

# 실험9 hls4ml quantization, bitstream generation and board test

작성자: 황현하

실습 조교 : 황현하, hien

## • 실험 목적

본 실험에서는 qkeras를 이용한 quantization 실험을 진행하고 hls4ml package를 통해 quantization이 board implementation에 어떠한 영향을 끼치는 지 확인해본다. 뿐만 아니라 실제로 해당 모델을 bitstream으로 generation하여 ZCU104 보드에 implementation까지 해보는 실험을 진행함으로써 전체적인 design flow를 경험해본다. 본 실험은 Quantization, bitstream generation, board test 순으로 진행되며 각 단계별로 model의 accuracy와 board utilization 등을 확인해본다.

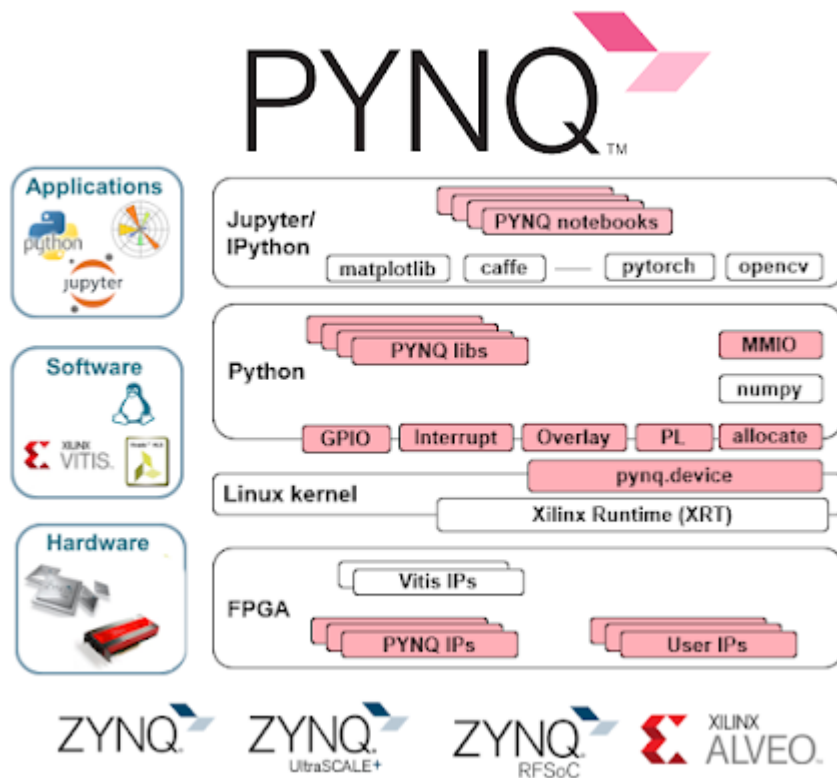
## • 실험 전에 준비해야할 내용

Qkeras는 기존의 Keras API를 확장하여 quantization까지 제공해주는 라이브러리이다. QKeras를 이용하면 몇몇 Keras layer, 특히 arithmetic operation과 activation을 수행하는 layer를 대체하여 Keras network의 quantized version을 생성할 수 있다. Qkeras에 대한 자세한 내용은 아래의 링크를 통해 확인할 수 있다.

<https://github.com/google/qkeras>

PYNQ는 Xilinx 플랫폼을 더 쉽게 사용하기 위해 만들어진 open-source 프로젝트이다. Python language와 library를 이용하여 사용자는 시스템을 좀더 편리하게 programmable logic과 microprocessor를 개발할 수 있다. PYNQ를 지원하는 보드는 Zynq, Zynq UltraScale+, Zynq RFSoc, Alveo등이 있으며, 다음과 같은 application에서 활용될 수 있다.

- parallel hardware execution
- high frame-rate video processing
- hardware accelerated algorithms
- real-time signal processing
- high bandwidth IO
- low latency control



.PYNQ는 hardware를 설계하기 위한 ASIC 형태의 design tool을 사용할 필요없이 software 개발자도 Xilinx의 플랫폼을 사용할 수 있게 도와주고, 빠른 prototyping을 통해 software interface와 framework을 쉽게 활용할 수 있다. PYNQ에 대한 자세한 내용은 아래의 링크에서 확인할 수 있다.

