3장 인공지능 활용

3.1 Tensorflow/keras 개요

3.2 ANN

3.3 CNN을 활용한 이미지 데이터 분석

3.1 Tenseoflow/keras 개요

1) Tensorflow

많은 사람이 인공지능 프레임 워크라고 알고 있는 텐서플로우는 구글 브레인 팀에서 개발한 수치 계산용 라이브러리(고성능 미분기) 이다. 머신러닝과 관련된 계산작업을 수행할 때는 기존의 컴퓨터 연산으로는 시간이 오래 걸리는 경우가 많은데, 텐서플로우는 CUDA (Computed Unified Device Architecture) 라는 NVIDA의 GPU 개발 툴을 이용해 많은 양의 연산을 병렬처리 할 수 있도록 도와준다.

<텐서플로우 로고>

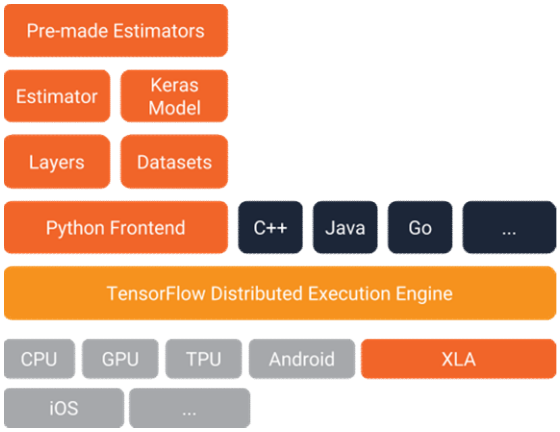
기존의 컴퓨터는 CPU를 사용해 연산하고, Python의 경우 Single-Core를 이용해 연산을 처리하지만 머신러닝의 세계에서는 복잡하고 어려운 연산을 처리하는 것보단 간단한 연산을 최대한 많이 그리고 빠르게 처리하는 것을 우선으로 한다. 이는 앞부분에서도 이야기한 벡터 연산과 연관이 있다. 벡터 연산은 단일로 처리할 경우 일반적인 연산모형과 다를게 없지만, 벡터를 모아 행렬로 처리하게 되면 많은 양을 한꺼번에 효율적으로 처리해줄 수 있다. 이를 이용해 머신러닝 알고리즘들을 적용하는 것이다. 이러한 행렬연산에 특화된 도구가 GPU이다. GPU는 CPU가 어렵고 복잡한 연산을 하기 위한 8~16개의 코어를 갖고 있는데 반해 몇백 개에서 몇천 개의 단순한 연산을 처리할 수 있도록 만들어졌다.

머신러닝을 위해 사용하는 Pandas, Numpy, Matplotlib 등등은 행렬연산을 기본으로 한다. 따라서 병렬처리가 가능한 GPU를 선호하는 것이다. 이러한 GPU를 이용한 연산을 통해 텐서플로우는 다음과 같은 기능들을 제공한다.

- Nympy 자료 구조와 비슷하지만, GPU를 지원한다.
- 분산 컴퓨팅을 지원한다.
- 고성능 최적화 기능을 제공하여 모든 종류의 손실함수를 쉽게 최소화 할

수 있다.

가장 저수준에서 텐서플로우 연산은 C++ 코드로 구현되어 있어 빠르고 효율적으로 동작하며, 고수준 API인 Keras 를 통해 이를 호출해 연산을 진행한다.



<텐서플로우의 구조>

2) Keras

케라스는 신경망을 쉽게 만들고, 훈련, 평가, 실행할 수 있는 텐서플로우의 API이다. 직관적이며 유연하므로 다양한 분야의 딥러닝 개발자들이 사용하고 있다. 텐서플로우의 2버전이 출시되면서 자체적으로 tf.keras를 번들로 포함했고 백 앤드에서 텐서플로우를 지원하도록 만들어 주었다. 케라스가 많은 머신러닝 개발자에게 사랑받는 이유는 다음과 같은 특징 덕분이다.

- 동일한 코드로 CPU와 GPU에서 실행할 수 있다.
- 사용하기 쉬운 API를 가지고 있어 딥러닝 모델을 쉽고 빠르게 생성해 낼수 있다.
- 입력과 다중 출력 등 어떤 네트워크 구조도 만들 수 있다.

케라스를 사용해 신경망을 생성하기 위해서는 다음과 같은 신경망 훈련에 관련된 요소들에 대해 이해하고 있어야 한다.

- 네트워크를 구성하는 층

- 입력 데이터와 그에 맞는 DATA Target
- 학습에 사용할 손실함수
- 학습 진행방식을 결정한 Optimizer

(1) Layer

Layer란 신경망을 구성하는 층으로 “하나 이상의 텐서를 입력받고 하나 이상의 텐서를 출력하는 데이터 처리 모듈”을 의미한다. 이 모듈에는 모두 가중치라는 상태를 지니고 있다. (가중치에 관한 이야기는 신경망에서 이어가도록 하자) Layer는 다음과 같은 기본적인 종류를 가지고 있다

- 완전 연결층 (Fully Connected Layer)
- 밀집층 (Dense Layer)
- 순환층(Recurrent Layer)
- 합성곱층(Convolution Layer)

각 레이어는 구현하고자 하는 목표에 맞게끔 다양한 조합으로 구성할 수 있으며, 이 책에서는 모든 Layer를 다룰 예정이다. 다음 코드는 밀집 층을 구현한 코드로 다음과 같이 정리 할 수 있다.

```
from tensorflow.keras import layers
layer = layers.Dense(16, input_shape=(512,)
```

- 첫 번째 차원이 512인 2차원 텐서를 입력(input의 개수가 512)
- 차원의 크기가 16인 변환된 텐서를 출력(output의 개수가 16)

