

# 실험1 서버 환경에서 MNIST 학습 및 Quantization

## 1. 실험 목적

신경회로망의 기본적인 개념인 training과 inference를 MNIST 문제의 실습을 통하여 이해하고, inference 신경망의 weight와 activation function의 quantization을 통하여 신경망의 복잡도와 인식률 즉 accuracy의 상관 관계를 이해한다.

- 서버 접속하여 MNIST model 만들기 위한 초기화 세팅 (Tensorflow API 이용하여 MNIST dataset을 다운받기)
- MNIST deep learning model 생성 (Tensorflow Keras API 통해 CNN model 생성)
- 생성한 MNIST model 학습
- 생성한 MNIST model을 activations 및 weight quantization 방법

## 2. 실험 전에 준비해야할 내용

인공신경망은 기계학습과 인지과학에서 생물학의 신경망에서 영감을 얻은 통계학적 학습 알고리즘이다. 인공신경망은 시냅스의 결합으로 네트워크를 형성한 인공 뉴런(노드)이 학습을 통해 시냅스의 결합 세기를 변화시켜, 문제 해결 능력을 가지는 모델 전반을 가리킨다. 인공신경망 학습하기 위해 사용하는 방법은 여러가지 있다.

- 1) Hebbian Learning (Hebbian Rule)
- 2) Perceptron Rule
- 3) Gradient Descent (Delta Rule, Least Mean Square)
- 4) Back propagation
- 5) 그 이외 응용기법

각각의 방법에 대한 자세한 설명(Neural Network의 학습 방법, Gradient Descent, Back-Propagation by 곽동현 IMCOMKING 2014. 6. 17.) : <https://newsight.tistory.com/70>

딥러닝 기술의 확산이 가속화 되고 있어 딥러닝 기술을 하드웨어, 즉 SoC로 구현하기 위한 연구들이 많이 진행되고 있다. 매우 많은 파라미터와 연산량을 가진 기존 딥러닝 네트워크를 하드웨어로 구현하려면 많은 비용과 전력 소모가 요구되므로 네트워크를 효과적으로 줄이기 위한 여러 가지 압축 방법 들이 제안되고 있다. 그러한 방법 중에 효과적인 최적화 기법 중 하나는 양자화 (Quantization) 방법이다. 뉴런의 활성화도 (activation) 및 학습 가중치 (weight)를 저장하는데 필요한 비트 수를 줄임으로써 동일한 양의 데이터 접근과 연산 비용 (칩 면적 및 에너지 소모 등)으로 더 많은 연산이 가능해지며 이로 인해 속도와 에너지 소모를 동시에 최적화할 수 있다.

- 효율적인 심층 신경망을 위한 양자화 알고리즘 및 방법론 (학위논문(박사)--서울대학교 대학원 :공과대학 컴퓨터공학부,2020. 2. 유승주.):

[https://dcollection.snu.ac.kr/public\\_resource/pdf/000000160741\\_20210912041841.pdf](https://dcollection.snu.ac.kr/public_resource/pdf/000000160741_20210912041841.pdf)

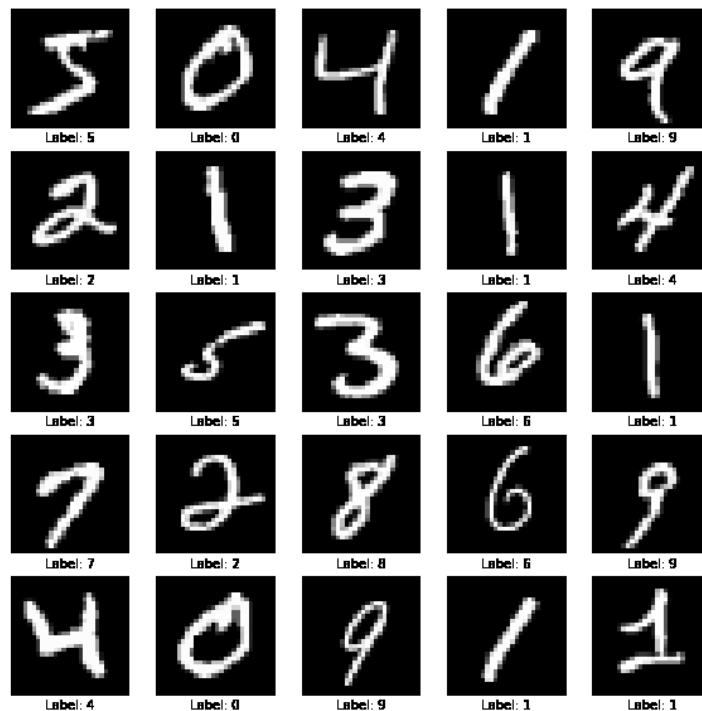
## MNIST 문제 및 데이터 세트

MNIST는 간단한 컴퓨터 비전 데이터 세트로 60,000개의 트레이닝 셋과 10,000개의 테스트 셋으로 이루어져 있다. MNIST는 간단한 컴퓨터 비전 데이터 세트로, 아래와 같이 손으로 쓰여진 이미지들로 구성되어 있다. 숫자는 0에서 1까지의 값을 갖는 고정 크기 이미지 (28x28 픽셀)로 크기가 표준화되어 있다.

