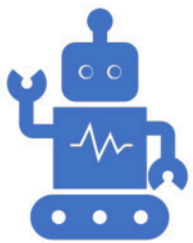
2020.08.24. ~ 2020.08.28.
경상대학교 407동 202호

Day 04



# 초보자를 위한 빅 데이터/ 머신러닝

---

# Index

# 오늘의 할 일 요약

- Colab 동작 방식을 익힌다.
- 0 부터 9 까지의  손 글씨 데이터를 분류하는 머신러닝모델을 만들어 본다.
- 지도 학습(분류) 동작 방식을 이해한다.
- 대용량 데이터를 위한 Custom Data pipeline를 익힌다.
- 동적인 구조를 표현 할 수 있는 SubClassing Model 을 생성하는 법을 익힌다.


colab

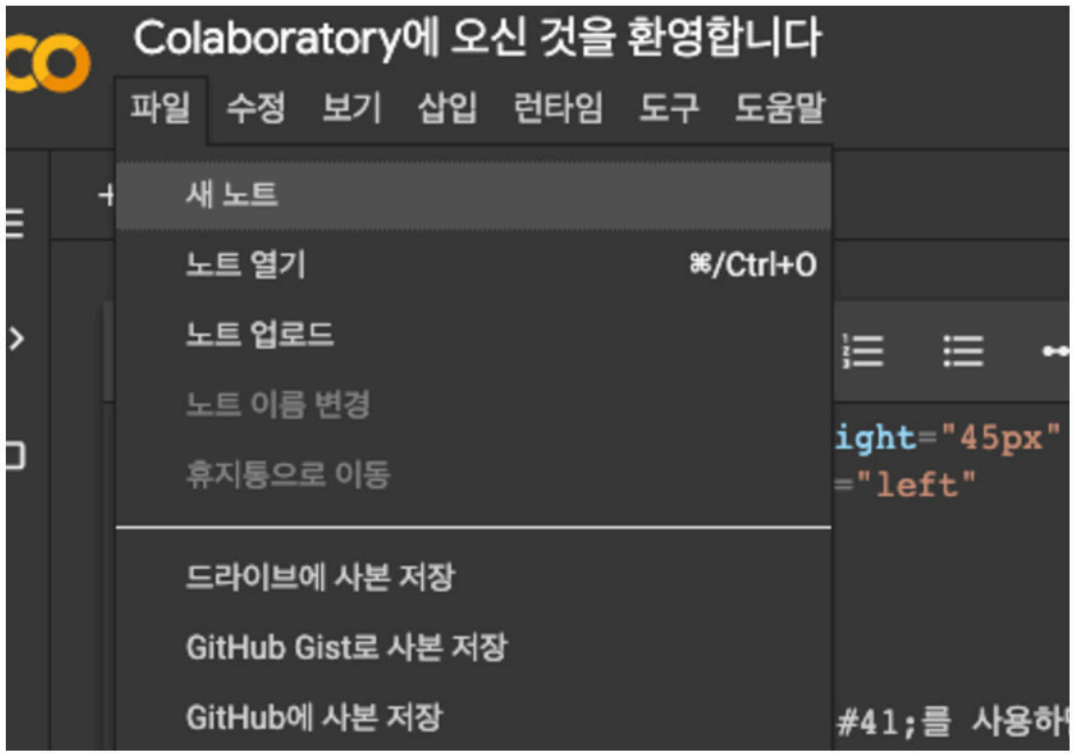
# Colab 사용법 : 1/4

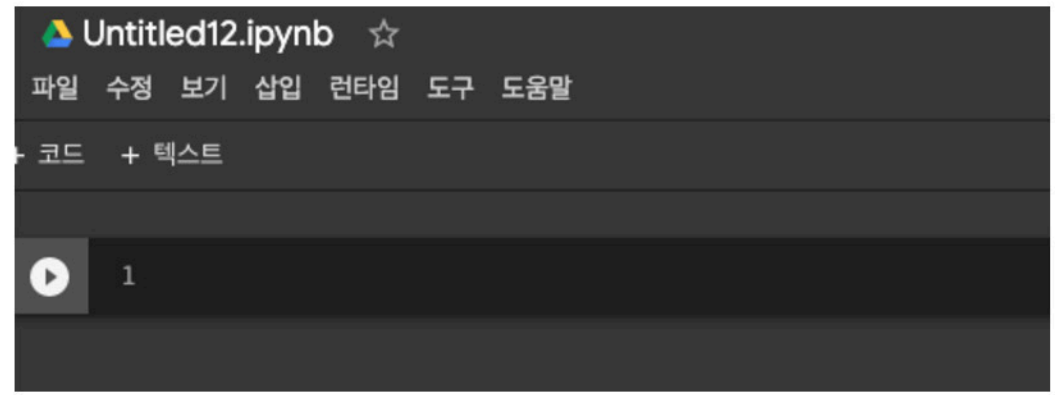
Colaboratory(colab)은 브라우저에서 python을 작성하고 실행할 수 있는 서비스

