: `'Password','cyclonevsoc'` creates a target configuration object with `'cyclonevsoc'` as the SSH password for Intel SoC boards.

#### `Port` — SSH connection port number
22 (default)

SSH port number specified as an integer.

Example: `'Port',22` creates a target configuration object with `22` as the SSH port number.

#### `Filename` — Name of the deployment file
`' '` (default)

Name of the deployment file specified as a string or character vector. When you do not specify the file name, the name of the bitstream is the file name.

Example: `'Filename','zcu102socinit"` creates a `'zcu102soc_init.dln'` file.

## Methods

#### Public Methods

| | |
|---|---|
| release | Release the connection to the target device |
| validateConnection | Validate SSH connection and deployed bitstream |

## Examples

### Create Target Object That Has a JTAG interface

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

  Target with properties:

       Vendor: 'Xilinx'
    Interface: JTAG
```

### Create Target Object That Has an Ethernet Interface and Set IP Address

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

  Target with properties:

       Vendor: 'Xilinx'
```

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

### Create Target Object That has a File Interface

```
hTarget = dlhdl.Target('Xilinx',Interface = 'File', Filename = 'zcu102socinit')
```

```
hTarget = 

  TargetFile with properties:

      Interface: File
           Path: './'
     FileHandle: -1
             CR: '←↵'
       Filename: 'zcu102socinit'
    IsConnected: 0
          hConn: []
         Vendor: 'Xilinx'
```

## See Also

**Functions**
release | validateConnection

**Objects**
dlhdl.Workflow

**Introduced in R2020b**

# release

**Class:** `dlhdl.Target`
**Package:** `dlhdl`

Release the connection to the target device

## Syntax

`release(targetObject)`

## Description

`release(targetObject)` releases the connection to the target board.

## Input Arguments

**`targetObject` — Instance of target object**
`dlhdl.Target` object

Instance of the target object, specified as an `dlhdl.Target` object.

## Examples

### Release Connection to Target Device

1. Create a `dlhdl.Target` object that has an Ethernet interface and SSH connection.

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

2. Create a `dlhdl.Workflow` object and deploy the object to the target board.

```
net = resnet18;
hW = dlhdl.Workflow('Network',net,'Bitstream','zcu102_single', 'Target', hTarget);
hW.deploy;
```

3. Obtain a prediction.

   Use this image to run the code:

```
% Load input images and resize them according to the network specifications
image = imread('zebra.jpeg');
inputImg = imresize(image, [224, 224]);
imshow(inputImg);
% Predict the outcome and optionally profile the results to measure performance.
[prediction, speed] = hW.predict(single(inputImg), 'Profile', 'on' );
```

4. Release the connection.

```
hTarget.release;
```

## See Also

validateConnection

**Introduced in R2020b**

# validateConnection

**Class:** dlhdl.Target
**Package:** dlhdl

Validate SSH connection and deployed bitstream

## Syntax

validateConnection(targetObject)

## Description

validateConnection(targetObject):

1. First validates the SSH connection for an Ethernet interface. This step is skipped for a JTAG interface.
2. Validates the connection for a deployed bitstream.

## Input Arguments

**targetObject — Instance of target object**
dlhdl.Target object

Instance of the target object, specified as an dlhdl.Target object.

## Examples

### Validate dlhdl.Target Object that has a JTAG Interface

Validate deployed bitstream and SSH connection to the target device.

1. Create a dlhdl.Target object with a JTAG interface.

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

2. Create a dlhdl.Workflow object and deploy the object to the target board.

```
snet = vgg19;
hW = dlhdl.Workflow('Network',snet,'Bitstream','arria10soc_single', 'Target', hTarget);
hW.deploy;
```

3. Validate the connection and bitstream.

```
hTarget.validateConnection
### Validating connection to bitstream over JTAG interface
### Bitstream connection over JTAG interface successful
```

### Validate dlhdl.Target Object that has an Ethernet Interface

Validate deployed bitstream and SSH connection to the target device.

1. Create a `dlhdl.Target` object that has an Ethernet interface.

   ```
   hTarget = dlhdl.Target('Xilinx','Interface','Ethernet','IPAddress','10.10.10.14');
   ```
2. Create a `dlhdl.Workflow` object and deploy the object to the target board.

   ```
   net = resnet18;
   hW = dlhdl.Workflow('Network',net,'Bitstream','zcu102_single', 'Target', hTarget);
   hW.deploy;
   ```
3. Validate the connection and bitstream.

   ```
   hTarget.validateConnection
   ### Validating connection to target over SSH
   ### SSH connection successful
   ### Validating connection to bitstream over Ethernet interface
   ### Bitstream connection over Ethernet interface successful
   ```

## See Also

`release`

**Introduced in R2020b**

# dlhdl.ProcessorConfig class

**Package:** `dlhdl`

Configure custom deep learning processor

## Description

Use the `dlhdl.ProcessorConfig` class to configure a custom processor, which is then passed on to the `dlhdl.buildProcessor` class to generate a custom deep learning processor.

# Creation

The `dlhdl.ProcessorConfig` class creates a custom processor configuration object that you can use to specify the processor parameters. The processor parameters are then used by the `dlhdl.buildProcessor` class to build and generate code for your custom deep learning processor.

`dlhdl.ProcessorConfig(Name,Value)` creates a custom processor configuration object, with additional options specified by one or more name-value arguments.

## Properties

#### System Level Properties

**`ReferenceDesign` — Reference design name**
'AXI-Stream DDR Memory Access : 3-AXIM' (default) | character vector

Reference design name, specified as a character vector

Example: `'AXI-Stream DDR memory Access : 3-AXIM'`

**`SynthesisTool` — Synthesis tool name**
'Xilinx Vivado' (default) | 'Altera Quartus II' | 'Xilinx ISE' | character vector

Synthesis tool name, specified as a character vector.

Example: `Xilinx Vivado`

**`SynthesisToolChipFamily` — Synthesis tool chip family name**
'Zynq Ultrascale+ (default) | 'Artix7' | 'Kintex7' | 'Kintex Ultrascale+' | 'Spartan7' | 'Virtex7' | 'Virtex Ultrascale+' | 'Zynq' | 'Arria 10' | 'Arria V GZ' | 'Cyclone 10 GX' | 'Stratix 10' | 'Stratix V' | character vector

Specify the target device chip family name as a character vector

Example: `'Zynq'`

**`TargetFrequency` — Target frequency in MHz**
200 (default) | integer

Specify the target board frequency in MHz.

Example: `220`

**`TargetPlatform` — Name of the target board**
'Xilinx Zynq Ultrascale+ MPSoC ZCU 102 Evaluation Kit' (default) | 'Generic Deep Learning Processor' | 'Altera Arria 10 SoC development kit' | 'Xilinx Zynq ZC706 evaluation kit' | character vector

Specify the name of the target board as a character vector.

Example: `'Xilinx Zynq ZC706 evaluation kit'`

**`Bitstream` — Name of the bitstream**
'arria10soc_single' | 'arria10soc_int8' | 'zc706_single' | 'zc706_int8' | 'zcu102_single' | 'zcu102_int8'

Specify the name of the bitstream whose processor configuration must be retrieved as a character vector.

Example: 'Bitstream','zcu102_single'

**Top Level Properties**

**`RunTimeControl` — Deep learning processor IP core mode setting**
'register' (default) | 'port' | character vector

Specify whether the runtime input signals to the deep learning processor IP core are implemented as registers or ports.

Example: `'register`

**`RunTimeStatus` — Feedback signals from deep learning processor IP core control**
'register' (default) | 'port'

Specify whether the runtime output signals from the deep learning processor IP core are implemented as registers or ports.

Example: 'register'

**`InputStreamControl` — Deep learning processor IP core input interface control**
'register' (default) | 'port' | character vector

Specify the deep learning processor IP core input interface option as a character vector

Example: `'register'`

**`OutputStreamControl` — Deep learning processor IP core output interface control**
'register' (default) | 'port' | character vector

Specify the deep learning processor IP core output interface option as a character vector

Example: `'port'`

**`ProcessorDataType` — Deep learning processor IP core module data type**
'single (default) | 'int8' | 'half' | character vector

Specify the deep learning processor IP core module data type as a character vector.

Example: `'single'`

#### Processing Module conv Properties

**`ModuleGeneration` — Enable or disable convolution module generation as a part of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the convolution module as a part of the deep learning processor configuration.

**`LRNBlockGeneration` — Enable or disable local response normalization (LRN) block generation as a part of the convolution module of the deep learning processor configuration**
'off' (default) | 'on' | character vector

Use this parameter to control generation of the LRN block as a part of the convolution module of the deep learning processor configuration.

**`SegmentationBlockGeneration` — Enable or disable segmentation block generation as a part of the convolution module of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the segmentation block as a part of the convolution module of the deep learning processor configuration.

**`ConvThreadNumber` — Number of parallel convolution processor kernel threads**
16 (default) | 4 | 9 | 16 | 25 | 36 | 64 | 256 | unsigned integer

This parameter is the number of parallel 3-by-3 convolution kernel threads that are a part of the `conv` module within the `dlhdl.ProcessorConfig` object.

**`InputMemorySize` — Cache block RAM (BRAM) sizes**
[227 227 3] (default) | 3D positive integer array

This parameter is a 3D matrix representing input image size limited by the `conv` module BRAM size within the `dlhdl.ProcessorConfig` object.

**`OutputMemorySize` — Cache block RAM (BRAM) sizes**
[227 227 3] (default) | 3D positive integer array

This parameter is a 3D matrix representing output image size limited by the `conv` module BRAM size within the `dlhdl.ProcessorConfig` object.

**`FeatureSizeLimit` — Maximum input and output feature size**
2048 (default) | positive integer

This parameter is a positive integer representing the maximum input and output feature size as a part of the `conv` module within the `dlhdl.ProcessorConfig` object.

#### Processing Module fc Properties

**`ModuleGeneration` — Enable or disable fully connected module generation as a part of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the fully connected module as a part of the deep learning processor configuration.

**`SoftmaxBlockGeneration` — Enable or disable Softmax block generation as a part of the fully connected module of the deep learning processor configuration**
'off' (default) | 'on' | character vector

Use this parameter to control generation of the Softmax block as a part of the fully connected module of the deep learning processor configuration. When you set this property to `off`, the Softmax layer is still implemented in software.

**`SigmoidBlockGeneration` — Enable or disable sigmoid block generation as a part of the fully connected module of the deep learning processor configuration**
'off' (default) | 'on' | character vector

Use this parameter to control generation of the sigmoid block as a part of the fully connected module of the deep learning processor configuration.

**`FCThreadNumber` — Number of parallel fully connected (fc) MAC threads**
4 (default) | 4 | 8 | 16 | 32 | 64 | unsigned integer

This parameter is the number of parallel fc MAC threads that are a part of the `fc` module within the `dlhdl.ProcessorConfig` object.

**`InputMemorySize` — Cache block RAM (BRAM) sizes**
25088 (default) | unsigned integer

This parameter is an unsigned integer representing cache BRAM size limited by the `fc` module BRAM size within the `dlhdl.ProcessorConfig` object.

**`OutputMemorySize` — Cache block RAM (BRAM) sizes**
4096 (default) | unsigned integer

This parameter is an unsigned integer representing cache BRAM size limited by the `fc` module BRAM size within the `dlhdl.ProcessorConfig` object.

#### Processing Module custom Properties

**`ModuleGeneration` — Enable or disable custom module generation as a part of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the adder module as a part of the deep learning processor configuration.

**`Addition` — Enable or disable addition layer generation as a part of the custom module of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the addition layer as a part of the custom module of the deep learning processor configuration.

**`Multiplication` — Enable or disable multiplication layer generation as a part of the custom module of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the multiplication layer as a part of the custom module of the deep learning processor configuration.

**`InputMemorySize` — Cache block RAM (BRAM) sizes**
40 (default) | unsigned integer

This parameter is an unsigned integer representing cache BRAM size limited by the `adder` module BRAM size within the `dlhdl.ProcessorConfig` object.

**`OutputMemorySize` — Cache block RAM (BRAM) sizes**
40 (default) | unsigned integer

This parameter is an unsigned integer representing cache BRAM size limited by the `adder` module BRAM size within the `dlhdl.ProcessorConfig` object.

## Methods

**Public Methods**

| | |
|---|---|
| getModuleProperty | Use the getModuleProperty method to get values of module properties within the dlhdl.ProcessorConfig object |
| setModuleProperty | Use the setModuleProperty method to set properties of modules within the dlhdl.ProcessorConfig object |
| estimatePerformance | Retrieve layer-level latencies and performance by using estimatePerformance method |
| estimateResources | Return estimated resources used by custom bitstream configuration |
| optimizeConfigurationForNetwork | Retrieve optimized network-specific deep learning processor configuration |
| openCustomLayerModel | Open a generated custom layer verification model to verify your custom layers |
| registerCustomLayer | Register the custom layer definition and Simulink model representation of the custom layer |
| verifyCustomLayerModel | Verify the functionality and accuracy of the custom layer by using the generated custom layer verification model |

## Examples

### Create a `ProcessorConfig` Object

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

```
hPC = dlhdl.ProcessorConfig
```

The result is:

```
hPC =

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

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

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

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

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

#### Modify Properties of ProcessorConfig Object

Modify the TargetPlatform, SynthesisTool, and TargetFrequency properties of hPC.

```
hPC.TargetPlatform = 'Xilinx Zynq ZC706 evaluation kit';
>> hPC.SynthesisTool = 'Xilinx Vivado';
>> hPC.TargetFrequency = 180;
hPC
```

The result is:

```
hPC =

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

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

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

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

                     System Level Properties
                             TargetPlatform: 'Xilinx Zynq ZC706 evaluation kit'
                            TargetFrequency: 180
                              SynthesisTool: 'Xilinx Vivado'
```

```
            ReferenceDesign: 'AXI-Stream DDR Memory Access : 3-AXIM'
     SynthesisToolChipFamily: 'Zynq'
     SynthesisToolDeviceName: 'xc7z045'
    SynthesisToolPackageName: 'ffg900'
     SynthesisToolSpeedValue: '-2'
```

### Retrieve ProcessorConfig object for zcu102_single bitstream

Retrieve the `ProcessorConfig` object for the `zcu102_single` bitstream and store the object in `hPC`.

```
hPC = dlhdl.ProcessorConfig('Bitstream','zcu102_single')
```

The result is:

```
hPC =

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

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

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

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

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

### Set the ProcessorConfig object module data type to int8

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

```
hPC = dlhdl.ProcessorConfig
```

Modify the `ProcessorDataType` to `int8`.

```
hPC.ProcessorDataType = 'int8'
```

The result is:

```
hPC =

                    Processing Module "conv"
                            ModuleGeneration: 'on'
```

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

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

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

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

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

## See Also

**Functions**
`dlhdl.buildProcessor`

**Classes**
`dlhdl.Workflow`

**Topics**
"Custom Processor Configuration Workflow"
"Deep Learning Processor IP Core"
"Estimate Performance of Deep Learning Network"
"Estimate Resource Utilization for Custom Processor Configuration"

**Introduced in R2020b**

# estimatePerformance

**Class:** dlhdl.ProcessorConfig
**Package:** dlhdl

Retrieve layer-level latencies and performance by using estimatePerformance method

## Syntax

```
estimatePerformance(processorConfigObject, network)
performance = estimatePerformance(processorConfigObject,network)
performance = estimatePerformance(processorConfigObject,network,Name,Value)
```

## Description

estimatePerformance(processorConfigObject, network) returns the layer-level latencies and network performance for the object specified by the network argument.

performance = estimatePerformance(processorConfigObject,network) returns a table containing the network object layer-level latencies and performance.

performance = estimatePerformance(processorConfigObject,network,Name,Value) returns a table containing the network object layer-level latencies and performance, with one or more arguments specified by optional name-value pair arguments.

## Input Arguments

**processorConfigObject — Instance of processor configuration object**
dlhdl.ProcessorConfig object

Instance of the processor configuration object, specified as a dlhdl.ProcessorConfig object.

**network — Network object**
SeriesNetwork object | DAGNetwork object | yolov2ObjectDetector object | dlquantizer object

Name of network object for performance estimate.

Example: estimatePerformance(snet)

### Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* Name *in quotes.*

**FrameCount — Number of frames to consider for the calculation of performance estimation**
1 (default) | integer

Number of frames to consider for the calculation of performance estimation, specified as a positive number integer.

Example: 'FrameCount',10

## Output Arguments

### performance — Network object performance
table

Network object performance for the ProcessorConfig object, returned as a table.

## Examples

### Estimate Performance of LogoNet Network

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

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

2 Create a dlhdl.ProcessorConfig object.

```
snet = getLogoNetwork;
hPC = dlhdl.ProcessorConfig;
```

3 To retrieve the layer-level latencies and performance for the LogoNet network, call the estimatePerformance method.

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

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


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                   39426458                  0.19713                       1          39426458              5.1
    ____conv_1             6822215                  0.03411
    ____maxpool_1          3755088                  0.01878
    ____conv_2            10440701                  0.05220
    ____maxpool_2          1447840                  0.00724
    ____conv_3             9362677                  0.04681
    ____maxpool_3          1765856                  0.00883
    ____conv_4             1377268                  0.00689
    ____maxpool_4            28098                  0.00014
    ____fc_1               2644886                  0.01322
    ____fc_2               1692534                  0.00846
    ____fc_3                 89295                  0.00045
 * The clock frequency of the DL processor is: 200MHz
```

### Estimate Performance of ResNet-18 for Multiple Frame Inputs

Estimate the performance of the ResNet-18 network for mulltiple frames by using the dlhdl.ProcessorConfig object.

Load the ResNet-18 network and save it to *net*

```
net = resnet18;
```

Create a `dlhdl.ProcessorConfig` object and save to *hPC*

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

Retrieve layer level latencies and performance in frames per second (FPS) for multiple frames by using the `estimatePerformance` method with `FrameNumber` as an optional input argument.

```
hPC.estimatePerformance(net,'FrameCount',10);
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convolution2DLayer'
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in software.
### Notice: The layer 'ClassificationLayer_predictions' with type 'nnet.cnn.layer.ClassificationOutputLayer' is implemented in software.


