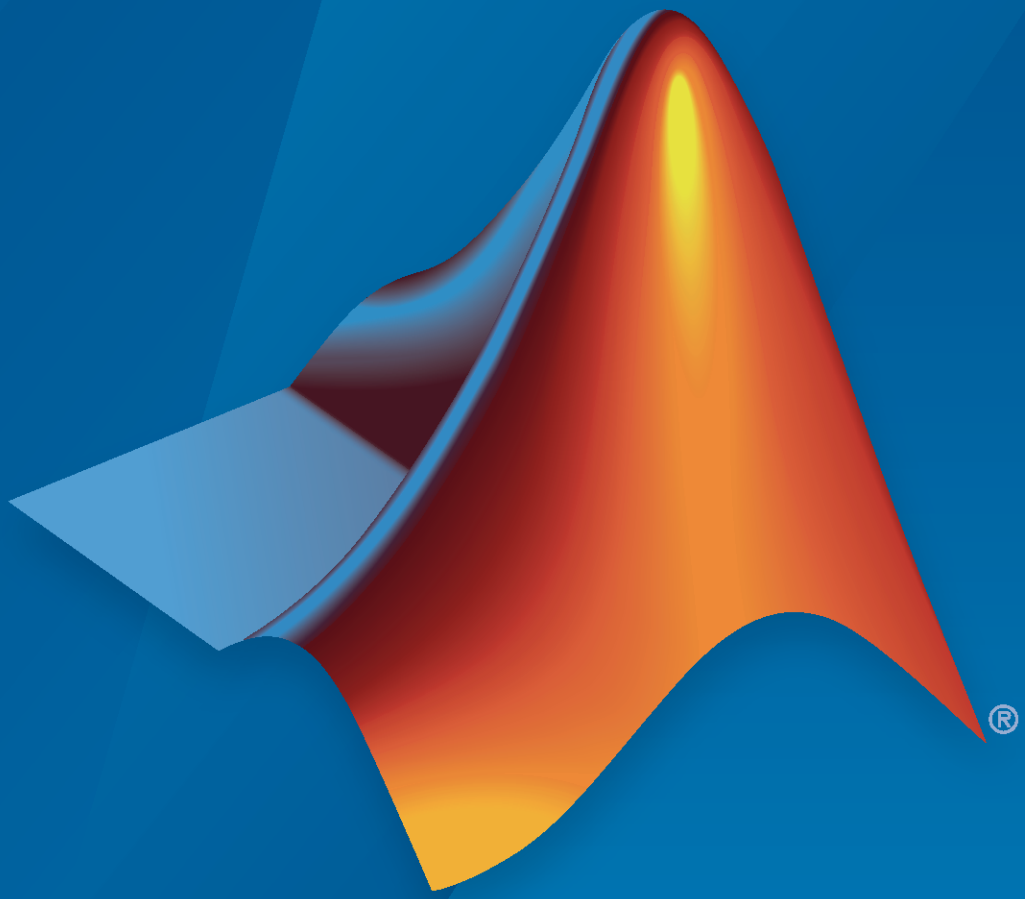
Deep Learning HDL Toolbox™

Reference



MATLAB®

R2021a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

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

Phone: 508-647-7000

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

Deep Learning HDL Toolbox™ Reference

Revision History

September 2020	Online only	New for Version 1.0 (R2020b)
March 2021	Online only	Revised for Version 1.1 (R2021a)

Contents

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

`dlhdl.Workflow` creates a workflow configuration object for you to specify the workflow to deploy your trained series network.

`dlhdl.Workflow (Name,Value)` creates a workflow configuration object for you to specify the workflow to deploy your trained deep learning network, with additional options specified by one or more name-value pair arguments.

Properties

`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.

Example: `'Bitstream','myfile.bit'` specifies that you want to deploy the trained network using your custom bitstream file `myfile.bit` which is in your current working directory.

Example: `'Bitstream','C:\myfolder\myfile.bit'` specifies that you want to deploy the trained network using your custom bitstream file `myfile.bit` that is located in the folder `'C:\myFolder'`.

`Network` — Name of the pretrained deep learning network that you want to import or the name of the quantized network object

' ' (default)

Deep learning network name specified as a variable

Example: `'network'`, `snet` creates a workflow object for the saved pretrained network, `snet`. To specify `snet`, you can import any of the existing supported pretrained networks or use transfer learning to adapt the network to your problem. For information on supported networks, see "Supported Pretrained Networks".

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 using the `dlquantizer` class. For information on supported networks, see "Supported Pretrained Networks".

Assign `VGG-19` to `snet`:

```
snet = vgg19;
```

`'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);
```

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 Using Custom Bitstream

```
snet = vgg19;
hW = dlhdl.Workflow('Network',snet,'Bitstream','myfile.bit', '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

Functions
`activations` | `compile` | `deploy` | `predict` | `getBuildInfo`

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

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

```
activations(imIn,layername)
activations(imIn,layername, Name,Value)
```

Description

`activations(imIn,layername)` returns intermediate layer activation data results for the image data in `imIn`, 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`.

`activations(imIn,layername, Name,Value)`returns intermediate layer activation data results for the image data in `imIn`, 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`.

Examples

Retrieve Layer Activation Results

Retrieve the activation results of the `LogoNet` `maxpool_3` layer for a given input image.

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

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

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


snet = getLogoNetwork();
hT = dlhdl.Target('Xilinx');
hW = dlhdl.Workflow('Network','snet','Bitstream','zcu102_single','target',hT);
image = imread('heineken.png');
inputImg = imresize(image, [227, 227]);
imIn = single(inputImg);
results = hW.activations(imIn,'maxpool_3','Profiler','on');
```

The result of the code execution is a 25-by-25-by-384 matrix for `results` and

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

```
                   LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                        -------------              -------------              ---------        ---------      ---------
Network                   32497812                    0.14772                     1          32497822             6.8
    conv_module           32497812                    0.14772
        conv_1             6953894                    0.03161
        maxpool_1          3305128                    0.01502
        conv_2            10397281                    0.04726
        maxpool_2          1207938                    0.00549
        conv_3             9267269                    0.04212
        maxpool_3          1366383                    0.00621
 * The clock frequency of the DL processor is: 220MHz
```

Input Arguments

`imIn` — Input resized image to deep learning network

single

Input image resized to match the input image layer image size of the deep learning network.

Example: To read an input image, resize it to 227x227, and convert it to single use:

Use this image to run the code:


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

Example: `imIn`

`layername` — Name of layer in deployed deep learning network

'' (default) | character vector

Name of the layer in the deployed deep learning network whose results are retrieved for the image specified in `imIn`.

The layer has to be of the type `Convolution`, `Fully Connected`, `Max Pooling`, `ReLU`, or `Dropout`. `Convolution` and `Fully Connected` layers are allowed as long as they are not followed by a `ReLU` layer.

Example: 'maxpool_3'

Name-Value Pair Arguments

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`.

`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'`

See Also
`compile` | `deploy` | `getBuildInfo` | `predict`

Introduced in R2020b

compile

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

Compile workflow object

Syntax

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

Description

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

`compile(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.

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.