- Python2/3 지원
- GPU/TPU 무료 액세스
- jupyter notebook 기반
- 간편한 공유
- 구글 클라우드 데이터 공유

- 구글 계정 필요
- 구독하는 클라우드 데이터 용량에 따라 데이터 용량에 제한이 있음

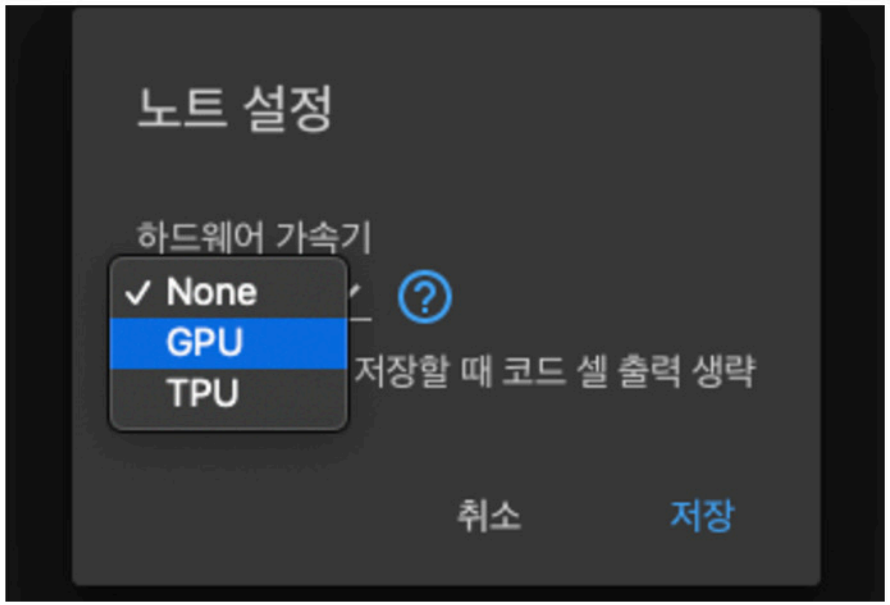
# Colab 사용법 : 2/4



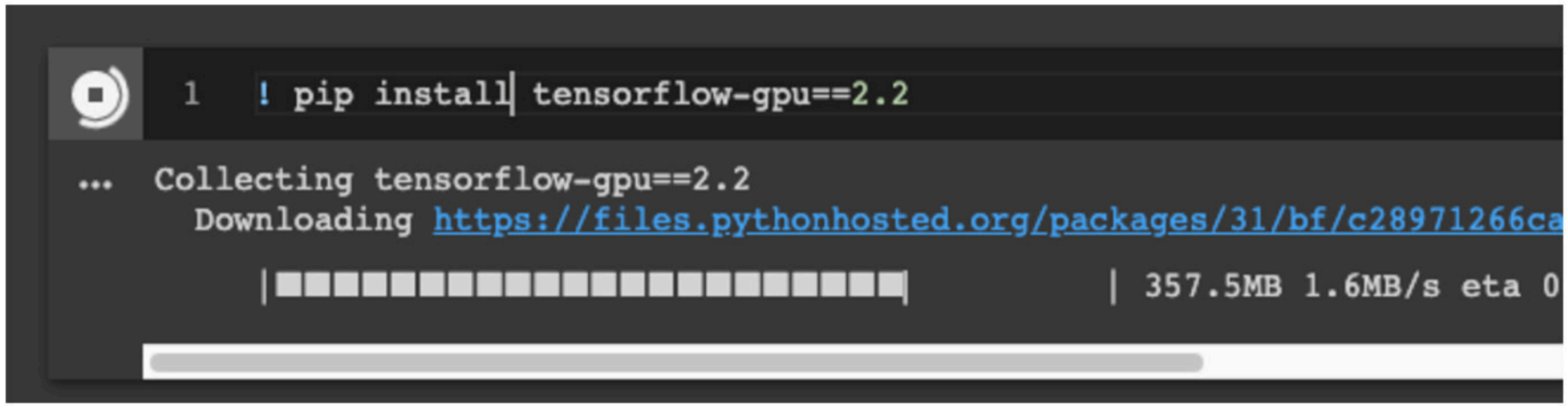

새노트 작성으로 새로운 파일을 생성하고 이 파일은 본인 계정의 구글 클라우드에 저장된다.

# Colab 사용법 : 3/4


GPU 사용을 위해서 런타임 -> 런타임 유형 변경 -> 하드웨어 가속기를 GPU로 설정 하고 저장한다.

# Colab 사용법 : 4/4




“! pip install tensorflow-gpu==2.2 “ 와 같이 ’!’를 사용해 shell 명령어를 사용해 원하는 버전의 python package를 설치 할 수 있다.

단 런타임을 다시 할당하면 설치된 내용은 사라지며 2020.08.13 기준 기본으로 설치된 tensorflow 버전은 2.3이다.

# MNIST 데이터 분류 (기본)

# 0. Hyperparameter 설정

```
leaning_rate = 1e-3
batch_size = 64
epochs = 4
```

**learning_rate** : 최적화 함수의 학습률
**batch_size**: 한번에 몇개의 데이터를 학습하는지 결정
**epochs** : 전체 데이터를 몇 회 반복 학습 하는지 결정

# 1. Python Package import

```
import os
import tensorflow as tf
from tensorflow import keras
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras import layers as l

print(tf.__version__)
```

```
import tensorflow as tf
# “tensorflow” 라는 이름의 모듈을 불러와 “tf” 라는 약칭으로 사용함.

from tensorflow import keras
# “tensorflow” 라는 모듈에서 ”keras” 라는 모듈을 불러옴
```