              Deep Learning Processor Estimator Performance Results

                   LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Total Latency     Frames/s
                         -------------             -------------              ---------        ---------       ---------
Network                   21219873                  0.10610                      10           210125220            9.5
    ____conv1              2165372                  0.01083
    ____pool1               646226                  0.00323
    ____res2a_branch2a      966221                  0.00483
    ____res2a_branch2b      966221                  0.00483
    ____res2a               210750                  0.00105
    ____res2b_branch2a      966221                  0.00483
    ____res2b_branch2b      966221                  0.00483
    ____res2b               210750                  0.00105
    ____res3a_branch1       540749                  0.00270
    ____res3a_branch2a      708564                  0.00354
    ____res3a_branch2b      919117                  0.00460
    ____res3a               105404                  0.00053
    ____res3b_branch2a      919117                  0.00460
    ____res3b_branch2b      919117                  0.00460
    ____res3b               105404                  0.00053
    ____res4a_branch1       509261                  0.00255
    ____res4a_branch2a      509261                  0.00255
    ____res4a_branch2b      905421                  0.00453
    ____res4a                52724                  0.00026
    ____res4b_branch2a      905421                  0.00453
    ____res4b_branch2b      905421                  0.00453
    ____res4b                52724                  0.00026
    ____res5a_branch1       751693                  0.00376
    ____res5a_branch2a      751693                  0.00376
    ____res5a_branch2b     1415373                  0.00708
    ____res5a                26368                  0.00013
    ____res5b_branch2a     1415373                  0.00708
    ____res5b_branch2b     1415373                  0.00708
    ____res5b                26368                  0.00013
    ____pool5                54594                  0.00027
    ____fc1000              207351                  0.00104
 * The clock frequency of the DL processor is: 200MHz
```

## Tips

To obtain the performance estimation for a `dlquantizer` object, set the `dlhdl.ProcessorConfig` object `ProcessorDataType` to `int8`.

## See Also

`estimateResources` | `getModuleProperty` | `optimizeConfigurationForNetwork` | `setModuleProperty`

### Topics

"Estimate Performance of Deep Learning Network"

**Introduced in R2021a**

# estimateResources

**Class:** `dlhdl.ProcessorConfig`
**Package:** `dlhdl`

Return estimated resources used by custom bitstream configuration

## Syntax

```
estimateResources(processorConfigObject)
resources = estimateResources(processorConfigObject)
estimateResources(processorConfigObject,Name,Value)
resources = estimateResources(processorConfigObject,Name,Value)
```

## Description

`estimateResources(processorConfigObject)` returns the estimated resources used by the custom bitstream configuration.

`resources = estimateResources(processorConfigObject)` returns a table containing the estimated resources used by the custom bitstream configuration.

`estimateResources(processorConfigObject,Name,Value)` returns the estimated resources used by the custom bitstream configuration, with additional options specified by one or more name-value arguments.

`resources = estimateResources(processorConfigObject,Name,Value)` returns the estimated resources used by the custom bitstream configuration, with additional options specified by one or more name-value arguments.

## Input Arguments

**`processorConfigObject` — Instance of processor configuration object**
`dlhdl.ProcessorConfig` object

Instance of the processor configuration object, specified as a `dlhdl.ProcessorConfig` object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

**`IncludeReferenceDesign` — Display reference design resource utilization**
false (default) | false | true | logical

Flag to enable the display of reference design resource utilization specified as `alogical` data type.

Example: `'IncludeReferenceDesign',true`

## Output Arguments

**`resources` — Custom processor configuration object resource utilization**
table

Resources used by the custom bitstream configuration, returned as a table.

## Examples

### Estimate Resources Used by Default Custom Processor Configuration

1. Create a default custom processor configuration object. Use the `dlhdl.ProcessorConfig` class.

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

2. To retrieve the resources used by the custom processor configuration, call the `estimateResources` method.

```
hPC.estimateResources;
```

3. Calling `estimateResources` returns these results:

```
              Deep Learning Processor Estimator Resource Results

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

### Estimate Resource Utilization for Custom Board and Reference Design

Rapidly prototype the deployment of deep learning networks to your custom board by using the `estimateResources` function. Estimate the resource utilization of the deep learning processor configuration for your custom board. Optimize the integration of custom IP cores and reference design into your system by using the `estimateResources` function to estimate the resource utilization of your reference design. The synthesis tool that you use must be in the list of tools supported by the `SynthesisTool` property of the `dlhdl.ProcessorConfig` object. For a list of supported tools and device families, see “SynthesisTool” on page 1-0 and “SynthesisToolChipFamily” on page 1-0 .

In this example, estimate the resource utilization for your custom board that has the Kintex® Ultrascale+™ chip family. Also estimate the resource utilization of the reference design for the Xilinx® Zynq® Ultrascale+™ MPSoC ZCU102 board.

#### Prerequisites

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

#### Estimate Resource Utilization for Kintex® Ultrascale+™ Board

To estimate the resource utilization for your custom board that has a Kintex® Ultrascale+™ chip family , use the `estimateResource` function of the `dlhdl.ProcessorConfig` object.

1. Add the `dlhdl_device_registration.m` file to the MATLAB® search path.
2. Create a `dlhdl.ProcessorConfig` object.
3. Update the `SynthesisToolChipFamily` and `SynthesisToolDeviceName` properties of the `dlhdl.ProcessorConfig` object.
4. Use the `estimateResources` function to retrieve the resource utilization for your custom board.

Deep Learning HDL Toolbox™ does not support lookup table (LUT) estimation for custom boards.

```
hPC = dlhdl.ProcessorConfig;
hPC.SynthesisToolChipFamily = 'Kintex Ultrascale+';
hPC.SynthesisToolDeviceName = 'xcku11p-ffva1156-1-e';
hPC.estimateResources
```

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

              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2928             600            298560
                        -------------    -------------    -------------
DL_Processor                381( 14%)        508( 85%)            0(  0%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

#### Estimate Resource Utilization for Custom Reference Design

Estimate the resource utilization for a reference design that you want to integrate into your system that has a Xilinx® Zynq® Ultrascale+™ MPSoC ZCU102 board. Use the `estimateResource` function with the `IncludeReferenceDesign` name-value argument. The `estimateResources` function uses the `ResourcesUsed.LogicElements`, `ResourcesUsed.DSP`, and `ResourcesUsed.RAM` information in the reference design plugin file to perform the resource estimation. To estimate resource utilization for your custom reference design, you must populate your reference design file with values for `ResourcesUsed.LogicElements`, `ResourcesUsed.DSP`, and `ResourcesUsed.RAM`. See “ResourcesUsed” on page 1-0 . The reference design used in this code is located at `$supportpackageinstallationfolder/Xilinx/boards/+DLZCU102/+matlab_libiio_3axi4_master_2019_1/plugin_rd.m`.

```
hPC_referencedesign = dlhdl.ProcessorConfig;
hPC_referencedesign.estimateResources('IncludeReferenceDesign',true)
```

```
              Deep Learning Processor Estimator Resource Results

                             DSPs          Block RAM*     LUTs(CLB/ALUT)
                        -------------    -------------    -------------
Available                    2520             912            274080
                        -------------    -------------    -------------
Total                       384( 16%)        586( 65%)       255074( 94%)
ReferenceDesign               3(  1%)         78(  9%)        35000( 13%)
DL_Processor                381( 16%)        508( 56%)       220074( 81%)
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

The `estimateResources` function returns the resource utilization for the reference design and for the deep learning processor configuration.

### Supporting Files

#### Device Registration File

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

```
type dlhdl_device_registration.m
```

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

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

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

## See Also

`estimatePerformance` | `getModuleProperty` | `optimizeConfigurationForNetwork` | `setModuleProperty`

### Topics

"Estimate Resource Utilization for Custom Processor Configuration"

**Introduced in R2021a**

# getModuleProperty

**Class:** dlhdl.ProcessorConfig
**Package:** dlhdl

Use the getModuleProperty method to get values of module properties within the dlhdl.ProcessorConfig object

## Syntax

```
getModuleProperty(processorConfigObject,ModuleName,ModulePropertyName)
```

## Description

The getModuleProperty(processorConfigObject,ModuleName,ModulePropertyName) method returns the value of the module property for modules within the dlhdl.ProcessorConfig object.

## Input Arguments

**processorConfigObject — Instance of processor configuration object**
dlhdl.ProcessorConfig object

Instance of the processor configuration object, specified as a dlhdl.ProcessorConfig object.

**ModuleName — Name of the module whose parameters are to be retrieved**
"conv" | "fc" | "custom" | 'conv' | 'fc' | 'custom' | character vector | string

The dlhdl.ProcessorConfig object module name, specified as a character vector or string.

**ModulePropertyName — Name of the module property whose value is to be retrieved**
character vector or string

'conv', 'fc', or 'custom' module properties specified as character vector.

Example: "ConvThreadNumber" or 'ConvThreadNumber'

This table lists module names and module property names.

| Module Name | Module Property Name |
| --- | --- |
| conv | “ModuleGeneration” on page 1-0 |
| conv | “LRNBlockGeneration” on page 1-0 |
| conv | “SegmentationBlockGeneration” on page 1-0 |
| conv | “ConvThreadNumber” on page 1-0 |
| conv | “InputMemorySize” on page 1-0 |
| conv | “OutputMemorySize” on page 1-0 |
| conv | “FeatureSizeLimit” on page 1-0 |

| | |
|---|---|
| fc | “ModuleGeneration” on page 1-0 |
| fc | “SoftmaxBlockGeneration” on page 1-0 |
| fc | “SigmoidBlockGeneration” on page 1-0 |
| fc | “FCThreadNumber” on page 1-0 |
| fc | “InputMemorySize” on page 1-0 |
| fc | “OutputMemorySize” on page 1-0 |
| custom | “ModuleGeneration” on page 1-0 |
| custom | “Addition” on page 1-0 |
| custom | “Multiplication” on page 1-0 |
| custom | “InputMemorySize” on page 1-0 |
| custom | “OutputMemorySize” on page 1-0 |

## Examples

### Retrieve ConvThreadNumber for conv Module Inside dlhdl.ProcessorConfig Object

1 Create an example object by using the `dlhdl.ProcessorConfig` class, and then use the `getModuleProperty` method to obtain the `ConvThreadNumber`.

```
hPC = dlhdl.ProcessorConfig;
hPC.getModuleProperty("conv","ConvThreadNumber")
```

2 Once you execute the code, the result is:

```
ans =

    16
```

### Retrieve InputMemorySize for fc Module Inside dlhdl.ProcessorConfig Object

1 Create an example object by using the `dlhdl.ProcessorConfig` class, and then use the `getModuleProperty` method to obtain the `ConvThreadNumber`.

```
hPC = dlhdl.ProcessorConfig;
hPC.getModuleProperty("fc","InputMemorySize")
```

2 Once you execute the code, the result is:

```
ans =

       25088
```

## See Also

`estimatePerformance` | `estimateResources` | `optimizeConfigurationForNetwork` | `setModuleProperty`

**Topics**

“Deep Learning Processor Architecture”
“Estimate Performance of Deep Learning Network”

“Estimate Resource Utilization for Custom Processor Configuration”

**Introduced in R2020b**

# openCustomLayerModel

**Class:** `dlhdl.ProcessorConfig`
**Package:** `dlhdl`

Open a generated custom layer verification model to verify your custom layers

## Syntax

```
openCustomLayerModel(processorConfigObject)
openCustomLayerModel(processorConfigObject,Name = Value)
```

## Description

`openCustomLayerModel(processorConfigObject)` opens a generated custom layer verification model to verify your custom layers.

`openCustomLayerModel(processorConfigObject,Name = Value)` opens a generated custom layer verification model to verify your custom layers, with one or more arguments specified by optional name-value arguments.

## Input Arguments

**`processorConfigObject` — Instance of processor configuration object**
`dlhdl.ProcessorConfig` object

Instance of the processor configuration object, specified as a `dlhdl.ProcessorConfig` object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

Example:

**`Network` — Network object**
`SeriesNetwork` (default) | `SeriesNetwork` | `DAGNetwork` object | `dlquantizer` object

Name of the network object that contains your registered custom layer. If you do not include the network argument, Deep Learning HDL Toolbox generates a custom network that includes your registered custom layer.

Example: Network = net

**`InputImages` — Input image**
*m*-by-*n*-by-*k* numeric array

Input image, specified as a m-by-n-by-k numeric array. m, n, and k must match the dimensions of the deep learning network input image layer. For example, for the ResNet-18 network, resize the input images to a 224-by-224-by-3 array.

Example: InputImages = image

Data Types: `single`

## Examples

### Generate and Open Model for Custom Signum Layer

1. Before you can generate a test bench model, you must create and register a signum layer. For instructions, see "Register Custom Signum Layer" on page 1-60.
2. Generate a test bench model for your custom layer.

```
image = randi(255, [2,2,4]);
% To generate a test network that contains your custom layer do not specify
%  the Network name-value pair argument
openCustomLayerModel(hPC, InputImages = single(image));
% To test your actual network with custom layer specify the Network name-value
% pair argument. For example, if your custom network name is myNet, at the command line enter:
%   openCustomLayerModel(hPC, Network = myNet, InputImages = single(image));
```

## See Also

`estimatePerformance` | `getModuleProperty` | `optimizeConfigurationForNetwork` | `registerCustomLayer` | `setModuleProperty`

**Introduced in R2022a**

# optimizeConfigurationForNetwork

**Class:** `dlhdl.ProcessorConfig`
**Package:** `dlhdl`

Retrieve optimized network-specific deep learning processor configuration

## Syntax

`optimizeConfigurationForNetwork(processorConfigObject,network)`

## Description

`optimizeConfigurationForNetwork(processorConfigObject,network)` returns an optimized deep learning processor configuration for the object specified by the `network` argument.

## Input Arguments

**`processorConfigObject` — Instance of processor configuration object**
`dlhdl.ProcessorConfig` object

Instance of the processor configuration object, specified as a `dlhdl.ProcessorConfig` object.

**`network` — Network object**
`SeriesNetwork` object | `DAGNetwork` object | `yolov2ObjectDetector` object | `dlquantizer` object

Name of network object for optimized deep learning processor configuration, specified as a `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, or `dlquantizer` object.

Example: `optimizeConfigurationForNetwork(snet)`

## Examples

### Generate Optimized Processor Configuration for `MobileNetV2` Network

1. Create a `dlhdl.ProcessorConfig` object.

```
net = mobilenetv2;
hPC = dlhdl.ProcessorConfig;
```

2. To retrieve an optimized processor configuration, call the `optimizeConfigurationForNetwork` method.

```
hPC.optimizeConfigurationForNetwork(net)
```

```
### Optimizing processor configuration for deep learning network begin.
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.layer.Convolution2DLayer'
### Note: Processing module "conv" property "InputMemorySize" changed from "[227 227 3]" to "[224 224 3]".
### Note: Processing module "conv" property "OutputMemorySize" changed from "[227 227 3]" to "[112 112 32]".
### Note: Processing module "conv" property "FeatureSizeLimit" changed from "2048" to "1280".
### Note: Processing module "conv" property "LRNBlockGeneration" changed from "on" to "off" because there is no LRN layer in the de
### Note: Processing module "fc" property "InputMemorySize" changed from "25088" to "1280".
### Note: Processing module "fc" property "OutputMemorySize" changed from "4096" to "1000".
```

```
                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'off'
                            ConvThreadNumber: 16
                             InputMemorySize: [224 224 3]
                            OutputMemorySize: [112 112 32]
                            FeatureSizeLimit: 1280

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 1280
                            OutputMemorySize: 1000

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

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

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

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

## See Also

estimatePerformance | estimateResources | getModuleProperty | setModuleProperty

### Topics

“Generate Custom Bitstream to Meet Custom Deep Learning Network Requirements”

**Introduced in R2021b**

# registerCustomLayer

**Class:** dlhdl.ProcessorConfig
**Package:** dlhdl

Register the custom layer definition and Simulink model representation of the custom layer

## Syntax

registerCustomLayer(processorConfigObject, 'Layer', Layer, 'Model', Model)

## Description

registerCustomLayer(processorConfigObject, 'Layer', Layer, 'Model', Model) registers a custom layer specified by the Layer argument and the Simulink® model representation of the custom layer, specified by the Model argument.

## Input Arguments

**processorConfigObject — Instance of processor configuration object**
dlhdl.ProcessorConfig object

Instance of the processor configuration object, specified as a dlhdl.ProcessorConfig object.

**Layer — Instance of custom layer definition**
nnet.layer.Layer

Class definition of the custom layer object, specified as an nnet.layer.Layer object.

Example: Layer = hSig

**Model — Simulink model representing custom layer**
'' (default)

Simulink model representing the custom layer, specified as a file name of the Simulink model on the MATLAB® path or absolute or relative path to the Simulink model.

Example: Model = 'myfile.slx'

Example: Model = 'C:\myfolder\myfile.slx'

## Examples

### Register Custom Signum Layer

1 Create a function that represents the custom signum layer. Save the function definition as SignumLayer.m.

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

    properties
        testPropertyValue1 single = 3;
```

```
        testPropertyValue2 single = 4;
    end

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

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

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

        end
    end
end
```

2. Create a variable `hSig`. Assign the custom signum layer function definition to `hSig`.

   ```
   hSig = SignumLayer('sLayer');
   ```
3. Create a Simulink model that represents the custom signum layer. Save the Simulink model as `SignumLayer.slx`.
4. Create a custom deep learning processor configuration object by using the `dlhdl.ProcessorConfig` class. Save the custom deep learning processor configuration as `hPC`.

   ```
   hPC = dlhdl.ProcessorConfig;
   ```
5. Use the `registerCustomLayer` method to register the custom signum layer definition and Simulink model.

   ```
   % If the Simulink model is on the MATLAB path, use:
   hPC.registerCustomLayer(Layer = hSig, Model = 'SignumLayer.slx');
   % If the Simulink model is in a folder called myLayers on your C drive, use:
   %   hPC.registerCustomLayer( Layer = hSig, Model = 'C:\myLayers\SignumLayer.slx');
   ```

## See Also

`estimatePerformance` | `getModuleProperty` | `openCustomLayerModel` | `optimizeConfigurationForNetwork` | `setModuleProperty`

**Introduced in R2022a**

# setModuleProperty

**Class:** `dlhdl.ProcessorConfig`
**Package:** `dlhdl`

Use the `setModuleProperty` method to set properties of modules within the `dlhdl.ProcessorConfig` object

## Syntax

`setModuleProperty(processorConfigObject,ModuleName,Name,Value)`

## Description

The `setModuleProperty(processorConfigObject,ModuleName,Name,Value)`method sets the properties of the module mentioned in `ModuleName` by using the values specified as `Name,Value` pairs.

## Input Arguments

**`processorConfigObject` — Instance of processor configuration object**
`dlhdl.ProcessorConfig` object

Instance of the processor configuration object, specified as a `dlhdl.ProcessorConfig` object.

**`ModuleName` — Name of the module whose parameters are to be set**
"conv" | "fc" | "custom" | 'conv' | 'fc' | 'custom' | string | character vector

The `dlhdl.ProcessorConfig` object module name, specified as a character vector or string.

### Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

Example:

**conv module parameters**

**`ModuleGeneration` — Enable or disable convolution module generation as a part of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the convolution module as a part of the deep learning processor configuration.

Example: `'ModuleGeneration', 'on'`

**`LRNBlockGeneration` — Enable or disable local response normalization (LRN) block generation as a part of the convolution module of the deep learning processor configuration**
'off' (default) | 'on' | character vector

Use this parameter to control generation of the LRN block as a part of the convolution module of the deep learning processor configuration.

Example: `'LRNBlockGeneration', 'on'`

**`SegmentationBlockGeneration` — Enable or disable segmentation block generation as a part of the convolution module of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the segmentation block as a part of the convolution module of the deep learning processor configuration.

**`ConvThreadNumber` — Number of parallel convolution processor kernel threads**
16 (default) | 4 | 9 | 16 | 25 | 36 | 64 | 256 | unsigned integer

This parameter is the number of parallel 3-by-3 convolution kernel threads that are a part of the `conv` module within the `dlhdl.ProcessorConfig` object.

Example: `'ConvThreadNumber', 64`

**`InputMemorySize` — Cache block RAM (BRAM) sizes**
[227 227 3] (default) | 3D positive integer array

This parameter is a 3D matrix representing input image size limited by the `conv` module BRAM size within the `dlhdl.ProcessorConfig` object.

Example: `'InputMemorySize', [227 227 3]`

**`OutputMemorySize` — Cache block RAM (BRAM) sizes**
[227 227 3] (default) | 3D positive integer array

This parameter is a 3D matrix representing output image size limited by the `conv` module BRAM size within the `dlhdl.ProcessorConfig` object.

Example: `'OutputMemorySize', [227 227 3]`

**`FeatureSizeLimit` — Maximum input and output feature size**
2048 (default) | positive integer

This parameter is a positive integer representing the maximum input and output feature size as a part of the `conv` module within the `dlhdl.ProcessorConfig` object.

Example: `'FeatureSizeLimit', 512`

**fc module parameters**

**`ModuleGeneration` — Enable or disable fully connected module generation as a part of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the fully connected module as a part of the deep learning processor configuration.

Example: `'ModuleGeneration', 'on'`

**`SoftmaxBlockGeneration` — Enable or disable Softmax block generation as a part of the fully connected module of the deep learning processor configuration**
'off' (default) | 'on' | character vector

Use this parameter to control generation of the Softmax block as a part of the fully connected module of the deep learning processor configuration. When you set this property to `off`, the Softmax layer is still implemented in software.

Example: `'SoftmaxBlockGeneration', 'on'`

**`SigmoidBlockGeneration` — Enable or disable sigmoid block generation as a part of the fully connected module of the deep learning processor configuration**
'off' (default) | 'on' | character vector

Use this parameter to control generation of the sigmoid block as a part of the fully connected module of the deep learning processor configuration.

**`FCThreadNumber` — Number of parallel fully connected (fc) MAC threads**
4 (default) | 4 | 8 | 16 | 32 | 64 | unsigned integer

This parameter is the number of parallel fc MAC threads that are a part of the `fc` module within the `dlhdl.ProcessorConfig` object.

Example: `'FCThreadNumber', 16`

**`InputMemorySize` — Cache block RAM (BRAM) sizes**
25088 (default) | unsigned integer

This parameter is an unsigned integer representing cache BRAM size limited by the `fc` module BRAM size within the `dlhdl.ProcessorConfig` object.

Example: `'InputMemorySize', 9216`

**`OutputMemorySize` — Cache block RAM (BRAM) sizes**
4096 (default) | unsigned integer

This parameter is an unsigned integer representing cache BRAM size limited by the `fc` module BRAM size within the `dlhdl.ProcessorConfig` object.

Example: `'OutputMemorySize', 4096`

**custom module properties**

**`ModuleGeneration` — Enable or disable adder module generation as a part of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the adder module as a part of the deep learning processor configuration.

Example: `'ModuleGeneration', 'on'`

**`Addition` — Enable or disable addition layer generation as a part of the custom module of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the addition layer as a part of the custom module of the deep learning processor configuration.

**`Multiplication` — Enable or disable multiplication layer generation as a part of the custom module of the deep learning processor configuration**
'on' (default) | 'off' | character vector

Use this parameter to control generation of the multiplication layer as a part of the custom module of the deep learning processor configuration.

**`InputMemorySize` — Cache block RAM (BRAM) sizes**
40 (default) | unsigned integer

This parameter is an unsigned integer representing cache BRAM size limited by the `adder` module BRAM size within the `dlhdl.ProcessorConfig` object.

Example: `'InputMemorySize', 40`

**`OutputMemorySize` — Cache block RAM (BRAM) sizes**
40 (default) | unsigned integer

This parameter is an unsigned integer representing cache BRAM size limited by the `adder` module BRAM size within the `dlhdl.ProcessorConfig` object.

Example: `'OutputMemorySize', 40`

## Examples

### Set Value for `ConvThreadNumber` Within `dlhdl.ProcessorConfig` Object

**1** Create an example object by using the `dlhdl.ProcessorConfig` class, and then use the `setModuleProperty` method to set the value for `convThreadNumber`.