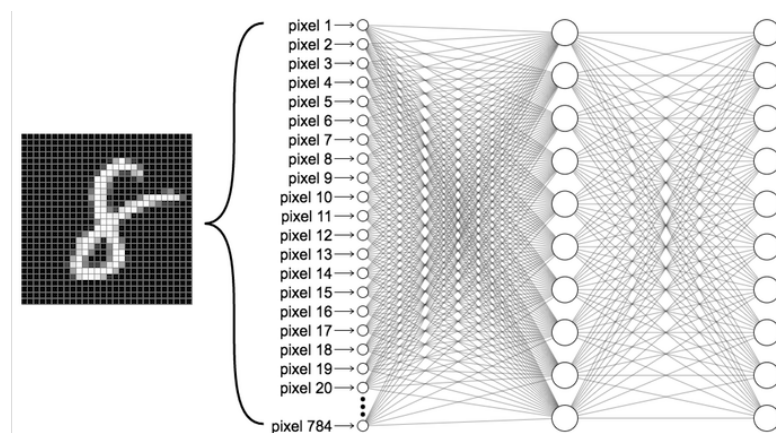
MNIST 데이터는 Yann LeCun의 웹사이트에서 제공한다. (THE MNIST DATABASE of handwritten digits Yann LeCun, Courant Institute, NYU Corinna Cortes, Google Labs, New YorkChristopher J.C. Burges, Microsoft Research, Redmond)

<http://yann.lecun.com/exdb/mnist/>

데이터셋 명	행렬 차원	데이터 종류
mnist.train.images	55000 x 784	학습 이미지 데이터
mnist.train.labels	55000 x 10	학습 라벨 데이터
mnist.test.images	10000 x 784	테스트용 이미지 데이터
mnist.test.labels	10000 x 10	테스트용 라벨 데이터
mnist.validation.images	5000 x 784	확인용 이미지 데이터
mnist.validation.labels	5000 x 10	확인용 라벨 데이터



네트워크의 훈련을 진행하기 위하여 이미지를 784개 pixel의 1-D numpy 배열로 변환하였다.



### 3. 실습 실험을 진행하기 위한 환경 세팅

서버 환경에서 MNIST 학습 및 Quantization 실습을 진행하기 위해 Terminal 프로그램을 이용해 제공된 서버 환경에 접속하여 Xilinx에서 지원하는 Vitis AI Library를 사용한다. Vitis AI Library를 통해 Deep Learning 알고리즘을 Xilinx FPGA 보드에서 구현하는 방식으로 변경할 수 있다.

- Xilinx FPGA Board : ZCU-104 Board (MPSOC)
- Host 서버 : Ubuntu 16.04 LTS

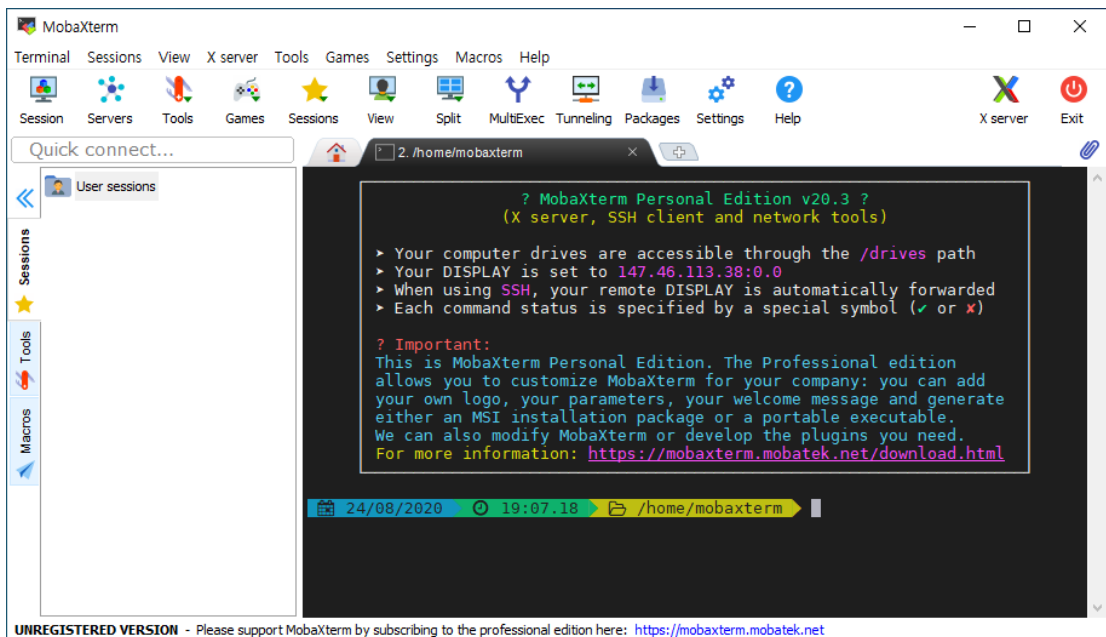
Host 서버에 접속하기 위해 terminal 프로그램을 설치하여 미리 만들어둔 Vitis-AI Docker 환경에서 실습을 진행한다. Host 서버는 서울대학교 전기정보공학부 실습용 서버이며 세팅된 환경에서 실습을 진행한다.