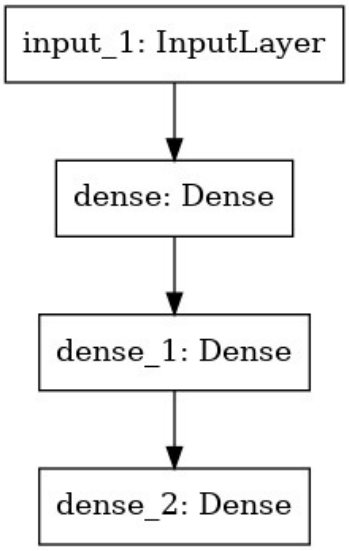
위와 같이 레이어를 쌓으면 모델이 완성된다. 이때, 케라스에서는 모델에 추가된 층을 서로 호환되게 맞추어 주기 때문에 크게 걱정하지 않아도 된다. 아래 예제는 케라스를 이용해 Dense Layer를 3개 쌓은 모델이다.

```
model = models.Sequential()
model.add(layers.Dense(16, input_shape=(512,))
model.add(layer.Dense(10))
model.add(layer.Dense(20))
```

(2) 모델 설계

Keras 에서 모델을 설계하는 방식은 크게 Sequential API를 사용하는 방식과 Keras functional API를 사용하는 방식으로 나뉜다. Sequential API는

Layer를 선형으로 쌓은 것으로 쉽고 간단하게 모델을 설계할 수 있다는 장점이 있다. Sequential API로 설계된 모델을 확인해 보면 다음과 같다.



<Sequential API를 이용해 생성한 model>

<예제1>

```
model = models.Sequential()
model.add(layers.Dense(16, input_shape=(512,))
model.add(layer.Dense(10))
model.add(layer.Dense(20))
```

<예제2>

```
model = keras.Sequential()
model.add(keras.Input(shape=(250, 250, 3)))  # 250x250 RGB images
model.add(layers.Conv2D(32, 5, strides=2, activation="relu"))
model.add(layers.Conv2D(32, 3, activation="relu"))
model.add(layers.MaxPooling2D(3))
```

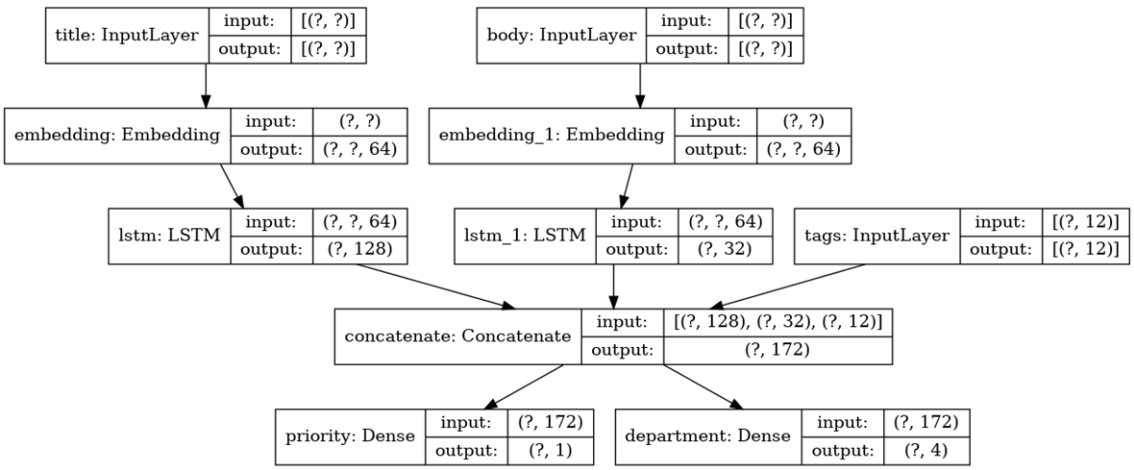
Sequential API는 선형적인 모델만 설계 가능하므로 여러 층을 공유하거나 다양한 종류의 입력과 출력을 사용하는 복잡한 모델을 만드는 일을 하기에는 명백한 한계가 존재한다. 따라서 이러한 단점을 해결하기 위해 Functional API를 사용하여 모델을 설계한다. Functional API는 Sequential API보다 더 유연한 모델을 설계하는 방법으로 비선형 Topology, 공유 Layer, 다중 입력, 다중 출력으로 모델을 처리할 수 있다. 위의 그림과 같은 똑같은 모델을

Functional API로 구현하기 위해선 다음과 같은 과정을 거친다.

1. 입력 노드를 생성한다.
2. 입력 노드 객체에서 레이어를 호출하여 새 노드를 형성
3. 레이어 추가
4. 입력과 출력을 지정하여 Model 생성

```
# 과정1
inputs = keras.Input(shape=(512,))

# 과정2
dense = layers.Dense(16, activation="relu")
x = dense(inputs)

# 과정3
x = layers.Dense(10, activation="relu")(x)
outputs = layers.Dense(20)(x)

# 과정4
model = keras.Model(inputs=inputs, outputs=outputs, name="mnist_model")
```

Functional API의 특징을 이용해 좀 더 복잡한 Model을 설계할 수도 있다.



<Functional API를 이용해 생성한 model>

```
num_tags = 12  # Number of unique issue tags
num_words = 10000  # Size of vocabulary obtained when preprocessing text data
num_departments = 4  # Number of departments for predictions

title_input = keras.Input(
    shape=(None,), name="title"
)  # Variable-length sequence of ints
body_input = keras.Input(shape=(None,), name="body")  # Variable-length sequence
of ints
tags_input = keras.Input(
    shape=(num_tags,), name="tags"
)  # Binary vectors of size `num_tags`

# Embed each word in the title into a 64-dimensional vector
title_features = layers.Embedding(num_words, 64)(title_input)
# Embed each word in the text into a 64-dimensional vector
body_features = layers.Embedding(num_words, 64)(body_input)

# Reduce sequence of embedded words in the title into a single 128-dimensional
vector
title_features = layers.LSTM(128)(title_features)
# Reduce sequence of embedded words in the body into a single 32-dimensional
vector
body_features = layers.LSTM(32)(body_features)

# Merge all available features into a single large vector via concatenation
x = layers.concatenate([title_features, body_features, tags_input])

# Stick a logistic regression for priority prediction on top of the features
priority_pred = layers.Dense(1, name="priority")(x)
# Stick a department classifier on top of the features
department_pred = layers.Dense(num_departments, name="department")(x)

# Instantiate an end-to-end model predicting both priority and department
model = keras.Model(
    inputs=[title_input, body_input, tags_input],
    outputs=[priority_pred, department_pred],
)
```

예제를 통해서 어느 정도 눈치챘겠지만 Sequential API는 Functional API로

도 생성할 수 있다. 이제 모델을 생성했으니 Keras를 이용해 Deep Learning Workflow 를 탐색해 보도록 하자.