```
hPC = dlhdl.ProcessorConfig;
hPC.setModuleProperty("conv","ConvThreadNumber",25)
hPC
```

**2** Once you execute the code, the result is:

```
hPC =

                    Processing Module "conv"
                            ModuleGeneration: 'on'
                          LRNBlockGeneration: 'on'
                 SegmentationBlockGeneration: 'on'
                            ConvThreadNumber: 25
                             InputMemorySize: [227 227 3]
                            OutputMemorySize: [227 227 3]
                            FeatureSizeLimit: 2048

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

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

              Processor Top Level Properties
                              RunTimeControl: 'register'
```

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

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

### Set Value for `InputMemorySize` Within `dlhdl.ProcessorConfig` Object

1 Create an example object by using the `dlhdl.ProcessorConfig` class, and then use the `setModuleProperty` method to set the value for `InputMemorySize`.

```
hPC = dlhdl.ProcessorConfig;
hPC.setModuleProperty("fc","InputMemorySize",25060)
hPC
```

2 Once you execute the code, the result is:

```
hPC =

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

                      Processing Module "fc"
                            ModuleGeneration: 'on'
                      SoftmaxBlockGeneration: 'off'
                              FCThreadNumber: 4
                             InputMemorySize: 25060
                            OutputMemorySize: 4096

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

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

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

### Set Value for `InputMemorySize` Within `dlhdl.ProcessorConfig` Object

1 Create an example object by using the `dlhdl.ProcessorConfig` class, and then use the `setModuleProperty` method to set the value for `InputMemorySize`.

```
hPC = dlhdl.ProcessorConfig;
hPC.setModuleProperty("custom","InputMemorySize",80)
hPC
```

2 Once you execute the code, the result is:

```
hPC =

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

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

                   Processing Module "adder"
                            ModuleGeneration: 'on'
                             InputMemorySize: 80
                            OutputMemorySize: 40

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

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

### Turn off conv module Within dlhdl.ProcessorConfig Object

1 Create an example object by using the `dlhdl.ProcessorConfig` class, and then use the `setModuleProperty` method to set the value for `ModuleGeneration`.

```
hPC = dlhdl.ProcessorConfig;
hPC.setModuleProperty("conv","ModuleGeneration", "off")
hPC
```

2 Once you execute the code, the result is:

```
hPC =

                    Processing Module "conv"
                            ModuleGeneration: 'off'

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

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

              Processor Top Level Properties
```

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

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

## See Also

`estimatePerformance` | `estimateResources` | `getModuleProperty` | `optimizeConfigurationForNetwork`

### Topics

"Deep Learning Processor Architecture"
"Estimate Performance of Deep Learning Network"
"Estimate Resource Utilization for Custom Processor Configuration"

**Introduced in R2020b**

# verifyCustomLayerModel

**Class:** `dlhdl.ProcessorConfig`
**Package:** `dlhdl`

Verify the functionality and accuracy of the custom layer by using the generated custom layer verification model

## Syntax

```
verifyCustomLayerModel(processorConfigObject)
```

## Description

`verifyCustomLayerModel(processorConfigObject)` verifies the functionality of the custom layer and prediction accuracy of the network with the custom layer by using the custom layer verification model, which you generated by using the `openCustomLayerModel` method.

## Input Arguments

**`processorConfigObject` — Instance of processor configuration object**
`dlhdl.ProcessorConfig` object

Instance of the processor configuration object, specified as a `dlhdl.ProcessorConfig` object.

## Examples

### Verify Custom Signum Layer Model

1. Generate the verification model for your custom signum layer. For instructions, see “Generate and Open Model for Custom Signum Layer” on page 1-57. Leave the generated model file open.
2. Verify the generated verification model for the custom signum layer by using the `verifyCustomLayerModel` method.

```
verifyCustomLayerModel(hPC)
```

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

## See Also

`estimatePerformance` | `getModuleProperty` | `optimizeConfigurationForNetwork` | `openCustomLayerModel` | `registerCustomLayer` | `setModuleProperty`

**Introduced in R2022a**

# dlhdl.buildProcessor

Build and generate custom processor IP

## Syntax

```
dlhdl.buildProcessor
dlhdl.buildProcessor(processorconfigobject)
dlhdl.buildProcessor(processorconfigobject,Name,Value)
```

## Description

`dlhdl.buildProcessor` generates a bitstream for the default `dlhdl.ProcessorConfig` object.

`dlhdl.buildProcessor(processorconfigobject)`generates a bitstream for the `processorconfigobject` object.

`dlhdl.buildProcessor(processorconfigobject,Name,Value)` generates a bitstream for the `processorconfigobject` object, with additional options specified by one or more `Name,Value` arguments.

## Examples

### Generate Custom Bitstream for Custom Processor Configuration

Create a custom processor configuration. Generate a bistream for the custom processor configuration.

Create a `dlhdl.ProcessorConfig` object. Save the object in `hPC`.

```
hPC = dlhdl.ProcessorConfig
```

Generate a custom bitstream for `hPC`

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

### Generate Custom Bitstream and Specify Project Folder Name, Deep Learning Processor IP Core Name, and Target Code Generation language

Create a custom deep learning processor configuration. When you generate code specify the project folder name, deep learning processor IP core name, and target code generation language.

```
hPC = dlhdl.ProcessorConfig;
dlhdl.buildProcessor(hPC,'ProjectFolder','fconlyprocessor_prj',...
'ProcessorName','fconlyprocessor','HDLCoderConfig',{'TargetLanguage','Verilog'});
```

## Input Arguments

**`processorconfigobject` — Name of the object generated by using `dlhdl.buildProcessor`**
(default) | variable

Name of the custom processor configuration object, specified as a variable of type `dlhdl.Processorconfig`.

Example: `dlhdl.buildProcessor(hPC)`

#### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

Example:

**`ProjectFolder` — Project folder name**
'dlhdl_prj' (default)

Name of project folder where generated files are saved

Example: `'ProjectFolder', 'fconlyprocessor_prj'`

**`ProcessorName` — Name of generated deep learning processor IP core**
'' (default) | character vector or string

Name of generated deep learning processor IP core

Example: `'ProcessorName', 'fconlyprocessor'`

**`TargetLanguage` — Target language**
'VHDL' (default) | 'Verilog' | character vector or string

Specify whether to generate VHDL or Verilog code.

Example: `'HDLCoderConfig',{'TargetLanguage','Verilog'}`

## See Also

`dlhdl.ProcessorConfig`

#### Topics

"Deep Learning Processor IP Core"
"Generate Custom Bitstream"
"Generate Custom Processor IP"

**Introduced in R2020b**

# hdlsetuptoolpath

Set up system environment to access FPGA synthesis software

## Syntax

```
hdlsetuptoolpath('ToolName',TOOLNAME,'ToolPath',TOOLPATH)
```

## Description

`hdlsetuptoolpath('ToolName',TOOLNAME,'ToolPath',TOOLPATH)` adds a third-party FPGA synthesis tool to your system path. It sets up the system environment variables for the synthesis tool. To configure one or more supported third-party FPGA synthesis tools to use with HDL Coder™, use the `hdlsetuptoolpath` function.

Before opening the HDL Workflow Advisor, add the tool to your system path. If you already have the HDL Workflow Advisor open, see “Add Synthesis Tool for Current HDL Workflow Advisor Session” (HDL Coder).

## Examples

### Set Up Intel Quartus Prime

The following command sets the synthesis tool path to point to an installed Intel Quartus® Prime Standard Edition 18.1 executable file. You must have already installed Altera® Quartus II.

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

### Set Up Intel Quartus Pro

The following command sets the synthesis tool path to point to an installed Intel Quartus Pro 19.2 executable file. You must have already installed Intel Quartus Pro.

```
hdlsetuptoolpath('ToolName','Intel Quartus Pro','ToolPath',...
 'C:\intel\19.2_pro\quartus\bin64\qpro.exe');
```

**Note** An installation of Quartus Pro contains both `quartus.exe` and `qpro.exe` executable files. When both tools are added to the path by using `hdlsetuptoolpath`, HDL Coder checks the tool availability before running the HDL Workflow Advisor.

### Set Up Xilinx ISE

The following command sets the synthesis tool path to point to an installed Xilinx ISE 14.7 executable file. You must have already installed Xilinx ISE.

```
hdlsetuptoolpath('ToolName','Xilinx ISE','ToolPath',...
 'C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\ise.exe');
```

#### Set Up Xilinx Vivado

The following command sets the synthesis tool path to point to an installed Vivado® Design Suite 2019.1 batch file. You must have already installed Xilinx Vivado.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
 'C:\Xilinx\Vivado\2019.1\bin\vivado.bat');
```

#### Set Up Microsemi Libero SoC

The following command sets the synthesis tool path to point to an installed Microsemi® Libero® Design Suite batch file. You must have already installed Microsemi Libero SoC.

```
hdlsetuptoolpath('ToolName','Microsemi Libero SoC','ToolPath',...
 'C:\Microsemi\Libero_SoC_v11.8\Designer\bin\libero.exe');
```

## Input Arguments

#### `TOOLNAME` — Synthesis tool name
character vector

Synthesis tool name, specified as a character vector.

Example: `'Xilinx Vivado'`

#### `TOOLPATH` — Full path to the synthesis tool executable or batch file
character vector

Full path to the synthesis tool executable or batch file, specified as a character vector.

Example: `'C:\Xilinx\Vivado\2018.3\bin\vivado.bat'`

## Tips

- If you have an icon for the tool on your Windows® desktop, you can find the full path to the synthesis tool.

  1. Right-click the icon and select **Properties**.
  2. Click the **Shortcut** tab.
- The `hdlsetuptoolpath` function changes the system path and system environment variables for only the current MATLAB session. To execute `hdlsetuptoolpath` programmatically when MATLAB starts, add `hdlsetuptoolpath` to your `startup.m` script.

## See Also
`startup` | `setenv`

#### Topics
“HDL Language Support and Supported Third-Party Tools and Hardware” (HDL Coder)
“Tool Setup” (HDL Coder)
“Add Synthesis Tool for Current HDL Workflow Advisor Session” (HDL Coder)

**Introduced in R2011a**

# dlquantizer

Quantize a deep neural network to 8-bit scaled integer data types

## Description

Use the `dlquantizer` object to reduce the memory requirement of a deep neural network by quantizing weights, biases, and activations to 8-bit scaled integer data types.

## Creation

### Syntax

```
quantObj = dlquantizer(net)
quantObj = dlquantizer(net,Name,Value)
```

#### Description

`quantObj = dlquantizer(net)` creates a `dlquantizer` object for the specified network.

`quantObj = dlquantizer(net,Name,Value)` creates a `dlquantizer` object for the specified network, with additional options specified by one or more name-value pair arguments.

Use `dlquantizer` to create an quantized network for GPU, FPGA, or CPU deployment. To learn about the products required to quantize and deploy the deep learning network to a GPU, FPGA, or CPU environment, see “Quantization Workflow Prerequisites”.

#### Input Arguments

**`net` — Pretrained neural network**
`DAGNetwork` object | `SeriesNetwork` object | `yolov2ObjectDetector` object | `ssdObjectDetector` object

Pretrained neural network, specified as a `DAGNetwork`, `SeriesNetwork`, `yolov2ObjectDetector`, or a `ssdObjectDetector` object.

Quantization of `yolov2ObjectDetector` and `ssdObjectDetector` networks requires a GPU Coder™ license.

## Properties

**`NetworkObject` — Pretrained neural network**
`DAGNetwork` object | `SeriesNetwork` object | `yolov2ObjectDetector` object | `ssdObjectDetector` object

Pre-trained neural network, specified as a `DAGNetwork`, `SeriesNetwork`, `yolov2ObjectDetector`, or a `ssdObjectDetector` object.

**`ExecutionEnvironment` — Execution environment**
'GPU' (default) | 'FPGA' | 'CPU' | 'MATLAB'

Specify the execution environment for the quantized network. When this parameter is not specified the default execution environment is GPU. To learn about the products required to quantize and deploy the deep learning network to a GPU, FPGA, or CPU environment, see “Quantization Workflow Prerequisites”.

Example: `'ExecutionEnvironment','FPGA'`

**`Simulation` — Enable or disable MATLAB simulation workflow**
'off (default) | 'on'

Enable or disable the MATLAB simulation workflow. When this parameter is set to on, the quantized network is validated by simulating the quantized network in MATLAB and comparing the single data type network prediction results to the simulated network prediction results.

Example: `'Simulation', 'on'`

## Object Functions

| | |
|---|---|
| calibrate | Simulate and collect ranges of a deep neural network |
| validate | Quantize and validate a deep neural network |
| quantize | Create quantized deep neural network |

## Examples

### Quantize a Neural Network for GPU Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network for GPU and explore the behavior of the quantized network. In this example, you quantize the squeezenet neural network after retraining the network to classify new images according to the Train Deep Learning Network to Classify New Images example. In this example, the memory required for the network is reduced approximately 75% through quantization while the accuracy of the network is not affected.

Load the pretrained network. `net` is the output network of the Train Deep Learning Network to Classify New Images example.

```
load squeezenetmerch
net
```

```
net = 
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Define calibration and validation data to use for quantization.

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 `MerchData` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

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

```
quantObj = dlquantizer(net);
```

Define a metric function to use to compare the behavior of the network before and after quantization. This example uses the `hComputeModelAccuracy` metric function.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % 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
```

Specify the metric function in a `dlquantizationOptions` object.

```
quantOpts = dlquantizationOptions('MetricFcn',{@(x)hComputeModelAccuracy(x, net, aug_valData)});
```

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.

```
calResults = calibrate(quantObj, aug_calData)
```

```
calResults=121×5 table
        Optimized Layer Name         Network Layer Name     Learnables / Activations    MinValue
```

```
    ____________________________    ____________________    ________________________    ________
    {'conv1_Weights'           }    {'conv1'           }           "Weights"              -0.91985
    {'conv1_Bias'              }    {'conv1'           }           "Bias"                 -0.07925
    {'fire2-squeeze1x1_Weights'}    {'fire2-squeeze1x1'}           "Weights"                 -1.38
    {'fire2-squeeze1x1_Bias'   }    {'fire2-squeeze1x1'}           "Bias"                 -0.11641
    {'fire2-expand1x1_Weights' }    {'fire2-expand1x1' }           "Weights"               -0.7406
    {'fire2-expand1x1_Bias'    }    {'fire2-expand1x1' }           "Bias"                -0.060056
    {'fire2-expand3x3_Weights' }    {'fire2-expand3x3' }           "Weights"              -0.74397
    {'fire2-expand3x3_Bias'    }    {'fire2-expand3x3' }           "Bias"                -0.051778
    {'fire3-squeeze1x1_Weights'}    {'fire3-squeeze1x1'}           "Weights"               -0.7712
    {'fire3-squeeze1x1_Bias'   }    {'fire3-squeeze1x1'}           "Bias"                 -0.10138
    {'fire3-expand1x1_Weights' }    {'fire3-expand1x1' }           "Weights"              -0.72035
    {'fire3-expand1x1_Bias'    }    {'fire3-expand1x1' }           "Bias"                -0.067029
    {'fire3-expand3x3_Weights' }    {'fire3-expand3x3' }           "Weights"              -0.61443
    {'fire3-expand3x3_Bias'    }    {'fire3-expand3x3' }           "Bias"                -0.053613
    {'fire4-squeeze1x1_Weights'}    {'fire4-squeeze1x1'}           "Weights"               -0.7422
    {'fire4-squeeze1x1_Bias'   }    {'fire4-squeeze1x1'}           "Bias"                 -0.10885
      ⋮
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
valResults = validate(quantObj, aug_valData, quantOpts)
```

```
valResults = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: [2×2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

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

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

```
valResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}                2.9003e+06
     {'Quantized'     }                7.3393e+05
```

In this example, the memory required for the network was reduced approximately 75% through quantization. The accuracy of the network is not affected.

The weights, biases, and activations of the convolution layers of the network specified in the dlquantizer object now use scaled 8-bit integer data types.

### Quantize a Neural Network for FPGA Target

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.

To run this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

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
```

Load the pretrained network.

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

```
snet = 

  SeriesNetwork with properties:

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

Define calibration and validation data to use for quantization.

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.

This example uses 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 a `dlquantizer` object and specify the network to quantize.

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

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

% Copyright 2020 The MathWorks, Inc.

    % Load ground truth
    groundTruth = dataStore.Labels;

    % Compare predicted label with 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
```

Specify the metric function in a `dlquantizationOptions` object.

