

# 실험6 Advanced features in hls4ml with MNIST dataset

작성자: 황현하, 박천명

실습 조교: 박천명, 이염미

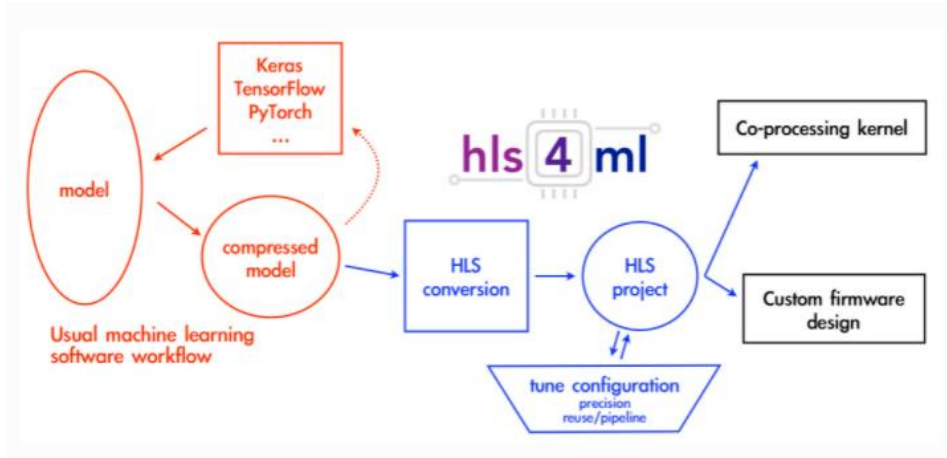
## 1. 실험 목적

본 실험에서는 FPGA에서 machine learning inference를 수행할 수 있게 도와주는 hls4ml package를 이용한 실습을 진행한다. 해당 실습은 두 개의 jupyter notebook (Part2 advanced config, Part3 compression)을 이용하여 진행된다. Part2에서는 Neural network의 layer 별 precision 및 reuse factor를 조절했을 때의 model의 차이점을 확인하고, Part3에서는 생성된 모델을 pruning 하여 model compression을 진행한다. Precision, reuse factor가 바뀔에 따라 FPGA utilization 및 model accuracy를 확인한다.

## 2. 실험 전에 준비해야할 내용

Hls4ml은 Keras와 PyTorch와 같은 open-source 패키지로 작성된 machine learning model을 FPGA 보드에 올리기 위해 HLS(High level synthesis) 코드로 빠르게 변환할 수 있게 도와주는 것을 목표로 만들어진 패키지이다. Hls4ml로 작성된 HLS 프로젝트는 IP를 만드는 데에 활용되어 해당 IP를 이용하면 더 복잡한 디자인에도 사용할 수 있다. 또한 CPU co-processing을 위한 kernel을 만들어 내는 데에도 활용될 수 있다. 사용자는 필요에 따라 수많은 parameter를 정의하여 알고리즘에 적합하게 이용할 수 있다. Hls4ml은 딥러닝 알고리즘의 FPGA implementation을 빠르게 prototyping할 수 있게 해주기 때문에 performance, resource utilization 그리고 latency 등의 requirement를 만족시키는 best design을 찾아내는 데에 필요한 시간을 크게 줄일 수 있다. Hls4ml의 공식 documentation은 아래의 링크에서 확인할 수 있다.

<https://fastmachinelearning.org/hls4ml>



### 3. 실습 실험을 진행하기 위한 환경 세팅

이 실습은 Host 서버에 접속하기 위해 terminal 프로그램을 설치하여 미리 만들어 놓은 Vltis-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](#)