# 2. Load Dataset

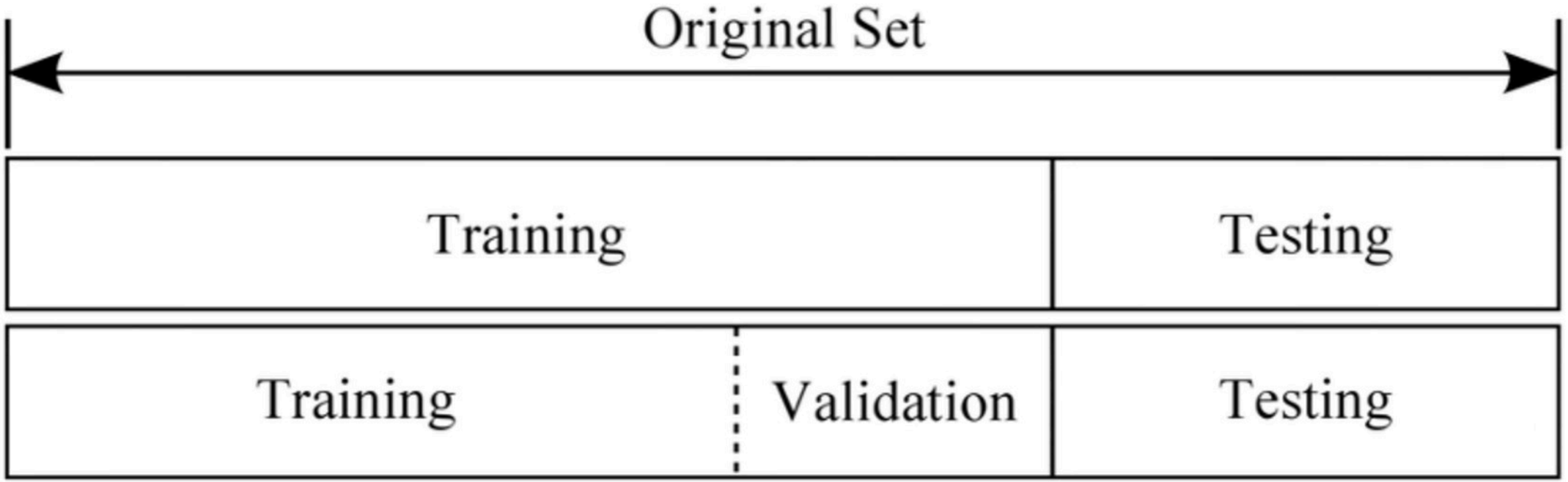
```
mnist = keras.datasets.mnist
(train_x, train_y), (test_x, test_y) = mnist.load_data()

print("Train image shape :", train_x.shape)
print("Test image shape :", test_x.shape)
print("Train target shape :",train_y.shape)
print("Test target shape :",test_y.shape)
```

```
Train image shape : (60000, 28, 28)
Test image shape : (10000, 28, 28)
Train target shape : (60000,)
Test target shape : (10000,)
```

```
keras.datasets.mnist.load_data()
# keras 모듈 내부의 mnist dataset을 불러옴.
# 불러온 데이터는 train_{x,y}, test_{x,y}로 4가지 Numpy 변수로 가지고 옴.
# Numpy 변수의 경우 *.shape를 통해 데이터 형태를 받을 수 있음
```

# 2. Load Dataset




Original Set : 모든 원본 데이터
Training : 학습에만 사용하는 데이터
Validation : 학습 중 모델 정확도 검증을 위한 데이터
Testing : 학습 후 모델 검증을 위한 데이터 **(학습과정에 사용하면 안됨)**

주로 Training, Validation, Testing을 각각 6:2:2 비율로 나누어 사용함

# 2.1. Dataset Preprocessing

```
train_x = train_x.astype(np.float32) / 255.0
train_x = np.reshape(train_x,(train_x.shape[0],-1))

test_x = test_x.astype(np.float32) / 255.0
test_x = np.reshape(test_x,(test_x.shape[0],-1))
```

학습을 위해 데이터를 사전에 처리하는 과정

```
train_x.astype(np.float32) / 255.0 # data type을 fp32로 변경하여 255로 나눔
# 데이터 범위를 0 ~ 255 에서 0 ~ 1.0 으로 정규화시킴
np.reshape(train_x,(train_x.shape[0],-1))
# [?,28,28,1]의 형태를 가지는 데이터를 [?,784]로 펼침
```

# 2.2. Training & Validation Dataset Split

```
data_size = int(len(train_x) * 0.2)
val_x = train_x[:data_size]
val_y = train_y[:data_size]
train_x = train_x[data_size:]
train_y = train_y[data_size:]
```

학습 데이터 셋을 8:2 비율로 학습, 검증 데이터셋을 분리하는 과정

```
data_size = int(len(train_x) * 0.2)
# 학습 데이터 셋의 개수(len)의 20%를 계산
val_x = train_x[:data_size]
# 개수를 기준으로 학습 데이터 셋을 나누어 검증 데이터 셋으로 보관
train_x = train_x[data_size:]
# 반대 기준으로 학습 데이터 셋을 나누어 학습 데이터 셋으로 보관
```