```
snet = alexnet;
hT = dlhdl.Target('Xilinx');
hW = dlhdl.Workflow('network', snet, '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                   227×227×3 images with 'zerocenter' normalization
     2   'conv1'    Convolution                   96 11×11×3 convolutions with stride [4  4] and padding [0  0  0  0
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 channels per element
     5   'pool1'    Max Pooling                   3×3 max pooling with stride [2  2] and padding [0  0  0  0]
     6   'conv2'    Grouped Convolution           2 groups of 128 5×5×48 convolutions with stride [1  1] and padding
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 channels per element
     9   'pool2'    Max Pooling                   3×3 max pooling with stride [2  2] and padding [0  0  0  0]
    10   'conv3'    Convolution                   384 3×3×256 convolutions with stride [1  1] and padding [1  1  1
    11   'relu3'    ReLU                          ReLU
    12   'conv4'    Grouped Convolution           2 groups of 192 3×3×192 convolutions with stride [1  1] and paddin
    13   'relu4'    ReLU                          ReLU
    14   'conv5'    Grouped Convolution           2 groups of 128 3×3×192 convolutions with stride [1  1] and paddin
    15   'relu5'    ReLU                          ReLU
    16   'pool5'    Max Pooling                   3×3 max pooling with stride [2  2] and padding [0  0  0  0]
    17   'fc6'      Fully Connected               4096 fully connected layer
    18   'relu6'    ReLU                          ReLU
    19   'drop6'    Dropout                       50% dropout
    20   'fc7'      Fully Connected               4096 fully connected layer
    21   'relu7'    ReLU                          ReLU
    22   'drop7'    Dropout                       50% dropout
    23   'fc8'      Fully Connected               1000 fully connected layer
    24   'prob'     Softmax                       softmax
    25   'output'   Classification Output         crossentropyex with 'tench' and 999 other classes

3 Memory Regions created.

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

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

### Allocating external memory buffers:

          offset_name          offset_address     allocated_space
    ___________________________    ______________    _______________

    "InputDataOffset"           "0x00000000"     "24.0 MB"
    "OutputResultOffset"        "0x01800000"     "4.0 MB"
    "SchedulerDataOffset"       "0x01c00000"     "4.0 MB"
    "SystemBufferOffset"        "0x02000000"     "28.0 MB"
    "InstructionDataOffset"     "0x03c00000"     "4.0 MB"
    "ConvWeightDataOffset"      "0x04000000"     "16.0 MB"
    "FCWeightDataOffset"        "0x05000000"     "224.0 MB"
    "EndOffset"                 "0x13000000"     "Total: 304.0 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.

   ```
   snet = resnet18;
   hTarget = dlhdl.Target('Xilinx');
   hW = dlhdl.Workflow('N',snet,'B','zcu102_single','T',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]
```

Input Arguments

Name-Value Pair Arguments

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`.

`InputFrameNumberLimit` — Maximum input frame number limit
integer

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

Example: `'InputFrameNumberLimit',30`

See Also

`deploy` | `getBuildInfo` | `predict`

Topics

“Use Compiler Output for System Integration”

Introduced in R2020b

deploy

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

Deploy the specified neural network to the target FPGA board

Syntax

`deploy`

Description

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

Examples

Deploy LogoNet to Intel Arria 10 SoC Development Kit

Note Before you run the `deploy` function, make sure that your host computer is connected to the Intel Arria 10 SoC board. For more information, see “Check Host Computer Connection to FPGA Boards”

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

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

### 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
```

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

Deploy Quantized LogoNet to Xilinx ZCU102 Development Kit

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
```

Use this image to run the code:

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

```
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);
hW.deploy

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

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


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

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

Introduced in R2020b

getBuildInfo

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

Retrieve bitstream resource utilization

Syntax

```
area = getBuildInfo
```

Description

`area = getBuildInfo` returns a structure containing the bitstream resource utilization.

Examples

Retrieve `arria10soc_single`Bitstream Resource Utilization

Retrieve the resource utilization for the `arria10soc_single` bitstream.

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
```

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');
```

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) utiliz


area =

  struct with fields:
```

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

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.

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(imds)
predict(imds, Name,Value)
```

Description

`predict(imds)` 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(imds, 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.

Examples

Predict Outcome and Profile Results

Note Before you run the `predict` function, make sure that your host computer is connected to the target device board. For more information, see “Configure Board-Specific Setup Information” .

Use this image to run the code:

```
% 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);

% 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);
% 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}


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


             Deep Learning Processor Profiler Performance Results

                     LastLayerLatency(cycles)   LastLayerLatency(seconds)       FramesNum      Total Latency     Frames/s
                           -------------              -------------               ---------        ---------       ---------
Network                    166206640                   1.10804                       1          166206873             0.9
    conv_module            156100737                   1.04067
        conv1_1              2174602                   0.01450
        conv1_2             15580687                   0.10387
        pool1                1976185                   0.01317
        conv2_1              7534356                   0.05023
        conv2_2             14623885                   0.09749
        pool2                1171628                   0.00781
        conv3_1              7540868                   0.05027
        conv3_2             14093791                   0.09396
        conv3_3             14093717                   0.09396
        conv3_4             14094381                   0.09396
        pool3                 766669                   0.00511
        conv4_1              6999620                   0.04666
        conv4_2             13725380                   0.09150
        conv4_3             13724671                   0.09150
        conv4_4             13725125                   0.09150
        pool4                 465360                   0.00310
        conv5_1              3424060                   0.02283
        conv5_2              3423759                   0.02283
        conv5_3              3424758                   0.02283
        conv5_4              3424461                   0.02283
        pool5                 113010                   0.00075
    fc_module              10105903                   0.06737
        fc6                 8397997                   0.05599
        fc7                 1370215                   0.00913
        fc8                  337689                   0.00225
 * The clock frequency of the DL processor is: 150MHz


ans =

    'zebra'
```

Obtain Prediction Results for Quantized LogoNet Network

Note Before you run the `predict` function, make sure that your host computer is connected to the target device board. For more information, see “Configure Board-Specific Setup Information” .

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
```

Use this image to run the code:

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');
Image = imageDatastore('heineken.png','Labels','Heineken');
dlquantObj.calibrate(Image);

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

% Load input images and resize them according to the network specifications
image = imread('heineken.png');
inputImg = imresize(image, [227, 227]);
imshow(inputImg);
imIn = single(inputImg);
% 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}

### Loading weights to FC Processor.
### FC Weights loaded. Current time is 12-Jun-2020 16:55:34
### 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                   13604105                 0.04535                     1           13604146            22.1
    conv_module           12033763                 0.04011
        conv_1             3339984                 0.01113
        maxpool_1          1490805                 0.00497
        conv_2             2866483                 0.00955
        maxpool_2           574102                 0.00191
        conv_3             2432474                 0.00811
        maxpool_3           700552                 0.00234
        conv_4              617505                 0.00206
        maxpool_4            11951                 0.00004
    fc_module              1570342                 0.00523
        fc_1                937715                 0.00313
        fc_2                599341                 0.00200
        fc_3                 33284                 0.00011
 * The clock frequency of the DL processor is: 300MHz
```

Input Arguments

`imds` — Input resized image to deep learning network

single

Input image resized to match the image input layer size of the deep learning network.

Example: To read an input image, resize it to 227x227, and convert it to single use:

Use this image to run the code:


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

Example: `imIn`

Name-Value Pair Arguments

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`.

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'`

See Also

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

Target device vendor name, specified as a character vector.

Example: `'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'`

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.

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
```

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
```

Description

`release` releases the connection to the target board.

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.

```
snet = alexnet;
hW = dlhdl.Workflow('Network',snet,'BitstreamName','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, [227, 227]);
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

Validate SSH connection and deployed bitstream

Syntax

`validateConnection`

Description

`validateConnection`:

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.