<http://www.pynq.io/>

## • 실습 진행을 위한 환경 세팅

이 실습은 Host 서버에 접속하기 위해 terminal 프로그램을 설치하여 미리 만들어 놓은 Vitis-AI Docker 환경에서 실습을 진행한다. Host 서버는 서울대학교 전기정보공학부 실습용 서버이며 세팅 된 환경에서 실습을 진행한다.

### a) Terminal 설치 및 사용

Host 서버에 접속하기 위해 MobaXterm terminal 프로그램을 설치하는 방법.

MobaXterm 다운로드 주소 <https://mobaxterm.mobatek.net/>

MobaXterm 실행하여 원격 서버 접속

- Host: 147.46.121.38
- Username: ai\_system10
- Password: ai\_system10

Session settings

SSH TelnetsRshXdmcpRDPVNCFTP SFTPSerialFileShellBrowserMoshAws S3WSL

Basic SSH settings

Remote host \*147.46.121.38☒ Specify usernameai\_system10Port22

Advanced SSH settings

Terminal settings

Network settings

Bookmark settings

Secure Shell (SSH) session

OK

Cancel

MobaXterm Master Password

Secure my stored passwords

Please enter a "Master Password" in order to protect all your stored passwords. The master password is used to encrypt the other stored passwords.

IMPORTANT: DO NOT FORGET YOUR MASTER PASSWORD, otherwise you will lose all your stored passwords!

My master password:

Re-type my master password:

Master password strength:

Prompt me for my master password

☒ only on new Windows account or new computer

☐ at every MobaXterm startup

☐ at every MobaXterm startup and after resuming from standby mode

OK

Cancel

## b) Vitis-AI docker 실행

Vitis-AI Docker 환경을 실행하는 명령어

```
$cd Vitis-AI
```

```
$docker pull xilinx/vitis-ai-cpu:latest
```

```
$/docker_run.sh xilinx/vitis-ai-cpu:latest
```

```
ai_system10@ECE-util1:~$ cd Vitis-AI/
ai_system10@ECE-util1:~/Vitis-AI$ ls
LICENSE  data  docker_run.sh  dsa  external  models  tools
README.md  demo  docs  examples  index.html  setup
ai_system10@ECE-util1:~/Vitis-AI$ docker pull xilinx/vitis-ai-cpu:latest
latest: Pulling from xilinx/vitis-ai-cpu
Digest: sha256:1d568blb77601a4e9989f969a74dfd9fd61102b713cb137edb83d76db11cea91
Status: Image is up to date for xilinx/vitis-ai-cpu:latest
docker.io/xilinx/vitis-ai-cpu:latest
ai_system10@ECE-util1:~/Vitis-AI$ ./docker_run.sh xilinx/vitis-ai-cpu:latest
NOTICE: BY INVOKING THIS SCRIPT AND USING THE SOFTWARE INSTALLED BY THE
SCRIPT, YOU AGREE ON BEHALF OF YOURSELF AND YOUR EMPLOYER (IF APPLICABLE)
TO BE BOUND TO THE LICENSE AGREEMENTS APPLICABLE TO THE SOFTWARE THAT YOU
INSTALL BY RUNNING THE SCRIPT.
Press any key to continue...
```

```
Do you agree to the terms and wish to proceed [y/n]? y
Setting up ai_system10 's environment in the Docker container...
Running as vitis-ai-user with ID 0 and group 0

=====
VITIS-AI
=====

Docker Image Version: 1.4.916
Build Date: 2021-07-20
VAI_ROOT: /opt/vitis_ai

For TensorFlow 1.15 Workflows do:
    conda activate vitis-ai-tensorflow
For Caffe Workflows do:
    conda activate vitis-ai-caffe
For PyTorch Workflows do:
    conda activate vitis-ai-pytorch
For TensorFlow 2.3 Workflows do:
    conda activate vitis-ai-tensorflow2
Vitis-AI /workspace >
```