# 3. Build Neural Net Model

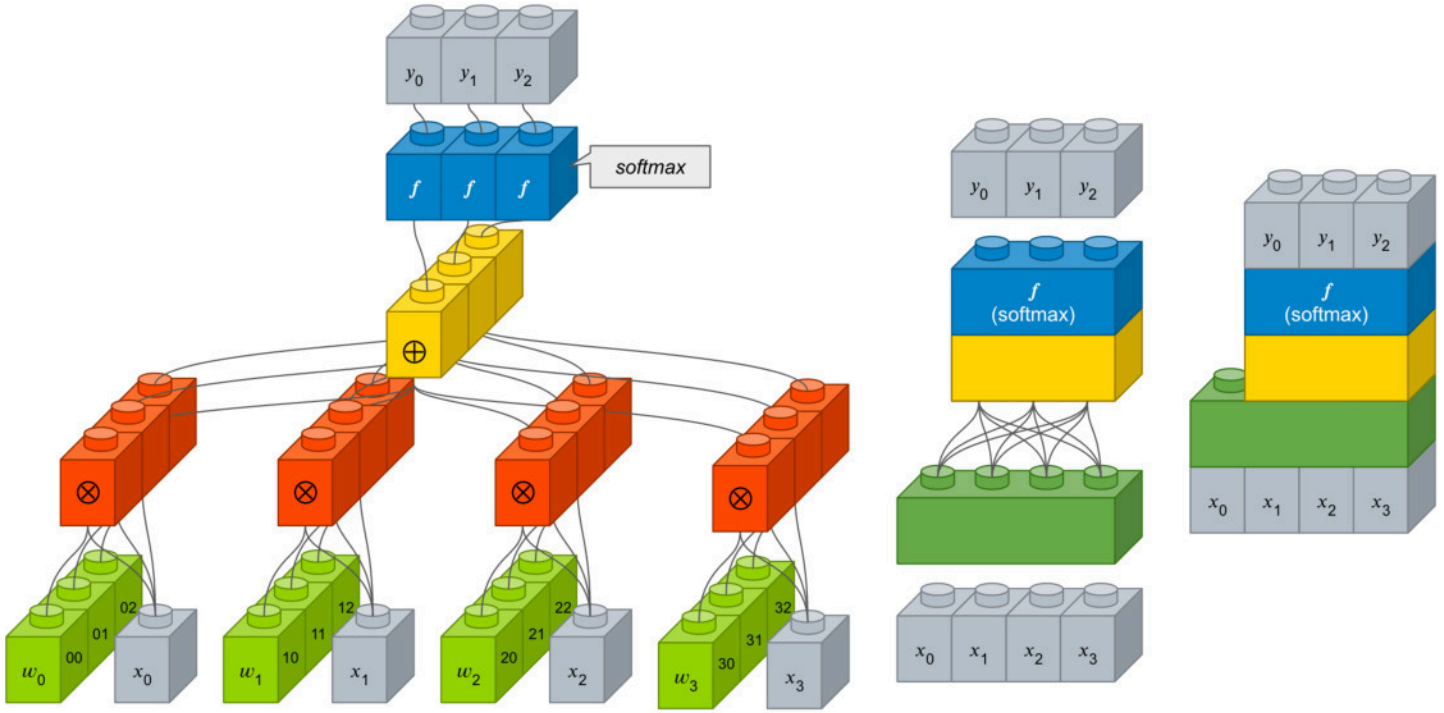
```
from tensorflow.keras import layers as l
layers = [l.Dense(128),l.ReLU(),
          l.Dense(256),l.ReLU(),
          l.Dense(512),l.ReLU(),
          l.Dense(10),l.Softmax()]
model = keras.Sequential(layers, name='MyModel’)
```

딥러닝 모델을 생성하는 과정

```
from tensorflow.keras import layers as l
# Tensorflow 페키지에서 keras 모듈 내부에 layers 모듈을 가져와 약칭을 ”l” 로 함
l.Dense(128),l.ReLU(),
# layers 내부의 Dense 레이어를 가져와 출력 개수를 128개로 설정함.
# layers 내부의 ReLU 레이어를 가져와 리스트에 추가함
model = keras.Sequential(layers, name='MyModel’)
# keras 모듈 내부의 Sequential 모듈을 가져와 레이어 리스트를 모델로 변환함
```

# 3. Build Neural Net Model

| | | |
|---|---|---|
| input_1: InputLayer | input: | [(?, 784)] |
| | output: | [(?, 784)] |
| dense: Dense | input: | (?, 784) |
| | output: | (?, 64) |
| dense_1: Dense | input: | (?, 64) |
| | output: | (?, 64) |
| dense_2: Dense | input: | (?, 64) |
| | output: | (?, 10) |




Sequential의 경우 레고를 쌓는것과 비슷하게 각층의 레이어를 조립할 수 있음

# 3. Build Neural Net Model

```
loss_funcion = keras.losses.SparseCategoricalCrossentropy()
optimizer = keras.optimizers.Adam(learning_rate=leaning_rate)

model.compile(optimizer=optimizer,
              loss=loss_funcion,
              metrics=[keras.metrics.SparseCategoricalAccuracy(name="acc")])
```

학습에 필요한 손실 함수 및 모델 내부 가중치를 조절하는 정책(Adam)을 생성

```
model.compile(optimizer=, loss=, metrics= )
# 모델 학습에 필요한 설정 값을 추가함
# optimizer: 모델 내부 가중치를 조정하는 정책
# loss : 손실 함수
# metrics : 정확도 측정을 위한 방법
```

# 4. Training

```
history = model.fit(x=train_x,
                    y=train_y,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_data=(val_x, val_y))
```

**Output:**

```
Epoch 1/4 750/750 [======] - 2s 2ms/step - loss: 0.5899 - acc: 0.8441 - val_loss: 0.2680 - val_acc: 0.9238
Epoch 2/4 750/750 [======] - 2s 2ms/step - loss: 0.2272 - acc: 0.9342 - val_loss: 0.1936 - val_acc: 0.9423
Epoch 3/4 750/750 [======] - 2s 2ms/step - loss: 0.1726 - acc: 0.9490 - val_loss: 0.1657 - val_acc: 0.9520
...
```