## Basic SSH settings

Remote host \* 147.46.121.38

☒ Specify username ai\_system10

Port 22

## Advanced SSH settings

## Terminal settings

## Network settings

## ★ Bookmark settings

Secure Shell (SSH) session



OK

Cancel

## 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

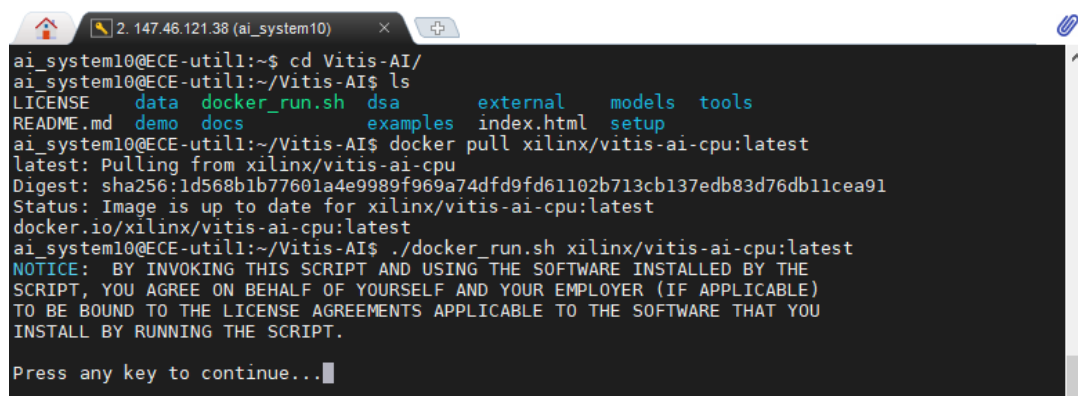
## b) Vitis-AI docker 실행

Vitis-AI Docker 환경을 실행하는 명령어

```
$ cd Vitis-AI
```

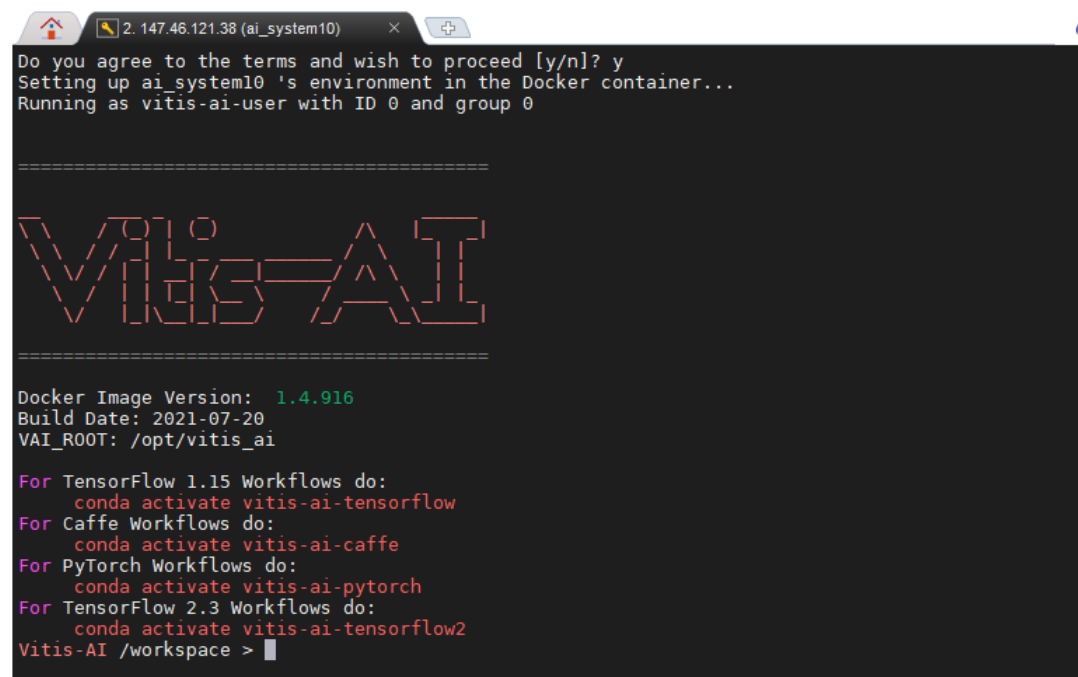
```
$ docker pull xilinx/vitis-ai-cpu:latest
```

```
$ ./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: sha256:1d568b1b77601a4e9989f969a74dfd9fd61102b713cb137edb83d76db11cea91
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.

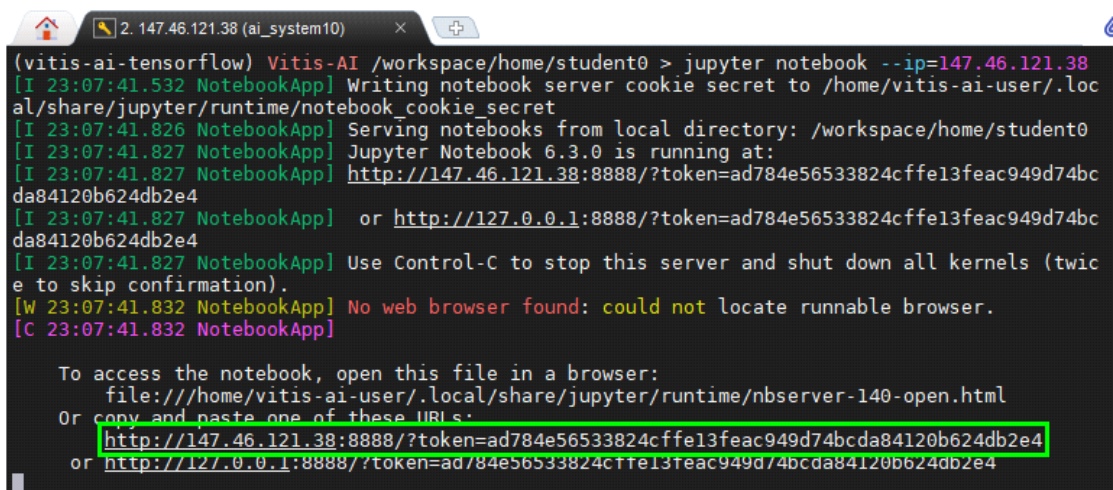
```
$ conda env create -f environment.yml
```