#### a) Terminal 설치 및 사용

Host 서버에 접속하기 위해 MobaXterm terminal 프로그램을 설치하는 방법

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

MobaXterm 실행



MobaXterm을 통해 원격 서버 접속 방법

## 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 > ■
```

Vitis-AI Docker 환경에서 사용하는 library에 따라 콘다 환경이 세팅되어 있다.

다음의 명령어를 이용해 tensorflow2 콘다 환경을 실행할 수 있다.

```
$conda activate vitis-ai-tensorflow2
```

```
$pip install matplotlib keras==2.2.5
```

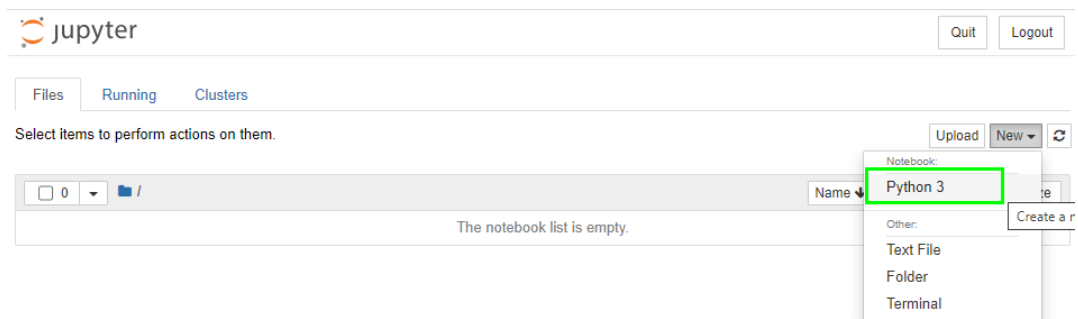
## c) Jupyter Notebook 실행

주피터 노트북 앱은 웹 브라우저를 통하여 노트북 문서들을 편집하고 실행하게 하는 서버-클라이언트 애플리케이션이다. 인터넷 접근없이 로컬 데스크탑에서 실행될 수 있다. 또는 인터넷을 통하여 리모트 서버에 설치될 수 있고 접근 가능하다.

이번 실습에서는 주피터 노트북 앱을 이용해 코드 작성을 진행한다.

```
$ jupyter notebook
```

Python script 작성



## 4. Jupyter Notebook을 통해 실습 실험

Host 서버에서 jupyter notebook을 이용해 MNIST dataset을 다운받아 deep learning model을 학습하고, Xilinx ZCU104 board에서 inference 하기 위한 quantized된 MNIST model을 만들어 본다.

- 서버 접속하여 MNIST model 만들기 위한 초기화 세팅 (Tensorflow API 이용하여 MNIST dataset을 다운받기)

- MNIST deep learning model 생성 (Tensorflow Keras API 통해 CNN model 생성)
- 생성한 MNIST model 학습
- 생성한 MNIST model을 activations 및 weight quantization 방법

## a) MNIST model 만들기 위한 초기화 세팅

Tensorflow 및 필요한 library 가지고 오기

```
import os
import numpy as np
import matplotlib.pyplot as plt

# Silence TensorFlow messages
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
```

학습 model을 저장하기 위한 폴더 생성

```
batch_size = 64
MODEL_DIR = './models'
FLOAT_MODEL = 'float_model.h5'
QAUNT_MODEL = 'quantized_model.h5'

if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)
```

MNIST dataset을 다운 받기

```
In [2]: (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

print('Training data: {}, {}'.format(x_train.shape, y_train.shape))
print('Test data: {}, {}'.format(x_test.shape, y_test.shape))