모델 학습하는 과정

```
history = model.fit(x=, y= , batch_size= ,epochs= , validation_data= )

# fit을 사용해 학습가능
# x: 입력데이터
# y: 입력라벨(지도학습)
# batch_size : 배치 사이즈
# epochs : 반복학습 횟수
# validation_data : 검증 데이터 셋
```

# 5. Model Evaluate

```
result = model.evaluate(x=test_x, y=test_y)
evaluate = dict(zip(model.metrics_names,result))
print(evaluate)
```

**Output:**

313/313 [==============================] - 1s 2ms/step - loss: 0.1345 - acc: 0.9579
{'loss': 0.1344558149576187, 'acc': 0.957899876976013}

모델 테스트 과정

```
result = model.evaluate(x=test_x, y=test_y)

# evaluate를 사용해 테스트 가능
# x: 입력 데이터
# y: 입력 라벨(지도 학습)
```

# 6. Model Save & Load (Keras Model, h5 format)

```
save_path = './mnist_cnn.h5'
save_point = model.save(save_path) # save
model = tf.keras.models.load_model(save_path) # load
```

```
# 불러온 모델을 테스트
result = model.evaluate(x=test_x, y=test_y)
evaluate = dict(zip(model.metrics_names,result))
print(evaluate)
```

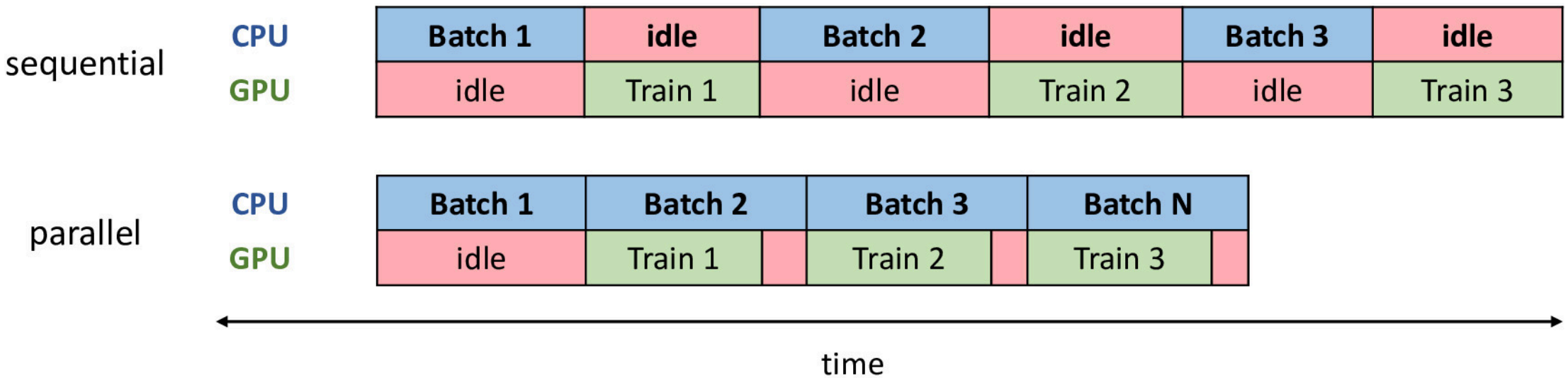
모델 저장 및 불러오기

```
model.save("저장경로")
model = tf.keras.models.load_model("불러올 경로")
# save를 사용해 모델을 '*.h5'로 저장 가능
# tf.keras.models.load_model를 사용해 저장된 '*.h5'를 모델로 불러올 수 있음
```