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,'BitstreamName','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.

```
snet = alexnet;
hW = dlhdl.Workflow('Network',snet,'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

`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) | 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'

Processing Module conv Properties

`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.

`KernelDataType` — Adder module kernel data type
single (default) | int8 | character vector

This parameter is a character vector that represents the module kernel data type.

Processing Module fc Properties

`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.

`KernelDataType` — Adder module kernel data type
single (default) | int8 | character vector

This parameter is a character vector that represents the module kernel data type.

Processing Module adder Properties

`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.

`KernelDataType` — Adder module kernel data type
single (default) | int8 | character vector

This parameter is a character vector that represents the module kernel data type.

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"
                           ConvThreadNumber: 16
                            InputMemorySize: [227  227    3]
                           OutputMemorySize: [227  227    3]
                           FeatureSizeLimit: 2048
                             KernelDataType: 'single'

                      Processing Module "fc"
                              FCThreadNumber: 4
                            InputMemorySize: 25088
                           OutputMemorySize: 4096
                             KernelDataType: 'single'

                   Processing Module "adder"
                            InputMemorySize: 40
                           OutputMemorySize: 40
                             KernelDataType: '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"
                           ConvThreadNumber: 16
                            InputMemorySize: [227  227    3]
                           OutputMemorySize: [227  227    3]
                           FeatureSizeLimit: 2048
                             KernelDataType: 'single'

                     Processing Module "fc"
                            FCThreadNumber: 4
                            InputMemorySize: 25088
                           OutputMemorySize: 4096
                             KernelDataType: 'single'

                 Processing Module "adder"
                            InputMemorySize: 40
                           OutputMemorySize: 40
                             KernelDataType: '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 UltraScale+'
                    SynthesisToolDeviceName: 'xczu9eg-ffvb1156-2-e'
                   SynthesisToolPackageName: ''
                    SynthesisToolSpeedValue: ''
```

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"
                         ConvThreadNumber: 16
                          InputMemorySize: [227  227    3]
                         OutputMemorySize: [227  227    3]
                         FeatureSizeLimit: 2048
                           KernelDataType: 'single'

                   Processing Module "fc"
                          FCThreadNumber: 4
                          InputMemorySize: 25088
                         OutputMemorySize: 4096
                           KernelDataType: 'single'

               Processing Module "adder"
                          InputMemorySize: 40
```

```
        OutputMemorySize: 40
          KernelDataType: '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: ''
```

See Also

Functions

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

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(network)
performance = estimatePerformance(network)
```

Description

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

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

Examples

Estimate Performance of LogoNet Network

Calculate the LogoNet network performance and layer-level latencies for the `hPC` `ProcessorConfig` object.

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
```

Create a `dlhdl.ProcessorConfig` object.

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

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

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

```
3 Memory Regions created.


             Deep Learning Processor Estimator Performance Results

                  LastFrameLatency(cycles)   LastFrameLatency(seconds)       FramesNum      Total Latency     Frames/s
                       -------------             -------------               ---------        ---------      ---------
Network                   39853460                  0.19927                     1           39853460             5.0
    ____conv_1             6825287                  0.03413
```

```
    ____maxpool_1          3755088                 0.01878
    ____conv_2            10440701                 0.05220
    ____maxpool_2          1447840                 0.00724
    ____conv_3             9393397                 0.04697
    ____maxpool_3          1765856                 0.00883
    ____conv_4             1770484                 0.00885
    ____maxpool_4            28098                 0.00014
    ____fc_1               2644884                 0.01322
    ____fc_2               1692532                 0.00846
    ____fc_3                 89293                 0.00045
 * The clock frequency of the DL processor is: 200MHz
```

Input Arguments

`network` — Network object

`SeriesNetwork` object | `DAGNetwork` object | `yolov2ObjectDetector` object | `dlquantizer` object

Name of network object for performance estimate.

Example: `estimatePerformance(snet)`

Output Arguments

`performance` — Network object performance

`table`

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

Tips

To obtain the performance estimation for a `dlquantizer` object, set the `dlhdl.ProcessorConfig` object `KernelDataType` data type to int8 for the conv, fc, and adder modules.

See Also

`estimateResources` | `getModuleProperty` | `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
resources = estimateResources
estimateResources('Name','Value')
resources = estimateResources('Name','Value')
```

Description

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

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

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

Examples

Estimate Resources Used by Default Custom Processor Configuration

Calculate the resources used by the default custom bitstream processor configuration object.

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

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

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

```
hPC.estimateResources;
```

Calling `estimateResources` returns these results:

```
             Deep Learning Processor Estimator Resource Results

                           DSPs          Block RAM*
                      -------------    -------------
DL_Processor                   368              508
    conv_module                343              459
    fc_module                   17               34
    adder_module                 8                6
    debug_module                 0                8
    sched_module                 0                1
```

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

Estimate LUT Resource Used by the Default Custom Processor Configuration

Calculate the LUT resources used by the default custom bitstream processor configuration object.

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

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

To retrieve the LUT resources utilized by the custom bitstream configuration, call the `estimateResources` method with `'LUT',true` as the name-value argument.

```
hPC.estimateResources('LUT',true);
```

```
               Deep Learning Processor Estimator Resource Results

                         DSPs          Block RAM*     LUTs(CLB/ALUT)
                     ------------    -------------    -------------
DL_Processor                368              508           207126
    conv_module             343              459
    fc_module                17               34
    adder_module              8                6
    debug_module              0                8
    sched_module              0                1
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

Input Arguments

Name-Value Pair Arguments

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`.

LUT — Display lookup table (LUT) resource utilization

false (default) | true | logical

Parameter that enables or disables the display of LUT resources used by the custom bitstream configuration.

Example: `'LUT',true`

Output Arguments

`resources` — Custom processor configuration object resource utilization

`table`

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

Tips

To obtain the resources used by the custom bitstream configuration for a different chip family object, set the `dlhdl.ProcessorConfig` object `SynthesisToolChipFamily` value to a different family. For a list of supported device families, see “SynthesisToolChipFamily” on page 1-0 .

```
hPC = dlhdl.ProcessorConfig;
hPC.SynthesisToolChipFamily ='Kintex7'
hPC.estimateResources
```

```
             Deep Learning Processor Estimator Resource Results

                           DSPs          Block RAM*
                       -------------    -------------
DL_Processor                    368              508
    conv_module                 343              459
    fc_module                    17               34
    adder_module                  8                6
    debug_module                  0                8
    sched_module                  0                1
* Block RAM represents Block RAM tiles in Xilinx devices and Block RAM bits in Intel devices
```

See Also

`estimatePerformance` | `getModuleProperty` | `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(ModuleName,ModulePropertyName)
```

Description

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

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
```

Retrieve `KernelDataType` for `adder` 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('adder','KernelDataType')
```

2 Once you execute the code, the result is:

```
ans =

    'single'
```

Input Arguments

`ModuleName` — Name of the module whose parameters are to be retrieved

'' (default) | 'conv' | 'fc' | 'adder' | character vector

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

`ModulePropertyName` — Name of the module property whose value is to be retrieved

character vector

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

Example: 'ConvThreadNumber'

This table lists module names and module property names.

Module Name	Module Property Name
`conv`	“ConvThreadNumber” on page 1-0
`conv`	“InputMemorySize” on page 1-0
`conv`	“OutputMemorySize” on page 1-0
`conv`	“FeatureSizeLimit” on page 1-0
`conv`	“KernelDataType” on page 1-0
`fc`	“FCThreadNumber” on page 1-0
`fc`	“InputMemorySize” on page 1-0
`fc`	“OutputMemorySize” on page 1-0
`fc`	“KernelDataType” on page 1-0
`adder`	“InputMemorySize” on page 1-0
`adder`	“OutputMemorySize” on page 1-0
`adder`	“KernelDataType” on page 1-0

See Also

`estimatePerformance` | `estimateResources` | `setModuleProperty`

Topics

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

Introduced in R2020b

setModuleProperty

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

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

Syntax

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

Description

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

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"
                           ConvThreadNumber: 25
                            InputMemorySize: [227  227    3]
                           OutputMemorySize: [227  227    3]
                           FeatureSizeLimit: 2048
                             KernelDataType: 'single'

                      Processing Module "fc"
                              FCThreadNumber: 4
                            InputMemorySize: 25088
                           OutputMemorySize: 4096
                             KernelDataType: 'single'