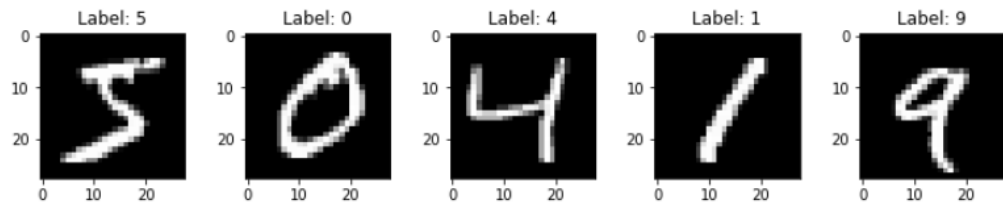
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz
11493376/11490434 [=====] - 3s 0us/step
Training data: (60000, 28, 28), (60000,)
Test data: (10000, 28, 28), (10000,)
```

MNIST data visualization

```
In [3]: fig, axs = plt.subplots(1, 5, figsize=(10, 10))
plt.tight_layout()

for i in range(5):
    axs[i].imshow(x_train[i], 'gray')
    axs[i].set_title('Label: {}'.format(y_train[i]))
```

MNIST 학습 실습



## Data normalization

```
# Data Normalization
x_train = x_train.reshape((60000,28,28,1)).astype('float32') / 255.0
y_train = keras.utils.to_categorical(y_train)

x_test = x_test.reshape((10000,28,28,1)).astype('float32') / 255.0
y_test = keras.utils.to_categorical(y_test)
```

학습 하기 위해 MNIST dataset을 dataset을 성질에 맞게 보통 다음 세가지로 분류한다. 각각 모델을 학습하고 검증하고 평가하는데에 목적을 서로 다르게 두고있다.

1. Train Set
2. Validation Set
3. Test Set

이번 실험에서는 아래와 같이 dataset을 정해진 비율로 구분하여 진행한다. 실험 2에서는 dataset에 비율을 hyper-parameter으로 이용하여 전체 데이터를 다시 학습시킴으로써 모델이 조금 더 튜닝되도록 만든다.

Training data (55,000개)		Test data (10,000개)		Validation data (5,000개)	
mnist.train		mnist.test		mnist.validation	
images	labels	images	labels	images	labels

```
# Create saperated datasets for train,validate,test
train_dataset = tf.data.Dataset.from_tensor_slices((x_train[:50000], y_train[:50000])).batch(batch_size)
val_dataset = tf.data.Dataset.from_tensor_slices((x_train[50000:], y_train[50000:])).batch(batch_size)
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(batch_size)
```



## b) MNIST 학습 model 생성

이번 실험에서는 Tensorflow Keras API를 통해 MNIST 학습 모델을 만들어 본다. Keras에서 컴파일하기 위해 필요한 두 개의 매개변수(hyper-parameter)가 있다. (hyper-parameter에 대해 실험 2번에서 자세히 배울 예정)

1. Optimizer
2. Loss function

Keras API에 지원된 옵티마이저 및 loss function에 대해 Keras API reference 홈페이지에 나와 있다.

사용가능한 옵티마이저의 종류:

- Stochastic Gradient Descent (SGD)
- AdaGrad
- RMSProp
- AdaDelta
- Adam

<https://keras.io/ko/optimizers/>

사용가능한 loss function의 종류:

- Mean squared error
- Mean absolute error
- Mean squared logarithmic error
- Categorical hinge
- Categorical cross entropy
- Sparse categorical cross entropy
- Binary cross entropy

<https://keras.io/ko/losses/>

아래 예시에서는 옵티마이저 RMSProp 사용하여 loss function은 Categorical cross entropy loss function을 사용한다.

## MNIST model 생성

```
def customcnn():
    # create a cnn model
    inputs = keras.Input(shape=(28,28,1))
    x = layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1))(inputs)
    x = layers.MaxPooling2D((2,2))(x)
    x = layers.Conv2D(64, (3,3), activation='relu')(x)
    x = layers.MaxPooling2D((2,2))(x)
    x = layers.Conv2D(64, (3,3), activation='relu')(x)
    x = layers.Flatten()(x)
    x = layers.Dense(64, activation='relu')(x)
    outputs = layers.Dense(10, activation='softmax')(x)

    model = keras.Model(inputs=inputs, outputs=outputs, name='mnist_customcnn_model')
    model.summary()

    # Compile the model
    model.compile(optimizer="rmsprop",
                  loss="categorical_crossentropy",
                  metrics=['accuracy']
                  )

    return model
```

생성된 모델 아키텍처 출력 예시

```
: # build cnn model
print("\nCreate custom cnn..")
model = customcnn()
```

Create custom cnn..  
Model: "mnist\_customcnn\_model"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 64)	36928
flatten (Flatten)	(None, 576)	0
dense (Dense)	(None, 64)	36928
dense_1 (Dense)	(None, 10)	650
=====		
Total params: 93,322		
Trainable params: 93,322		
Non-trainable params: 0		

## c) 생성한 MNIST model 학습

### 5 epochs 만큼 학습

