실험5 Basic tutorial of hls4ml with MNIST dataset

작성자: 황현하

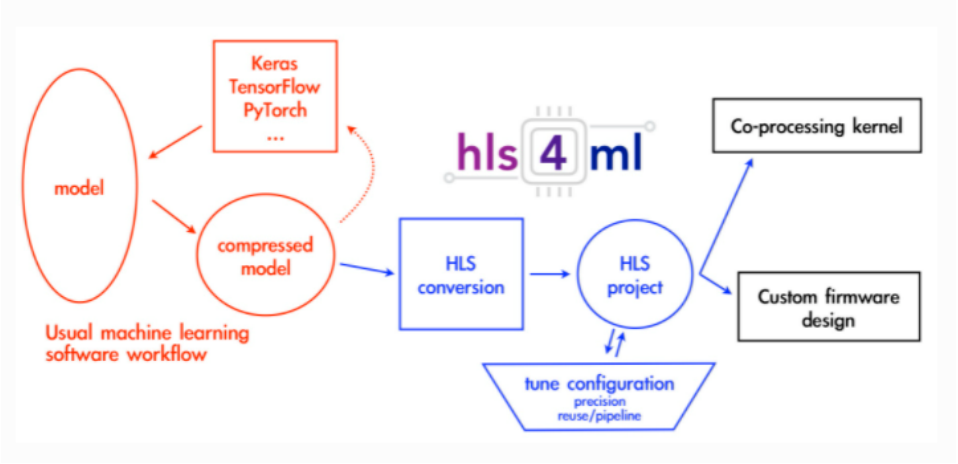
• 실험 목적

본 실험에서는 FPGA에서 machine learning inference를 수행할 수 있게 도와주는 hls4ml package를 이용한 기초적인 실습을 진행한다. hls4ml이 딥러닝 모델을 HLS(High level synthesis) 언어로 변환해주는 과정을 실험을 통해 이해하는 것을 목표로 한다. 구체적으로는 MNIST데이터 셋을 분류하는 모델을 hls4ml 모델로 변환하여 성능 변화를 확인한다. 또한 해당 모델을 IP로 export하여 FPGA utilization을 확인한다.

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



- **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 전기정보공학보 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/