                   Processing Module "adder"
                            InputMemorySize: 40
                           OutputMemorySize: 40
                             KernelDataType: '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"
                           ConvThreadNumber: 16
                            InputMemorySize: [227  227    3]
                           OutputMemorySize: [227  227    3]
                           FeatureSizeLimit: 2048
                             KernelDataType: 'single'

                      Processing Module "fc"
                               FCThreadNumber: 4
                            InputMemorySize: 25060
                           OutputMemorySize: 4096
                             KernelDataType: 'single'

                   Processing Module "adder"
                            InputMemorySize: 40
                           OutputMemorySize: 40
                             KernelDataType: '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('adder','InputMemorySize',80)
hPC
```

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

```
hPC =

                    Processing Module "conv"
                           ConvThreadNumber: 16
                            InputMemorySize: [227  227    3]
                           OutputMemorySize: [227  227    3]
                           FeatureSizeLimit: 2048
                             KernelDataType: 'single'

                      Processing Module "fc"
                               FCThreadNumber: 4
                            InputMemorySize: 25088
                           OutputMemorySize: 4096
                             KernelDataType: 'single'

                   Processing Module "adder"
                            InputMemorySize: 80
                           OutputMemorySize: 40
                             KernelDataType: '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: ''
```

Input Arguments

`ModuleName` — Name of the module whose parameters are to be set

'' (default) | 'conv' | 'fc' | 'adder' | character vector

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

Name-Value Arguments

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`.

Example:

conv module parameters

`ConvThreadNumber` — Number of parallel convolution processor kernel threads

16 (default) | 4 | 9 | 16 | 25 | 36 | 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

512 (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`

`KernelDataType` — Adder module kernel data type
single (default) | int8 | character vector

This parameter is a character vector that represents the module kernel data type.

Example: `'KernelDataType', 'int8'`

fc module parameters

`FCThreadNumber` — Number of parallel fully connected (fc) MAC threads
4 (default) | 4 | 8 | 16 | 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
9216 (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`

`KernelDataType` — Adder module kernel data type
single (default) | int8 | character vector

This parameter is a character vector that represents the module kernel data type.

Example: `'KernelDataType', 'int8'`

adder module properties

`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`

`KernelDataType` — Adder module kernel data type
single (default) | int8 | character vector

This parameter is a character vector that represents the module kernel data type.

Example: `'KernelDataType', 'int8'`

See Also

`estimatePerformance` | `estimateResources` | `getModuleProperty`

Topics
“Deep Learning Processor Architecture”
“Estimate Performance of Deep Learning Network”
“Estimate Resource Utilization for Custom Processor Configuration”

Introduced in R2020b

dlhdl.buildProcessor

Build and generate custom processor IP

Syntax

```
dlhdl.buildProcessor
dlhdl.buildProcessor(processorconfigobject)
```

Description

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

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

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.builProcessor(hPC)
```

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)`

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

`setenv` | `startup`

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 `ssdObjectDetector` networks requires the `ExecutionEnvironment` property to be set to `'FPGA'`.

Properties

`NetworkObject` — 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 `ssdObjectDetector` networks requires the `ExecutionEnvironment` property to be set to `'FPGA'`.

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

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

Examples

Specify FPGA Execution Environment

- This example shows how to specify an FPGA execution environment.

```
net = vgg19;
quantobj = dlquantizer(net,'ExecutionEnvironment','FPGA');
```

Quantize a Neural Network

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

```
net
```

```
net =

  DAGNetwork with properties:
```

```
        Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 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. Save this function in a local file.

```
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 =

  95x5 table

                Optimized Layer Name                    Network Layer Name       Learnables / Activations     MinVa
    ____________________________________________    ____________________    ____________________________    ______

    {'conv1_relu_conv1_Weights'                  }    {'relu_conv1'           }         "Weights"                -0.9
    {'conv1_relu_conv1_Bias'                     }    {'relu_conv1'           }         "Bias"                   -0.0
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Weights'}    {'fire2-relu_squeeze1x1'}         "Weights"                   -
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Bias'   }    {'fire2-relu_squeeze1x1'}         "Bias"                   -0.1
    {'fire2-expand1x1_fire2-relu_expand1x1_Weights'  }    {'fire2-relu_expand1x1' }         "Weights"                 -0.
    {'fire2-expand1x1_fire2-relu_expand1x1_Bias'     }    {'fire2-relu_expand1x1' }         "Bias"                  -0.06
    {'fire2-expand3x3_fire2-relu_expand3x3_Weights'  }    {'fire2-relu_expand3x3' }         "Weights"                -0.7
    {'fire2-expand3x3_fire2-relu_expand3x3_Bias'     }    {'fire2-relu_expand3x3' }         "Bias"                  -0.05
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Weights'}    {'fire3-relu_squeeze1x1'}         "Weights"                -0.7
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Bias'   }    {'fire3-relu_squeeze1x1'}         "Bias"                   -0.1
    {'fire3-expand1x1_fire3-relu_expand1x1_Weights'  }    {'fire3-relu_expand1x1' }         "Weights"                -0.7
    {'fire3-expand1x1_fire3-relu_expand1x1_Bias'     }    {'fire3-relu_expand1x1' }         "Bias"                  -0.06
    {'fire3-expand3x3_fire3-relu_expand3x3_Weights'  }    {'fire3-relu_expand3x3' }         "Weights"                -0.6
    {'fire3-expand3x3_fire3-relu_expand3x3_Bias'     }    {'fire3-relu_expand3x3' }         "Bias"                  -0.05
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Weights'}    {'fire4-relu_squeeze1x1'}         "Weights"                -0.7
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Bias'   }    {'fire4-relu_squeeze1x1'}         "Bias"                   -0.1
...
```

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: [1x1 struct]
```

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

```
valResults.MetricResults.Result
```

```
ans =

  2x3 table

    NetworkImplementation    MetricOutput    LearnableParameterMemory(bytes)
    _____________________    ____________    _______________________________

     {'Floating-Point'}           1                   2.9003e+06
     {'Quantized'     }           1                   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 Execution Environment

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and explore the behavior of the quantized network. In this example, you quantize the `LogoNet` neural network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results from the target device.

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 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.

```
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.

Load the network to quantize into the base workspace.

```
net
```

```
net =

  DAGNetwork with properties:

         Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 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);
```

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.