```
# Train the model for 5 epochs using a dataset
print("\nFit on dataset..")
history = model.fit(train_dataset, epochs=5, validation_data=val_dataset)
```

```
Fit on dataset..
Epoch 1/5
782/782 [=====] - 8s 10ms/step - loss: 0.1982 - accuracy: 0.9387 - val_loss: 0.0633 - val_accuracy: 0.9827
Epoch 2/5
782/782 [=====] - 7s 10ms/step - loss: 0.0512 - accuracy: 0.9845 - val_loss: 0.0558 - val_accuracy: 0.9858
Epoch 3/5
782/782 [=====] - 7s 9ms/step - loss: 0.0346 - accuracy: 0.9898 - val_loss: 0.0471 - val_accuracy: 0.9877
Epoch 4/5
782/782 [=====] - 7s 9ms/step - loss: 0.0251 - accuracy: 0.9927 - val_loss: 0.0432 - val_accuracy: 0.9898
Epoch 5/5
782/782 [=====] - 7s 9ms/step - loss: 0.0187 - accuracy: 0.9948 - val_loss: 0.0446 - val_accuracy: 0.9897
```

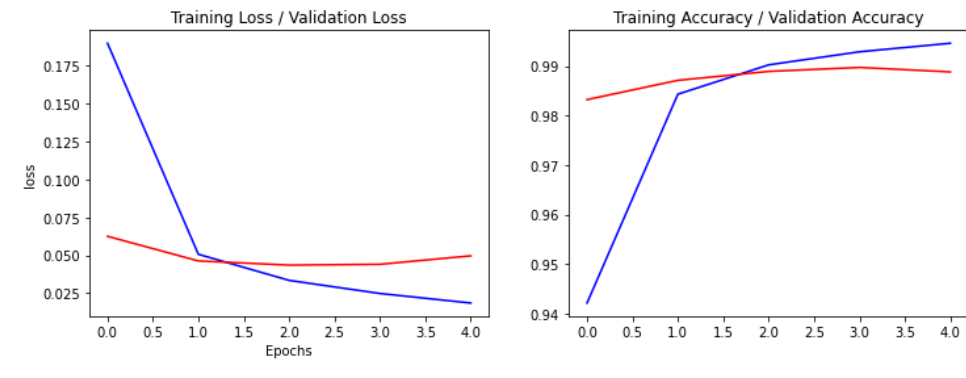
### 학습 과정을 그래프로 출력

Training Loss 와 Validation Loss 그래프를 통해 model overfitting 되는지 확인할 수 있다. (파란색 : Training, 빨간색 : Validation) 학습 속도 조절을 위해 epochs 값과 batch\_size 값을 임의로 조정해놓아서 과적합이 발생하는 시점을 확인할 수 없지만, 보통 epochs 값이 지나치게 클 수록 과적합이 발생하여 실제 검증 정확도 val\_acc는 점점 하락하는 것을 확인할 수 있다.

```
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

axs[0].plot(history.history['loss'], 'b')
axs[0].plot(history.history['val_loss'], 'r')
axs[0].set_title('Training Loss / Validation Loss')
axs[0].set_xlabel='Epochs', ylabel='loss'

axs[1].plot(history.history['accuracy'], 'b')
axs[1].plot(history.history['val_accuracy'], 'r')
axs[1].set_title('Training Accuracy / Validation Accuracy')
```



## 테스트 셋에서의 loss와 accuracy 확인

```
# Evaluate model with test data
print("\nEvaluate model on test dataset..")
loss, acc = model.evaluate(test_dataset) # returns loss and metrics
print("Test Loss: %.3f" % loss)
print("Test Accuracy: %.3f" % acc)
```

```
Evaluate model on test dataset..
157/157 [=====] - 1s 3ms/step - loss: 0.0358 - accuracy: 0.9894
Test Loss: 0.036
Test Accuracy: 0.989
```

## 학습된 모델을 저장