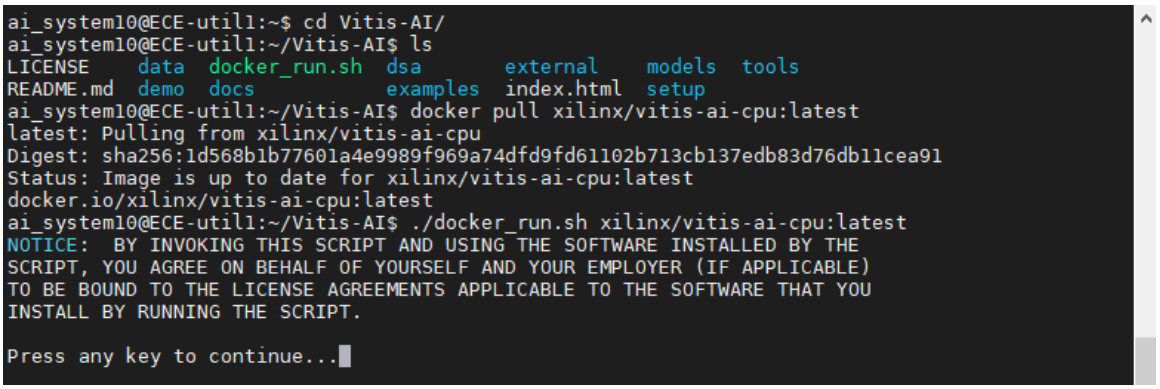
b) Vitis-AI docker 실행

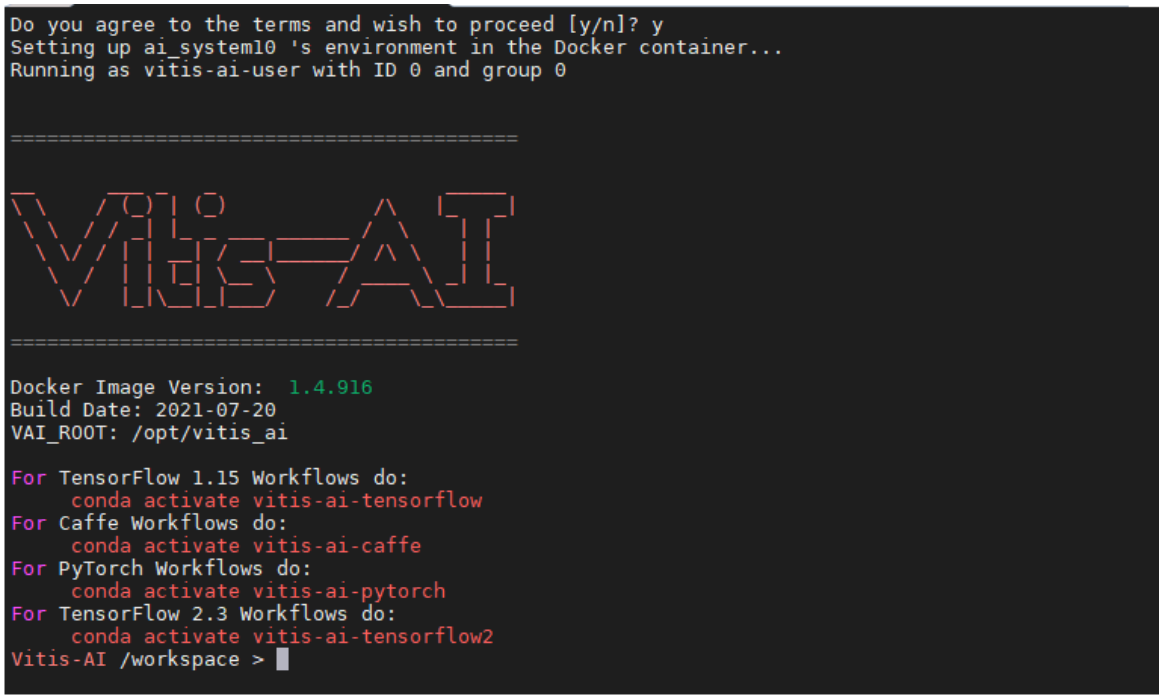
VItis-AI Docker 환경을 실행하는 명령어

```
$cd Vitis-AI
```

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

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





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.

```yaml
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.

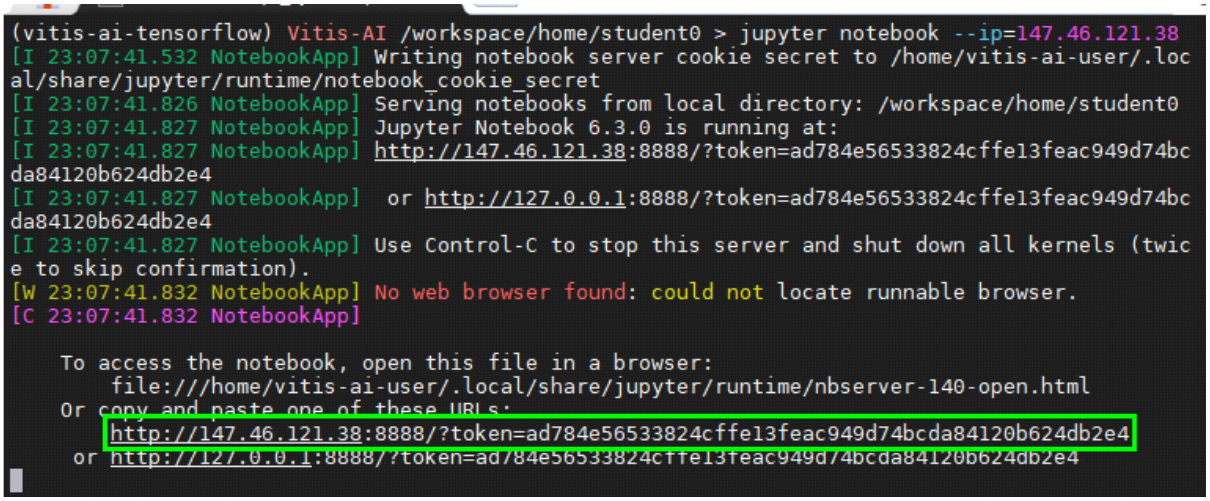
$conda env create -f environment.yml

$conda activate hls4ml

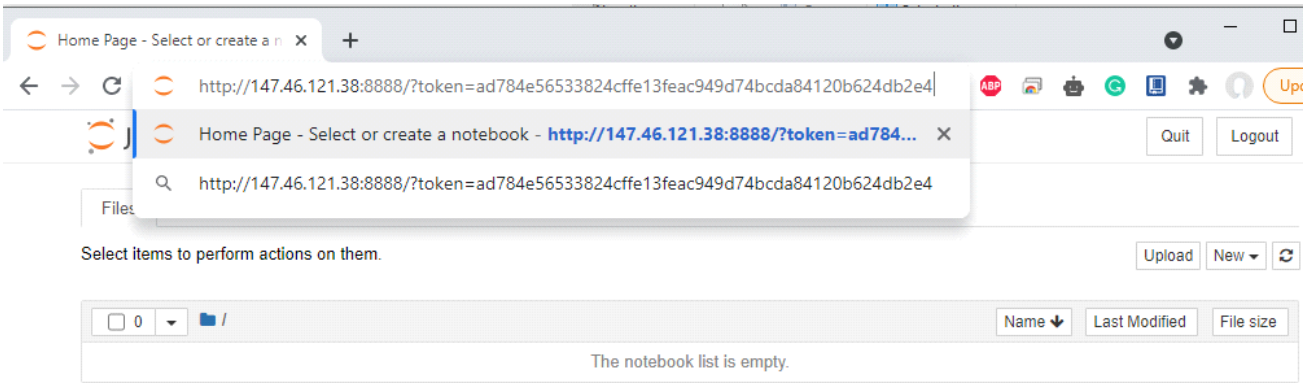
- **Jupyter notebook 실행**

기존 실험과 마찬가지로 본 실험에서도 jupyter notebook을 이용하여 실험을 진행한다.

$ jupyter notebook



주소를 복사하여 인터넷 브라우저에서 jupyter notebook 접속



- **Jupyter Notebook을 이용한 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