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

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
```

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

```
validateOut = 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
```

The weights, biases, and activations of the convolution layers of the network specified in the `dlquantizer` object now use scaled 8-bit integer data types.

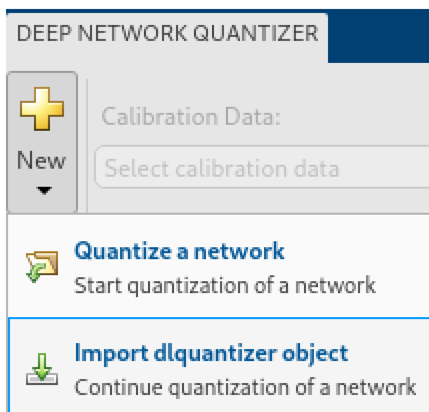
### Import a `dlquantizer` Object into the Deep Network Quantizer App

This example shows you how to import a `dlquantizer` object from the base workspace into the **Deep Network Quantizer** app. This allows you to begin quantization of a deep neural network using the command line or the app, and resume your work later in the app.

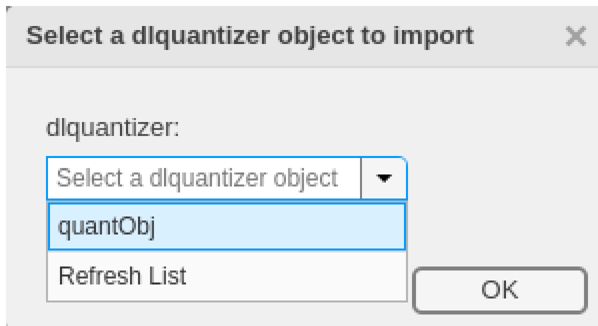
Open the **Deep Network Quantizer** app.

```
deepNetworkQuantizer
```

In the app, click **New** and select `Import dlquantizer object`.

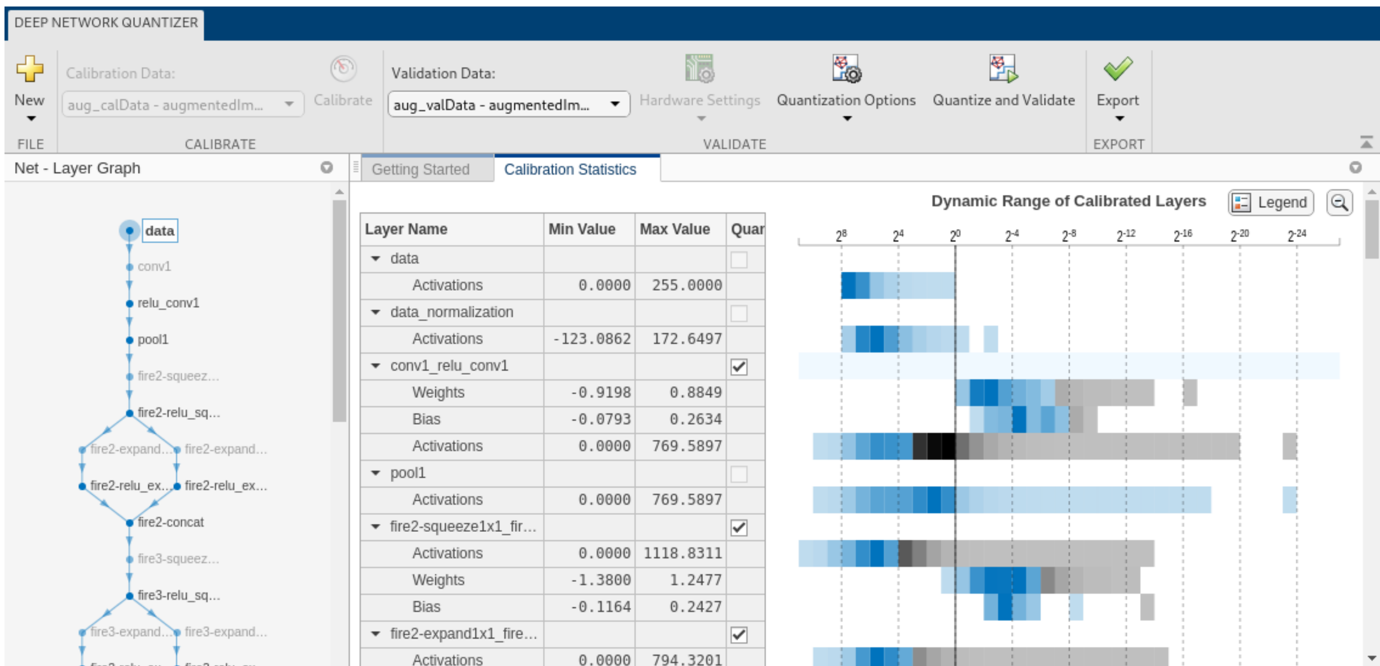


In the dialog, select the `dlquantizer` object to import from the base workspace. For this example, use `quantObj` that you create in the above example Quantize a Neural Network for GPU Target.




The app imports any data contained in the `dlquantizer` object that was collected at the command line. This data can include the network to quantize, calibration data, validation data, and calibration statistics.

The app displays a table containing the calibration data contained in the imported `dlquantizer` object, `quantObj`. To the right of the table, the app displays histograms of the dynamic ranges of the parameters. The gray regions of the histograms indicate data that cannot be represented by the quantized representation. For more information on how to interpret these histograms, see "Quantization of Deep Neural Networks".




## Quantize a Network for FPGA Deployment

To explore the behavior of a neural network that has quantized convolution layers, use the **Deep Network Quantizer** app. This example quantizes the learnable parameters of the convolution layers of the `LogoNet` neural network.

For this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

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

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

Load the pretrained network.

```
snet = getLogoNetwork;
```

```
snet = 

  SeriesNetwork with properties:

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

Define calibration and validation data to use for quantization.

The app uses calibration data to exercise the network and collect the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network. The app also exercises the dynamic ranges of the activations in all layers of the LogoNet network. For the best quantization results, the calibration data must be representative of inputs to the LogoNet network.

After quantization, the app uses the validation data set to test the network to understand the effects of the limited range and precision of the quantized learnable parameters of the convolution layers in the network.

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

Expedite the calibration and validation process by using a subset of the `calibrationData` and `validationData`. Store the new reduced calibration data set in `calibrationData_concise` and the new reduced validation data set in `validationData_concise`.

```
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');
calibrationData_concise = calibrationData.subset(1:20);
validationData_concise = validationData.subset(1:1);
```

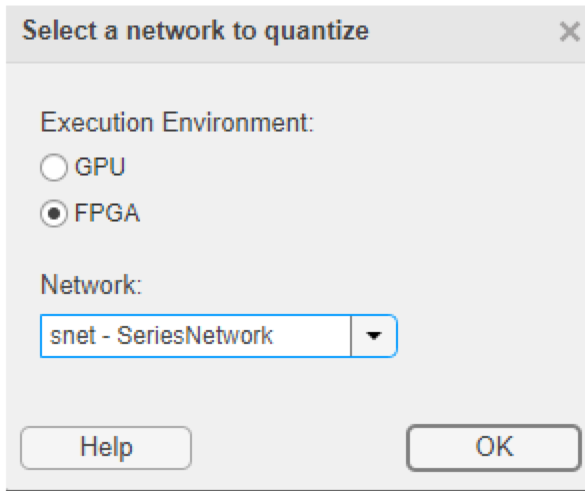
At the MATLAB command prompt, open the Deep Network Quantizer app.

```
deepNetworkQuantizer
```

Click **New** and select `Quantize a network`.

The app verifies your execution environment.

Select the execution environment and the network to quantize from the base workspace. For this example, select a FPGA execution environment and the series network `snet`.



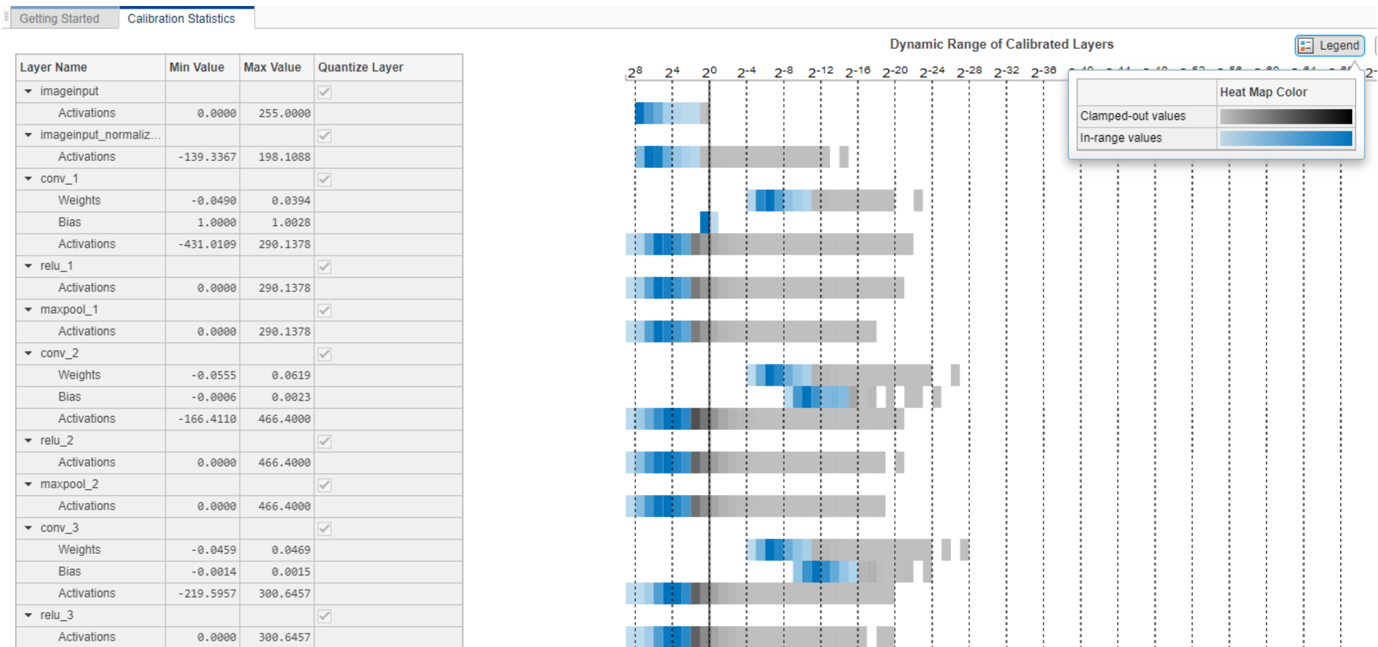

The app displays the layer graph of the selected network.

In the **Calibrate** section of the app toolstrip, under **Calibration Data**, select the `augmentedImageDatastore` object from the base workspace containing the calibration data `calibrationData_concise`.

Click **Calibrate**.

The **Deep Network Quantizer** app uses the calibration data to exercise the network and collect range information for the learnable parameters in the network layers.

When the calibration is complete, the app displays a table containing the weights and biases in the convolution and fully connected layers of the network. Also displayed are the dynamic ranges of the activations in all layers of the network and their minimum and maximum values during the calibration. The app displays histograms of the dynamic ranges of the parameters. The gray regions of the histograms indicate data that cannot be represented by the quantized representation. For more information on how to interpret these histograms, see “Quantization of Deep Neural Networks”.




In the **Quantize** column of the table, indicate whether to quantize the learnable parameters in the layer. You cannot quantize layers that are not convolution layers. Layers that are not quantized remain in single-precision.

In the **Validate** section of the app toolstrip, under **Validation Data**, select the `augmentedImageDatastore` object from the base workspace containing the validation data `validationData_concise`.

In the **Hardware Settings** section of the toolstrip, select from the options listed in the table:

| **Simulation Environment** | **Action** |
|---|---|
| MATLAB (Simulate in MATLAB) | Simulates the quantized network in MATLAB. Validates the quantized network by comparing performance to single-precision version of the network. |
| Intel Arria 10 SoC (arria10soc_int8) | Deploys the quantized network to an Intel Arria 10 SoC board by using the `arria10soc_int8` bitstream. Validates the quantized network by comparing performance to single-precision version of the network. |
| Xilinx ZCU102 (zcu102_int8) | Deploys the quantized network to a Xilinx Zynq UltraScale+ MPSoC ZCU102 10 SoC board by using the `zcu102_int8` bitstream. Validates the quantized network by comparing performance to single-precision version of the network. |

| | |
|---|---|
| Xilinx ZC706 (zc706_int8) | Deploys the quantized network to a Xilinx Zynq-7000 ZC706 board by using the `zc706_int8` bitstream. Validates the quantized network by comparing performance to single-precision version of the network. |

When you select the `Intel Arria 10 SoC (arria10soc_int8)`, `Xilinx ZCU102 (zcu102_int8)`, or `Xilinx ZC706 (zc706_int8)` options, select the interface to use to deploy and validate the quantized network. The **Target** interface options are listed in this table.

| Target Option | Action |
|---|---|
| JTAG | Programs the target FPGA board selected in **Simulation Environment** by using a JTAG cable. For more information, see “JTAG Connection” |
| Ethernet | Programs the target FPGA board selected in **Simulation Environment** through the Ethernet interface. Specify the IP address for your target board in **IP Address**. |

For this example, select `Xilinx ZCU102 (zcu102_int8)`, select **Ethernet**, and enter the board IP address.

In the **Validate** section of the app toolstrip, under **Quantization Options**, select the **Default** metric function.

Click **Quantize and Validate**.

The **Deep Network Quantizer** app quantizes the weights, activations, and biases of convolution layers in the network to scaled 8-bit integer data types and uses the validation data to exercise the network. The app determines a metric function to use for the validation based on the type of network that is being quantized.

| Type of Network | Metric Function |
| --- | --- |
| Classification | **Top-1 Accuracy** – Accuracy of the network |
| Object Detection | **Average Precision** – Average precision over all detection results. See `evaluateDetectionPrecision`. |
| Regression | **MSE** – Mean squared error of the network |
| Semantic Segmentation | `evaluateSemanticSegmentation` – Evaluate semantic segmentation data set against ground truth |
| Single Shot Detector (SSD) | **WeightedIOU** – Average IoU of each class, weighted by the number of pixels in that class |

When the validation is complete, the app displays the results of the validation, including:

- Metric function used for validation
- Result of the metric function before and after quantization

Validation Summary

Validation Results

**Number of samples: 1**

| Metric | Floating-Point Network Results | Quantized Network Results | Percent Change |
| --- | --- | --- | --- |
| FramesPerSecond | 5.5102 | 19.1158 | 246.9166 |
| Number of Threads (Convolution) | 16.0000 | 64.0000 | 300.0000 |
| Number of Threads (Fully Connected) | 4.0000 | 16.0000 | 300.0000 |
| LUT Utilization (%) | 93.5610 | 79.2440 | 15.3023 |
| BlockRAM Utilization (%) | 63.7061 | 49.6711 | 22.0310 |
| DSP Utilization (%) | 14.7222 | 30.5952 | 107.8167 |
| Top-1 Accuracy | 1.0000 | 1.0000 | 0.0000 |

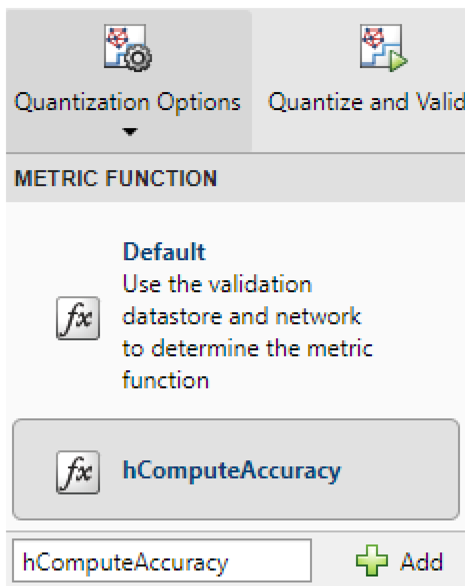
If you want to use a different metric function for validation, for example to use the Top-5 accuracy metric function instead of the default Top-1 accuracy metric function, you can define a custom metric function. Save this function in a local file.

```
function accuracy = hComputeAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % Compare predicted label with 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
```

To revalidate the network by using this custom metric function, under **Quantization Options**, enter the name of the custom metric function `hComputeAccuracy`. Select **Add** to add `hComputeAccuracy` to the list of metric functions available in the app. Select `hComputeAccuracy` as the metric function to use.

The custom metric function must be on the path. If the metric function is not on the path, this step produces an error.




Click **Quantize and Validate**.

The app quantizes the network and displays the validation results for the custom metric function.

Validation Summary

Validation Results

**Number of samples:** 1
**Metric function:** Custom metric function

| Metric | Floating-Point Network Results | Quantized Network Results | Percent Change |
|---|---|---|---|
| hComputeAccuracy | 1.0000 | 1.0000 | 0.0000 |

The app displays only scalar values in the validation results table. To view the validation results for a custom metric function with nonscalar output, export the `dlquantizer` object, then validate the quantized network by using the `validate` function in the MATLAB command window.

After quantizing and validating the network, you can choose to export the quantized network.

Click the **Export** button. In the drop-down list, select `Export Quantizer` to create a `dlquantizer` object in the base workspace. You can deploy the quantized network to your target FPGA board and retrieve the prediction results by using MATLAB. See, “Deploy Quantized Network Example”.

## See Also

**Apps**
**Deep Network Quantizer**

**Functions**
`calibrate` | `validate` | `dlquantizationOptions`

**Topics**
“Quantization of Deep Neural Networks”
“Quantize Residual Network Trained for Image Classification and Generate CUDA Code”
“Quantize Object Detectors and Generate CUDA Code”
“Quantize Network for FPGA Deployment”
“Deploy Quantized Network Example”
“Classify Images on an FPGA Using a Quantized DAG Network”
“Code Generation for Quantized Deep Learning Network on Raspberry Pi” (MATLAB Coder)

**Introduced in R2020a**

# dlquantizationOptions

Options for quantizing a trained deep neural network

## Description

The `dlquantizationOptions` object provides options for quantizing a trained deep neural network to scaled 8-bit integer data types. Use the `dlquantizationOptions` object to define the metric function to use that compares the accuracy of the network before and after quantization.

To learn about the products required to quantize a deep neural network, see “Quantization Workflow Prerequisites”.

## Creation

### Syntax

```
quantOpts = dlquantizationOptions
quantOpts = dlquantizationOptions(Name,Value)
```

#### Description

`quantOpts = dlquantizationOptions` creates a `dlquantizationOptions` object with default property values.

`quantOpts = dlquantizationOptions(Name,Value)` creates a `dlquantizationOptions` object with additional properties specified by one or more name-value pair arguments.

### Properties

**`MetricFcn` — Function to use for calculating validation metrics**
cell array of function handles

Cell array of function handles specifying the functions for calculating validation metrics of quantized network.

Example: `options = dlquantizationOptions('MetricFcn', {@(x)hComputeModelAccuracy(x, net, groundTruth)});`

Data Types: `cell`

#### FPGA Execution Environment Options

**`Bitstream` — Bitstream name**
'zcu102_int8' | 'zc706_int8' | 'arria10soc_int8'

*This property affects FPGA targeting only.*

Name of the FPGA bitstream specified as a character vector.

Example: 'Bitstream', 'zcu102_int8'

**Target — Name of the dlhdl.Target object**
hT

*This property affects FPGA targeting only.*

Name of the dlhdl.Target object that has the board name and board interface information.

Example: 'Target', hT

## Examples

### Quantize a Neural Network for GPU Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network for GPU and explore the behavior of the quantized network. In this example, you quantize the squeezenet neural network after retraining the network to classify new images according to the Train Deep Learning Network to Classify New Images example. In this example, the memory required for the network is reduced approximately 75% through quantization while the accuracy of the network is not affected.