```
# Save model
path = os.path.join(MODEL_DIR, FLOAT_MODEL)
print("\nSave trained model to{}".format(path))
model.save(path)
```

Save trained model to ./models/float\_model.h5.

models 폴더에 float\_model.h5 파일 생성된다.

jupyter Quit Logout

Files Running Clusters

Select items to perform actions on them.

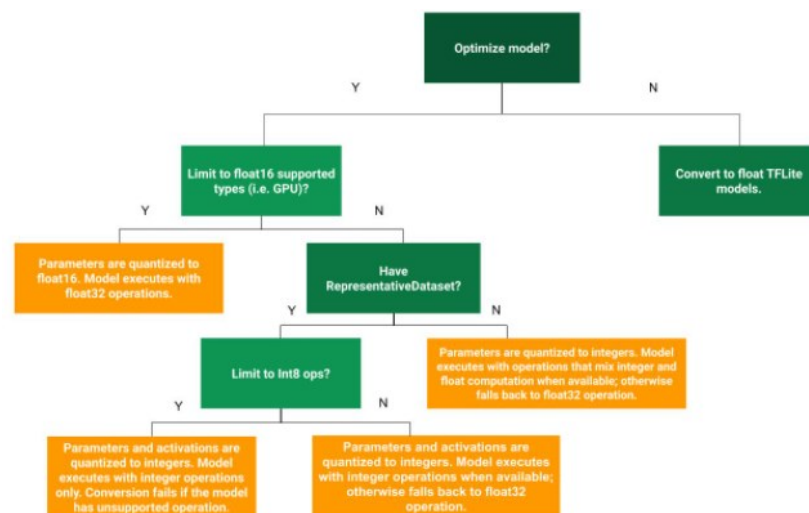
Upload New 🔗

	Name	Last Modified	File size
<input type="checkbox"/>	0		
<input type="checkbox"/>	..	seconds ago	
<input type="checkbox"/>	float_model.h5	an hour ago	798 kB

## d) MNIST model quantization

훈련 후 양자화는 모델 정확성을 거의 저하시키지 않으면서 CPU 및 하드웨어 가속기 지연 시간을 개선하고 모델 크기를 줄일 수 있는 변환 기술이다. TensorFlow Lite 변환기를 사용하여 TensorFlow Lite 형식으로 변환할 때 이미 훈련된 부동 TensorFlow 모델을 양자화할 수 있다. 선택할 수 있는 몇 가지 훈련 후 양자화 옵션이 있다. 다음은 선택 항목과 선택 항목이 제공하는 이점에 대한 요약표이다.

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU



### 32-bit tflite model : not quantized

```
model = tf.keras.models.load_model('./models/float_model.h5')
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
#saving converted model in "converted_model.tflite" file
open("./models/converted_model.tflite", "wb").write(tflite_model)
```

### 16-bit float quantization model

가중치를 16 bit 부동 소수점 숫자에 대한 IEEE 표준인 float16으로 양자화하여 부동 소수점 모델의 크기를 줄일 수 있다. 가중치의 float16 양자화를 활성화하려면 다음 스텝을 사

용한다.

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]
tflite_quant_model_float16 = converter.convert()
#saving converted model in "converted_model.tflite" file
open("./models/converted_quant_model_float16.tflite", "wb").write(tflite_quant_model_float16)
```

## 8-bit integer quantization model

완전한 정수 양자화의 경우, 모델에 있는 모든 부동 소수점 텐서의 범위, 즉 (최소, 최대)를 보정하거나 추정해야 합니다. 가중치 및 편향과 같은 상수 텐서와 달리 모델 입력, 활성화 (중간 계층의 출력) 및 모델 출력과 같은 가변 텐서는 몇 가지 추론 주기를 실행하지 않는 한 보정할 수 없습니다. 결과적으로, 변환기는 이를 보정하기 위해 대표적 데이터셋이 필요합니다. 이 데이터셋은 훈련 또는 검증 데이터의 작은 하위 집합(약 100 ~ 500개 샘플)일 수 있습니다. 아래의 `representative_data_gen()` 함수를 참조하세요.

```
def representative_data_gen():
    for input_value in tf.data.Dataset.from_tensor_slices(x_test).batch(1).take(100):
        yield [input_value]
```

## 8-bit activations with 8-bit weight

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
# Ensure that if any ops can't be quantized, the converter throws an error
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
# Set the input and output tensors to uint8 (APIs added in r2.3)
converter.inference_input_type = tf.uint8
converter.inference_output_type = tf.uint8

tflite_model_quant_int8 = converter.convert()
#saving converted model in "converted_model.tflite" file
open("./models/converted_quant_model_int8.tflite", "wb").write(tflite_model_quant_int8)
```

## 16-bit activations with 8-bit weight

'정수 전용' 방식과 유사하지만 활성화는 16 bit 범위에 따라 양자화되고 가중치는 8 bit 정수로 양자화되고 바이어스는 64 bit 정수로 양자화된다. 이를 16x8 양자화라고 한다.