현재까지의 실험 환경 세팅 과정은 기존의 실험 방식과 동일하다. 하지만 hls4ml 패키지를 이용하기 위해서는 conda 환경을 새로 설정해 주어야 한다. 실험에 필요한 환경은 제공되는 코드의 environment.yml에 적혀 있다.

```

name: hls4ml
channels:
  - conda-forge
dependencies:
  - python=3.7
  - jupyterhub
  - pydot
  - graphviz
  - pip
  - pip:
    - jupyter
    - tensorflow==2.3.1
    - git+https://github.com/google/qkeras.git#egg=qkeras
    - scikit-learn
    - git+https://github.com/thesps/conifer.git
    - matplotlib
    - pandas
    - pyyaml
    - seaborn

```

위와 같은 환경을 이용하기 위해 다음 명령어를 이용하면 된다.

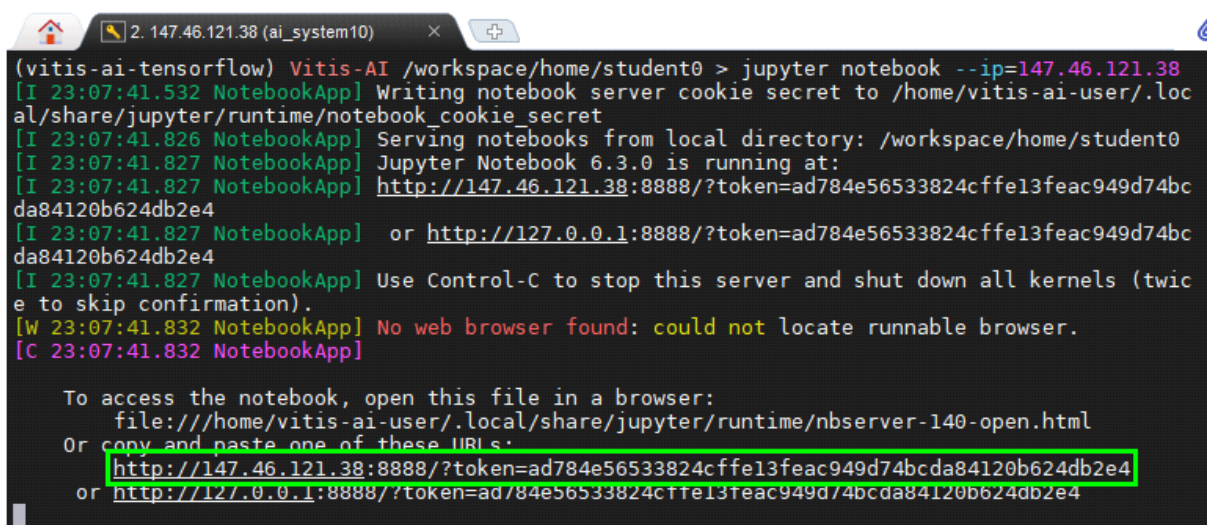
```
$conda env create -f environment.yml
```

```
$conda activate hls4ml
```

## • Jupyter notebook 실행

기존 실험과 마찬가지로 본 실험에서도 jupyter notebook을 이용하여 실험을 진행한다.

```
$ jupyter notebook --ip=147.46.121.38
```