# MNIST 데이터 분류 (심화)

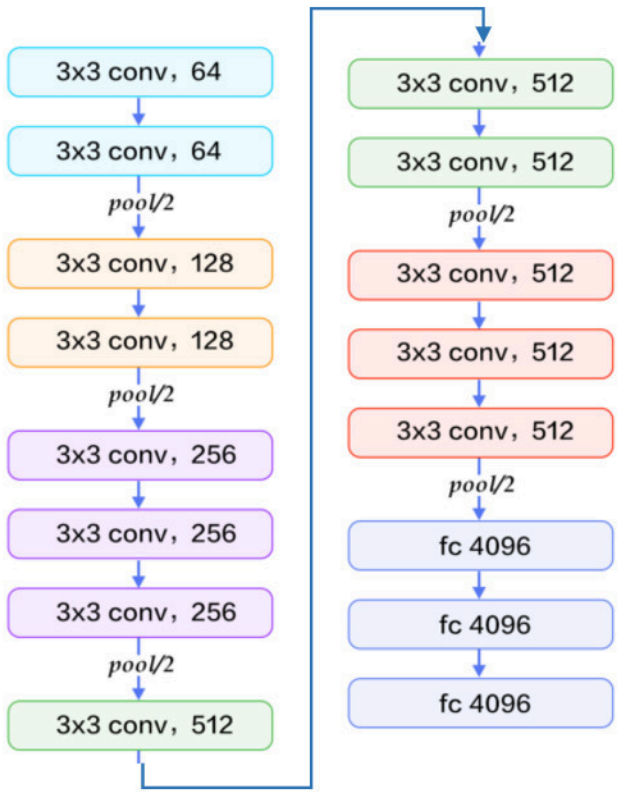
# 1. Custom DataLoader & Custom Model

Batch : data load, data preprocessing 을 포함 CPU에서 진행됨
Train : 모델 학습, GPU에서 진행됨

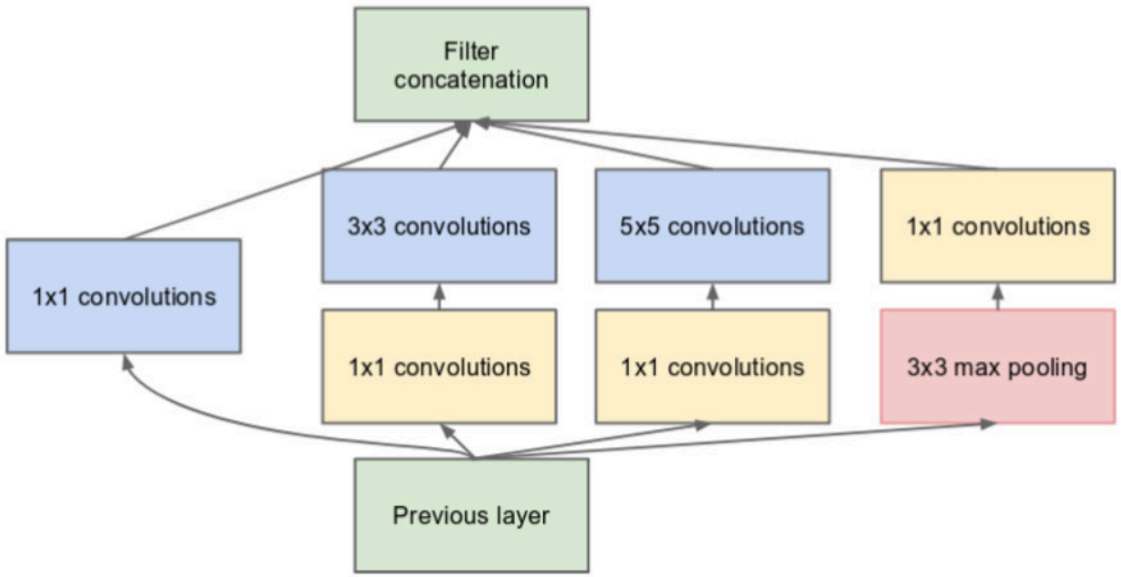



가능한 GPU(가속기) 사용이 100%가 될 수 있도록 낭비가 없도록 해야함!

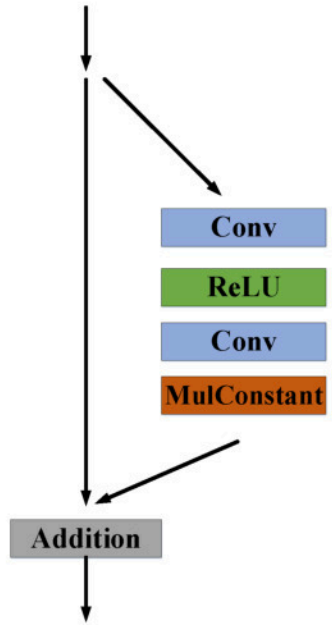
# 1. Custom DataLoader & Custom Model




VGG16 Model




Inception block




residual block

Sequential 로 표현 가능

Sequential 로 불가능

Keras.Sequential 형태로는 표현 할 수 있는 모델 구조가 한정되어 있음

Inception, residual 같은 복잡한 구조를 표현 하기 위해서는 SubClassing 방식이 필요함.

# 2. Custom Dataset Preprocessing

```
# validation / training Split Done

train_dataset = tf.data.Dataset.from_tensor_slices((train_x, train_y))
valid_dataset = tf.data.Dataset.from_tensor_slices((val_x, val_y))
test_dataset = tf.data.Dataset.from_tensor_slices((test_x, test_y))
```

커스텀 데이터 로더를 생성 tf.data.Dataset이

```
*_dataset = tf.data.Dataset.from_tensor_slices((*_x, *_y))
```

# **tf.data.Dataset.from_tensor_slices 함수를** 사용해 데이터,라벨을 입력하여 커스텀 데이터 셋을 생성

# 2. Custom Dataset Preprocessing

```
def pre_processing(image,target):
    image = tf.reshape(image,[28*28]) # Flatten
    image = tf.cast(image,dtype=tf.float32) # cast to fp32
    image /=255.0 # normalization
    return image, targe
```

pre_processing 함수를 사용해 데이터 전처리

```
image = tf.reshape(image,[28*28]) # Flatten
# 데이터 셋의 형태를 [28,28,1]인 이미지 형태에서 [784]로 펼처줌

image = tf.cast(image,dtype=tf.float32) # cast to fp32
# 데이터 타입을 uint8 에서 fp32로 변경

image /=255.0 # normalization
# 데이터 범위를 0~255에서 0~1.0으로 변경
```

# 2. Custom Dataset Preprocessing

```
def pre_processing(image,target):
    image = tf.reshape(image,[28*28]) # Flatten
    image = tf.cast(image,dtype=tf.float32) # cast to fp32
    image /=255.0 # normalization
    return image,targe

train_dataset = train_dataset.map(pre_processing)
valid_dataset = valid_dataset.map(pre_processing)
test_dataset = test_dataset.map(pre_processing)
```

custom dataset 과 전처리 함수를 연결 하는 과정

Dataset.map 을 사용해 전처리 함수(pre_processing)와 데이터 셋을 연결 할 수 있음

# 2. Custom Dataset Preprocessing

```
train_dataset = train_dataset.shuffle(train_data_size).batch(batch_size)
test_dataset = test_dataset.batch(batch_size)
valid_dataset = valid_dataset.batch(batch_size)
```