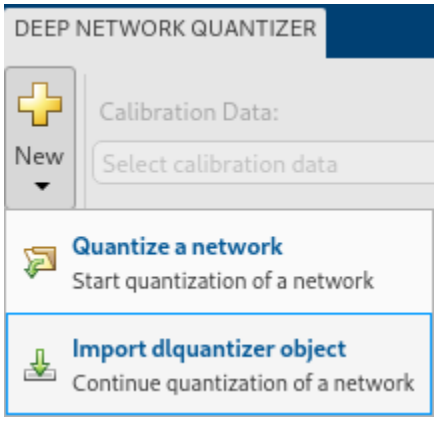
```
calResults = calibrate(quantObj, aug_calData)
```

```
calResults =

  95x5 table

                Optimized Layer Name                      Network Layer Name       Learnables / Activations     MinVa
    ____________________________________________    ____________________________    ________________________    ______

    {'conv1_relu_conv1_Weights'                   }    {'relu_conv1'           }        "Weights"                -0.9
    {'conv1_relu_conv1_Bias'                      }    {'relu_conv1'           }        "Bias"                   -0.0
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Weights'}    {'fire2-relu_squeeze1x1'}        "Weights"                   -
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Bias'   }    {'fire2-relu_squeeze1x1'}        "Bias"                   -0.1
    {'fire2-expand1x1_fire2-relu_expand1x1_Weights'  }    {'fire2-relu_expand1x1' }        "Weights"                 -0
    {'fire2-expand1x1_fire2-relu_expand1x1_Bias'     }    {'fire2-relu_expand1x1' }        "Bias"                 -0.06
    {'fire2-expand3x3_fire2-relu_expand3x3_Weights'  }    {'fire2-relu_expand3x3' }        "Weights"                -0.7
    {'fire2-expand3x3_fire2-relu_expand3x3_Bias'     }    {'fire2-relu_expand3x3' }        "Bias"                 -0.05
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Weights'}    {'fire3-relu_squeeze1x1'}        "Weights"                -0.7
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Bias'   }    {'fire3-relu_squeeze1x1'}        "Bias"                   -0.1
    {'fire3-expand1x1_fire3-relu_expand1x1_Weights'  }    {'fire3-relu_expand1x1' }        "Weights"                -0.7
    {'fire3-expand1x1_fire3-relu_expand1x1_Bias'     }    {'fire3-relu_expand1x1' }        "Bias"                 -0.06
    {'fire3-expand3x3_fire3-relu_expand3x3_Weights'  }    {'fire3-relu_expand3x3' }        "Weights"                -0.6
    {'fire3-expand3x3_fire3-relu_expand3x3_Bias'     }    {'fire3-relu_expand3x3' }        "Bias"                 -0.05
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Weights'}    {'fire4-relu_squeeze1x1'}        "Weights"                -0.7
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Bias'   }    {'fire4-relu_squeeze1x1'}        "Bias"                   -0.1
...
```

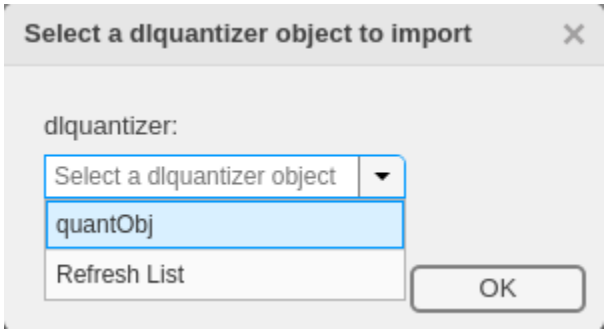
Open the **Deep Network Quantizer** app.

```
deepNetworkQuantizer
```

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

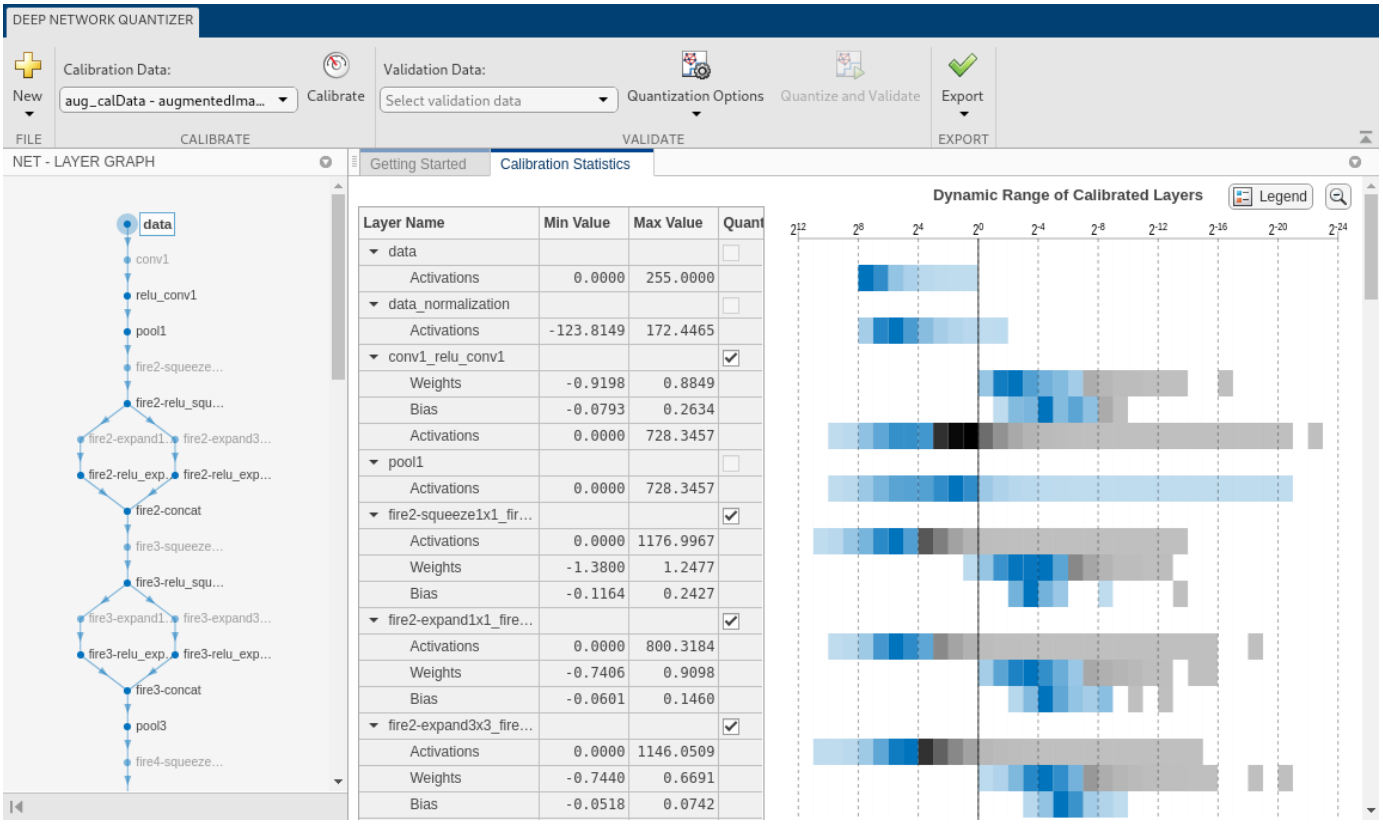


In the dialog, select the `dlquantizer` object to import from the base workspace.



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 = vaidationData.subset(1:1);
```

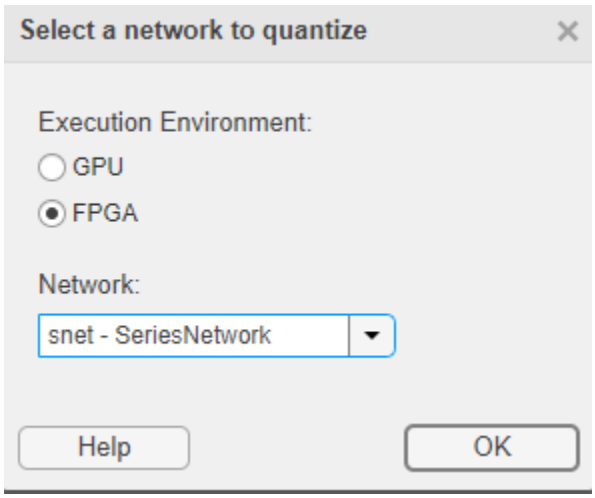
At the MATLAB command prompt, open the Deep Network Quantizer app.

```
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`.