```
$ conda activate hls4ml
```

c) Jupyter notebook 실행

기존 실험과 마찬가지로 본 실험에서도 jupyter notebook을 이용하여 실험을 진행한다.

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



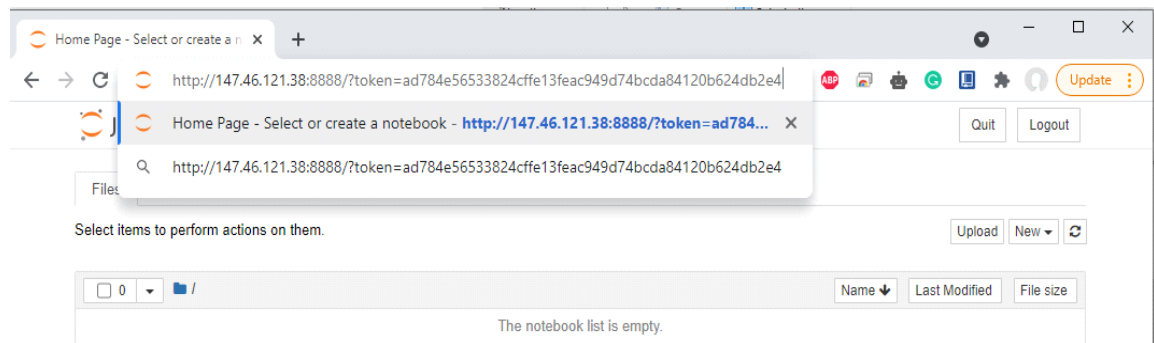
```

(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



## 4. Jupyter Notebook을 이용한 실습

### 1) Part 2\_advanced\_config

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)
```

a) Load model

Load the model trained in 'part1\_tutorial'.

```
from tensorflow.keras.models import load_model
model = load_model('model_mnist_cnn/KERAS_check_best_model.h5')
y_keras = model.predict(x_test)
```

b) Make hls4ml config & model

This time, we'll create a config with finer granularity. When we print the config dictionary, you'll notice that an entry is created for each named Layer of the model. See for the first layer, for example:

HLSConfig:

LayerName:

conv2d:

Precision:

weight: ap\_fixed<16,6>

bias: ap\_fixed<16,6>

ReuseFactor: 1

Taken 'out of the box' this config will set all the parameters to the same settings as in part 1, but we can use it as a template to start modifying things

```

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

config = hls4ml.utils.config_from_keras_model(model, granularity='name')

config['Backend'] = 'VivadoAccelerator'
config['OutputDir'] = 'mnist-hls-test'
config['ProjectName'] = 'myproject_mnist_cnn'
config['XilinxPart'] = 'xczu7ev-ffvc1156-2-e'
config['Board'] = 'zcu104'
config['ClockPeriod'] = 5
config['IOType'] = 'io_stream'

config['LayerName']['input_1']['Trace'] = True
config['LayerName']['conv2d']['Trace'] = True
config['LayerName']['conv2d_relu']['Trace'] = True
config['LayerName']['max_pooling2d']['Trace'] = True
config['LayerName']['conv2d_1']['Trace'] = True
config['LayerName']['conv2d_1_relu']['Trace'] = True
config['LayerName']['max_pooling2d_1']['Trace'] = True
config['LayerName']['dense']['Trace'] = True
config['LayerName']['dense_linear']['Trace'] = True
config['LayerName']['activation']['Trace'] = True