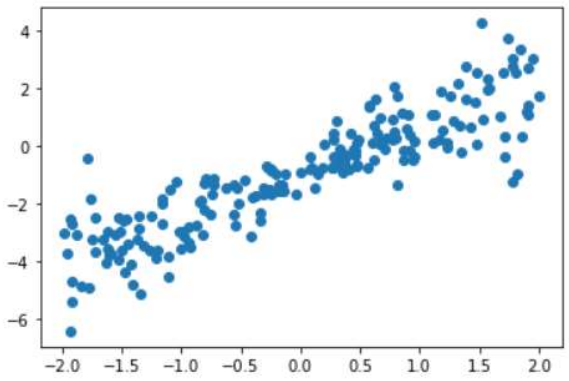
(3) model 생성 및 학습

Model의 생성 및 학습 과정은 다음과 같은 단계로 이루어진다.

1. Input Tensor와 Target Tensor로 이루어진 Training Dataset 을 정의한다.
2. Input Tensor와 Target Tensor를 연결하는 Model을 정의한다.
3. Loss function, Optimizer, Metric 값을 지정하여 Training 과정을 설정한다.
4. Training Dataset 을 이용해 Model.fit() 함수를 이용해 Model을 학습시킨다.
5. 학습된 Model 평가

위와 같은 단계를 이용해 간단한 Sequential model을 생성하고 학습시켜 보도록 하자. 우리는 간단한 회귀 분석 모델을 생성하고 학습시켜볼 예정이다.

다음과 같은 Dataset 을 보고 데이터의 분포를 예측해보자.



<선형 dataset>

위와 같은 Dataset 을 확인했을 때 우리는 직관적으로 $\hat{y}=wx+b$ 의 model을 예측한다. 이와같이 머신러닝이란 우리가 예측하지 못하는 문제를 예측하는데서 시작하는 것이 아니라, 우리가 예측할수 있는 문제를 확실하게 예측할수 있도록 만드는데서 시작해야 한다.

우선 컴퓨터로 이 문제를 해결하는 데 필요한 모듈을 호출하자.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(0) # 랜덤 모듈사용시 항상 같은결과를 나타내기위해 seed값 고정
```

- tensorflow : 모델을 생성하고 학습시키기 위한 모듈
- numpy : Dataset 을 생성하기 위한 모듈
- matplotlib : Dataset 과 예측한 Data를 시각화하기 위한 모듈

필요한 모듈을 호출하고 위의 그림과 같은 Dataset 을 생성하기 위해 함수를 만들고 데이터를 생성하고 확인하면 위와 같은 Dataset이 생성된 걸 확인할 수 있다.

```
def make_random_data():
    x = np.random.uniform(low=-2, high=2, size=200)
    y = []
    for t in x:
        r = np.random.normal(loc=0.0, scale=(0.5 + t*t/3), size=None)
        y.append(r)
    return  x, 1.726*x -0.84 + np.array(y)
```

```
x, y = make_random_data()
plt.plot(x, y, 'o')
plt.show()
```

이제 생성된 Dataset 을 학습시키기 전 200개의 샘플에서 150개를 훈련 세트로, 나머지 50개를 테스트 세트로 나눈다. 이는 train data를 이용해 model을 학습시킨 후 학습이 제대로 됐는지 확인하기 위한 testset으로 분리하는 과정이다.

```
x_train, y_train = x[:150], y[:150]
x_test, y_test = x[150:], y[150:]
```

모델에 입력되는 Tensor인 x_train은 1차원 Data set이므로 input_dim을 1로 지정해주고, 출력 Tensor를 결정하는 y_train도 1차원 Dataset 이므로 1개의 Unit으로 모델을 생성해 준다. 만약 아직도 이 과정이 익숙하지 않는다면 모델 설계 부분으로 돌아가 다시 복습하도록 하자.

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(units=1, input_dim=1))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

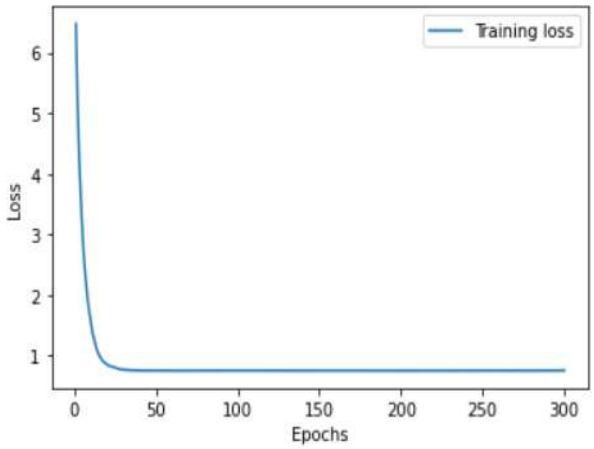
이제 모델을 생성했으니 실제로 모델의 학습 과정을 설정해 주자. 이 과정은 모델을 빌드하고 실행하기 전의 단계로 학습 시 필요한 Loss function, Optimizer, Metric 값을 지정하는 단계이다.