```
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'))


model.build()
```

d) Train the model

We'll use Adam optimizer with categorical crossentropy loss. The callbacks will decay the learning rate and save the model into a directory 'model_mnist_cnn' 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.

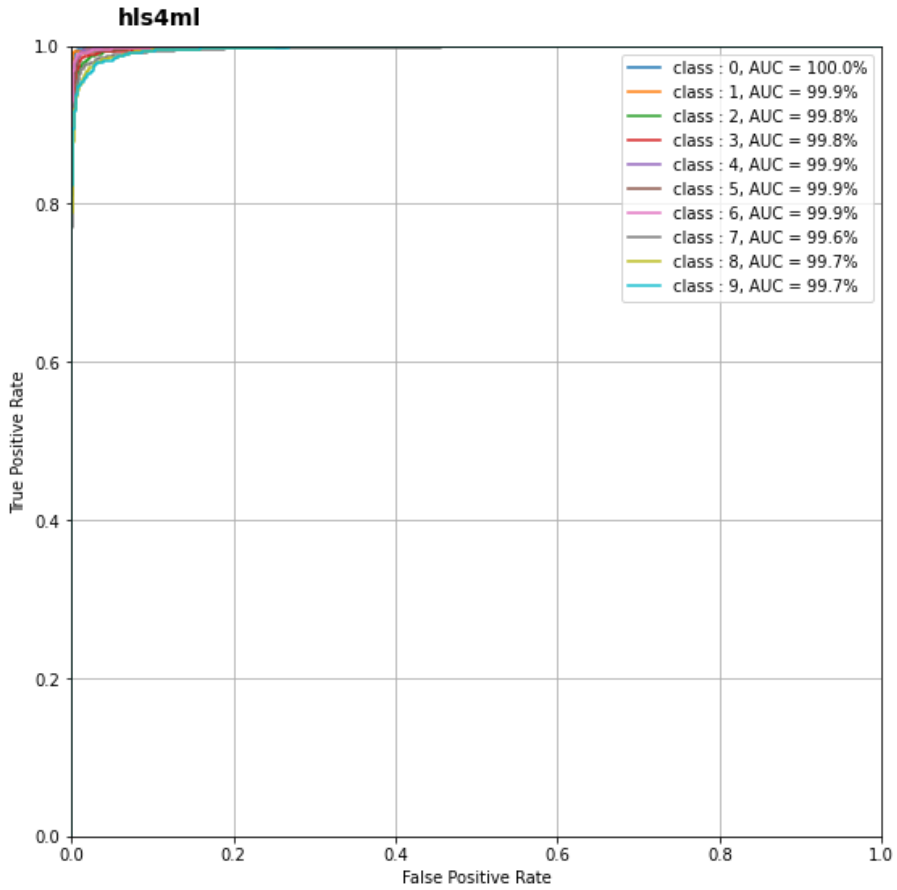
```
from tensorflow.keras.optimizers import Adam
from callbacks import all_callbacks
import os
train =True