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("-----")

```

### c) Profiling

As you can see, we can choose the precision of everything in our Neural Network. This is a powerful way to tune the performance, but it's also complicated. The tools in `hls4ml.model.profiling` can help you choose the right precision for your model. (That said, training your model with quantization built in can get around this problem, and that is introduced in Part 4. So, don't go too far down the rabbit hole of tuning your data types without first trying out quantization aware training with QKeras.)



The first thing to try is to numerically profile your model. This method plots the distribution of the weights (and biases) as a box and whisker plot. The grey boxes show the values which can be represented with the data types used in the hls\_model. Generally, you need the box to overlap completely with the whisker 'to the right' (large values) otherwise you'll get saturation & wrap-around issues. It can be okay for the box not to overlap completely 'to the left' (small values), but finding how small you can go is a matter of trial-and-error.

Providing data, in this case just using the first 1000 examples for speed, will show the same distributions captured at the output of each layer.

```
from hls4ml.model.profiling import numerical
hls_model = keras_to_hls(config)
numerical(model=model, hls_model=hls_model, X=x_test[:1000])
```

#### d) Customize

Let's just try setting the precision of the first layer weights to something more narrow than 16 bits. Using fewer bits can save resources in the FPGA. After inspecting the profiling plot above, let's try 8 bits with 1 integer bit.

Then create a new HLSModel, and display the profiling with the new config. This time, just display the weight profile by not providing any data 'X'. Then create the HLSModel and display the architecture. Notice the box around the weights of the first layer reflects the different precision.

```
config['HLSConfig']['LayerName']['conv2d']['Precision']['weight'] = 'ap_fixed<8,2>'
config['HLSConfig']['LayerName']['conv2d']['Precision']['bias'] = 'ap_fixed<8,2>'
config['OutputDir'] = 'mnist-hls-test2'
config['ProjectName'] = 'myproject_mnist_cnn2'
hls_model = keras_to_hls(config)
numerical(model=model, hls_model=hls_model, X=x_test[:1000])
hls4ml.utils.plot_model(hls_model, show_shapes=True, show_precision=True, to_file=None)
```

#### e) Trace

When we start using customised precision throughout the model, it can be useful to collect the output from each layer to find out when things have gone wrong. We enable this trace collection by setting Trace = True for each layer whose output we want to collect.

```
for layer in config['HLSConfig']['LayerName'].keys():
    config['HLSConfig']['LayerName'][layer]['Trace'] = True
hls_model = keras_to_hls(config)
```

#### f) Compile, trace, predict

Now we need to check that this model performance is still good after reducing the precision. We compile the `hls_model`, and now use the `hls_model.trace` method to collect the model output, and also the output for all the layers we enabled tracing for. This returns a dictionary with keys corresponding to the layer names of the model. Stored at that key is the array of values output by that layer, sampled from the provided data. A helper function `get_ymodel_keras` will return the same dictionary for the Keras model.

We'll just run the trace for the first 1000 examples, since it takes a bit longer and uses more memory than just running `predict`.

```
hls_model.compile()
hls4ml_pred, hls4ml_trace = hls_model.trace(x_test[:1000])
keras_trace = hls4ml.model.profiling.get_ymodel_keras(model, x_test[:1000])
y_hls = hls_model.predict(x_test)
```

#### g) Inspect

Now we can print out, make plots, or do any other more detailed analysis on the output of each layer to make sure we haven't made the performance worse. And if we have, we can quickly find out where. Let's just print the output of the first layer, for the first sample, for both the Keras and hls4ml models.

```
print("Keras layer 'conv2d', first sample:")
print(keras_trace['conv2d'][0])
print("hls4ml layer 'conv2d', first sample:")
print(hls4ml_trace['conv2d'][0])
```

#### h) Compare

Let's see if we lost performance by using 8 bits for the weights of the first layer by inspecting the accuracy and ROC curve.