Dataset 설정을 진행

```
train_dataset = train_dataset.shuffle(train_data_size).batch(batch_size)
# Dataset.shuffle(버퍼사이즈)를 사용해 데이터를 순서를 임의로 만듦
# Dataset.batch(배치사이즈)를 사용해 데이터를 한번에 몇 개씩 가지고 올지 설정
```

Shuffle, Batch 둘 다 dataset을 반환함

# 2. Custom Dataset Preprocessing

```
print(train_dataset.element_spec)
print(test_dataset.element_spec)
print(valid_dataset.element_spec)
```

Dataset 설정을 진행

shuffle, batch둘다 dataset을 반환함, dataset.element_spec 으로 출력되는 데이터 형태를 확인 할 수 있음

# 3. Custom Model Build

```
class SubBlock(keras.Model):
    def __init__(self,units):
        super(SubBlock,self).__init__() # 꼭 해줘야 함
        self.block_1 = l.Dense(units)
        self.act = l.ReLU()

    def call(self,inputs):
        tensor = self.block_1(inputs)
        return self.act(tensor)
```

SubClassing 방식을 사용해 커스텀 모델을 생성

def __init__(self,units): 생성자
def call(self,inputs): 모델이 호출(feedforward)될 때 실행

super(SubBlock,self).__init__()
# 부모의 생성자를 실행 SubClassing 방식을 사용할 때 꼭 호출 해야함

# 3. Custom Model Build

```
class MyModel(keras.Model):
    def __init__(self,name):
        super(MyModel,self).__init__(name=name)
        self.layer_0 = SubBlock(128)
        self.layer_1 = SubBlock(256)
        self.layer_2 = SubBlock(512)
        self.drop = l.Dropout(0.5)
        self.layer_3 = l.Dense(10)

    def call(self,inputs):
        tensor = self.layer_0(inputs)
        tensor = self.layer_1(tensor)
        tensor = self.layer_2(tensor)
        tensor = self.drop(tensor)
        tensor = self.layer_3(tensor)
        return tensor
```

앞서 생성한 Dense + Relu Layer인 SubBlock을 사용해 모델을 구성

__init__에서 모델에 필요한 layer를 생성

call에서 생성한 레이어를 호출하여 입력 받은 데이터(inputs)로 Feedforward를 진행

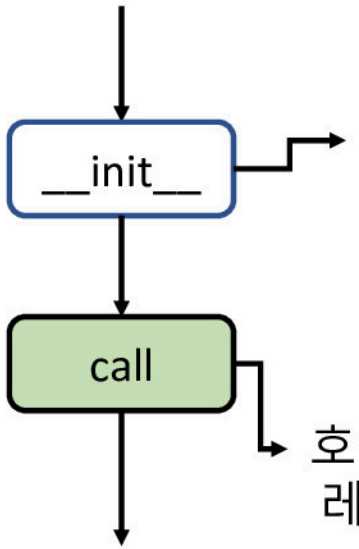
# 3. Custom Model Build

```
def build_model(input_shape,summary,name):
    model = MyModel(name = "MyModel")
    model.build(input_shape)
    if summary: model.summary()
    return model
```

Custom 모델을 생성하고, build 시키는 과정

```
model.build(input_shape)
# SubClassing 모델의 경우 build(입력 데이터 형태)를 호출해야 모델 구조를 볼 수 있음
model.summary()
# 모델 구조를 출력함
```

# 3. Custom Model Build




모델에 필요한 레이어만 생성됨
연결 구조는 모름

Call(feedforward) 가
호출되면서 메모리를 할당되고 각
레이어의 연결구조를 알 수 있음

```
class MyModel(keras.Model):
    def __init__(self,name):
        super(MyModel,self).__init__(name=name)
        self.layer_0 = SubBlock(128)
        self.layer_1 = SubBlock(256)
        self.layer_2 = SubBlock(512)
        self.drop = l.Dropout(0.5)
        self.layer_3 = l.Dense(10)

    def call(self,inputs):
        tensor = self.layer_0(inputs)
        tensor = self.layer_1(tensor)
        tensor = self.layer_2(tensor)
        tensor = self.drop(tensor)
        tensor = self.layer_3(tensor)
        return tensor
```

model.build(input_shape)는 입력 데이터 형태를 사용해 더미 데이터를 만들어 모델에 강제로 호출을 진행함. 호출이 진행되면서 연결구조를 알 수 있게 되어 model.summary()를 통해 내부 구조를 확인 할 수 있음.

# 3. Custom Model Build

```
input_shape = (None,28*28)

model = build_model(input_shape,True,'MyModel')
loss_funcion =
            keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = keras.optimizers.Adam()
```

학습을 위한 손실 함수 및 최적화 함수를 정하는 과정

```
model = build_model(input_shape,True,'MyModel’)
# build_model을 통해 모델을 생성하고 가지고 옴

keras.losses.SparseCategoricalCrossentropy(from_logits=True)
# 손실 함수 설정

optimizer = keras.optimizers.Adam()
# 최적화 함수 설정
```

# 3. Custom Model Build

```
train_loss = keras.metrics.Mean(name='train_loss')
train_accuracy =
      keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = keras.metrics.Mean(name='test_loss')
test_accuracy =
      keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
```

```
keras.metrics.Mean
# 입력 값을 받을 때 계산하는 방식이 평균인 측정방법

*_accuracy = keras.metrics.SparseCategoricalAccuracy(name='train_accuracy’)
# 정확도 측정을 위한 방법
```

# 4. Train step build

```
def train_step(images,targets):
    with tf.GradientTape() as tape:
        pred = model(images, training=True)
        loss = loss_funcion(targets, pred)
    # 기울기 계산
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients,model.trainable_variables))
    # Logging
    train_loss(loss)
    train_accuracy(targets, pred)
```