```
model.compile(optimizer='sgd', loss='mse', metrics = ['accuracy'])
```

- mse : 회귀 문제이므로 평균 제곱 오차를(mean squared error) 손실함수로 사용
- sgd : 가장 기본적인 optimizer로 많이 사용되는 확률적 경사 하강법 사용한다.
- accuracy : 훈련 과정을 관찰하는 지표로 accuracy를 지정하면 학습 과정에서 정확도를 수정한다.

```
history = model.fit(x_train, y_train, epochs=300,
                    validation_split=0.3)
```

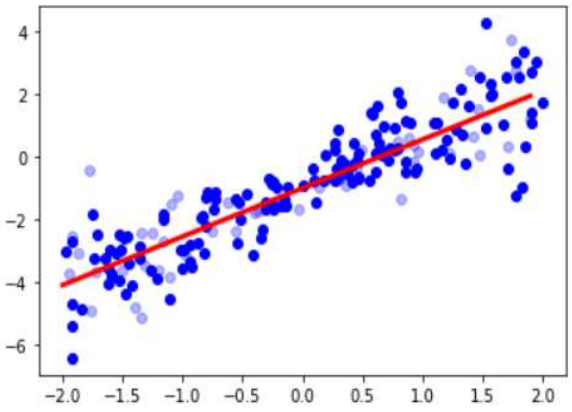
```
epochs = np.arange(1, 300+1)
plt.plot(epochs, history.history['loss'], label='Training loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



<학습시 Epch에 따른 loss값>

학습이 진행된 후에는 model에 저장된 가중치를 이용해 실제 model이 어떻게 학습되었는지 확인할 수 있다. 아래의 그림은 학습된 model의 가중치를 호출해 시각화한 그림이다.

```
x_arr = np.arange(-2, 2, 0.1)
y_arr = model.predict(x_arr)

plt.figure()
plt.plot(x_train, y_train, 'bo')
plt.plot(x_test, y_test, 'bo', alpha=0.3)
plt.plot(x_arr, y_arr, '-r', lw=3)
plt.show()
```



<학습후 생성된 model의 시각화>

(4) model 저장 및 그 외

model이란 생성한 layer와 가중치들을 저장해놓은 값으로 학습시키는 과정이 생각보다 오래 걸리는 경우가 많다. 성능이 좋은 GPU를 이용한다 해도 2시간을 넘기는 경우가 많으며, 대용량의 Dataset 과 복잡한 model을 구현할수록 그 시간은 계속 늘어난다. 이를 위해 Tensorflow는 model을 학습 중이나 학습이 끝난 후 저장할 수 있는 기능을 제공한다. 물론 이를 저장해 공유할 수도 있다.

기본적으로 모델을 저장하는 방법은 model 객체의 save 함수를 호출하는 것이다. 이때 확장자는 HDF5(.h5)를 사용한다.

```
model.save('simple_model.h5')
```

tf.keras.models.load_model() 함수를 이용한 모델 전체를 load 할수 있다.

```
model = tf.keras.models.load_model('simple_model.h5')
```

학습 도중 모델을 저장하기 위해서는 checkpoint 라는 방법을 이용해 자동 저장한다. 이는 모델을 재사용하거나 학습을 이어서 할 수 있도록 도와준다. ModelCheckpoint callback을 사용하기 위해서는 checkpoint 위치와 학습시 callback을 전달할 수 있도록 설정해 줘야 한다.

```
checkpoint_path = "training_ckpt/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

#가중치를 전달하는 callback 객체 생성
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)
```

앞서 학습시킨 예제에 callbacks 인자값을 추가하여 callback을 학습에 전달하도록 한다.

```
history = model.fit(x_train, y_train, epochs=300,
                    validation_split=0.3,callbacks=[cp_callback])
```

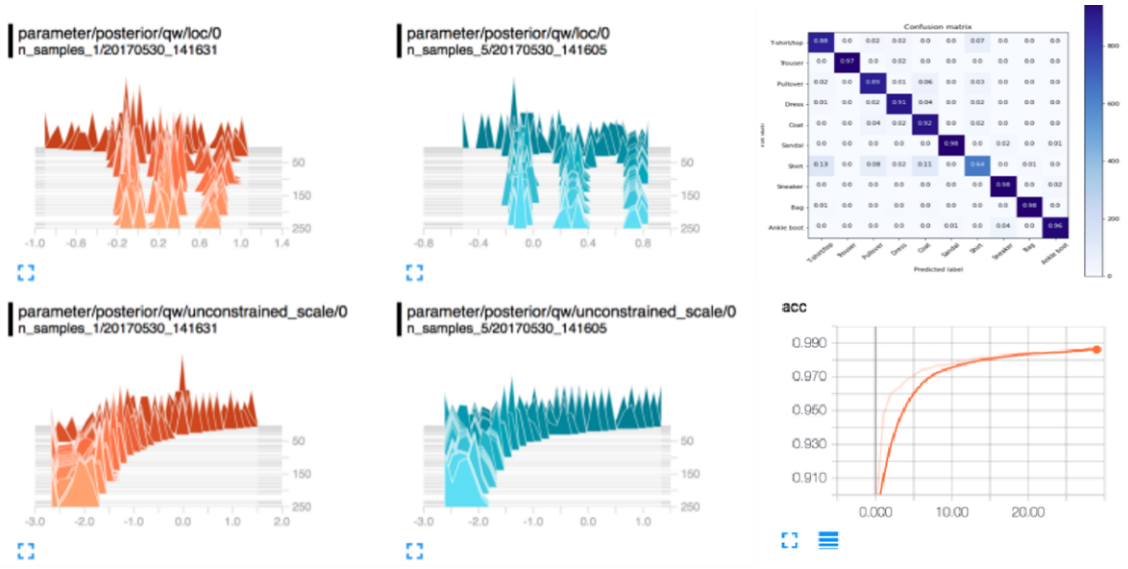
이와 같은 과정으로 만들어지는 checkpoint 파일은 epoch가 종료될 때마다 업데이트된다. 이후 저장된 가중치를 재사용하기 위해서는 모델 객체를 다시 만들고 가중치를 load 하면 된다.