Load the pretrained network. net is the output network of the Train Deep Learning Network to Classify New Images example.

```
load squeezenetmerch
net
```

```
net = 
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Define calibration and validation data to use for quantization.

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 MerchData data set. Define an augmentedImageDatastore object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
```

```
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

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

```
quantObj = dlquantizer(net);
```

Define a metric function to use to compare the behavior of the network before and after quantization. This example uses the `hComputeModelAccuracy` metric function.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % 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
```

Specify the metric function in a `dlquantizationOptions` object.

```
quantOpts = dlquantizationOptions('MetricFcn',{@(x)hComputeModelAccuracy(x, net, aug_valData)});
```

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.

```
calResults = calibrate(quantObj, aug_calData)
```

```
calResults=121×5 table
        Optimized Layer Name           Network Layer Name      Learnables / Activations    MinValue
    ____________________________    ____________________    ________________________    _________

    {'conv1_Weights'           }    {'conv1'           }           "Weights"              -0.91985
    {'conv1_Bias'              }    {'conv1'           }           "Bias"                 -0.07925
    {'fire2-squeeze1x1_Weights'}    {'fire2-squeeze1x1'}           "Weights"                 -1.38
    {'fire2-squeeze1x1_Bias'   }    {'fire2-squeeze1x1'}           "Bias"                 -0.11641
    {'fire2-expand1x1_Weights' }    {'fire2-expand1x1' }           "Weights"               -0.7406
    {'fire2-expand1x1_Bias'    }    {'fire2-expand1x1' }           "Bias"                -0.060056
    {'fire2-expand3x3_Weights' }    {'fire2-expand3x3' }           "Weights"              -0.74397
    {'fire2-expand3x3_Bias'    }    {'fire2-expand3x3' }           "Bias"                -0.051778
    {'fire3-squeeze1x1_Weights'}    {'fire3-squeeze1x1'}           "Weights"               -0.7712
    {'fire3-squeeze1x1_Bias'   }    {'fire3-squeeze1x1'}           "Bias"                 -0.10138
    {'fire3-expand1x1_Weights' }    {'fire3-expand1x1' }           "Weights"              -0.72035
    {'fire3-expand1x1_Bias'    }    {'fire3-expand1x1' }           "Bias"                -0.067029
```

```
    {'fire3-expand3x3_Weights' }    {'fire3-expand3x3' }          "Weights"            -0.6144
    {'fire3-expand3x3_Bias'    }    {'fire3-expand3x3' }          "Bias"             -0.05361
    {'fire4-squeeze1x1_Weights'}    {'fire4-squeeze1x1'}          "Weights"             -0.742
    {'fire4-squeeze1x1_Bias'   }    {'fire4-squeeze1x1'}          "Bias"               -0.1088
      ⋮
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
valResults = validate(quantObj, aug_valData, quantOpts)
```

```
valResults = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: [2×2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

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

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

```
valResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}                2.9003e+06
     {'Quantized'     }                7.3393e+05
```

In this example, the memory required for the network was reduced approximately 75% through quantization. The accuracy of the network is not affected.

The weights, biases, and activations of the convolution layers of the network specified in the dlquantizer object now use scaled 8-bit integer data types.

### Quantize a Neural Network for FPGA Target

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.

To run this example, you need the products listed under FPGA in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

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
```

Load the pretrained network.

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

```
snet = 

  SeriesNetwork with properties:

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

Define calibration and validation data to use for quantization.

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.

This example uses 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 a dlquantizer object and specify the network to quantize.

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

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

% Copyright 2020 The MathWorks, Inc.

    % Load ground truth
    groundTruth = dataStore.Labels;

    % Compare predicted label with 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
```

Specify the metric function in a `dlquantizationOptions` object.

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

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
```

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

```
validateOut = 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
```

The weights, biases, and activations of the convolution layers of the network specified in the dlquantizer object now use scaled 8-bit integer data types.

## See Also

**Apps**
**Deep Network Quantizer**

**Functions**
calibrate | validate | dlquantizer

**Topics**
“Quantization of Deep Neural Networks”

**Introduced in R2020a**

# calibrate

Simulate and collect ranges of a deep neural network

## Syntax

```
calibrationResults = calibrate(quantObj, calData)
calibrationResults = calibrate(quantObj, calData,Name,Value)
```

## Description

`calibrationResults = calibrate(quantObj, calData)` 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 specified by `dlquantizer` object, `quantObj`, using the data specified by `calData`.

`calibrationResults = calibrate(quantObj, calData,Name,Value)` 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 specified by `dlquantizer` object, `quantObj`, using the data specified by `calData`, with additional arguments specified by one or more name-value pair arguments.

To learn about the products required to quantize a deep neural network, see “Quantization Workflow Prerequisites”

## Examples

### Quantize a Neural Network for GPU Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network for GPU and explore the behavior of the quantized network. In this example, you quantize the squeezenet neural network after retraining the network to classify new images according to the Train Deep Learning Network to Classify New Images example. In this example, the memory required for the network is reduced approximately 75% through quantization while the accuracy of the network is not affected.

Load the pretrained network. `net` is the output network of the Train Deep Learning Network to Classify New Images example.

```
load squeezenetmerch
net
```

```
net = 
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Define calibration and validation data to use for quantization.

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 `MerchData` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

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

```
quantObj = dlquantizer(net);
```

Define a metric function to use to compare the behavior of the network before and after quantization. This example uses the `hComputeModelAccuracy` metric function.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % 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
```

Specify the metric function in a `dlquantizationOptions` object.

```
quantOpts = dlquantizationOptions('MetricFcn',{@(x)hComputeModelAccuracy(x, net, aug_valData)});
```

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.

```
calResults = calibrate(quantObj, aug_calData)
```

```
calResults=121×5 table
          Optimized Layer Name           Network Layer Name     Learnables / Activations    MinValue
    ____________________________    ____________________    ________________________    _________

    {'conv1_Weights'           }    {'conv1'           }           "Weights"             -0.91985
    {'conv1_Bias'              }    {'conv1'           }           "Bias"                -0.07925
    {'fire2-squeeze1x1_Weights'}    {'fire2-squeeze1x1'}           "Weights"                -1.38
    {'fire2-squeeze1x1_Bias'   }    {'fire2-squeeze1x1'}           "Bias"                -0.11641
    {'fire2-expand1x1_Weights' }    {'fire2-expand1x1' }           "Weights"              -0.7406
    {'fire2-expand1x1_Bias'    }    {'fire2-expand1x1' }           "Bias"               -0.060056
    {'fire2-expand3x3_Weights' }    {'fire2-expand3x3' }           "Weights"             -0.74397
    {'fire2-expand3x3_Bias'    }    {'fire2-expand3x3' }           "Bias"               -0.051778
    {'fire3-squeeze1x1_Weights'}    {'fire3-squeeze1x1'}           "Weights"              -0.7712
    {'fire3-squeeze1x1_Bias'   }    {'fire3-squeeze1x1'}           "Bias"                -0.10138
    {'fire3-expand1x1_Weights' }    {'fire3-expand1x1' }           "Weights"             -0.72035
    {'fire3-expand1x1_Bias'    }    {'fire3-expand1x1' }           "Bias"               -0.067029
    {'fire3-expand3x3_Weights' }    {'fire3-expand3x3' }           "Weights"             -0.61443
    {'fire3-expand3x3_Bias'    }    {'fire3-expand3x3' }           "Bias"               -0.053613
    {'fire4-squeeze1x1_Weights'}    {'fire4-squeeze1x1'}           "Weights"              -0.7422
    {'fire4-squeeze1x1_Bias'   }    {'fire4-squeeze1x1'}           "Bias"                -0.10885
      ⋮
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
valResults = validate(quantObj, aug_valData, quantOpts)
```

```
valResults = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: [2×2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

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

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

```
valResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}                2.9003e+06
     {'Quantized'     }                7.3393e+05
```

In this example, the memory required for the network was reduced approximately 75% through quantization. The accuracy of the network is not affected.

The weights, biases, and activations of the convolution layers of the network specified in the dlquantizer object now use scaled 8-bit integer data types.

### Quantize a Neural Network for FPGA Target

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.

To run this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

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
```

Load the pretrained network.

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

```
snet = 

  SeriesNetwork with properties:

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

Define calibration and validation data to use for quantization.

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.

This example uses 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 a `dlquantizer` object and specify the network to quantize.

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

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

% Copyright 2020 The MathWorks, Inc.

    % Load ground truth
    groundTruth = dataStore.Labels;

    % Compare predicted label with 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
```

Specify the metric function in a `dlquantizationOptions` object.

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

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
```

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

```
validateOut = 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
```

The weights, biases, and activations of the convolution layers of the network specified in the `dlquantizer` object now use scaled 8-bit integer data types.

## Input Arguments

**`quantObj` — Network to quantize**
`dlquantizer` object

`dlquantizer` object containing the network to quantize.

**`calData` — Data to use for calibration of quantized network**
`imageDatastore` object | `augmentedImageDatastore` object | `pixelLabelImageDatastore` object

Data to use for calibration of quantized network, specified as an `imageDatastore` object, an `augmentedImageDatastore` object, or a `pixelLabelImageDatastore` object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose* `Name` *in quotes.*

Example: `calResults = calibrate(quantObj, calData, 'UseGPU', 'on')`

### `UseGPU` — Logical flag to use GPU for calibration

'off' (default) | 'on'

Logical flag to use a GPU for calibration when the `dlquantizer` object `ExecutionEnvironment` is set to `'FPGA'` or `'CPU'`.

Example: `'UseGPU', 'on'`

## Output Arguments

### `calibrationResults` — Dynamic ranges of network

table

Dynamic ranges of layers of the network, returned as a table. Each row in the table displays the minimum and maximum values of a learnable parameter of a convolution layer of the optimized network. The software uses these minimum and maximum values to determine the scaling for the data type of the quantized parameter.

## See Also

**Apps**
**Deep Network Quantizer**

**Functions**
`validate` | `dlquantizer` | `dlquantizationOptions`

**Topics**
"Quantization of Deep Neural Networks"

**Introduced in R2020a**

# validate

Quantize and validate a deep neural network

## Syntax

```
validationResults = validate(quantObj, valData)
validationResults = validate(quantObj, valData, quantOpts)
```

## Description

`validationResults = validate(quantObj, valData)` quantizes the weights, biases, and activations in the convolution layers of the network, and validates the network specified by `dlquantizer` object, `quantObj` and using the data specified by `valData`.

`validationResults = validate(quantObj, valData, quantOpts)` quantizes the weights, biases, and activations in the convolution layers of the network, and validates the network specified by `dlquantizer` object, `quantObj`, using the data specified by `valData`, and the optional argument `quantOpts` that specifies a metric function to evaluate the performance of the quantized network.

To learn about the products required to quantize a deep neural network, see “Quantization Workflow Prerequisites”.

## Examples

### Quantize a Neural Network for GPU Target

This example shows how to quantize learnable parameters in the convolution layers of a neural network for GPU and explore the behavior of the quantized network. In this example, you quantize the squeezenet neural network after retraining the network to classify new images according to the Train Deep Learning Network to Classify New Images example. In this example, the memory required for the network is reduced approximately 75% through quantization while the accuracy of the network is not affected.

Load the pretrained network. `net` is the output network of the Train Deep Learning Network to Classify New Images example.

```
load squeezenetmerch
net
```

```
net = 
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Define calibration and validation data to use for quantization.

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 `MerchData` data set. Define an `augmentedImageDatastore` object to resize the data for the network. Then, split the data into calibration and validation data sets.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

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

```
quantObj = dlquantizer(net);
```

Define a metric function to use to compare the behavior of the network before and after quantization. This example uses the `hComputeModelAccuracy` metric function.

```
function accuracy = hComputeModelAccuracy(predictionScores, net, dataStore)
%% Computes model-level accuracy statistics

    % Load ground truth
    tmp = readall(dataStore);
    groundTruth = tmp.response;

    % 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
```

Specify the metric function in a `dlquantizationOptions` object.

```
quantOpts = dlquantizationOptions('MetricFcn',{@(x)hComputeModelAccuracy(x, net, aug_valData)});
```

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.

```
calResults = calibrate(quantObj, aug_calData)
```

```
calResults=121×5 table
        Optimized Layer Name           Network Layer Name     Learnables / Activations    MinValue
    ____________________________    ____________________    ________________________    _________

    {'conv1_Weights'            }    {'conv1'            }          "Weights"              -0.91985
    {'conv1_Bias'               }    {'conv1'            }          "Bias"                 -0.07925
    {'fire2-squeeze1x1_Weights'}    {'fire2-squeeze1x1'}          "Weights"                 -1.38
    {'fire2-squeeze1x1_Bias'   }    {'fire2-squeeze1x1'}          "Bias"                 -0.11641
    {'fire2-expand1x1_Weights' }    {'fire2-expand1x1' }          "Weights"                -0.7406
    {'fire2-expand1x1_Bias'    }    {'fire2-expand1x1' }          "Bias"                -0.060056
    {'fire2-expand3x3_Weights' }    {'fire2-expand3x3' }          "Weights"               -0.74397
    {'fire2-expand3x3_Bias'    }    {'fire2-expand3x3' }          "Bias"                -0.051778
    {'fire3-squeeze1x1_Weights'}    {'fire3-squeeze1x1'}          "Weights"                -0.7712
    {'fire3-squeeze1x1_Bias'   }    {'fire3-squeeze1x1'}          "Bias"                 -0.10138
    {'fire3-expand1x1_Weights' }    {'fire3-expand1x1' }          "Weights"               -0.72035
    {'fire3-expand1x1_Bias'    }    {'fire3-expand1x1' }          "Bias"                -0.067029
    {'fire3-expand3x3_Weights' }    {'fire3-expand3x3' }          "Weights"               -0.61443
    {'fire3-expand3x3_Bias'    }    {'fire3-expand3x3' }          "Bias"                -0.053613
    {'fire4-squeeze1x1_Weights'}    {'fire4-squeeze1x1'}          "Weights"                -0.7422
    {'fire4-squeeze1x1_Bias'   }    {'fire4-squeeze1x1'}          "Bias"                 -0.10885
      ⋮
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```
valResults = validate(quantObj, aug_valData, quantOpts)
```

```
valResults = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: [2×2 table]
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

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

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

```
valResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}               2.9003e+06
     {'Quantized'     }               7.3393e+05
```

In this example, the memory required for the network was reduced approximately 75% through quantization. The accuracy of the network is not affected.

The weights, biases, and activations of the convolution layers of the network specified in the dlquantizer object now use scaled 8-bit integer data types.

### Quantize a Neural Network for FPGA Target

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.

To run this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

For additional requirements, see “Quantization Workflow Prerequisites”.

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
```

Load the pretrained network.

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

```
snet = 

  SeriesNetwork with properties:

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

Define calibration and validation data to use for quantization.

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.

This example uses 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 a `dlquantizer` object and specify the network to quantize.

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

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

% Copyright 2020 The MathWorks, Inc.

    % Load ground truth
    groundTruth = dataStore.Labels;

    % Compare predicted label with 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
```

Specify the metric function in a `dlquantizationOptions` object.

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

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
```

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

```
validateOut = 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
```

The weights, biases, and activations of the convolution layers of the network specified in the `dlquantizer` object now use scaled 8-bit integer data types.

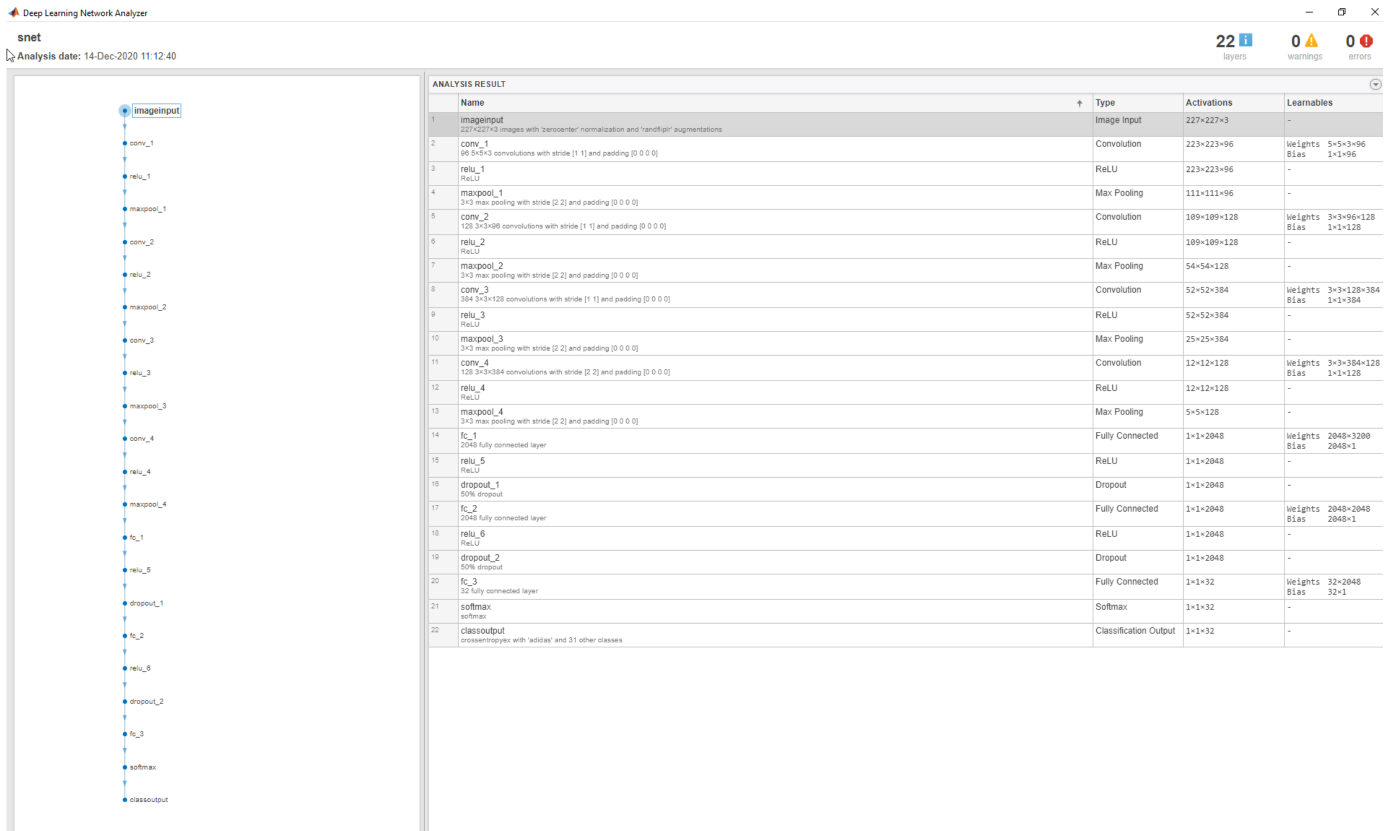
### Validate Quantized Network by Using MATLAB Simulation

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and validate the quantized network. Rapidly prototype the quantized network by using MATLAB based simulation to validate the quantized network. For this type of simulation, you do not need hardware FPGA board from the prototyping process. In this example, you quantize the LogoNet neural network.

For this example, you need the products listed under `FPGA` in “Quantization Workflow Prerequisites”.

Load the pretrained network and analyze the network architecture.

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




Define calibration and validation data to use for quantization.