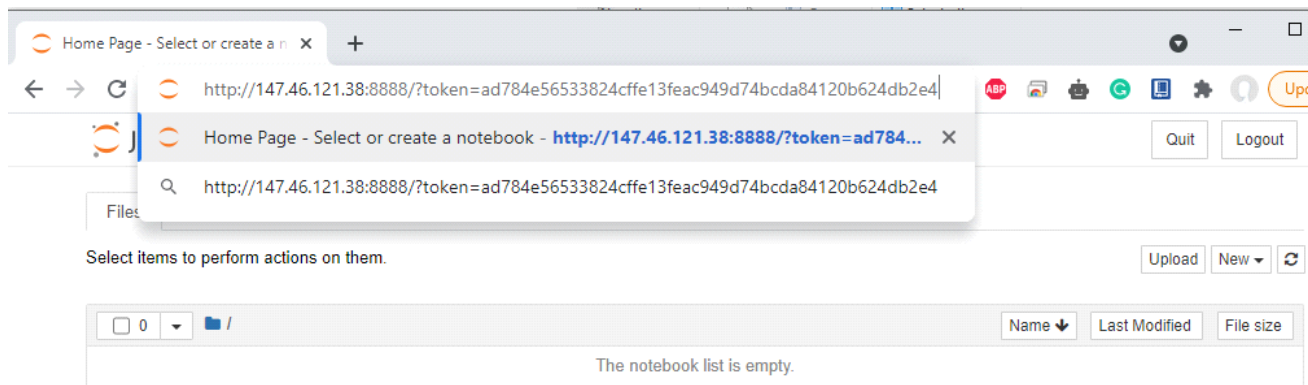
```

(vitis-ai-tensorflow) Vitis-AI /workspace/home/student0 > jupyter notebook --ip=147.46.121.38
[I 23:07:41.532 NotebookApp] Writing notebook server cookie secret to /home/vitis-ai-user/.local/share/jupyter/runtime/notebook_cookie_secret
[I 23:07:41.826 NotebookApp] Serving notebooks from local directory: /workspace/home/student0
[I 23:07:41.827 NotebookApp] Jupyter Notebook 6.3.0 is running at:
[I 23:07:41.827 NotebookApp] http://147.46.121.38:8888/?token=ad784e56533824cffe13feac949d74bcda84120b624db2e4
[I 23:07:41.827 NotebookApp] or http://127.0.0.1:8888/?token=ad784e56533824cffe13feac949d74bcda84120b624db2e4
[I 23:07:41.827 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 23:07:41.832 NotebookApp] No web browser found: could not locate runnable browser.
[C 23:07:41.832 NotebookApp]

To access the notebook, open this file in a browser:
file:///home/vitis-ai-user/.local/share/jupyter/runtime/nbserver-140-open.html
Or copy and paste one of these URLs:
http://147.46.121.38:8888/?token=ad784e56533824cffe13feac949d74bcda84120b624db2e4
or http://127.0.0.1:8888/?token=ad784e56533824cffe13feac949d74bcda84120b624db2e4

```

주소를 복사하여 인터넷 브라우저에서 jupyter notebook 접속



- **Hls4ml Quantization 실습**

a) Load MNIST dataset from tf.keras

```
import tensorflow as tf
import numpy as np
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

b) Print the the dataset shape

```
print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)
(60000, 28, 28, 1) (10000, 28, 28, 1) (60000, 10) (10000, 10)
```

c) Construct the model

This time we're going to use QKeras layers. QKeras is "Quantized Keras" for deep heterogeneous quantization of ML models.

<https://github.com/google/qkeras>

It is maintained by Google and recently support for QKeras model is added to hls4ml.

```

from keras.layers import QDense, QActivation
from keras.quantizers import quantized_bits, quantized_relu
from keras.convolutional import QConv2D
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, InputLayer, MaxPooling2D, Activation

input_shape=(28,28,1)

model = Sequential()
model.add(InputLayer(input_shape=input_shape))
model.add(QConv2D(16, kernel_size=(3, 3), kernel_quantizer=quantized_bits(6,0,alpha=1), bias_quantizer=quantized_bits(6,0,alpha=1)))
model.add(QActivation(activation=quantized_relu(6)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(QConv2D(16, kernel_size=(3, 3), kernel_quantizer=quantized_bits(6,0,alpha=1), bias_quantizer=quantized_bits(6,0,alpha=1)))
model.add(QActivation(activation=quantized_relu(6)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(QDense(10, kernel_quantizer=quantized_bits(6,0,alpha=1), bias_quantizer=quantized_bits(6,0,alpha=1)))
model.add(Activation(activation='softmax'))

model.build()

```

#### d) Train sparse

Let's train with model sparsity again, since QKeras layers are prunable.