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(units=1, input_dim=1))

model.load_weights(checkpoint_path) #checkpoint_path = "training_ckpt/cp.ckpt"
```

학습이 완료된 이후 학습된 모델을 평가하거나 학습 과정을 분석하기 위해서는 수치화된 결과보다 시각화된 결과를 받는 게 좀 더 직관적일 때가 있다. 이를 위해서 Tensorflow 는 Tensorboard를 제공한다.

Tensorboard 는 머신러닝 학습 과정에 대한 다음과 같은 다양한 시각화 분석 도구를 제공한다.

- 손실 및 정확도와 같은 측정항목 추적 및 시각화
- 모델 그래프(작업 및 레이어) 시각화
- 시간의 경과에 따라 달라지는 가중치, 편향, 기타 텐서의 히스토그램 확인
- 저차원 공간에 임베딩 투영
- 이미지, 텍스트, 오디오 데이터 표시
- TensorFlow 프로그램 프로파일링



<Tensorboard를 이용한 다양한 시각화>

Tensorboard 는 웹서비스의 형태로 동작하며 이를 위해선 다음과 같은 내용을 모델 컴파일 시 추가해야 한다.

<적용전>

```
model.compile(optimizer='sgd', loss='mse')
history = model.fit(x_train, y_train, epochs=300,
                    validation_split=0.3)
```

<적용후>

```
callback_list = [tf.keras.callbacks.TensorBoard(log_dir='logs')]
model.compile(optimizer='sgd', loss='mse')
history = model.fit(x_train, y_train, epochs=300,
                    callbacks=callback_list, validation_split=0.3)
```

학습이 진행된 이후 TensorBoard 실행하기 위해서는 다음과 같은 코드를 입력하면 된다. 이때, Native 한 환경에서 실행하는 경우는 크게 문제 되지 않지만 GoogleColab과 같이 클라우드 플랫폼을 이용하는 환경에서는 동작에 문제가 발생할 수도 있다. 주로 logdir을 따로 생성할 수 없어 발생하는 문제들이다. 따라서 문제가 발생한다면 각 플랫폼의 가이드를 참고하도록 하자.

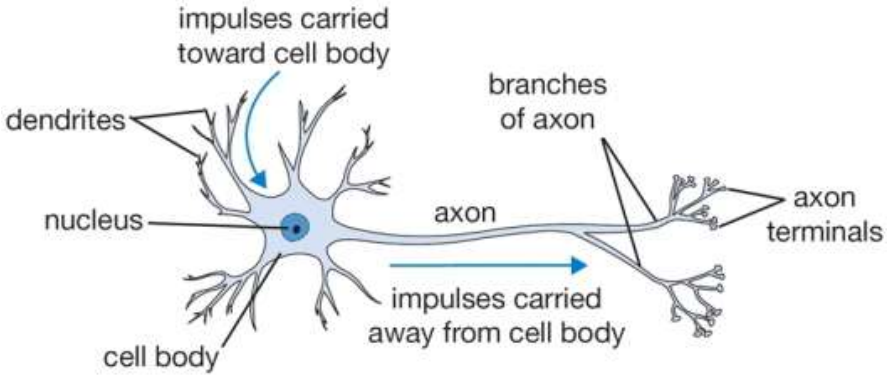
```
logdir = './logs'
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)
%load_ext tensorboard
%tensorboard --logdir=./logs
```

이로써 기본적인 Tensorflow/keras 사용법과 이를 이용해 model을 생성하고 학습시키는 기본적인 과정을 진행했다. 좀더 다양한 기법은 텐서플로우 공식 사이트인 https://www.tensorflow.org/tutorials?hl=ko 에서 제공하고 있으니 참고하길 바란다.

3.2 ANN

(1) 인공신경망의 구조

인간의 신경은 뉴런이라는 기본단위로 이루어져 있으며, 이들이 모여 신경과 기관을 구성한다. 사람과 같이 생각하도록 프로그래밍하기 위해 인간은 신경을 구현하기로 했고, 이는 인공신경망(ANN)을 만들게 되었다.



<사람의 뉴런>

인공신경망은 인간의 신경을 수학적 요소로 간단하면서 가변성 있게 구연해놓은 딥러닝의 핵심적인 요소이며, 우리는 인공신경망을 구성하고 있는 요소를 신경이 아닌 뉴런이라고 이야기 할 것이다. 이제부터 뉴런을 만들고 직접 학습시켜 보는 과정을 통해 인공신경망이 어떻게 구성되어 있는지 확인해 보도록 하자.

매캘러와 피츠는 생물학적 뉴런을 통한 가장 기본적인 모델을 제안했는데, 이 모델은 이진(0/1) 입력과 이진 출력을 이용한 연산이 가능할 수 있도록 만들었다. 이렇게 만든 모델은 어떤 논리 명제도 계산할 수 있으며, 실제로 이러한 모델을 이용해 AND, OR 연산이 가능한 퍼셉트론을 만들 수 있다. 물론 아래와 같이 AND OR XOR 연산을 고전적인 프로그래밍 방법을 이용해 만들 수 있지만, 학습이 가능한 상태로 만들기 위해선 다음과 같이 주어진 Dataset을 학습할 수 있도록 만들어야 한다.

다음은 파이썬을 이용해 AND 연산함수를 구현한 코드이다.