This example uses the `logos_dataset` data set. The data set consists of 320 images. Each image is 227-by-227 in size and has three color channels (RGB). Create an `augmentedImageDatastore` object to use for calibration and validation. Expedite the calibration and validation process by reducing the calibration data set to 20 images. The MATLAB simulation workflow has a maximum limit of five images when validating the quantized network. Reduce the validation data set sizes to five images.

```
curDir = pwd;
newDir = fullfile(matlabroot,'examples','deeplearning_shared','data','logos_dataset.zip');
copyfile(newDir,curDir,'f');
unzip('logos_dataset.zip');
imageData = imageDatastore(fullfile(curDir,'logos_dataset'),...
 'IncludeSubfolders',true,'FileExtensions','.JPG','LabelSource','foldernames');
[calibrationData, validationData] = splitEachLabel(imageData, 0.5,'randomized');
calibrationData_reduced = calibrationData.subset(1:20);
validationData_reduced = validationData.subset(1:5);
```

Create a quantized network by using the `dlquantizer` object. To use the MATLAB simulation environment set Simulation to on.

```
dlQuantObj = dlquantizer(snet,'ExecutionEnvironment','FPGA','Simulation','on')
```

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

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

```
ans =

  35×5 table

        Optimized Layer Name         Network Layer Name    Learnables / Activations     MinValue
    ____________________________    __________________    ________________________    ___________

    {'conv_1_Weights'          }     {'conv_1'     }           "Weights"                -0.048978
    {'conv_1_Bias'             }     {'conv_1'     }           "Bias"                     0.99996
    {'conv_2_Weights'          }     {'conv_2'     }           "Weights"                -0.055518
    {'conv_2_Bias'             }     {'conv_2'     }           "Bias"                 -0.00061171
    {'conv_3_Weights'          }     {'conv_3'     }           "Weights"                -0.045942
    {'conv_3_Bias'             }     {'conv_3'     }           "Bias"                  -0.0013998
    {'conv_4_Weights'          }     {'conv_4'     }           "Weights"                -0.045967
    {'conv_4_Bias'             }     {'conv_4'     }           "Bias"                    -0.00164
    {'fc_1_Weights'            }     {'fc_1'       }           "Weights"                -0.051394
    {'fc_1_Bias'               }     {'fc_1'       }           "Bias"                 -0.00052319
    {'fc_2_Weights'            }     {'fc_2'       }           "Weights"                 -0.05016
    {'fc_2_Bias'               }     {'fc_2'       }           "Bias"                  -0.0017564
    {'fc_3_Weights'            }     {'fc_3'       }           "Weights"                -0.050706
    {'fc_3_Bias'               }     {'fc_3'       }           "Bias"                    -0.02951
    {'imageinput'              }     {'imageinput' }           "Activations"                    0
    {'imageinput_normalization'}     {'imageinput' }           "Activations"              -139.34
    {'conv_1'                  }     {'conv_1'     }           "Activations"              -431.01
    {'relu_1'                  }     {'relu_1'     }           "Activations"                    0
    {'maxpool_1'               }     {'maxpool_1'  }           "Activations"                    0
    {'conv_2'                  }     {'conv_2'     }           "Activations"              -166.41
    {'relu_2'                  }     {'relu_2'     }           "Activations"                    0
    {'maxpool_2'               }     {'maxpool_2'  }           "Activations"                    0
    {'conv_3'                  }     {'conv_3'     }           "Activations"               -219.6
    {'relu_3'                  }     {'relu_3'     }           "Activations"                    0
    {'maxpool_3'               }     {'maxpool_3'  }           "Activations"                    0
    {'conv_4'                  }     {'conv_4'     }           "Activations"              -245.37
    {'relu_4'                  }     {'relu_4'     }           "Activations"                    0
    {'maxpool_4'               }     {'maxpool_4'  }           "Activations"                    0
    {'fc_1'                    }     {'fc_1'       }           "Activations"              -123.79
    {'relu_5'                  }     {'relu_5'     }           "Activations"                    0
    {'fc_2'                    }     {'fc_2'       }           "Activations"              -16.557
    {'relu_6'                  }     {'relu_6'     }           "Activations"                    0
    {'fc_3'                    }     {'fc_3'       }           "Activations"              -13.049
    {'softmax'                 }     {'softmax'    }           "Activations"           1.4971e-22
    {'classoutput'             }     {'classoutput'}           "Activations"           1.4971e-22
```

Set your target metric function and create a `dlquantizationOptions` object with the target metric function and the validation data set. In this example the target metric function calculates the Top-5 accuracy.

```
options = dlquantizationOptions('MetricFcn', {@(x)hComputeAccuracy(x,snet,validationData_reduced)});
```

**Note** If no custom metric function is specified, the default metric function will be used for validation. The default metric function uses at most 5 files from the validation datastore when the MATLAB simulation environment is selected. Custom metric functions do not have this restriction.

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network. The `validate` function simulates the quantized network in MATLAB. The `validate`

function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the single data type network object to the results of the quantized network object.

```
prediction = dlQuantObj.validate(validationData_reduced,options)
```

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

prediction = 

  struct with fields:

       NumSamples: 5
    MetricResults: [1×1 struct]
```

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

```
validateOut = prediction.MetricResults.Result
```

```
validateOut =

  2×2 table

    NetworkImplementation    MetricOutput
    _____________________    ____________

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

#### Validate Quantized Neural Network for CPU Target

This example shows how to quantize and validate a neural network for a CPU target. This workflow is similar to other execution environments, but before validating you must establish a `raspi` connection.

First, load your network. This example uses the pretrained network `squeezeNet`.

```
load squeezenetmerch
net
```

```
net = 
  DAGNetwork with properties:

         Layers: [68×1 nnet.cnn.layer.Layer]
    Connections: [75×2 table]
     InputNames: {'data'}
    OutputNames: {'new_classoutput'}
```

Then define your calibration and validation data, `calDS` and `valDS` respectively.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[calData, valData] = splitEachLabel(imds, 0.7, 'randomized');
aug_calData = augmentedImageDatastore([227 227], calData);
aug_valData = augmentedImageDatastore([227 227], valData);
```

Create the `dlquantizer object.`

```
dq =  dlquantizer(net,'ExecutionEnvironment','CPU')
```

```
dq =
  dlquantizer with properties:

           NetworkObject: [1×1 DAGNetwork]
    ExecutionEnvironment: 'CPU'
```

`Calibrate the network.`

```
calResults = calibrate(dq, aug_calData)
```

```
calResults=121×5 table
        Optimized Layer Name          Network Layer Name     Learnables / Activations    MinValue
    ____________________________    ____________________    ________________________    ________

    {'conv1_Weights'           }    {'conv1'           }           "Weights"             -0.9198
    {'conv1_Bias'              }    {'conv1'           }           "Bias"                -0.0792
    {'fire2-squeeze1x1_Weights'}    {'fire2-squeeze1x1'}           "Weights"                -1.3
    {'fire2-squeeze1x1_Bias'   }    {'fire2-squeeze1x1'}           "Bias"                -0.1164
    {'fire2-expand1x1_Weights' }    {'fire2-expand1x1' }           "Weights"              -0.740
    {'fire2-expand1x1_Bias'    }    {'fire2-expand1x1' }           "Bias"               -0.06005
    {'fire2-expand3x3_Weights' }    {'fire2-expand3x3' }           "Weights"             -0.7439
    {'fire2-expand3x3_Bias'    }    {'fire2-expand3x3' }           "Bias"               -0.05177
    {'fire3-squeeze1x1_Weights'}    {'fire3-squeeze1x1'}           "Weights"              -0.771
    {'fire3-squeeze1x1_Bias'   }    {'fire3-squeeze1x1'}           "Bias"                -0.1013
    {'fire3-expand1x1_Weights' }    {'fire3-expand1x1' }           "Weights"             -0.7203
    {'fire3-expand1x1_Bias'    }    {'fire3-expand1x1' }           "Bias"               -0.06702
    {'fire3-expand3x3_Weights' }    {'fire3-expand3x3' }           "Weights"             -0.6144
    {'fire3-expand3x3_Bias'    }    {'fire3-expand3x3' }           "Bias"               -0.05361
    {'fire4-squeeze1x1_Weights'}    {'fire4-squeeze1x1'}           "Weights"              -0.742
    {'fire4-squeeze1x1_Bias'   }    {'fire4-squeeze1x1'}           "Bias"                -0.1088
      ⋮
```

Establish `raspi` connection. This step is unique to CPU targets. If you are connecting to your Raspberry Pi hardware board for the first time, you must also specify the `Username`, `DeviceAddress`, and `Password`.

```
r = raspi;
%  r = raspi('raspiname','username','password');
```

Use the MATLAB Support Package for Raspberry Pi function, `raspi`, to create a connection to the Raspberry Pi. In the above code, replace:

- `raspiname` with the name or address of your Raspberry Pi
- `username` with your user name
- `password` with your password

For more information about `raspi`, see raspi.

Validate the quantized network with the `validate` function.

```
valResults = validate(dq, aug_valData)
```

```
### Starting application: 'codegen\lib\validate_predict_int8\pil\validate_predict_int8.elf'
    To terminate execution: clear validate_predict_int8_pil
### Launching application validate_predict_int8.elf...
### Host application produced the following standard output (stdout) and standard error (stderr)

valResults = struct with fields:
       NumSamples: 20
    MetricResults: [1×1 struct]
       Statistics: []
```

Examine the validation output to see the performance of the quantized network.

```
valResults.MetricResults.Result
```

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

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

## Input Arguments

**`quantObj` — Network to quantize**
`dlquantizer` object

`dlquantizer` object specifying the network to quantize.

**`valData` — Data to use for validation of quantized network**
`imageDataStore` object | `augmentedImageDataStore` object | `pixelLabelImageDataStore` object

Data to use for validation of quantized network, specified as an `imageDataStore` object, an `augmentedImageDataStore` object, or a `pixelLabelImageDataStore` object.

**`quantOpts` — Options for quantizing network**
`dlQuantizationOptions` object

Options for quantizing the network, specified as a `dlquantizationOptions` object.

## Output Arguments

**`validationResults` — Results of quantization of network**
struct

Results of quantization of the network, returned as a struct. The struct contains these fields.

- `NumSamples` – The number of sample inputs used to validate the network.
- `MetricResults` – Struct containing results of the metric function defined in the `dlquantizationOptions` object. When more than one metric function is specified in the `dlquantizationOptions` object, `MetricResults` is an array of structs.

  `MetricResults` contains these fields.

| Field | Description |
| --- | --- |
| `MetricFunction` | Function used to determine the performance of the quantized network. This function is specified in the `dlquantizationOptions` object. |
| `Result` | Table indicating the results of the metric function before and after quantization.<br><br>The first row in the table contains the information for the original, floating-point implementation. The second row contains the information for the quantized implementation. The output of the metric function is displayed in the `MetricOutput` column. |

## See Also

**Apps**
**Deep Network Quantizer**

**Functions**
`calibrate` | `dlquantizer` | `dlquantizationOptions`

**Topics**
"Quantization of Deep Neural Networks"

**Introduced in R2020a**

# dlhdl.Simulator class

**Package:** `dlhdl`

Create an object that retrieves intermediate layer results and validate deep learning network prediction accuracy

## Description

Use the `dlhdl.Simulator` class to creates a simulator object that you use to retrieve intermediate layer results and verify the accuracy of prediction results for your deep learning network without the need for hardware targets.

# Creation

`simObj=dlhdl.Simulator('Network',Network,'ProcessorConfig',ProcessorConfig)` creates a `dlhdl.Simulator` object that you use to retrieve intermediate layer results and verify the accuracy of prediction results for your deep learning network without the need for hardware targets.

### Input Arguments

#### `Network` — Network object

`SeriesNetwork` object | `DAGNetwork` object | `yolov2ObjectDetector` object | `dlquantizer` object

Name of the deep learning network object.

**Example:** `'network'`, `net` creates a workflow object for the saved pretrained network `net`. To specify `net`, you can import any of the existing supported pretrained networks or use transfer learning to adapt the network to your issue. See "Supported Pretrained Networks".

```
net = resnet18;
hPC = dlhdl.ProcessorConfig;
simObj = dlhdl.Simulator('Network',net,'Bitstream',hPC);
```

**Example:** `'network'`, `dlquantizeObj` creates a workflow object for the quantized network object `dlquantizeObj`. To specify `dlquantizeObj`, you can import any of the supported existing pretrained networks and create an object by using the `dlquantizer` class. For information on supported networks, see "Supported Pretrained Networks".

```
net = resnet18;
dlquantObj = dlquantizer(net,'ExecutionEnvironment','FPGA');
dlquantObj.calibrate(imdsTrain);
simObj = dlhdl.Simulator('Network',dlquantObj,'Bitstream',hPC);
```

#### `ProcessorConfig` — `dlhdl.ProcessorConfig` object

`hPC`

Deep learning processor configuration object, specified as a `dlhdl.ProcessorConfig` object

Example: `'ProcessorConfig',hPC`

```
hPC = dlhdl.ProcessorConfig;
simObj = dlhdl.Simulator('Network',resnet18,'ProcessorConfig',hPC);
```

## Methods

### Public Methods

| | |
|---|---|
| activations | Retrieve intermediate layers results for dlhdl.Simulator object |
| predict | Retrieve prediction results for dlhdl.Simulator object |

## Examples

### Create a `dlhdl.Simulator` Object for the ResNet-18 Network

1 Retrieve the deep learning processor configuration for the `zcu102_single` bitstream and save to `hPC`.

```
hPC = dlhdl.ProcessorConfig('Bitstream','zcu102_single');
```

2 Create a `dlhdl.Simulator` object with resnet18 as the network and `hPC` as the `ProcessorConfig` object.

```
simObj = dlhdl.Simulator('Network',resnet18,'ProcessorConfig',hPC);
```

### Create a `dlhdl.Simulator` Object for the resnet18 Network and int8 data type Deep Learning Processor Configuration

1 Create a deep learning processor configuration that has the int8 data type and save it to `hPC`.

```
hPC = dlhdl.ProcessorConfig;
hPC.ProcessorDataType = 'int8';
```

2 Create a `dlquantizer` object with ResNet-18 as the network and FPGA execution environment. Calibrate the quantized network object by using the `calibrate` function.

```
net = resnet18;
dlQuantObj = dlquantizer(net,'ExecutionEnvironment','FPGA');
dlQuantObj.calibrate(imageDataStore);
```

3 Create a `dlhdl.Simulator` object with `dlQuantObj` as the network and `hPC` as the `ProcessorConfig` object.

```
simObj = dlhdl.Simulator('Network',dlQuantObj,'ProcessorConfig',hPC);
```

## See Also

`dlhdl.Workflow` | `dlhdl.Target`

### Topics

"Prototype and Verify Deep Learning Networks Without Target Hardware"

**Introduced in R2021b**

# activations

**Class:** `dlhdl.Simulator`
**Package:** `dlhdl`

Retrieve intermediate layers results for `dlhdl.Simulator` object

## Syntax

```
activations(simulatorObject,image,layername)
result = activations(simulatorObject,image,layername)
```

## Description

`activations(simulatorObject,image,layername)` returns intermediate layer activation data results for the image data in `image` and the name of the layer specified in `layername`. The result size depends on the output size of the layer. The layer output size can be retrieved by using the `analyzeNetwork` function.

`result = activations(simulatorObject,image,layername)` stores the intermediate layer activation data results for the image data in `image` and the name of the layer specified in `layername` in `result`. The result size depends on the output size of the layer. The layer output size can be retrieved by using the `analyzeNetwork` function.

## Input Arguments

**`simulatorObject` — Instance of simulator object**
`dlhdl.Simulator` object

Instance of the simulator object, specified as an `dlhdl.Simulator` object.

**`image` — Input image**
*m*-by-*n*-by-*k* numeric array

Input image, specified as a *m*-by-*n*-by-*k* numeric array. *m* , *n*, and *k* must match the dimensions of the deep learning network input image layer. For example, for the LogoNet network, resize the input images to a 227-by-227-by-3 array.

Data Types: `single`

**`layername` — Layer to extract activations from**
'' (default) | character vector

Layer to extract activations from, specified as a character vector.

To compute the activations of a `SeriesNetwork` object, specify the layer as a character vector corresponding to the layer name.

To compute the activations of a `DAGNetwork` object, specify the layer as a character vector corresponding to the layer name. If the layer has multiple outputs, specify the layer and output as the layer name, followed by the character `"/"`, followed by the name of the layer output. That is, `layername` of the form `'layerName/outputName'`.

These limitations apply when extracting activations from layers:

- A `maxPooling2DLayer` which has the `HasUnpoolingIndices` argument set to `true` has three outputs `out`, `indices`, and `size`. Activations are supported only for the output `out`.
- Activations for a `ReLU`layer in the branch of a DAG network are supported when the layer is not preceded by a `Convolution`, `Fully Connected`, or `Addition` layer.
- Activations for the `dropout` layer is not supported.

Example: `'conv1'`

Example: `'mpool/out'`

## Output Arguments

**`result` — Intermediate layer activation data**
`array of single`

Intermediate layer activation data, returned as an array of singles. The array size depends on the layer output size. For example, for the ResNet-18 network `pool1` layer, the size of the returned result array is 56-by-56-by-64.

## Examples

### Prototype and Verify Deep Learning Networks Without Target Hardware

Rapidly prototype your custom deep learning network and bitstream by visualizing intermediate layer activation results and verifying prediction accuracy without target hardware by emulating the network and bitstream. To emulate the network and bitstream, create a `dlhdl.Simulator` object. Use the `dlhdl.Simulator` object to:

- Retrieve intermediate layer results by using the `activations` function.
- Verify prediction accuracy by using the `predict` function.

In this example, retrieve the intermediate layer activation results and verify the prediction accuracy for the ResNet-18 network and deep learning processor configuration for the `zcu102_single` bitstream.

#### Prerequisites

- Deep Learning Toolbox ™
- Deep Learning HDL Toolbox ™
- Deep Learning Toolbox Model for ResNet-18 Network
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- Image Processing Toolbox ™
- MATLAB Coder Interface for Deep learning Libraries™

#### Load Pretrained SeriesNetwork

To load the pretrained network ResNet-18, enter:

```
snet = resnet18;
```

To view the layers of the pretrained network, enter:

```
analyzeNetwork(snet);
```

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

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

#### Define Training and Validation Data Sets

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

```
curDir = pwd;
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
'IncludeSubfolders',true, ...
'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

#### Replace Final Layers

The fully connected layer and the classification layer of the pretrained network `net` are configured for 1000 classes. These two layers `fc1000` and `ClassificationLayer_predictions` in ResNet-18 contain information on how to combine the features that the network extracts into class probabilities and predicted labels. These layers must be fine-tuned for the new classification problem. Extract all the layers, except the last two layers, from the pretrained network.

```
lgraph = layerGraph(snet)
```

```
lgraph = 
  LayerGraph with properties:

         Layers: [71×1 nnet.cnn.layer.Layer]
    Connections: [78×2 table]
     InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}
```

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

```
numClasses = 5
```

```
newLearnableLayer = fullyConnectedLayer(numClasses, ...
'Name','new_fc', ...
'WeightLearnRateFactor',10, ...
'BiasLearnRateFactor',10);
lgraph = replaceLayer(lgraph,'fc1000',newLearnableLayer);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassLayer);
```