```

import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

print("Keras Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_keras, axis=1))))
print("hls4ml Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_hls, axis=1))))

mnist_classes=['0','1','2','3','4','5','6','7','8','9']

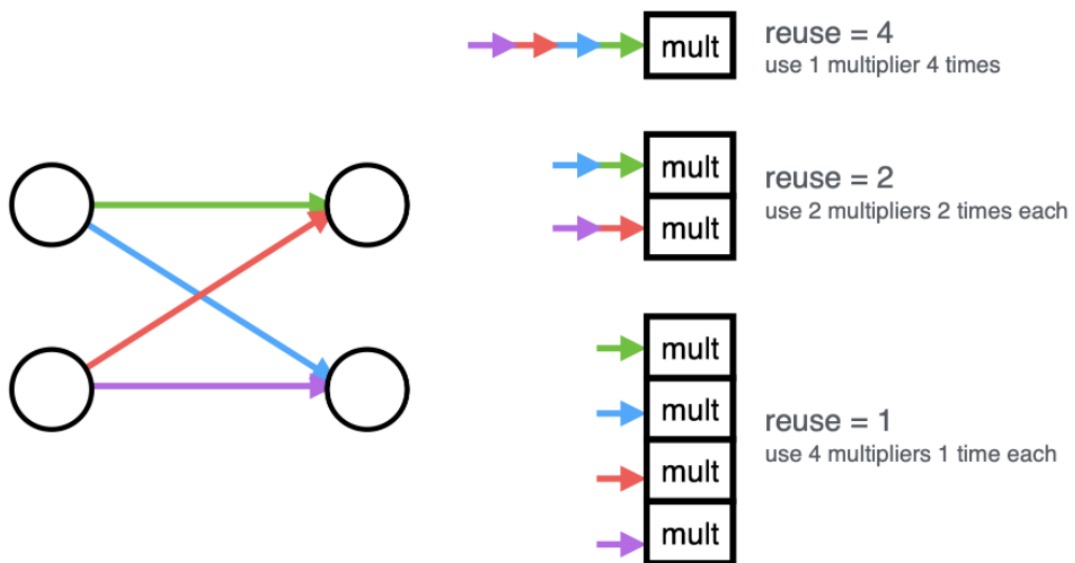
fig, ax = plt.subplots(figsize=(9, 9))
_ = plotting.makeRoc(y_test, y_keras, mnist_classes)
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_hls, mnist_classes, linestyle='--')

from matplotlib.lines import Line2D
lines = [Line2D([0], [0], ls='--'),
         Line2D([0], [0], ls='--')]
from matplotlib.legend import Legend
leg = Legend(ax, lines, labels=['keras', 'hls4ml'],
            loc='lower right', frameon=False)
ax.add_artist(leg)

```

#### i) ReuseFactor

Now let's look at the other configuration parameter: ReuseFactor. Recall that ReuseFactor is our mechanism for tuning the parallelism:



```

config['OutputDir']='mnist-hls-test2'
config['ProjectName'] = 'mnist-hls-test2'
config['HLSConfig']['LayerName']['conv2d']['ReuseFactor'] = 8
config['HLSConfig']['LayerName']['conv2d_1']['ReuseFactor'] = 8
config['HLSConfig']['LayerName']['dense']['ReuseFactor'] = 8
hls_model = keras_to_hls(config)

print("-----")
print("Configuration")
plotting.print_dict(config)
print("-----")

hls_model.compile()
y_hls = hls_model.predict(x_test)
print("Keras Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_keras, axis=1))))
print("hls4ml Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_hls, axis=1))))
plt.figure(figsize=(9, 9))
_ = plotting.makeRoc(y_test, y_keras, mnist_classes)
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_hls, mnist_classes, linestyle='--')

```

#### j) Build model

Now build the model. This can take several minutes.

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-test2/vivado_hls.log
```

```

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

```

#### k) Print report, compare this to the report from Exercise 1

(뒤에 실행된 mnist-hls-test가 exercise 1에서 실행한 report)