```
def and_fun1(x,y):
    if x == 1:
        if y==1:
            return 1
        else:
            return 0
    else:
        return 0
```

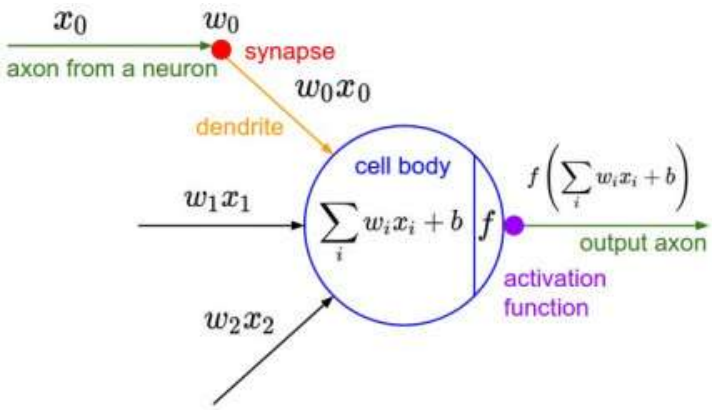
일반적으로 AND 연산을 함수로 구현한다면 위와 같은 방식으로 구현할 수 있겠지만, and 연산이 0과 연산하면 0을 출력한다는 특성, 그리고 파이썬의 특성을 동시에 적용하면 다음과 같은 코드를 작성할 수도 있다.

```
def and_fun2(x,y):
    if 0 in (x,y):
        return 0
    else:
        return 1
```

같은 방식으로 OR 연산도 작성할 수 있다.

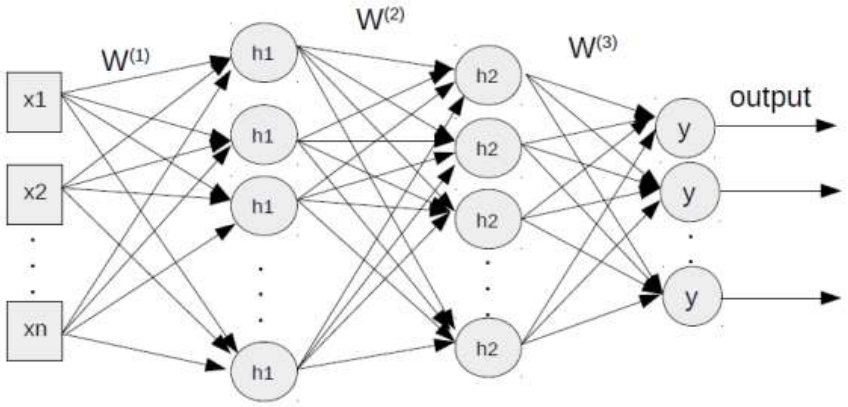
(2) 퍼셉트론

앞서 제시한 이진 출력을 가지는 단순화 된 원리로 동작하는 신경 세포를 MCP 뉴런이라 부르고 이를 기초로 1957년 프랑크 로젠블라크는 퍼셉트론(perceptron) 학습 규칙을 제안하게 된다. 이 모델은 입력과 출력이 이진 데이터가 아닌 숫자이며, 각각의 입력 연결은 가중치와 연관돼 있다. 그리고 이 가중치에 대한 합을 활성화 함수를 이용해 출력해 낸다.



<퍼셉트론의 기본구조>

퍼셉트론은 층이 하나뿐인 TLU(입력에 가중치 합을 계산한 다음 계단 함수를 적용하는 인공 뉴런)로 구성된다. 각 TLU는 모든 입력에 연결되어 있고, 한 개의 레이어에 모든 뉴런이 이전 레이어의 모든 뉴런과 연결되어 있을 때 이를 Dense Layer라고 부른다. 퍼셉트론의 입력은 input layer로 구성되어 있고, 그다음 편향을 특성이 더해지게 된다. 이때 편향은 항상 1을 출력하게 입력하여 가중치와 곱하여 사용하게 된다. 이를 시각적으로 표현하면 다음과 같이 표현할 수 있다.



<다층 퍼셉트론의 구조>

단층 퍼셉트론을 이용해 AND 연산을 구현해 보도록 하자. 우리는 AND 연산을 제어문을 이용해 구현했지만, 텐서플로우를 이용하면 AND 연산이 가능한 퍼셉트론을 간단하게 학습시키고 구현할 수 있다. 물론 텐서플로우와 같은 딥러닝 프레임 워크가 아닌 Native 한 파이썬 코드로 학습시키는 코드를 작성할 수 있지만, 이 책에서는 구현하지 않도록 하겠다.

AND 연산을 위한 모델을 생성하고 학습시키기 전 우리는 학습시킬 데이터를 준비해야 한다. and 연산은 4가지 경우의 수만 존재하기 때문에 x_data에는 학습돼야 할 데이터, y_data에는 x_data에 대한 정답 label 값을 입력하여 준비한다.

```
x_data = [[0,0], [0,1], [1,0], [1,1]]
y_data = [[0], [0], [0], [1]]
```

이제 모델을 생성해야 할 단계이다. 모델 생성은 Sequential API를 호출하는

방식과 Functional API를 호출하는 방식 중 원하는 방식을 사용하면 되지만, 이 책에서는 입문자도 쉽게 이해할 수 있는 Sequential API를 호출하여 모델을 생성할 것이다.

```
model = keras.Sequential()
model.add(keras.layers.Dense(1, activation='sigmoid', input_shape=(2,)))
#x_data.shape 의 (4,2)중 2에 영향을 받는다.
```

모델 생성 후 학습을 위해 손실함수와 옵티마이저 알고리즘을 하자. AND 연산은 0과 1을 분류하는 이진 분류 문제를 해결하는 것으로 이를 학습시키기 위해선 binary cross entropy를 손실함수로 지정해야 한다. 또한 옵티마이저 알고리즘은 가장 많이 쓰이고 효율적인 확률적 경사하강법을 사용하도록 했다. 손실함수의 경우 학습돼야 할 label에 영향을 받고 이에 대한 분류가 명확하게 이루어져 있지만, 옵티마이저의 경우 알고리즘 간의 특징을 알아도 무엇을 적용해야 할지 모르는 경우들이 많다. 학습 시 문제점이 발생하고 그 문제점들을 해결하기 위해 다양한 알고리즘을 시도하는 것이 이러한 문제를 해결하는 방법이 될 수 있다.