```

from tensorflow_model_optimization.python.core.sparsity.keras import prune, pruning_callbacks, pruning_schedule
from tensorflow_model_optimization.sparsity.keras import strip_pruning
pruning_params = {"pruning_schedule" : pruning_schedule.ConstantSparsity(0.75, begin_step=0, frequency=100)}
model = prune.prune_low_magnitude(model, **pruning_params)

```

#### e) Train the model

We'll use the same settings as the model for part 1: Adam optimizer with categorical crossentropy loss.

The callbacks will decay the learning rate and save the model into a directory 'model\_mnist\_cnn4'

The model isn't very complex, so this should just take a few minutes even on the CPU.

If you've restarted the notebook kernel after training once, set `train = False` to load the trained model rather than training again.



```

from tensorflow.keras.optimizers import Adam
from callbacks import all_callbacks

train = False

if train:
    adam = Adam(lr=0.0001)
    model.compile(optimizer=adam, loss=['categorical_crossentropy'], metrics=['accuracy'])
    callbacks = all_callbacks(stop_patience = 1000,
                             lr_factor = 0.5,
                             lr_patience = 10,
                             lr_epsilon = 0.000001,
                             lr_cooldown = 2,
                             lr_minimum = 0.0000001,
                             outputDir = 'model_mnist_cnn4')
    callbacks.callbacks.append(pruning_callbacks.UpdatePruningStep())
    model.fit(x_train, y_train, batch_size=128,
              epochs=30, validation_split=0.2, shuffle=True,
              callbacks = callbacks.callbacks)
    model = strip_pruning(model)
    model.save('model_mnist_cnn4/KERAS_check_best_model.h5')
else:
    from qkeras.utils import load_qmodel
    model = load_qmodel('model_mnist_cnn4/KERAS_check_best_model.h5')

```

#### f) Check performance

How does this model which was trained using 6-bits, and 75% sparsity model compare against the original model? Let's report the accuracy and make a ROC curve. The quantized, pruned model is shown with solid lines, the unpruned model from part 1 is shown with dashed lines.

We should also check that hls4ml can respect the choice to use 6-bits throughout the model, and match the accuracy. We'll generate a configuration from this Quantized model, and plot its performance as the dotted line.

The generated configuration is printed out. You'll notice that it uses 7 bits for the type, but we specified 6!? That's just because QKeras doesn't count the sign-bit when we specify the number of bits, so the type that actually gets used needs 1 more.

We also use the `OutputRoundingSaturationMode` optimizer pass of `hls4ml` to set the Activation layers to round, rather than truncate, the cast. This is important for getting good model accuracy when using small bit precision activations. And we'll set a different data type for the tables used in the Softmax, just for a bit of extra performance.

**\*\*Make sure you've trained the model from part 1\*\***

```
import hls4ml
from hls4ml.converters.keras_to_hls import keras_to_hls
import plotting
import yaml

hls4ml.model.optimizer.OutputRoundingSaturationMode, layers = ['Activation']
hls4ml.model.optimizer.OutputRoundingSaturationMode, rounding_mode = 'AP_RND'
hls4ml.model.optimizer.OutputRoundingSaturationMode, saturation_mode = 'AP_SAT'

config = hls4ml.utils.config_from_keras_model(model, granularity='name')
config['Backend'] = 'VivadoAccelerator'
config['OutputDir'] = 'mnist-hls-test4'
config['ProjectName'] = 'myproject_mnist_cnn4'
config['XilinxPart'] = 'xczu7ev-ffvc1156-2-e'
config['Board'] = 'zcu104'
config['ClockPeriod'] = 5
config['IOType'] = 'io_stream'
config['HLSConfig'] = {}
config['HLSConfig']['Model'] = {}
config['HLSConfig']['Model'] = config['Model']
config['HLSConfig']['LayerName'] = config['LayerName']

del config['Model']
del config['LayerName']
config['AcceleratorConfig'] = {}
config['AcceleratorConfig']['Interface'] = 'axi_stream'
config['AcceleratorConfig']['Driver'] = 'python'
config['AcceleratorConfig']['Precision'] = {}
config['AcceleratorConfig']['Precision']['Input'] = 'float'
config['AcceleratorConfig']['Precision']['Output'] = 'float'
config['KerasModel'] = model

print("-----")
print("Configuration")
plotting.print_dict(config)
print("-----")
hls_model = keras_to_hls(config)
hls_model.compile()
y_qkeras = model.predict(x_test)
y_hls = hls_model.predict(x_test)
```