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

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

## 2) Part 3\_compression

#### 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 dataset shape

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

c) Construct a model

We'll use the same architecture as in part 1

```
from tensorflow.keras.layers import Dropout, Flatten, Dense, Activation, BatchNormalization, Conv2D, MaxPooling2D, InputLayer
from tensorflow.keras.models import Sequential

input_shape=(28,28,1)

model = Sequential()
model.add(InputLayer(input_shape=input_shape))
model.add(Conv2D(16, kernel_size=(3, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(16, kernel_size=(3, 3), activation="relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(10))
model.add(Activation(activation='softmax'))
```

d) Train sparse

This time we'll use the Tensorflow model optimization sparsity to train a sparse model (forcing many weights to '0'). In this instance, the target sparsity is 75%

```
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\_cnn3'. 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_cnn3')
    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_cnn3/KERAS_check_best_model.h5')
else:
    from tensorflow.keras.models import load_model
    model = load_model('model_mnist_cnn3/KERAS_check_best_model.h5')
```

f) Check sparsity

Make a quick check that the model was indeed trained sparse. We'll just make a histogram of the weights of the 1st layer, and hopefully observe a large peak in the bin containing '0'. Note logarithmic y axis.

```
import matplotlib.pyplot as plt

model.summary()
w = model.layers[0].weights[0].numpy()
h, b = np.histogram(w, bins=100)
plt.figure(figsize=(7,7))
plt.bar(b[:-1], h, width=b[1]-b[0])
plt.semilogy()
print('% of zeros = {}'.format(np.sum(w==0)/np.size(w)))
```

g) Check performance

How does this 75% sparse model compare against the unpruned model? Let's report the accuracy and make a ROC curve. The pruned model is shown with solid lines, the unpruned model from part 1 is shown with dashed lines. **Make sure you've trained the model from part 1**

```
import plotting
import matplotlib.pyplot as plt
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)
y_prune = model.predict(x_test)

print("Accuracy unpruned: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_ref, axis=1))))
print("Accuracy pruned: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_prune, axis=1))))

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_prune, mnist_classes, linestyle='--')

from matplotlib.lines import Line2D
lines = [Line2D([0], [0], ls='-'),
         Line2D([0], [0], ls='--')]
from matplotlib.legend import Legend
leg = Legend(ax, lines, labels=['unpruned', 'pruned'],
            loc='lower right', frameon=False)
ax.add_artist(leg)
```

h) Convert the model to FPGA firmware with hls4ml

Let's use the default configuration: ap\_fixed<16,6> precision everywhere and ReuseFactor=1, so we can compare with the part 1 model. We need to use strip\_pruning to change the layer types back to their originals.

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-test3/vivado_hls.log
```

```

import hls4ml
from hls4ml.converters.keras_to_hls import keras_to_hls
import yaml

config = hls4ml.utils.config_from_keras_model(model, granularity='name')
config['Backend'] = 'VivadoAccelerator'
config['OutputDir'] = 'mnist-hls-test3'
config['ProjectName'] = 'myproject_mnist_cnn3'
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()
os.environ['PATH'] = '/workspace/home/Xilinx/Vivado/2019.2/bin:' + os.environ['PATH']
hls_model.build(csim=False, synth=True, export=True)

```

- i) Print report, compare this to the report from exercise 1

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

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

## 5. 실험 후 보고서에 포함될 내용

1. 실습 과정을 통해 첫번째 conv layer에서 weight와 bias의 precision을 ap\_fixed<16,6>에서 ap\_fixed<8,2>로 변경하였다. Profiling을 통해 parameter 값이 어떻게 변화했는지 설명하시오.



2. Model Compression 실습에서 Pruning을 통해 model sparsity의 변화가 있었다. model sparsity를 0.2, 0.4, 0.6, 0.8로 변경시키며 accuracy변화를 작성하라. 이러한 과정을 통해 model 성능의 변화가 있었는지 적고, 변화가 있다면 왜 변했는지, 변화가 없다면 왜 변하지 않았는지 설명하라.

3. Advanced configuration, Compression 실험을 통해 생성된 Vivado HLS의 report를 확인할 수 있다. 지난 시간에 Part1. tutorial에서 확인한 report와 이번 실습 시간에 확인한 report를 비교하여 utilization이 어떻게 변화했는지 작성하라. 또한 각각의 실험을 통해 생성된 IP가 FPGA board로 implementation이 가능한지 적고 근거를 제시하라.

제출 기한: 2021.11.19(금) 오후 11:59

제출 양식: 이름\_학번\_보고서6.pdf (ex. 홍길동\_2021-12345\_보고서6.pdf)

\*보고서는 pdf로 변환하여 제출