#### Train Network

The network requires input images of size 224-by-224-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images, such as randomly

flipping the training images along the vertical axis and randomly translating them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
'RandXReflection',true, ...
'RandXTranslation',pixelRange, ...
'RandYTranslation',pixelRange);
```

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

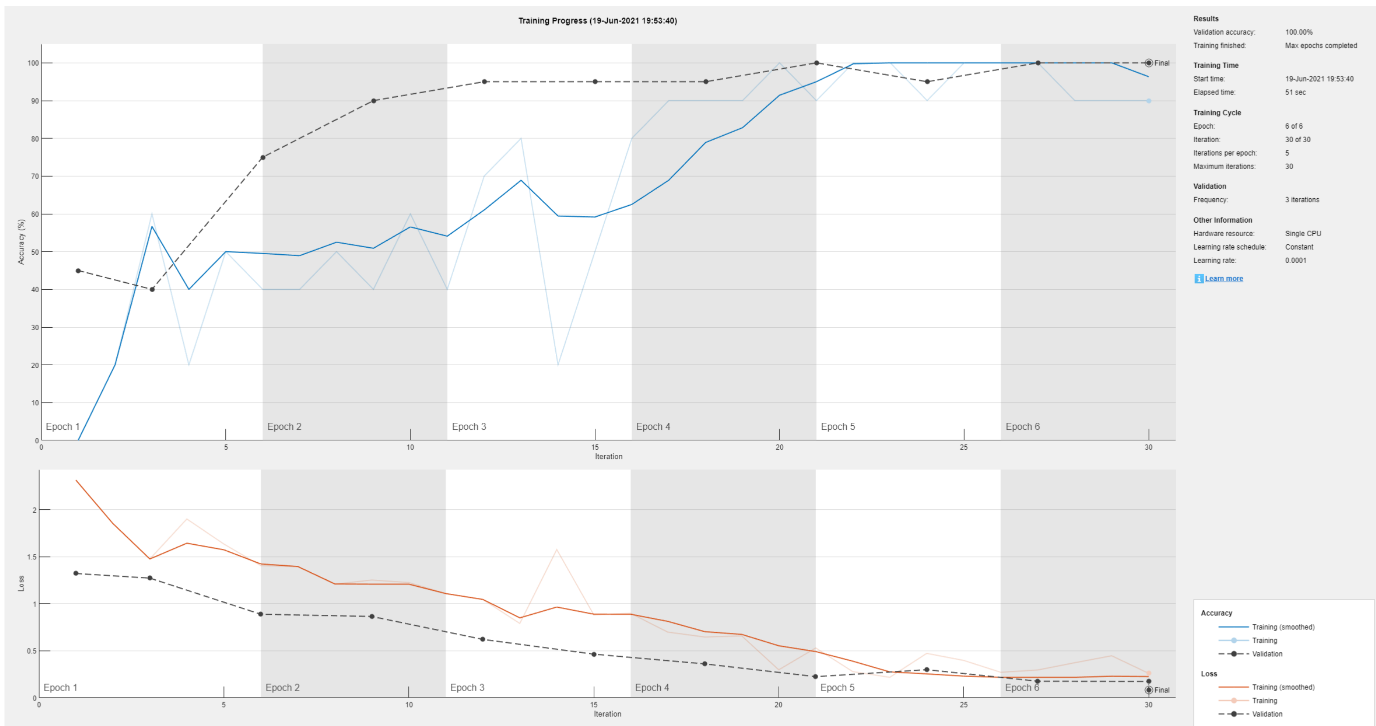
```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
'DataAugmentation',imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. Specify the mini-batch size and validation data. The software validates the network for every `ValidationFrequency` iteration during training.

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

Train the network that consists of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a supported GPU device. See “GPU Support by Release” (Parallel Computing Toolbox)). Otherwise, the network uses a CPU (requires MATLAB Coder Interface for Deep learning Libraries™). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value argument of `trainingOptions`.

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




### Retrieve Deep Learning Processor Configuration

Use the `dlhdl.ProcessorConfig` object to retrieve the deep learning processor configuration for the `zcu102_single` bitstream.

```
hPC = dlhdl.ProcessorConfig('Bitstream','zcu102_single');
```

### Create Simulator Object

Create a `dlhdl.Simulator` object with ResNet-18 as the network and `hPC` as the `ProcessorConfig` object.

```
simObj = dlhdl.Simulator('Network',netTransfer,'ProcessorConfig',hPC);
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'out_data' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented
Compiling leg: conv1>>pool1 ...
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented i
Compiling leg: conv1>>pool1 ... complete.
Compiling leg: res2a_branch2a>>res2a_branch2b ...
### Notice: The layer 'pool1' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softw
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented i
Compiling leg: res2a_branch2a>>res2a_branch2b ... complete.
Compiling leg: res2b_branch2a>>res2b_branch2b ...
### Notice: The layer 'res2a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented i
Compiling leg: res2b_branch2a>>res2b_branch2b ... complete.
Compiling leg: res3a_branch1 ...
### Notice: The layer 'res2b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented i
```

```
Compiling leg: res3a_branch1 ... complete.
Compiling leg: res3a_branch2a>>res3a_branch2b ...
### Notice: The layer 'res2b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res3a_branch2a>>res3a_branch2b ... complete.
Compiling leg: res3b_branch2a>>res3b_branch2b ...
### Notice: The layer 'res3a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res3b_branch2a>>res3b_branch2b ... complete.
Compiling leg: res4a_branch1 ...
### Notice: The layer 'res3b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res4a_branch1 ... complete.
Compiling leg: res4a_branch2a>>res4a_branch2b ...
### Notice: The layer 'res3b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res4a_branch2a>>res4a_branch2b ... complete.
Compiling leg: res4b_branch2a>>res4b_branch2b ...
### Notice: The layer 'res4a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res4b_branch2a>>res4b_branch2b ... complete.
Compiling leg: res5a_branch1 ...
### Notice: The layer 'res4b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res5a_branch1 ... complete.
Compiling leg: res5a_branch2a>>res5a_branch2b ...
### Notice: The layer 'res4b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res5a_branch2a>>res5a_branch2b ... complete.
Compiling leg: res5b_branch2a>>res5b_branch2b ...
### Notice: The layer 'res5a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res5b_branch2a>>res5b_branch2b ... complete.
Compiling leg: pool5 ...
### Notice: The layer 'res5b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: pool5 ... complete.
Compiling leg: new_fc ...
### Notice: The layer 'pool5' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: new_fc ... complete.
### Should not enter here. It means a component is unaccounted for in MATLAB Emulation.
### Notice: The layer 'new_fc' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softw
### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'new_classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is i
```

**Load Image for Prediction and Intermediate Layer Activation Results**

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

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

#### Show Activations of First Maxpool Layer

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

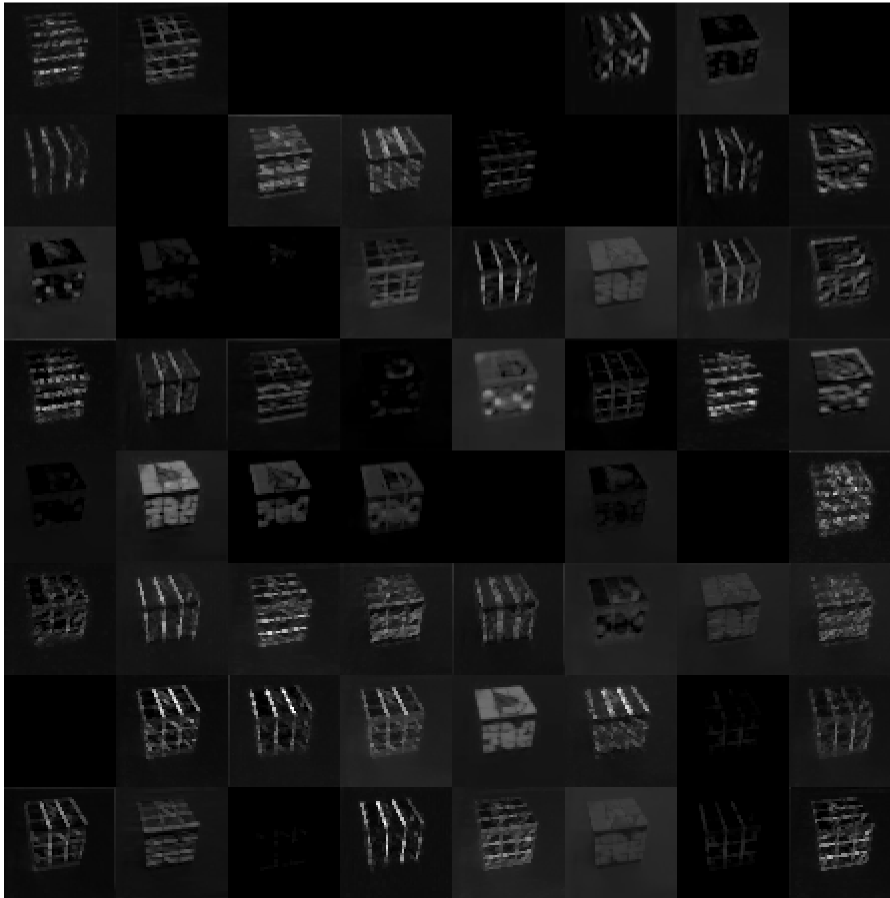
```
act1 = simObj.activations(single(inputImg),'pool1');
```

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

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

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

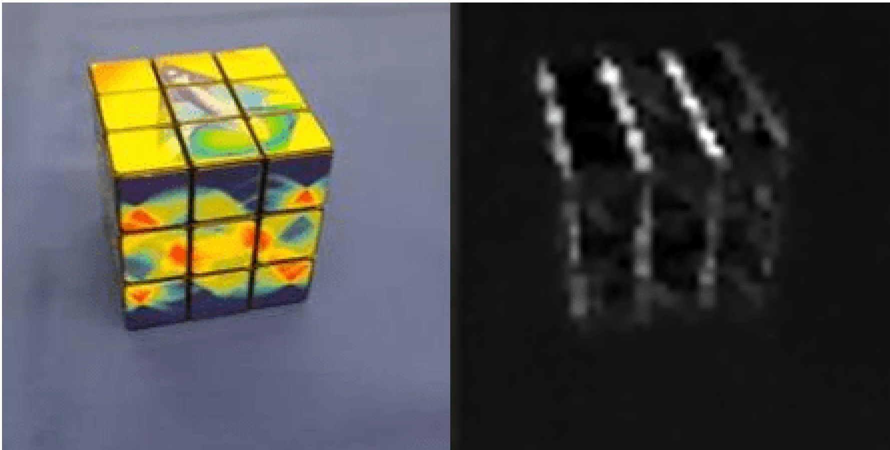
```
I = imtile(mat2gray(act1),'GridSize',[8 8]);
imshow(I)
```



### Find Strongest Activation Channel

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

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

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



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

#### Verify Prediction Results

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

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

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

## See Also

`dlhdl.Simulator` | `predict`

**Topics**

"Prototype and Verify Deep Learning Networks Without Target Hardware"

**Introduced in R2021b**

# predict

**Class:** `dlhdl.Simulator`
**Package:** `dlhdl`

Retrieve prediction results for `dlhdl.Simulator` object

## Syntax

```
prediction = predict(simulatorObject,image)
```

## Description

`prediction = predict(simulatorObject,image)` returns a table containing the percentage prediction values for the input image specified by `image`.

## Input Arguments

**`simulatorObject` — Instance of simulator object**
`dlhdl.Simulator` object

Instance of the simulator object, specified as an `dlhdl.Simulator` object.

**`image` — Input image**
*m*-by-*n*-by-*k* numeric array

Input image, specified as a *m*-by-*n*-by-*k* numeric array. *m*, *n*, and *k* must match the dimensions of the deep learning network input image layer. For example, for the LogoNet network, resize the input images to a 227-by-227-by-3 array.

Data Types: `single`

## Output Arguments

**`prediction` — Network prediction for input image**
`table`

Deep learning network prediction for the input image specified by `image`.

## Examples

### Prototype and Verify Deep Learning Networks Without Target Hardware

Rapidly prototype your custom deep learning network and bitstream by visualizing intermediate layer activation results and verifying prediction accuracy without target hardware by emulating the network and bitstream. To emulate the network and bitstream, create a `dlhdl.Simulator` object. Use the `dlhdl.Simulator` object to:

- Retrieve intermediate layer results by using the `activations` function.

- Verify prediction accuracy by using the `predict` function.

In this example, retrieve the intermediate layer activation results and verify the prediction accuracy for the ResNet-18 network and deep learning processor configuration for the `zcu102_single` bitstream.

#### Prerequisites

- Deep Learning Toolbox ™
- Deep Learning HDL Toolbox ™
- Deep Learning Toolbox Model for ResNet-18 Network
- Deep Learning HDL Toolbox Support Package for Xilinx FPGA and SoC Devices
- Image Processing Toolbox ™
- MATLAB Coder Interface for Deep learning Libraries™

#### Load Pretrained SeriesNetwork

To load the pretrained network ResNet-18, enter:

```
snet = resnet18;
```

To view the layers of the pretrained network, enter:

```
analyzeNetwork(snet);
```

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

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

#### Define Training and Validation Data Sets

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

```
curDir = pwd;
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
'IncludeSubfolders',true, ...
'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

#### Replace Final Layers

The fully connected layer and the classification layer of the pretrained network `net` are configured for 1000 classes. These two layers `fc1000` and `ClassificationLayer_predictions` in ResNet-18 contain information on how to combine the features that the network extracts into class probabilities and predicted labels. These layers must be fine-tuned for the new classification problem. Extract all the layers, except the last two layers, from the pretrained network.

```
lgraph = layerGraph(snet)
```

```
lgraph =
  LayerGraph with properties:
```

```
        Layers: [71×1 nnet.cnn.layer.Layer]
   Connections: [78×2 table]
    InputNames: {'data'}
   OutputNames: {'ClassificationLayer_predictions'}
```

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

```
numClasses = 5
```

```
newLearnableLayer = fullyConnectedLayer(numClasses, ...
'Name','new_fc', ...
'WeightLearnRateFactor',10, ...
'BiasLearnRateFactor',10);
lgraph = replaceLayer(lgraph,'fc1000',newLearnableLayer);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassLayer);
```

### Train Network

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

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
'RandXReflection',true, ...
'RandXTranslation',pixelRange, ...
'RandYTranslation',pixelRange);
```

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

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

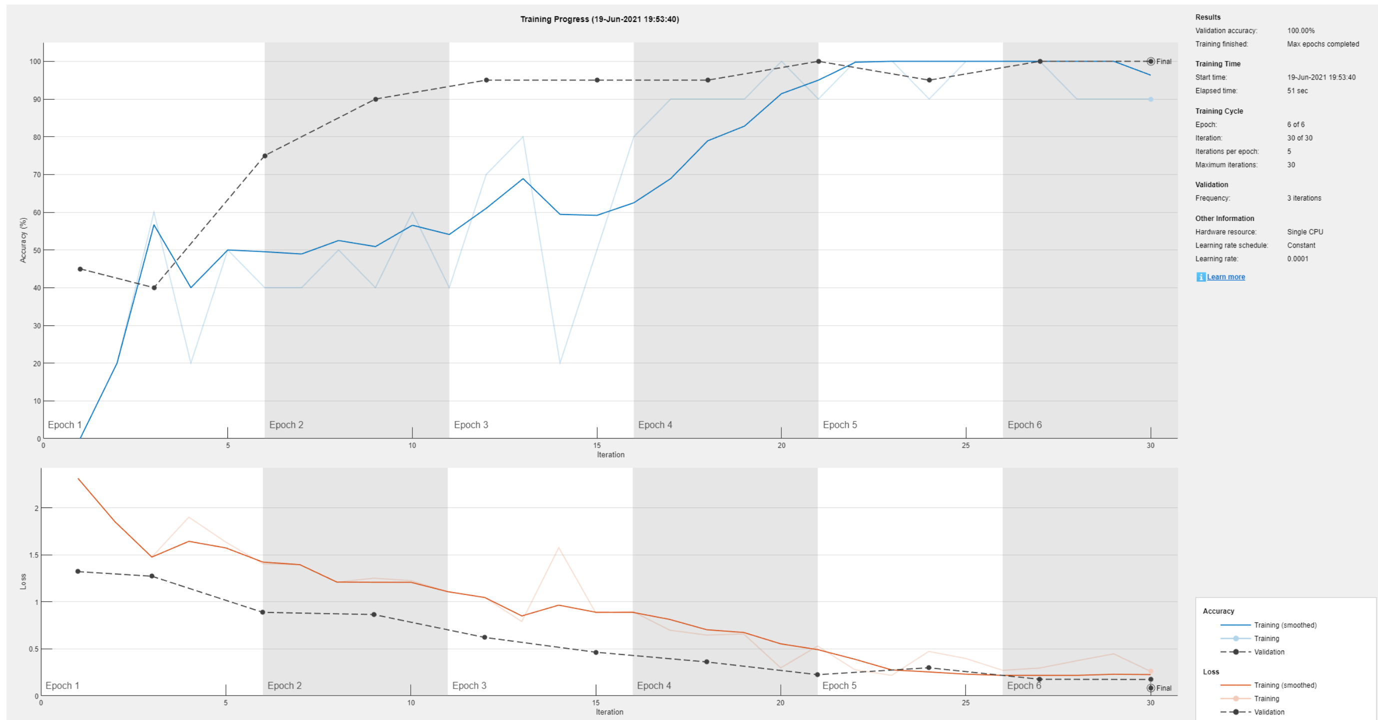
Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. Specify the mini-batch size and validation data. The software validates the network for every `ValidationFrequency` iteration during training.

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

Train the network that consists of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a supported GPU device. See “GPU Support by Release” (Parallel Computing Toolbox)). Otherwise, the network uses a CPU

(requires MATLAB Coder Interface for Deep learning Libraries™). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value argument of `trainingOptions`.