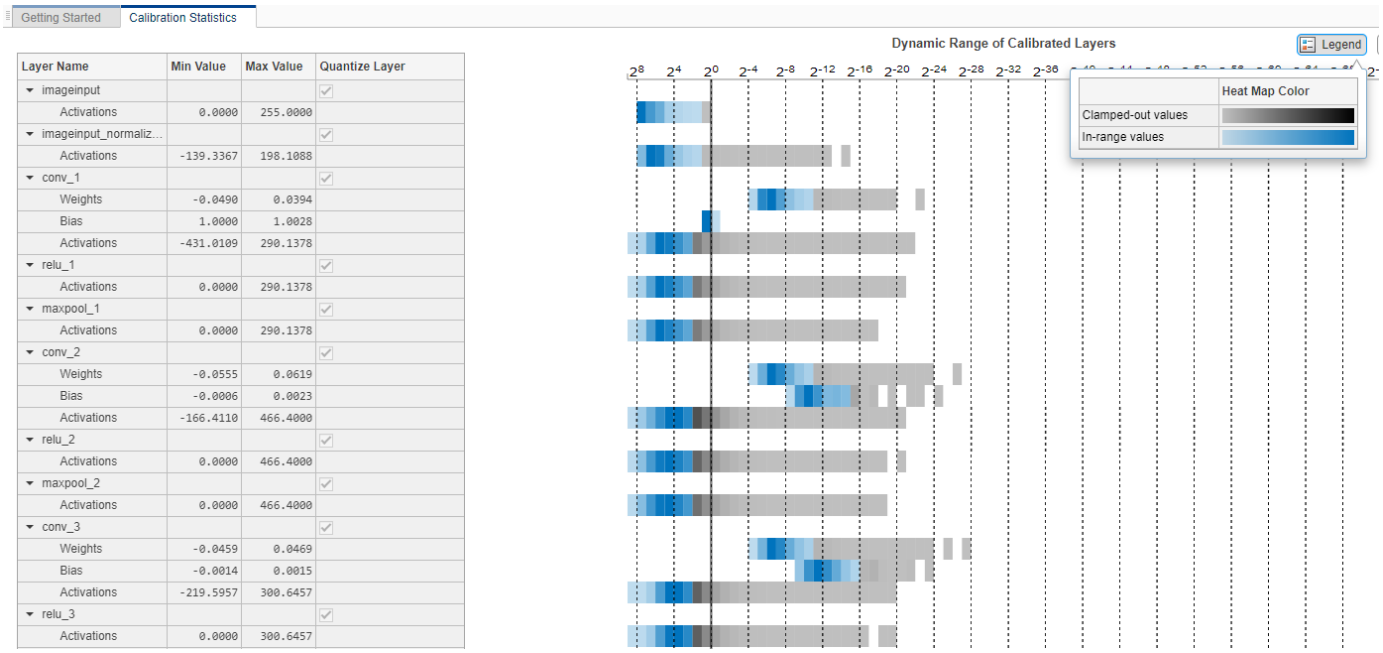
The app displays the layer graph of the selected network.

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.

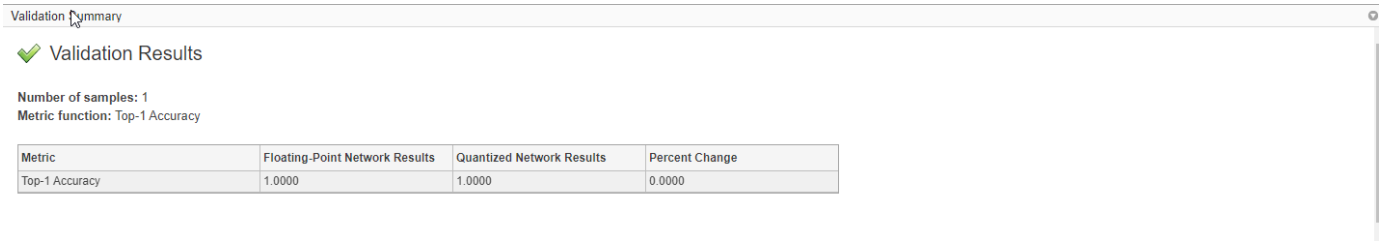
Click **Quantize and Validate**.

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



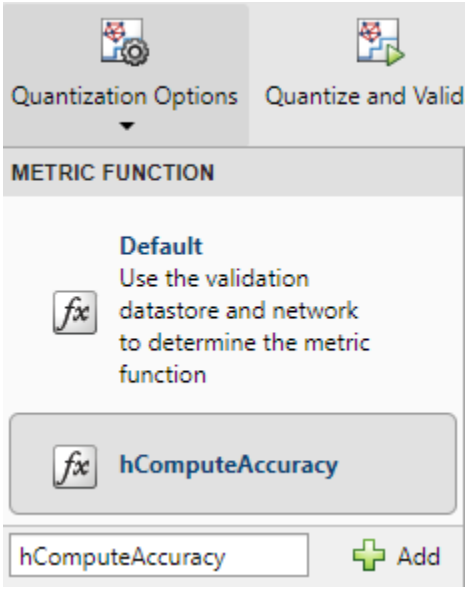
If you want to use a different metric function for validation, for example to use the Top-5 accuracy metric function instead of the default Top-1 accuracy metric function, you can define a custom metric function. Save this function in a local file.

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

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

    % Compare 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
```

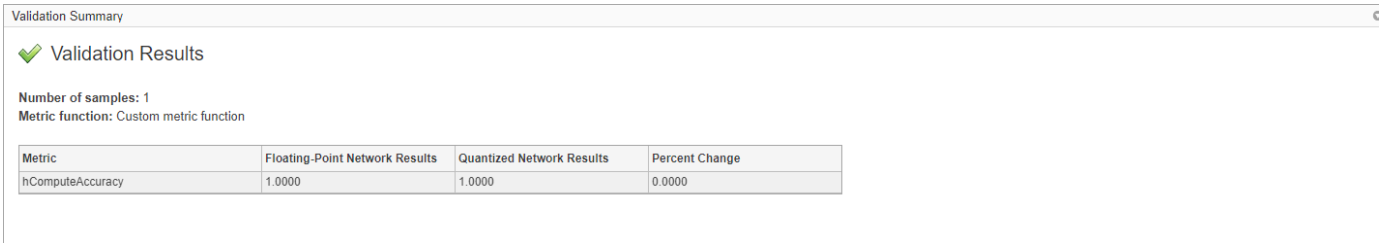
To revalidate the network by using this custom metric function, under **Quantization Options**, enter the name of the custom metric function `hComputeAccuracy`. Select **Add** to add `hComputeAccuracy` to the list of metric functions available in the app. Select `hComputeAccuracy` as the metric function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



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”.

If the performance of the quantized network is not satisfactory, you can choose to not quantize some layers by clearing the layer in the table. Click **Quantize and Validate** again.

See Also

Apps
Deep Network Quantizer

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

Topics
“Quantization of Deep Neural Networks”

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 as `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

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

```
net
```

```
net =

  DAGNetwork with properties:

         Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 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. Save this function in a local file.

```
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 =

  95x5 table

                Optimized Layer Name                    Network Layer Name       Learnables / Activations     MinVa
    ____________________________________________    ____________________    ________________________    ______

    {'conv1_relu_conv1_Weights'                  }    {'relu_conv1'          }        "Weights"               -0.9
    {'conv1_relu_conv1_Bias'                     }    {'relu_conv1'          }        "Bias"                  -0.0
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Weights'}    {'fire2-relu_squeeze1x1'}        "Weights"
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Bias'   }    {'fire2-relu_squeeze1x1'}        "Bias"                  -0.1
    {'fire2-expand1x1_fire2-relu_expand1x1_Weights'  }    {'fire2-relu_expand1x1' }        "Weights"                -0
    {'fire2-expand1x1_fire2-relu_expand1x1_Bias'     }    {'fire2-relu_expand1x1' }        "Bias"                  -0.00
    {'fire2-expand3x3_fire2-relu_expand3x3_Weights'  }    {'fire2-relu_expand3x3' }        "Weights"               -0.7
    {'fire2-expand3x3_fire2-relu_expand3x3_Bias'     }    {'fire2-relu_expand3x3' }        "Bias"                  -0.05
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Weights'}    {'fire3-relu_squeeze1x1'}        "Weights"               -0.7
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Bias'   }    {'fire3-relu_squeeze1x1'}        "Bias"                  -0.1
    {'fire3-expand1x1_fire3-relu_expand1x1_Weights'  }    {'fire3-relu_expand1x1' }        "Weights"               -0.7
    {'fire3-expand1x1_fire3-relu_expand1x1_Bias'     }    {'fire3-relu_expand1x1' }        "Bias"                  -0.00
    {'fire3-expand3x3_fire3-relu_expand3x3_Weights'  }    {'fire3-relu_expand3x3' }        "Weights"               -0.6
    {'fire3-expand3x3_fire3-relu_expand3x3_Bias'     }    {'fire3-relu_expand3x3' }        "Bias"                  -0.05
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Weights'}    {'fire4-relu_squeeze1x1'}        "Weights"               -0.7
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Bias'   }    {'fire4-relu_squeeze1x1'}        "Bias"                  -0.1
...
```