```
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import load_model

model_ref = load_model('model_mnist_cnn/KERAS_check_best_model.h5')
y_ref = model_ref.predict(x_test)

print("Accuracy baseline: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_ref, axis=1))))
print("Accuracy pruned, quantized: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_qkeras, axis=1))))
print("Accuracy hls4ml: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_hls, axis=1))))
```

Accuracy baseline: 0.957  
Accuracy pruned, quantized: 0.9226  
Accuracy hls4ml: 0.9234

```

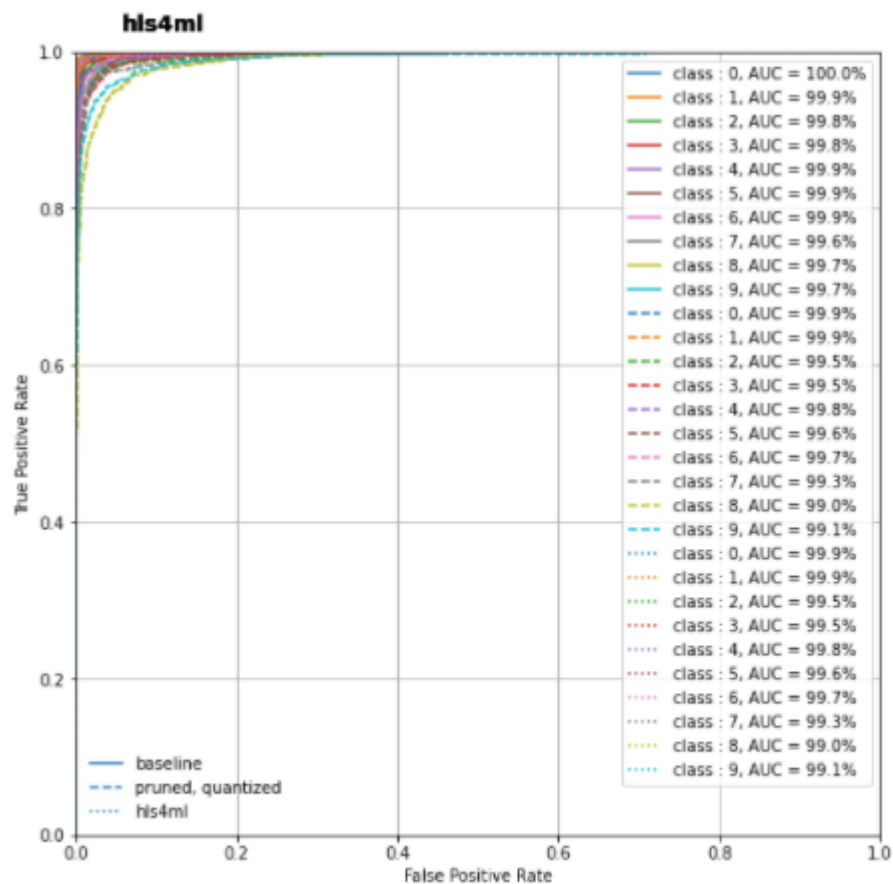
import matplotlib.pyplot as plt
mnist_classes=['0','1','2','3','4','5','6','7','8','9']

fig, ax = plt.subplots(figsize=(9, 9))
_ = plotting.makeRoc(y_test, y_ref, mnist_classes)
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_qkeras, mnist_classes, linestyle='---')
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_hls, mnist_classes, linestyle=':')

from matplotlib.lines import Line2D
lines = [Line2D([0], [0], ls='--'),
         Line2D([0], [0], ls='---'),
         Line2D([0], [0], ls=':')]
from matplotlib.legend import Legend
leg = Legend(ax, lines, labels=['baseline', 'pruned, quantized', 'hls4ml'],
            loc='lower left', frameon=False)
ax.add_artist(leg)

```

<matplotlib.legend.Legend at 0x7f4c49d2aa10>



g) Synthesize

Now let's synthesize this quantized, pruned model.

The synthesis will take a while

While the C-Synthesis is running, we can monitor the progress looking at the log file by opening a terminal from the notebook home, and executing:

```
tail -f mnist-hls-test4/vivado_hls.log
```

```
import os
os.environ['PATH'] = '/workspace/home/Xilinx/Vivado/2019.2/bin:' + os.environ['PATH']
hls_model.build(csim=False, synth=True, export=True)
```

h) Check the reports

Print out the reports generated by Vivado HLS. Pay attention to the 'Utilization Estimates' section in particular this time.

```
hls4ml.report.read_vivado_report(config['OutputDir'])
```

```
=====
== Utilization Estimates
=====
* Summary:
```

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	44	-
FIFO	1	-	163	667	-
Instance	119	138	38492	110362	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	90	-
Register	-	-	10	-	-
Total	120	138	38665	111163	0
Available	624	1728	460800	230400	96
Utilization (%)	19	7	8	48	0

```
=====
```

Print the report for the model trained in part 1. Now, compared to the model from part 1, this model has been trained with low-precision quantization, and 75% pruning. You should be able to see that we have saved a lot of resource compared to where we started in part 1. At the same time, referring to the ROC curve above, the model performance is pretty much identical even with this drastic compression!

**\*\*Note you need to have trained and synthesized the model from part 1\*\***

```
hls4ml.report.read_vivado_report('mnist-hls-test')
```

Print the report for the model trained in part 3. Both these models were trained with 75% sparsity, but the new model uses 6-bit precision as well. You can see how Vivado HLS has moved multiplication operations from DSPs into LUTs, reducing the "critical" resource usage.

**\*\*Note you need to have trained and synthesized the model from part 3\*\***

```
hls4ml.report.read_vivado_report('mnist-hls-test3')
```

- **Hls4ml bitstream generation**

a) Load MNIST dataset from tf.keras

```
import tensorflow as tf
import numpy as np
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)
```

b) Save the test dataset to files so that we can use them on the Pynq card later.

```
np.save('y_test.npy', y_test)
np.save('X_test.npy', x_test)
```

c) Check the reports

Print out the reports generated by Vivado HLS. Pay attention to the Utilization Estimates' section in particular this time.

```
hls4ml,report,read_vivado_report('mnist-hls-test')
```

```
hls4ml,report,read_vivado_report('mnist-hls-test2')
```

```
hls4ml,report,read_vivado_report('mnist-hls-test3')
```

```
hls4ml,report,read_vivado_report('mnist-hls-test4')
```

d) Choose the project for making a bitstream file.

```
prjdir=YOUR_PROJECT_DIR
```

e) Generate bitstream

```
import os
os.environ['PATH'] = '/workspace/home/Xilinx/Vivado/2019.2/bin:' + os.environ['PATH']
curr_dir = os.getcwd()
os.chdir(prjdir)
try:
    os.system('vivado -mode batch -source design.tcl')
except:
    print("Something went wrong, check the Vivado logs")
os.chdir(curr_dir)
```

f) Check the .bit file and .hwh file

## • Board Test

a) Check the network condition and the USB connection for ZCU 104 board

b) move generated bitstream, hwh, x\_test.npy, y\_test.npy files to the board using terminal.

c) By connecting to the IP adress of the board, open the jupyter notebook file on the board

## Part 6: Board Test

This notebook is to run on the PYNQ! You'll need the bitfile and test dataset file from part 5. We will load the bitfile we generated onto the PL of the PYNQ SoC.

This notebook is to run on the PYNQ. You'll need the bitfile and test dataset file from previous parts. We will load the bitfile we generated onto the PL of the PYNQ SoC.

More details :

[https://pynq.readthedocs.io/en/latest/overlay\\_design\\_methodology/python\\_overlay\\_api.html](https://pynq.readthedocs.io/en/latest/overlay_design_methodology/python_overlay_api.html)

d) Check accuracy of the model on the board.