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_cnn')
    model.fit(x_train, y_train, batch_size=128,
              epochs=10, validation_split=0.2, shuffle=True,
              callbacks = callbacks.callbacks)
else:
    from tensorflow.keras.models import load_model
    model = load_model('model_mnist_cnn/KERAS_check_best_model.h5')
```

e) Check performance

Check the accuracy and make a ROC curve

```
import plotting
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
mnist_classes=['0','1','2','3','4','5','6','7','8','9']
y_keras = model.predict(x_test)
print("Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_keras, axis=1))))
plt.figure(figsize=(9,9))
_ = plotting.makeRoc(y_test, y_keras, mnist_classes)
```

```
Accuracy: 0.957
```



f) Convert the model to FPGA firmware with hls4ml (Make an hls4ml config & model)

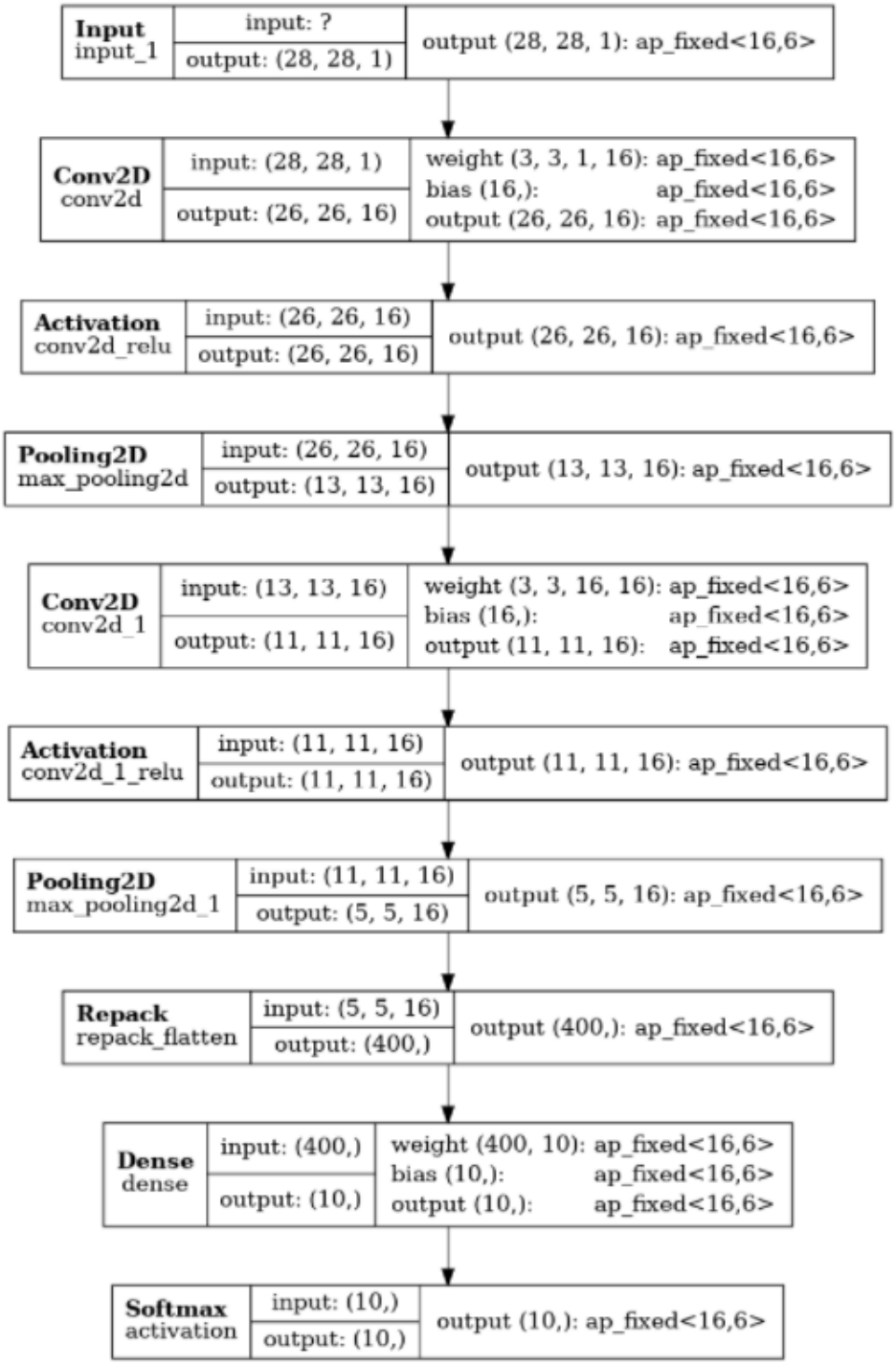
Now we will go through the steps to convert the model we trained to a low-latency optimized FPGA firmware with hls4ml. First, we will evaluate its classification performance to make sure we haven't lost accuracy using the fixed-point data types. Then we will synthesize the model with Vivado HLS and check the metrics of latency and FPGA resource usage.

Now we will go through the steps to convert the model we trained to a low-latency optimized FPGA firmware with hls4ml. First, we will evaluate its classification performance to make sure we haven't lost accuracy using the fixed-point data types. Then we will synthesize