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: [1x1 struct]
```

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

```
valResults.MetricResults.Result
```

```
ans =

  2x3 table

    NetworkImplementation    MetricOutput    LearnableParameterMemory(bytes)
    _____________________    ____________    _______________________________

     {'Floating-Point'}           1                   2.9003e+06
     {'Quantized'     }           1                   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 Execution Environment

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and explore the behavior of the quantized network. In this example, you quantize the `LogoNet` neural network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results from the target device.

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 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.

```
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` | `dlquantizer` | `validate`

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

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

```
net
```

```
net =

  DAGNetwork with properties:

         Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 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. Save this function in a local file.

```
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 =

  95x5 table

                Optimized Layer Name                    Network Layer Name       Learnables / Activations     MinVa
```

```
    ____________________________________________________    ____________________________    ____________________________    _____

    {'conv1_relu_conv1_Weights'                       }    {'relu_conv1'           }         "Weights"                -0.9
    {'conv1_relu_conv1_Bias'                          }    {'relu_conv1'           }         "Bias"                   -0.0
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Weights'}    {'fire2-relu_squeeze1x1'}         "Weights"                    -
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Bias'   }    {'fire2-relu_squeeze1x1'}         "Bias"                   -0.1
    {'fire2-expand1x1_fire2-relu_expand1x1_Weights'  }    {'fire2-relu_expand1x1' }         "Weights"                 -0.
    {'fire2-expand1x1_fire2-relu_expand1x1_Bias'     }    {'fire2-relu_expand1x1' }         "Bias"                  -0.06
    {'fire2-expand3x3_fire2-relu_expand3x3_Weights'  }    {'fire2-relu_expand3x3' }         "Weights"                -0.7
    {'fire2-expand3x3_fire2-relu_expand3x3_Bias'     }    {'fire2-relu_expand3x3' }         "Bias"                  -0.05
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Weights'}    {'fire3-relu_squeeze1x1'}         "Weights"                -0.7
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Bias'   }    {'fire3-relu_squeeze1x1'}         "Bias"                   -0.1
    {'fire3-expand1x1_fire3-relu_expand1x1_Weights'  }    {'fire3-relu_expand1x1' }         "Weights"                -0.7
    {'fire3-expand1x1_fire3-relu_expand1x1_Bias'     }    {'fire3-relu_expand1x1' }         "Bias"                  -0.06
    {'fire3-expand3x3_fire3-relu_expand3x3_Weights'  }    {'fire3-relu_expand3x3' }         "Weights"                -0.6
    {'fire3-expand3x3_fire3-relu_expand3x3_Bias'     }    {'fire3-relu_expand3x3' }         "Bias"                  -0.05
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Weights'}    {'fire4-relu_squeeze1x1'}         "Weights"                -0.7
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Bias'   }    {'fire4-relu_squeeze1x1'}         "Bias"                   -0.1
...
```

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: [1x1 struct]
```

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

```
valResults.MetricResults.Result
```

```
ans =

  2x3 table

    NetworkImplementation    MetricOutput    LearnableParameterMemory(bytes)
    _____________________    ____________    _______________________________

     {'Floating-Point'}           1                   2.9003e+06
     {'Quantized'     }           1                   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 Execution Environment

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and explore the behavior of the quantized network. In this example, you quantize the `LogoNet` neural network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results from the target device.

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 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.

```
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 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`.

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
`dlquantizationOptions` | `dlquantizer` | `validate`

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

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

```
net
```

```
net =

  DAGNetwork with properties:

         Layers: [68x1 nnet.cnn.layer.Layer]
    Connections: [75x2 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. Save this function in a local file.

```
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 =

  95x5 table

                Optimized Layer Name                    Network Layer Name       Learnables / Activations     MinVa
    ____________________________________________    ____________________________    ____________________________    ____

    {'conv1_relu_conv1_Weights'                 }    {'relu_conv1'           }        "Weights"                -0.9
```

```
    {'conv1_relu_conv1_Bias'                        }    {'relu_conv1'           }        "Bias"                 -0.0
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Weights'}    {'fire2-relu_squeeze1x1'}        "Weights"                 -
    {'fire2-squeeze1x1_fire2-relu_squeeze1x1_Bias'   }    {'fire2-relu_squeeze1x1'}        "Bias"                 -0.1
    {'fire2-expand1x1_fire2-relu_expand1x1_Weights'  }    {'fire2-relu_expand1x1' }        "Weights"               -0.
    {'fire2-expand1x1_fire2-relu_expand1x1_Bias'     }    {'fire2-relu_expand1x1' }        "Bias"                -0.00
    {'fire2-expand3x3_fire2-relu_expand3x3_Weights'  }    {'fire2-relu_expand3x3' }        "Weights"              -0.7
    {'fire2-expand3x3_fire2-relu_expand3x3_Bias'     }    {'fire2-relu_expand3x3' }        "Bias"                -0.05
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Weights'}    {'fire3-relu_squeeze1x1'}        "Weights"              -0.7
    {'fire3-squeeze1x1_fire3-relu_squeeze1x1_Bias'   }    {'fire3-relu_squeeze1x1'}        "Bias"                 -0.1
    {'fire3-expand1x1_fire3-relu_expand1x1_Weights'  }    {'fire3-relu_expand1x1' }        "Weights"              -0.7
    {'fire3-expand1x1_fire3-relu_expand1x1_Bias'     }    {'fire3-relu_expand1x1' }        "Bias"                -0.00
    {'fire3-expand3x3_fire3-relu_expand3x3_Weights'  }    {'fire3-relu_expand3x3' }        "Weights"              -0.6
    {'fire3-expand3x3_fire3-relu_expand3x3_Bias'     }    {'fire3-relu_expand3x3' }        "Bias"                -0.05
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Weights'}    {'fire4-relu_squeeze1x1'}        "Weights"              -0.7
    {'fire4-squeeze1x1_fire4-relu_squeeze1x1_Bias'   }    {'fire4-relu_squeeze1x1'}        "Bias"                 -0.1
...
```

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: [1x1 struct]
```

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

```
valResults.MetricResults.Result
```

```
ans =

  2x3 table

    NetworkImplementation    MetricOutput    LearnableParameterMemory(bytes)
    _____________________    ____________    _______________________________

     {'Floating-Point'}           1                   2.9003e+06
     {'Quantized'     }           1                   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 Execution Environment

This example shows how to quantize learnable parameters in the convolution layers of a neural network, and explore the behavior of the quantized network. In this example, you quantize the `LogoNet` neural network. Quantization helps reduce the memory requirement of a deep neural network by quantizing weights, biases and activations of network layers to 8-bit scaled integer data types. Use MATLAB® to retrieve the prediction results from the target device.