```
from pynq import DefaultHierarchy, DefaultIP, allocate
from pynq import Overlay
from datetime import datetime
import pynq.lib.dma
import numpy as np

class NeuralNetworkOverlay(Overlay):
    def __init__(self, bitfile_name, dtbo=None, download=True, ignore_version=False, device=None):
        super().__init__(bitfile_name, dtbo=dtbo, download=download, ignore_version=ignore_version, device=device)

    def _print_dt(self, timea, timeb, N):
        dt = (timeb - timea)
        dts = dt.seconds + dt.microseconds * 10**-6
        rate = N / dts
        print("Classified {} samples in {} seconds ({} inferences / s)".format(N, dts, rate))
        return dts, rate

    def predict(self, X, y_shape, dtype=np.float32, debug=None, profile=False, encode=None, decode=None):
        if profile:
            timea = datetime.now()
        if encode is not None:
            X = encode(X)
        with allocate(shape=X.shape, dtype=dtype) as input_buffer, \
             allocate(shape=y_shape, dtype=dtype) as output_buffer:
            input_buffer[:] = X
            self.hier_0.axi_dma_0.sendchannel.transfer(input_buffer)
            self.hier_0.axi_dma_0.recvchannel.transfer(output_buffer)
            if debug:
                print("Transfer OK")
            self.hier_0.axi_dma_0.sendchannel.wait()
            if debug:
                print("Send OK")
            self.hier_0.axi_dma_0.recvchannel.wait()
            if debug:
                print("Receive OK")
            result = output_buffer.copy()
        if decode is not None:
            result = decode(result)
        if profile:
            timeb = datetime.now()
            dts, rate = self._print_dt(timea, timeb, len(X))
            return result, dts, rate
        return result
```

```
if __name__ == '__main__':
    x_test = np.load('x_test.npy')
    y_test = np.load('y_test.npy')

    N = 100

    overlay = NeuralNetworkOverlay('design_1.bit')

    out = overlay.predict(x_test[:N], [N, 10])
    predicted = out.argmax(axis=1)

    correct = (predicted == y_test[:N]).sum()
    accuracy = correct / N

    print("Model Accuracy: %.2f%%"%(accuracy*100))
```

## • 실험 후 보고서에 포함될 내용

1. 본 실험에서는 QKeras를 통해 Quantization을 수행하였다. 실험에서 각 layer의 weight와 activation을 몇 bit로 quantize하였는지 적고 기존 Keras 모델에 비해 board utilization과 accuracy가 어떻게 변했는지 논의하시오.

2. QKeras의 weight와 activation의 quantized bit을 자유롭게 변화시키며(4가지 이상의 case) MNIST dataset의 testset에 대한 accuracy와 model size가 어떻게 변화하는 지 작성하시오.

3. 본 실험에서는 이전에 수행했던 실험과 vivado report를 비교하여 bitstream generation을 하는 과정이 포함되어 있다. 어떤 프로젝트를 bitstream generation에 활용하였는지 적고 왜 해당 프로젝트를 활용하였는지 이유를 작성하시오.(각 project의 utilization와 accuracy 결과를 포함하여 작성)

4. 본 실험을 통해 board test를 수행하면 MNIST dataset의 testset에 대한 accuracy를 확인할 수 있다. 1. Keras 또는 QKeras model의 accuracy, 2. hls4ml model의 accuracy, 3. board test를 통해 확인한 accuracy를 비교하고 변화가 있다면 왜 변했는지, 변화가 없다면 왜 변하지 않았는지 설명하라.

제출 기한 : 2021.12.03(금) 오후 11:59

제출 양식 : 이름\_학번\_보고서9.pdf (ex. 홍길동\_2021-12345\_보고서9.pdf)

\*보고서는 pdf로 변환하여 제출