```

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.representative_dataset = representative_data_gen
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_ops = [tf.lite.OpsSet.EXPERIMENTAL_TFLITE_BUILTINS_ACTIVATIONS_INT16_WEIGHTS_INT8]
tflite_quant_model_act16_wei_8 = converter.convert()
#saving converted model in "converted_model.tflite" file
open("./models/converted_quant_model_act16_wei_8.tflite", "wb").write(tflite_quant_model_act16_wei_8)

```

INFO:tensorflow:Assets written to: /tmp/tmpj0w9xxra/assets

INFO:tensorflow:Assets written to: /tmp/tmpj0w9xxra/assets

105328

## Quantization Model와 32-bit float model 비교

```

print("32-bit Float model in Mb:",
      os.path.getsize('./models/converted_model.tflite') / float(2**20))
print("16-bit Float Quantized model in Mb:",
      os.path.getsize('./models/converted_quant_model_float16.tflite') / float(2**20))
print("Compression ratio:",
      os.path.getsize('./models/converted_model.tflite')/os.path.getsize('./models/converted_quant_model_float16.tflite'))

```

32-bit Float model in Mb: 0.3596458435058594  
 16-bit Float Quantized model in Mb: 0.183135986328125  
 Compression ratio: 1.9638185302449591

```

print("32-bit Float model in Mb:",
      os.path.getsize('./models/converted_model.tflite') / float(2**20))
print("8-bit int Quantized model in Mb:",
      os.path.getsize('./models/converted_quant_model_int8.tflite') / float(2**20))
print("Compression ratio:",
      os.path.getsize('./models/converted_model.tflite')/os.path.getsize('./models/converted_quant_model_int8.tflite'))

```

32-bit Float model in Mb: 0.3596458435058594  
 8-bit int Quantized model in Mb: 0.0996246337890625  
 Compression ratio: 3.610009189768724

```

print("32-bit Float model in Mb:",
      os.path.getsize('./models/converted_model.tflite') / float(2**20))
print("16-bit(A) 8-bit(W) int Quantized model in Mb:",
      os.path.getsize('./models/converted_quant_model_act16_wei_8.tflite') / float(2**20))
print("Compression ratio:",
      os.path.getsize('./models/converted_model.tflite')/os.path.getsize('./models/converted_quant_model_act16_wei_8.tflite'))

```

32-bit Float model in Mb: 0.3596458435058594  
 16-bit(A) 8-bit(W) int Quantized model in Mb: 0.1004486083984375  
 Compression ratio: 3.5803964757709252

## Xilinx ZCU104 FPGA 위한 quantization model

실험 3번에서 Xilinx ZCU104 FPGA 보드에 MNIST quantization model를 올려 model 정확도 확인하기 위해 아래와 같이 Vitis AI API를 이용해 8-bit INT quantization model를 만들어 본다. (Vitis AI API에서는 8-bit INT quantization만 지원)

Keras Model quantization을 하기 위해 Vitis AI API를 사용해야한다. Vitis AI은 Xilinx에서 만든 Embedded Xilinx Board 위한 neural network quantization API이며 아래와 같이 API를 사용할 수 있다.

```
import tensorflow.keras.models as models
from tensorflow_model_optimization.quantization.keras import vitis_quantize
```

Vitis AI API github page (<https://github.com/Xilinx/Vitis-AI>)

저장된 MNIST model 가지고 오기

```
# Load the floating point trained model
print('Load float model..')
path = os.path.join(MODEL_DIR, FLOAT_MODEL)
try:
    float_model = models.load_model(path)
except:
    print('\nError:load float model failed!')

Load float model..
```

get input dimensions of the floating-point model

```
# get input dimensions of the floating-point model
height = float_model.input_shape[1]
width = float_model.input_shape[2]
```

vitis\_quatize을 사용해 Xilinx 보드 위한 quantization model 생성

```
# Run quantization
print('\nRun quantization..')
quantizer = vitis_quantize.VitisQuantizer(float_model)
quantized_model = quantizer.quantize_model(calib_dataset=test_dataset)
```

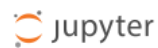
```
Run quantization..
[VAI INFO] Start CrossLayerEqualization...
10/10 [=====] - 1s 63ms/step
[VAI INFO] CrossLayerEqualization Done.
[VAI INFO] Start Quantize Calibration...
157/157 [=====] - 3s 19ms/step
[VAI INFO] Quantize Calibration Done.
[VAI INFO] Start Post-Quantize Adjustment...
[VAI INFO] Post-Quantize Adjustment Done.
[VAI INFO] Quantization Finished.
```



## Quantization model 저장

```
# Save quantized model
path = os.path.join(MODEL_DIR, QAUNT_MODEL)
quantized_model.save(path)
print('\nSaved quantized model as', path)
```

Saved quantized model as ./models/quantized\_model.h5

[Quit](#)[Logout](#)[Files](#)[Running](#)[Clusters](#)

Select items to perform actions on them.

[Upload](#)[New](#)

<input type="checkbox"/> 0	<a href="#">/ models</a>	Name	Last Modified	File size
	..		seconds ago	
<input type="checkbox"/>	float_model.h5		9 minutes ago	798 kB
<input type="checkbox"/>	quantized_model.h5		2 minutes ago	454 kB

## Quantized model 검증

```
path = os.path.join(MODEL_DIR, QAUNT_MODEL)
with vitis_quantize.quantize_scope():
    quantized_model = models.load_model(path, compile=False)
```

```
# Compile the model
print('\nCompile model..')
quantized_model.compile(optimizer="rmsprop",
                        loss="categorical_crossentropy",
                        metrics=['accuracy'])
```

Compile model..