학습을 위한 함수

```
with tf.GradientTape() as tape
# 연산을 기록하여 기울기를 계산하는 객체
gradients = tape.gradient(loss, model.trainable_variables)
# 기록된 연산 내용을 사용, 오차와 모델 내부 가중치를 입력 받아 기울기를 계산
optimizer.apply_gradients(zip(gradients,model.trainable_variables))
# 계산된 기울기로 모델 가중치 값을 수정
```

# 4. Train step build

```
def train_step(images,targets):
    with tf.GradientTape() as tape:
        pred = model(images, training=True)
        loss = loss_funcion(targets, pred)
    # 기울기 계산
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients,model.trainable_variables))
    # Logging
    train_loss(loss)
    train_accuracy(targets, pred)
```

학습을 위한 함수
train_loss(loss)
train_accuracy(targets, pred)
# 계산된 오차(손실) 및 예측 정확도를 계산

# 4. Test step build

```
def test_step(images,targets):
    pred = model(images, training=False)
    loss = loss_funcion(targets, pred)

    test_loss(loss)
    test_accuracy(targets,pred)
```

검증 및 테스트를 위한 함수

```
pred = model(images, training=False)
# 학습에 사용하는 것이 아니기 때문에 training 값을 False로 입력, 학습 여부에 따라 동작 방법이 달라지는 레이어가 있음(Dropout, BatchNorm)
```

학습이 아니기 때문에 tf.GradientTape()를 사용하지 않음
기울기를 계산할 필요가 없기 때문

# 5. Training

```
for epoch in range(epochs):
      train_loss.reset_states()
      train_accuracy.reset_states()
      test_loss.reset_states()
      test_accuracy.reset_states()

      TRAIN
      VALIDATION
      Message_Print
```

학습 파이프 라인

```
for epoch in range(epochs)
# epochs 만큼의 반복을 가지고 내부에서 학습을 반복하는 2중 for 구조를 가짐

metrics.reset_states()
# epoch 정확한 값을 출력하기 위해 내부 값을 비움
```

# 5. Training

## *TRAIN*

```
pbar = tqdm(train_dataset,total=(train_data_size//batch_size)+1)
for images, targets in pbar:
   train_step(images,targets)
   pbar.set_description('[%d/%d] Loss: %0.4f, Acc: %0.4f'% (epoch+1,
      epochs, train_loss.result(), train_accuracy.result()))
```

학습 파이프 라인 ***TRAIN*** 과정

```
pbar = tqdm(train_dataset,total=(train_data_size//batch_size)+1)
# 학습 진행과정을 나타내기 위해 tqdm 패키지로 train_dataset을 래핑

for images, targets in pbar:
# pbar를 반복해 데이터와 라벨을 받음
```

# 5. Training

## *VALIDATION*

```
for images, targets in valid_dataset:
      test_step(images,targets)
```

학습 파이프 라인 ***VALIDATION*** 과정

매 epoch 마다 Valid_dataset을 반복해
데이터, 라벨을 받아 test_step(데이터,라벨)로 검증 작업을 진행

# 5. Training

## *Message_Print*

```
itr = optimizer.iterations.numpy()
msg = """
Epoch\t: [%d/%d]
Itr\t: %d
------Train------
Loss\t: %0.4f
Acc\t: %0.4f
------Vaild------
Loss\t: %0.4f
Acc\t: %0.4f
"""
print(msg % (epoch+1, epochs, itr,
train_loss.result(), train_accuracy.result(),
test_loss.result(), test_accuracy.result()))
```

학습 파이프 라인 ***Message_Print*** 과정

optimizer.iterations.numpy()
# 옵티마이저의 스텝 횟수를 가지고 옴
# *.numpy()의 경우 Tensorflow의 데이터를 numpy 데이터로 변환해 가지고 옴

metrics.result()
# 계산된 메트릭 값을 반환함

# 6. Model Evaluate

```
test_loss.reset_states()
test_accuracy.reset_states()

for images, targets in tqdm(test_dataset):
      test_step(images,targets)

msg = """
------Test------
Loss\t: %0.4f
Acc\t: %0.4f
---------------- """
print(msg % (test_loss.result(), test_accuracy.result()))
```

학습(for문)을 나와 테스트 데이터로 모델의 정확도를 측정

# 7. Model Save & Load (Tensorflow SavedModel format)

```
tf.saved_model.save(model,'MyModel')
load_model = tf.saved_model.load('MyModel’)
```

tf.saved.model을 사용해 모델의 저장 및 불러오기가 가능
save(모델, 저장경로)를 입력하면 저장경로로 모델을 저장함.

저장된 모델은 폴더로 관리되며 모델구조, 모델 가중치, 모델 실행 그래프로 관리된다.

```
load_model = tf.saved_model.load('MyModel’)
# 저장된 모델 경로를 입력하여 모델을 불러온다.
```

SavedModel 폴더 구조

# 7. Model Save & Load (Tensorflow SavedModel format)

```
for images, targets in tqdm(test_dataset):
    pred = load_model(images,False)
    loss = loss_funcion(targets, pred)
    test_loss(loss)
    test_accuracy(targets,pred)

msg = """------Test------
         Loss\t: %0.4f
         Acc\t: %0.4f
         ----------------"""
print(msg % (test_loss.result(), test_accuracy.result()))
```

테스트를 진행하여 이전 테스트와 비교하여 모델과 가중치가 정확하게 불러와졌는지 확인한다.