```
import hls4ml
from hls4ml.converters.keras_to_hls import keras_to_hls


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['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)
```

g) Visualize what we created. The model architecture is shown, annotated with the shape and data types

```
hls4ml.utils.plot_model(hls_model, show_shapes=True, show_precision=True, to_file=None)
```



h) Compile, predict

Now we need to check that this model performance is still good. We compile the hls_model, and then use hls_model.predict to execute the FPGA firmware with bit-accurate emulation on the CPU.

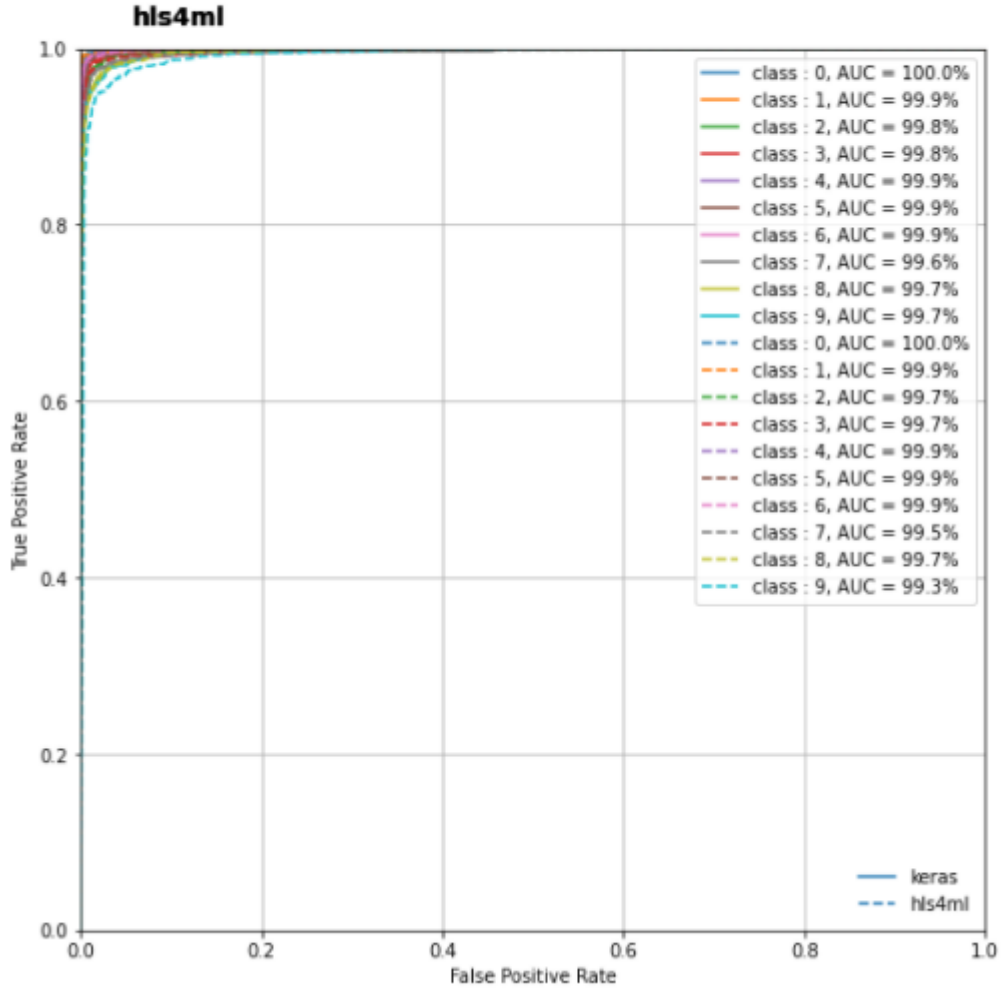
```
hls_model.compile()
y_hls4ml    = hls_model.predict(x_test)
```

i) Compare

Now let's see how the performance compares to Keras:

```
acc_hls4ml = accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_hls4ml, axis=1))
acc_keras=accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_keras, axis=1))
print('Accuracy hls4ml:     {}'.format(acc_hls4ml))
print('Accuracy keras:      {}'.format(acc_keras))

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_hls4ml, 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)
```

```
Accuracy hls4ml:      0.9567
Accuracy keras:       0.957

<matplotlib.legend.Legend at 0x7f89c9895710>
```



j) synthesize

Now we'll actually use Vivado HLS to synthesize the model. We can run the build using a method of our hls_model object. After running this step, we can integrate the generated IP into a workflow to compile for a specific FPGA board. In this case, we'll just review the reports that Vivado HLS generates, checking the latency and resource usage.

This can take a few tens of 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-test/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)
```

k) check the reports

Print out the reports generated by Vivado HLS. Pay attention to the Latency and the 'Utilization Estimates' sections

```
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         |       87|   4301|   71708|  252405|    0|
|Memory           |        -|      -|       -|       -|    -|
|Multiplexer      |        -|      -|       -|      90|    -|
|Register         |        -|      -|      10|       -|    -|
+-----------------+---------+-------+--------+--------+-----+
|Total            |       88|   4301|   71881|  253206|    0|
+-----------------+---------+-------+--------+--------+-----+
|Available        |      624|   1728|  460800|  230400|   96|
+-----------------+---------+-------+--------+--------+-----+
|Utilization (%)  |       14|    248|      15|     109|    0|
+-----------------+---------+-------+--------+--------+-----+
```

- **실험 후 보고서에 포함될 내용**

1. Keras model과 Hls4ml model의 accuracy를 비교하고, 변화가 있다면 왜 변했는지, 변화가 없다면 왜 변하지 않았는지 설명하라.

2. Keras model에서는 각 layer의 데이터 type은 floating point로 구성되어 있다. 실험에서 생성된 hls4ml model에서는 각 layer가 어떤 데이터 type을 가지는 지 적고 무엇을 의미하는 지 작성하시오.

3. 실험 과정에서 config['ProjectName']을 통해 Vivado HLS project가 생성된 경로를 확인할 수 있다. 해당 project를 Vivado HLS가 설치된 PC로 옮겨서 vivado_hls 파일을 클릭하

면 HLS 프로젝트를 실행할 수 있다. Conv, dense, pooling, activation(ReLU, sigmoid) layer의 연산이 HLS code로 어떻게 작성되었는지 각각의 code를 첨부하고 각 코드에 대해 설명하라.

4. 이번 실험을 통해 생성된 Vivado HLS의 report를 확인할 수 있다. 실험을 통해 생성된 IP가 FPGA board로 implementation이 가능한 지 적고 근거를 제시하라.

제출 기한 : 2021.11.17(수) 오후 11:59

제출 양식 : 이름_학번_보고서5.pdf (ex. 홍길동_2021-12345_보고서6.pdf)

*보고서는 pdf로 변환하여 제출