```
# Evaluate model with test data
print("\nEvaluate model on test Dataset")
loss, acc = quantized_model.evaluate(test_dataset) # returns loss and metrics
print("Test Loss: %.3f" % loss)
print("Test Accuracy: %.3f" % acc)
```

Evaluate model on test Dataset

157/157 [=====] - 1s 6ms/step - loss: 0.0362 - accuracy: 0.9893

Test Loss: 0.036

Test Accuracy: 0.989

## Xilinx ZCU104 보드에 올리기위한 xmodel 생성 및 저장

```
!echo "-----"
!echo "COMPILING MODEL FOR ZCU104.."
!echo "-----"


!vai_c_tensorflow2 \
    --model ./models/quantized_model.h5 \
    --arch /opt/vitis_ai/compiler/arch/DPU CZDX8G/ZCU104/arch.json \
    --output_dir ./compiled_model/zcu104 \
    --net_name customcnn

!echo "-----"
!echo "MODEL COMPILED"
!echo "-----"
```

```
-----
COMPILING MODEL FOR ZCU104..
-----
*****
* VITIS_AI Compilation - Xilinx Inc.
*****
[INFO] Namespace(batchsize=1, inputs_shape=None, layout='NHWC', model_files=
['./models/quantized_model.h5'], model_type='tensorflow2', named_inputs_shape=N
one, out_filename='/tmp/customcnn_org.xmodel', proto=None)
[INFO] tensorflow2 model: /workspace/home/TA/models/quantized_model.h5
[INFO] keras version: 2.4.0
[INFO] Tensorflow Keras model type: functional
[INFO] parse raw model      :100%| 15/15 [00:00<00:00, 7563.66it/s]
[INFO] infer shape (NHWC)  :100%| 26/26 [00:00<00:00, 20247.29it/s]
[INFO] perform level-0 opt :100%| 2/2 [00:00<00:00, 471.80it/s]
[INFO] perform level-1 opt :100%| 2/2 [00:00<00:00, 1754.57it/s]
[INFO] generate xmodel      :100%| 26/26 [00:00<00:00, 5446.33it/s]
[INFO] dump xmodel: /tmp/customcnn_org.xmodel
[UNIOLOG][INFO] Target architecture: DPU CZDX8G_ISA0_B4096_MAX_BG2
[UNIOLOG][INFO] Compile mode: dpu
[UNIOLOG][INFO] Debug mode: function
[UNIOLOG][INFO] Target architecture: DPU CZDX8G_ISA0_B4096_MAX_BG2
[UNIOLOG][INFO] Graph name: mnist_customcnn_model, with op num: 42
[UNIOLOG][INFO] Begin to compile...
[UNIOLOG][INFO] Total device subgraph number 3, DPU subgraph number 1
[UNIOLOG][INFO] Compile done.
[UNIOLOG][INFO] The meta json is saved to "/workspace/home/TA/./compiled_model/z
cu104/meta.json"
[UNIOLOG][INFO] The compiled xmodel is saved to "/workspace/home/TA/./compiled_m
odel/zcu104/customcnn.xmodel"
[UNIOLOG][INFO] The compiled xmodel's md5sum is d5361d62af759b47d783153476f7872
b, and has been saved to "/workspace/home/TA/./compiled_model/zcu104/md5sum.tx
t"
-----
MODEL COMPILED
-----
```

생성된 모델을 보드에 다운로드

생성 모델은 [compiled\\_model/zcu104](#) 폴더에 저장됩니다.

 Quit Logout

Files Running Clusters

Select items to perform actions on them. Upload New ▾ ↻

<input type="checkbox"/> 0 ▾	/ compiled_model / zcu104	Name ▾	Last Modified	File size
	..		seconds ago	
<input type="checkbox"/>	customcnn.xmodel		a minute ago	172 kB
<input type="checkbox"/>	md5sum.txt		a minute ago	33 B
<input type="checkbox"/>	meta.json		a minute ago	178 B

## 5. 실험 후 보고서에 포함될 내용

- MNIST 학습 모델이 테스트셋에 대한 Accuracy 결과
- Quantization을 수행 후 테스트셋에 대한 Accuracy 결과
- Data Normalization을 수행하는 목적이 무엇인지 작성
- Optimizer에 따라 MNIST 학습 모델이 테스트셋에 대한 Accuracy가 달라지는지 확인
- Model 학습 시 model overfitting이 무엇인지 설명하고, 확인할 수 있는 방법을 작성
- Quantization을 수행하는 목적은 무엇인지 장단점에 대해 작성