```
optimizer = keras.optimizers.SGD(lr=0.1)
model.compile(optimizer=optimizer, loss="binary_crossentropy",
              metrics=['accuracy'])
model.fit(x_data, y_data, batch_size=4, epochs=50)
#4개의 샘플을 한번에 넣어 50번 학습
```

이제 학습이 완료되었으니 모델을 이용해 x_data가 제대로 학습되었는지 확인해 보자.

```
model.predict(x_data)
```

<결과>

```
array([[0.02703479],
       [0.2042268 ],
       [0.20302524],
       [0.7017505 ]], dtype=float32)
```

위와 같은 실행 결과를 0 또는 1로 표현해주기 위해 np.round 함수를 이용해

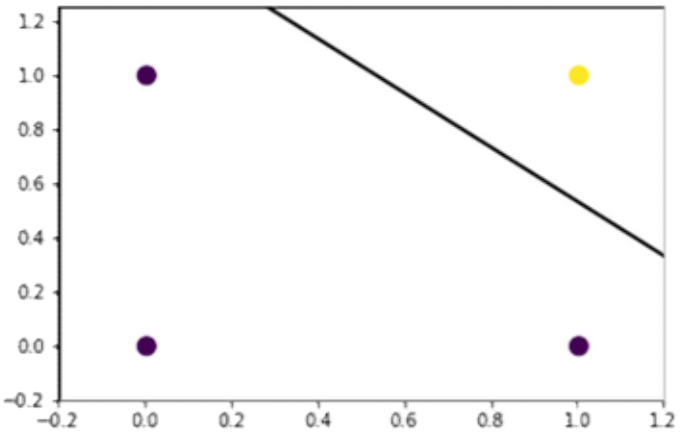
변환해 주도록 하자.

```
np.round(model.predict(x_data))
```

<결과>

```
array([[0.],
       [0.],
       [0.],
       [1.]], dtype=float32)
```

실제 학습된 값을 시각화하면 다음과 같이 나타낼 수 있다. 아래의 직선은 모델의 학습된 결과로 [[0,0], [0,1], [1,0], [1,1]]를 이진 분류한 걸 확인할 수 있다.



<AND 연산이 가능한 model 의 시각화>

전체적인 코드는 다음과 같다.

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

# Data
x_data = [[0,0], [0,1], [1,0], [1,1]]
y_data = [[0], [0], [0], [1]]

x_data = np.array(x_data)
y_data = np.array(y_data)

model = keras.Sequential()
model.add(keras.layers.Dense(1, activation='sigmoid', input_shape=(2,)))
optimizer = keras.optimizers.SGD(lr=0.1)
model.fit(x_data, y_data, batch_size=4, epochs=50)

model.predict(x_data)
np.round(model.predict(x_data))
```

AND 연산을 구현하기 위한 과정으로 입력이 (0, 0), (1, 0), (0, 1) 인 경우 0, (1, 1) 인 경우 1로 출력되는 선형 모델을 학습시켰다. 이와 동일한 OR 연산이 가능한 퍼셉트론을 만들고 학습시키기 위해서는 같은 모델에 대해 다른 Dataset 으로 훈련하면 가능하지만, XOR 연산의 경우 같은 모델을 이용하여 학습시킨다면 제대로 학습이 이루어지지 않는 문제가 발생한다. 이는 AND나 OR 연산에 공통점이 존재하고 XOR 연산에는 앞선 2개의 연산과는 차이점이 존재한다는 것을 의미한다.

XOR 연산을 해결하기 위해서 다음과 같은 모델을 고안할 수 있다.

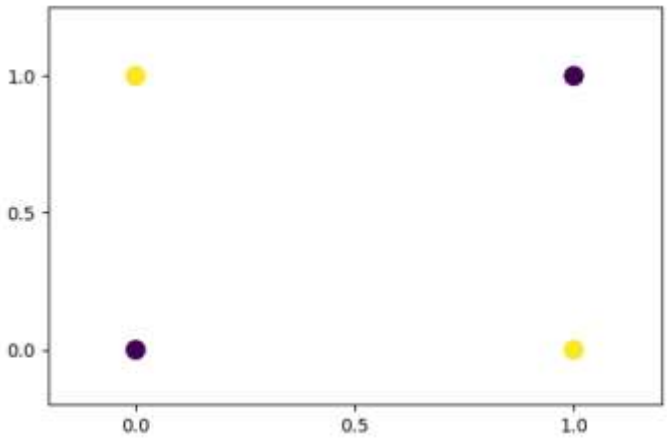
```
model = keras.Sequential()
model.add(keras.layers.Dense(2, activation='sigmoid', input_shape=(2,)))
model.add(keras.layers.Dense(1, activation='sigmoid'))
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 2)                 6
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 3
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```

위에서 제시한 모델은 AND 연산의 해결을 위한 모델과 비교했을 때 XOR 연산의 학습 시 2가지 차이점이 눈에 띈다.

1. Dense Layer가 추가되었다.
2. Input data를 받는 Layer의 노드가 2개로 증가하였다.

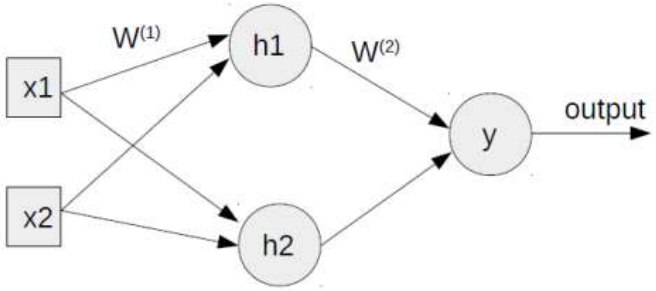
모델이 변경된 이유에 대해 생각해보자. AND 연산의 경우 위에서 제시한 것과 같이 직선 한 개로 분류할 수 있다. 하지만 XOR 연산을 위한 Dataset 을 시각화한다면 다음과 같이 표현할 수 있는데, 이를 직선 한 개로 분류할 수 없는 문제가 발생하다.



<XOR dataset>

이러한 문제를 해결하기 위해 다층 퍼셉트론으로 구현 되었다. 두 개의 직선으로 분류하기 위해 입력 데이터를 받는 layer에 노드를 2개로 증가하였고, 2개의 노드에서 출력된 값을 1개의 값으로 출력하기 위해 1개짜리 노드를 갖고있

는 layer를 추가 하였다. 이를 시각적으로 표현하면 다음과 같이 표현할수 있다.



<XOR를 구현하기 위한 다층 Layer>

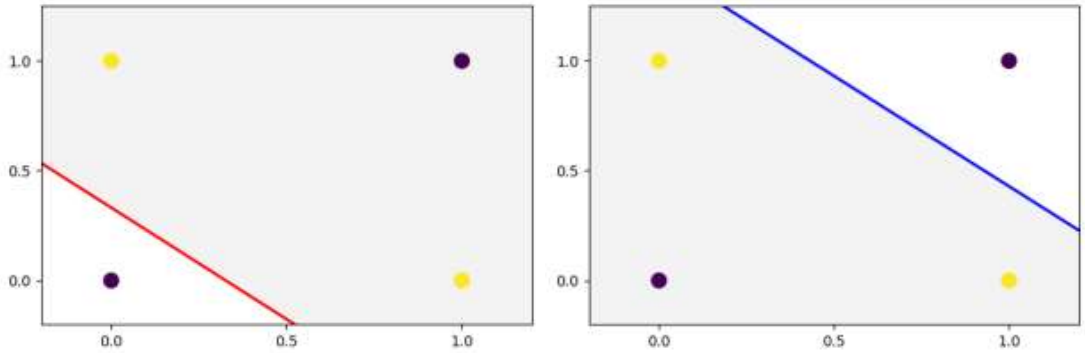
전체 코드는 다음과 같다.