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 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.

```
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.

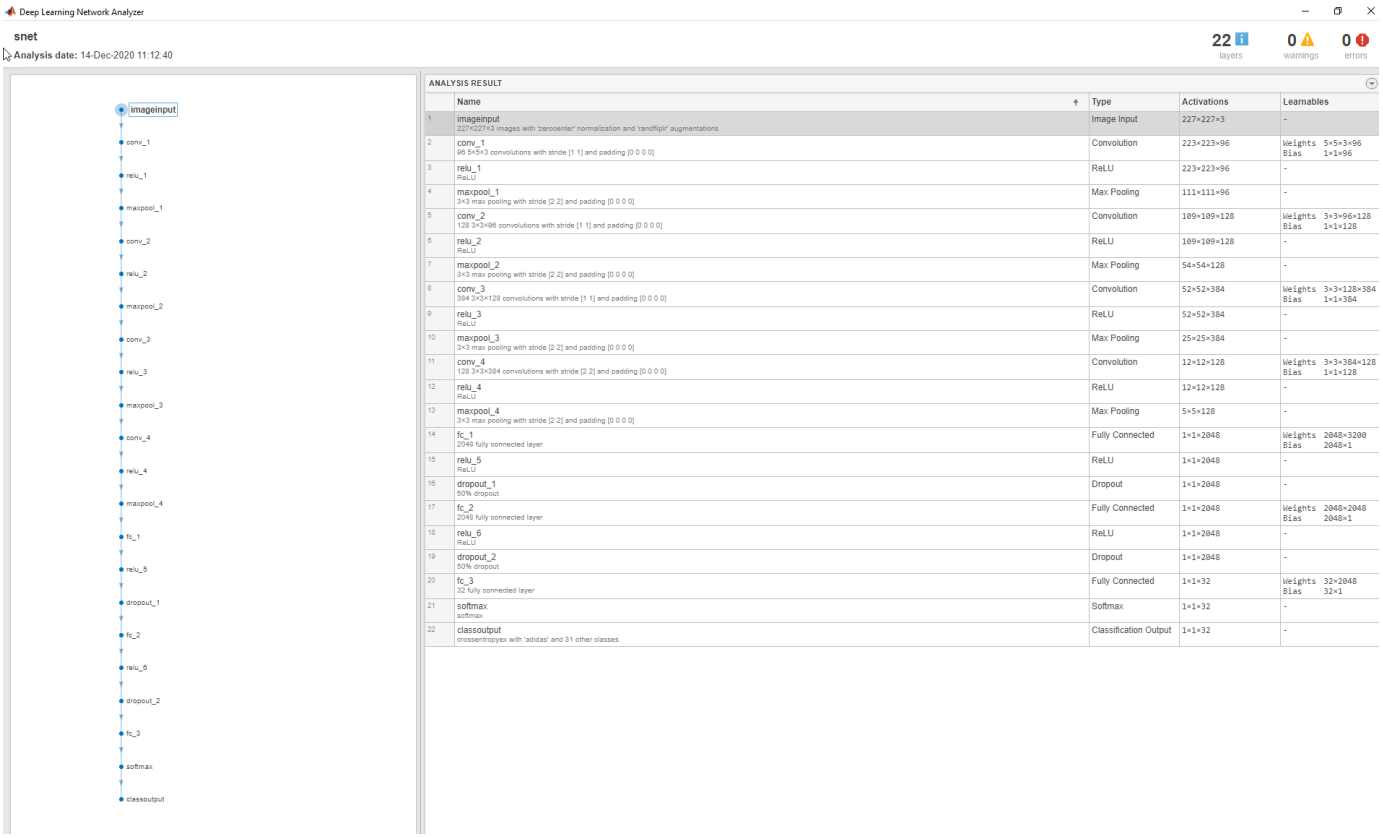
Validate Quantized Network by Using MATLAB Simulation

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);
```



	Name	Type	Activations	Learnables
1	imageinput 227×227×3 images with 'zerocenter' normalization and 'randfliplr' augmentations	Image Input	227×227×3	-
2	conv_1 96 5×5×3 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	223×223×96	Weights 5×5×3×96 Bias 1×1×96
3	relu_1 ReLU	ReLU	223×223×96	-
4	maxpool_1 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	111×111×96	-
5	conv_2 128 3×3×96 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	109×109×128	Weights 3×3×96×128 Bias 1×1×128
6	relu_2 ReLU	ReLU	109×109×128	-
7	maxpool_2 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	54×54×128	-
8	conv_3 384 3×3×128 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	52×52×384	Weights 3×3×128×384 Bias 1×1×384
9	relu_3 ReLU	ReLU	52×52×384	-
10	maxpool_3 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	25×25×384	-
11	conv_4 128 3×3×384 convolutions with stride [2 2] and padding [0 0 0 0]	Convolution	12×12×128	Weights 3×3×384×128 Bias 1×1×128
12	relu_4 ReLU	ReLU	12×12×128	-
13	maxpool_4 3×3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	5×5×128	-
14	fc_1 2048 fully connected layer	Fully Connected	1×1×2048	Weights 2048×3200 Bias 2048×1
15	relu_5 ReLU	ReLU	1×1×2048	-
16	dropout_1 50% dropout	Dropout	1×1×2048	-
17	fc_2 2048 fully connected layer	Fully Connected	1×1×2048	Weights 2048×2048 Bias 2048×1
18	relu_6 ReLU	ReLU	1×1×2048	-
19	dropout_2 50% dropout	Dropout	1×1×2048	-
20	fc_3 32 fully connected layer	Fully Connected	1×1×32	Weights 32×2048 Bias 32×1
21	softmax softmax	Softmax	1×1×32	-
22	classoutput crossentropyex with 'adidas' and 31 other classes	Classification Output	1×1×32	-

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"                -0.04897
    {'conv_1_Bias'             }    {'conv_1'     }          "Bias"                     0.9999
    {'conv_2_Weights'          }    {'conv_2'     }          "Weights"                -0.05551
    {'conv_2_Bias'             }    {'conv_2'     }          "Bias"                  -0.000611
    {'conv_3_Weights'          }    {'conv_3'     }          "Weights"                -0.04594
    {'conv_3_Bias'             }    {'conv_3'     }          "Bias"                   -0.00139
    {'conv_4_Weights'          }    {'conv_4'     }          "Weights"                -0.04596
    {'conv_4_Bias'             }    {'conv_4'     }          "Bias"                    -0.0016
    {'fc_1_Weights'            }    {'fc_1'       }          "Weights"                -0.05139
    {'fc_1_Bias'               }    {'fc_1'       }          "Bias"                  -0.000523
    {'fc_2_Weights'            }    {'fc_2'       }          "Weights"                 -0.0501
    {'fc_2_Bias'               }    {'fc_2'       }          "Bias"                   -0.00175
    {'fc_3_Weights'            }    {'fc_3'       }          "Weights"                -0.05070
    {'fc_3_Bias'               }    {'fc_3'       }          "Bias"                    -0.0295
    {'imageinput'              }    {'imageinput' }          "Activations"                   0
    {'imageinput_normalization'}    {'imageinput' }          "Activations"              -139.3
    {'conv_1'                  }    {'conv_1'     }          "Activations"              -431.0
    {'relu_1'                  }    {'relu_1'     }          "Activations"                   0
    {'maxpool_1'               }    {'maxpool_1'  }          "Activations"                   0
    {'conv_2'                  }    {'conv_2'     }          "Activations"              -166.4
    {'relu_2'                  }    {'relu_2'     }          "Activations"                   0
    {'maxpool_2'               }    {'maxpool_2'  }          "Activations"                   0
    {'conv_3'                  }    {'conv_3'     }          "Activations"               -219.
    {'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
```

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. 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` | `dlquantizationOptions` | `dlquantizer`

Topics
“Quantization of Deep Neural Networks”

Introduced in R2020a