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




### Retrieve Deep Learning Processor Configuration

Use the `dlhdl.ProcessorConfig` object to retrieve the deep learning processor configuration for the `zcu102_single` bitstream.

```
hPC = dlhdl.ProcessorConfig('Bitstream','zcu102_single');
```

### Create Simulator Object

Create a `dlhdl.Simulator` object with ResNet-18 as the network and `hPC` as the `ProcessorConfig` object.

```
simObj = dlhdl.Simulator('Network',netTransfer,'ProcessorConfig',hPC);
```

```
### Optimizing series network: Fused 'nnet.cnn.layer.BatchNormalizationLayer' into 'nnet.cnn.laye
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'out_data' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented
Compiling leg: conv1>>pool1 ...
### Notice: The layer 'data' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: conv1>>pool1 ... complete.
Compiling leg: res2a_branch2a>>res2a_branch2b ...
### Notice: The layer 'pool1' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res2a_branch2a>>res2a_branch2b ... complete.
Compiling leg: res2b_branch2a>>res2b_branch2b ...
### Notice: The layer 'res2a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
```

```
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res2b_branch2a>>res2b_branch2b ... complete.
Compiling leg: res3a_branch1 ...
### Notice: The layer 'res2b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res3a_branch1 ... complete.
Compiling leg: res3a_branch2a>>res3a_branch2b ...
### Notice: The layer 'res2b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res3a_branch2a>>res3a_branch2b ... complete.
Compiling leg: res3b_branch2a>>res3b_branch2b ...
### Notice: The layer 'res3a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res3b_branch2a>>res3b_branch2b ... complete.
Compiling leg: res4a_branch1 ...
### Notice: The layer 'res3b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res4a_branch1 ... complete.
Compiling leg: res4a_branch2a>>res4a_branch2b ...
### Notice: The layer 'res3b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res4a_branch2a>>res4a_branch2b ... complete.
Compiling leg: res4b_branch2a>>res4b_branch2b ...
### Notice: The layer 'res4a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res4b_branch2a>>res4b_branch2b ... complete.
Compiling leg: res5a_branch1 ...
### Notice: The layer 'res4b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res5a_branch1 ... complete.
Compiling leg: res5a_branch2a>>res5a_branch2b ...
### Notice: The layer 'res4b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res5a_branch2a>>res5a_branch2b ... complete.
Compiling leg: res5b_branch2a>>res5b_branch2b ...
### Notice: The layer 'res5a_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: res5b_branch2a>>res5b_branch2b ... complete.
Compiling leg: pool5 ...
### Notice: The layer 'res5b_relu' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in s
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: pool5 ... complete.
Compiling leg: new_fc ...
### Notice: The layer 'pool5' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softwa
### Notice: The layer 'output' with type 'nnet.cnn.layer.RegressionOutputLayer' is implemented in
Compiling leg: new_fc ... complete.
### Should not enter here. It means a component is unaccounted for in MATLAB Emulation.
### Notice: The layer 'new_fc' with type 'nnet.cnn.layer.ImageInputLayer' is implemented in softw
### Notice: The layer 'prob' with type 'nnet.cnn.layer.SoftmaxLayer' is implemented in software.
### Notice: The layer 'new_classoutput' with type 'nnet.cnn.layer.ClassificationOutputLayer' is i
```

**Load Image for Prediction and Intermediate Layer Activation Results**

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

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

#### Show Activations of First Maxpool Layer

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

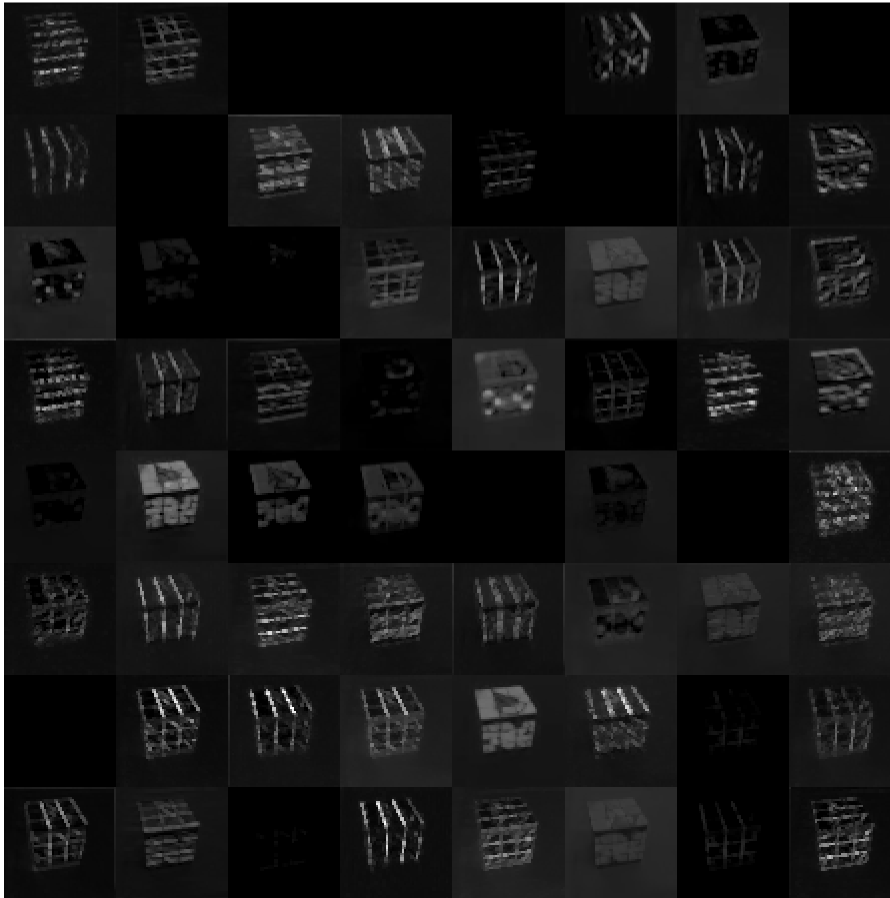
```
act1 = simObj.activations(single(inputImg),'pool1');
```

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

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

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

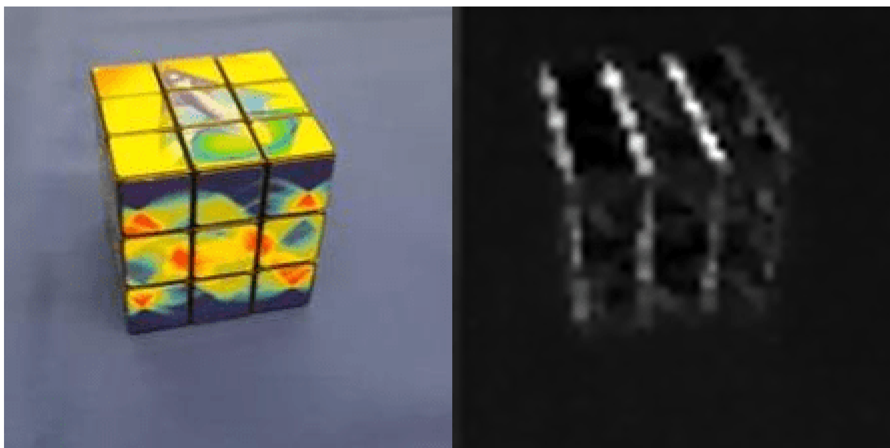
```
I = imtile(mat2gray(act1),'GridSize',[8 8]);
imshow(I)
```



### Find Strongest Activation Channel

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

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

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



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

#### Verify Prediction Results

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

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

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

## See Also

`activations` | `dlhdl.Simulator`

#### Topics

"Prototype and Verify Deep Learning Networks Without Target Hardware"

**Introduced in R2021b**

# hdlcoder.ReferenceDesign class

**Package:** `hdlcoder`

Reference design registration object that describes SoC reference design

## Description

`refdesign = hdlcoder.ReferenceDesign('SynthesisTool', toolname)` creates a reference design object that you use to register a custom reference design for an SoC platform.

To specify the characteristics of your reference design, set the properties of the reference design object.

Use a reference design tool version that is compatible with the supported tool version. If you choose a different tool version, it is possible that HDL Coder is unable to create the reference design project for IP core integration.

# Creation

`refdesign = hdlcoder.ReferenceDesign('SynthesisTool',toolname)` creates a reference design object that you use to register a custom reference design for an SoC platform.

### Input Arguments

**`toolname` — Synthesis tool name**
`Xilinx Vivado` (default) | `Altera Quartus II` | `Xilinx ISE` | `Xilinx Vivado`

Synthesis tool name, specified as a character vector.

Example: `'Altera Quartus II'`

## Properties

**`ReferenceDesignName` — Reference design name**
`''` (default) | character vector

Reference design name, specified as a character vector. In the HDL Workflow Advisor, this name appears in the **Reference design** drop-down list.

Example: `'Default system (Vivado 2015.4)'`

**`BoardName` — Board name**
`''` (default) | character vector

Board associated with this reference design, specified as a character vector.

Example: `'Enclustra Mars ZX3 with PM3 base board'`

**`SupportedToolVersion` — Supported tool version**
`{}` (default) | cell array of character vectors

One or more tool versions that work with this reference design, specified as a cell array of character vectors.

Example: `{'2015.4'}`

Example: `{'13.7','14.0'}`

**`CustomConstraints` — Design constraint file (optional)**
`{}` (default) | cell array of character vectors

One or more design constraint files, specified as a cell array of character vectors. This property is optional.

Example: `{'MarsZX3_PM3.xdc'}`

Example: `{'MyDesign.qsf'}`

**`CustomFiles` — Relative path to required file or folder (optional)**
`{}` (default) | cell array of character vectors

One or more relative paths to files or folders that the reference design requires, specified as a cell array of character vectors. This property is optional.

Examples of required files or folders:

- Existing IP core used in the reference design.

  For example, if the IP core, *`my_ip_core`*, is in the reference design folder, set `CustomFiles` to `{'`*`my_ip_core`*`']`
- PS7 definition XML file.

  For example, to include a PS7 definition XML file, *`ps7_system_prj.xml`*, in a folder, *`data`*, set `CustomFiles` to `{fullfile('`*`data`*`', '`*`ps7_system_prj.xml`*`')}`
- Folder containing existing IP cores used in the reference design. HDL Coder supports only a specific IP core folder name for each synthesis tool:

  - For Altera Qsys, IP core files must be in a folder named `ip`. Set `CustomFiles` to `{'ip'}`.
  - For Xilinx Vivado, IP core files, or a zip file containing the IP core files, must be in a folder named `ipcore`. Set `CustomFiles` to `{'ipcore'}`.
  - For Xilinx EDK, IP core files must be in a folder named `pcores`. Set `CustomFiles` to `{'pcores'}`.

**Note** To add IP modules to the reference design, it is recommended to create an IP repository folder that contains these IP modules, and then use the `addIPRepository` method.

Example: `{'my_ip_core'}`

Example: `{fullfile('data', 'ps7_system_prj.xml')}`

Example: `{'ip'}`

Example: `{'ipcore'}`

Example: `{'pcores'}`

**`DeviceTreeName` — Linux device tree name**
character vector

Specify the device tree file name. For an example that shows how to use different device tree file names when mapping the DUT ports to different AXI4-Stream channels, see Dynamically Create Master Only or Slave Only or Both Master and Slave Reference Designs (HDL Coder).

Example: `'devicetree_axistream_iio.dtb'`

**`AddJTAGMATLABasAXIMasterParameter` — Control visibility of JTAG AXI Manager IP**
`true` (default) | `false` | logical data type

Specify whether you want the parameter **Insert JTAG AXI Manager (HDL Verifier required)** to be displayed in the **Set Target Reference Design** task of the HDL Workflow Advisor. By default, this property value is set to `true`. The parameter is displayed in the **Set Target Reference Design** task. After you enable this property, to specify whether you want the code generator to insert the JTAG AXI Manager IP, use the `JTAGMATLABasAXIMasterDefaultValue` property. If you do not want the parameter to be displayed, set the property value to `false`.

This property is optional.

Example: `'false'`

**`JTAGMATLABasAXIMasterDefaultValue` — Specify whether to insert AXI Manager IP**
`'off'` (default) | `'on'` | character vector

Specify whether you want the code generator to insert the JTAG AXI Manager IP. The values that you specify are the choices for the **Insert JTAG AXI Manager (HDL Verifier required)** drop-down in the **Set Target Reference Design** task of the HDL Workflow Advisor. To specify insertion of the JTAG AXI Manager automatically, before you set this property to `on`, set the `AddJTAGMATLABasAXIMasterParameter` property to `true`.

This property is optional.

Example: `'on'`

**`IPCacheZipFile` — IP cache file to include in the project**
`''` (default) | `'ipcache.zip'` | character vector

Specify the IP cache zip file to include in your project. When you run the `IP Core Generation` workflow in the HDL Workflow Advisor, the code generator extracts this file in the **Create Project** task. The **Build FPGA Bitstream** task reuses the IP cache, which accelerates reference design synthesis.

This property is optional.

Example: `'ipcache.zip'`

**`ReportTimingFailure` — Report timing failures as warnings or errors**
`'hdlcoder.ReportTiming.Warning'` (default) | `'hdlcoder.ReportTiming.Error'`

Specify whether you want the code generator to report timing failures in the **Build FPGA Bitstream** task as warnings or errors. When you run the `IP Core Generation` workflow in the HDL Workflow Advisor, by default, the code generator reports any timing failures as error. If you have implemented the custom logic to resolve timing failures, you can specify these failures to be reported as warning instead of error. To learn more, see “Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows” (HDL Coder).

This property is optional.

Example: `'hdlcoder.ReportTiming.Warning'`

**`HasProcessingSystem` — Specify if reference design has existing Processing System (PS)**
`true` (default) | `false` | logical data type

Specify if the reference design has an existing PS.

Example: `'false'`

**`GenerateIPCoreDeviceTreeNodes` — Enable generation of device tree nodes for HDL Coder IP core**
`false` (default) | `true` | logical data type

Enable generation of device tree nodes for an HDL Coder generated IP core, and then insert the nodes into the device tree. To enable the generation of device tree nodes for the IP core, `HasProcessingSystem` must be set to `true`.

Do not enable this property if you do not need any additional device tree nodes to be inserted into the registered device tree for the generated IP core.

Example: `'true'`

**`ResourcesUsed` — Board resources used by reference design**
structure

Board resources used by reference design, returned as a structure with the fields:

**`LogicElements` — Reference design resources utilized by FPGA lookup tables (LUTs)**
0 (default)

Reference design resources utilized by FPGA lookup tables (LUTs), specified as a number.

Example: `hRD.ResourcesUsed.LogicElements = 100`

**`DSP` — Reference design resources utilized by FPGA DSP slices**
0 (default)

Reference design resources utilized by FPGA DSP slices, specified as a number.

Example: `hRD.ResourcesUsed.DSP = 3`

**`RAM` — Reference design resources utilized by FPGA board RAM resources**
0 (default)

Reference design resources utilized by FPGA board RAM resources, specified as a number.

Example: `hRD.ResourcesUsed.RAM = 32000`

## Methods

### Public Methods

| | |
|---|---|
| CallbackCustomProgrammingMethod | Function handle for custom callback function that gets executed during Program Target Device task in the Workflow Advisor |

| | |
|---|---|
| CustomizeReferenceDesignFcn | Function handle for callback function that gets executed before Set Target Interface task in the HDL Workflow Advisor |
| EmbeddedCoderSupportPackage | Specify whether to use an Embedded Coder support package |
| PostBuildBitstreamFcn | Function handle for callback function that gets executed after Build FPGA Bitstream task in the HDL Workflow Advisor |
| PostCreateProjectFcn | Function handle for callback function that gets executed after Create Project task in the HDL Workflow Advisor |
| PostSWInterfaceFcn | Function handle for custom callback function that gets executed after Generate Software Interface task in the HDL Workflow Advisor |
| PostTargetInterfaceFcn | Function handle for callback function that gets executed after Set Target Interface task in the HDL Workflow Advisor |
| PostTargetReferenceDesignFcn | Function handle for callback function that gets executed after Set Target Reference Design task in the HDL Workflow Advisor |
| addAXI4MasterInterface | Add and define AXI4 Master interface |
| addAXI4SlaveInterface | Add and define AXI4 slave interface |
| addAXI4StreamInterface | Add AXI4-Stream interface |
| addAXI4StreamVideoInterface | Add AXI4-Stream Video interface |
| addClockInterface | Add clock and reset interface |
| addCustomEDKDesign | Specify Xilinx EDK MHS project file |
| addCustomQsysDesign | Specify Altera Qsys project file |
| addCustomVivadoDesign | Specify Xilinx Vivado exported block design Tcl file |
| addDeviceTree | Add device tree for reference design object |
| addDeviceTreeIncludeDirectory | Specify the path of an include file to compile the device tree against |
| addIPRepository | Include IP modules from your IP repository folder in your custom reference design |
| addInternalIOInterface | Add and define internal IO interface between generated IP core and existing IP cores |
| addParameter | Add and define custom parameters for your reference design |
| validateReferenceDesign | Check property values in reference design object |

## See Also

`hdlcoder.Board`

### Topics

"Define Custom Board and Reference Design for Zynq Workflow" (HDL Coder)
"Define Custom Board and Reference Design for Intel SoC Workflow" (HDL Coder)
"Register a Custom Board" (HDL Coder)
"Register a Custom Reference Design" (HDL Coder)
"Define Custom Parameters and Callback Functions for Custom Reference Design" (HDL Coder)
"Board and Reference Design Registration System" (HDL Coder)

**Introduced in R2015a**

# registerDeepLearningMemoryAddressSpace

**Class:** `hdlcoder.ReferenceDesign`
**Package:** `hdlcoder`

Add memory address space to reference design

## Syntax

`registerDeepLearningMemoryAddressSpace(baseAddr,addrRange)`

## Description

`registerDeepLearningMemoryAddressSpace(baseAddr,addrRange)` registers memory address space accessible by the deep learning processor IP core to your reference design. The deep learning processor IP core uses this registered memory space to store the inputs to the deep learning network, network weights, and intermediate computations performed by the deep learning processor.

## Input Arguments

**`baseAddr` — Base address of the deep learning memory address space**
0x0 (default) | uint32

Base address of the deep learning memory address space, in bytes as `uint32`.

Example: 0x80000000

**`addrRange` — Address range of the deep learning memory address space**
0x0 (default) | uint32

Address range of the deep learning memory address space, in bytes as `uint32`.

Example: 0x40000000

## Requirements

The registered memory address space must be accessible by the AXI4 Master interfaces in your reference design.

## Tips

- A minimum of 32 MB of memory address space is required to run any deep learning network. To run a variety of deep learning networks, it is recommended to register a minimum of 512 MB of memory address space.
- The `compile` method of the `dlhdl.Workflow` object generates the memory address space for a deep learning network. See `compile`. Determine the memory address space for your deep learning network by using the `compile` method of the `dlhdl.Workflow` object.

## See Also

`registerDeepLearningTargetInterface` | `validateReferenceDesignForDeepLearning`

**Introduced in R2021b**

# registerDeepLearningTargetInterface

**Class:** `hdlcoder.ReferenceDesign`
**Package:** `hdlcoder`

Add and register a target interface

## Syntax

`registerDeepLearningTargetInterface(interfaceType)`

## Description

`registerDeepLearningTargetInterface(interfaceType)` registers a target interface to an `hdlcoder.ReferenceDesign` object. Use the registered target interface to interact with the generated deep learning processor IP core by using MATLAB.

## Input Arguments

**`interfaceType` — Target interface type**
'JTAG' (default) | 'JTAG' | string | character vector

Target interface type, specified as a string or character vector.

Example: 'JTAG'

## Limitations

To register a JTAG target interface to your reference design, your reference design must meet these requirements.

- The reference design must include JTAG MATLAB as AXI Master IP.
- The JTAG MATLAB as AXI Master IP must be one of the masters to the AXI4-Slave interface in the reference design.
- The JTAG MATLAB as AXI Master IP must have access to the same memory as the AXI4-Master interfaces in the reference design.

## See Also

`registerDeepLearningMemoryAddressSpace` | `validateReferenceDesignForDeepLearning`

**Introduced in R2021b**

# validateReferenceDesignForDeepLearning

**Class:** `hdlcoder.ReferenceDesign`
**Package:** `hdlcoder`

Checks property values in reference design object

## Syntax

```
validateReferenceDesignForDeepLearning
```

## Description

`validateReferenceDesignForDeepLearning` checks that the `hdlcoder.ReferenceDesign` object is compatible with the deep learning processor IP core generation workflow.

## Reference Design Requirements

The `validateReferenceDesign` method checks your `hdlcoder.ReferenceDesign` object for these requirements:

- Your reference design must have an AXI4 Slave interface. You use this interface to access registers in the deep learning processor IP core.
- Your reference design must have three AXI4 Master interfaces with `InterfaceID` values of `"AXI4 Master Activation Data"`, `"AXI4 Master Weight Data"`, and `"AXI4 Master Debug"`. The deep learning processor IP core uses these AXI4 Master interfaces to access memory for network storage and intermediate calculations.
- Your reference design must have a registered memory address space. Use the `registerDeepLearningMemoryAddressSpace` to register a memory address space. To run a variety of deep learning networks on your deep learning processor IP core, it is recommended to register a minimum of 512 MB of memory address space.

## Tips

Register a target interface for your reference design by using the `registerDeepLearningTargetInterface` method. You can use the registered target interface to run your deep learning network on the generated deep learning processor IP core by using MATLAB and a `dlhdl.Workflow` object.

## See Also

`registerDeepLearningMemoryAddressSpace` | `registerDeepLearningTargetInterface`

**Introduced in R2021b**