```
x_data = [[0,0], [0,1], [1,0], [1,1]]
y_data = [[0], [1], [1], [0]]

x_data = np.array(x_data)
y_data = np.array(y_data)

model = keras.Sequential()
model.add(keras.layers.Dense(2, activation='sigmoid', input_shape=(2,)))
model.add(keras.layers.Dense(1, activation='sigmoid'))

optimizer = keras.optimizers.SGD(lr=0.1)
model.compile(optimizer=optimizer, loss="binary_crossentropy",
              metrics=['accuracy'])

model.fit(x_data, y_data, batch_size=4, epochs=3000)

model.predict(x_data)
np.round(model.predict(x_data))
```

<결과>

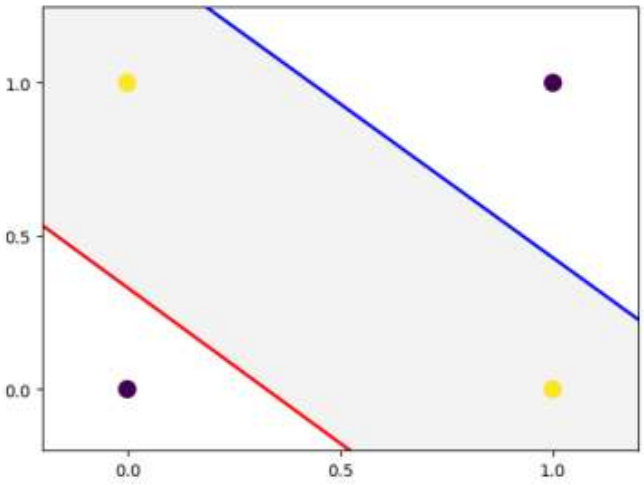
```
array([[0.],
       [1.],
       [1.],
       [0.]], dtype=float32)
```

분류 과정을 확인하면 (0,0) 과 다른 점들을 분류하기 위해 1개의 직선이 사용되었고, (1,1) 과 다른 점들을 분류하기 위해 다른 점을 분류하기 위해 또 다른 직선을 사용하였다.



<다층 Layer에 구현된 직선1>

이에 대한 교집합을 적용하면 xor 연산이 가능해지는 것을 확인할 수 있다.



<다층 Layer에 구현된 직선2>

(3) 다층 퍼셉트론의 활용

다층 퍼셉트론은 분류작업과 회귀 예측작업에 사용될 수 있다. AND 연산과 XOR 연산의 학습된 모델과 관련한 출력값은 0과 1 사이의 실수였다. 이는 양성 클래스에 대한 예측 확률로 해석할 수 있지만 이를 좀 더 확장한다면 다중 레이블에 적용할 수 있다. 이처럼 XOR 문제를 해결하기 위한 다층 퍼셉트론 훈련 방법은 1986년 데이브드 루멜하트, 제프리 힌터 등이 역전파(backpropagation) 훈련 알고리즘을 세상에 공개하면서 가능해졌다. 실제로 AND 연산과 XOR 연산을 학습하는 데 있어 우리는 같은 과정을 거치는 코드에 서로 다른 Dataset과 모델만을 사용했을 뿐이다. 역 전파법은 다층 퍼셉트

론이 어떤 결과를 예측한 이후, 예측이 틀린 경우 Weight와 Bias 값을 조정해야 하는데, 단층 퍼셉트론에서는 간단하게 이 값을 업데이트 가능했지만 다층 퍼셉트론에서는 Weight와 Bias 값의 구조가 어려워 업데이트하지 못한 문제를 해결했다.

이번에는 MNIST 라는 Dataset 의 손글씨 숫자를 분류하는 다층 퍼셉트론을 구현하여 학습시켜 보자. 이 Dataset 은 머신러닝 알고리즘을 테스트하여 비교하기 위한 Data set으로 주로 사용되고 있다. 데이터는 사람이 250명의 사람이 쓴 손글씨이다. 널리 쓰이는 Dataset 이니 만큼 tensorflow 에 기본적으로 내장되어 있어 손쉽게 불러올 수 있다.

```
from tensorflow.keras.datasets import mnist
```

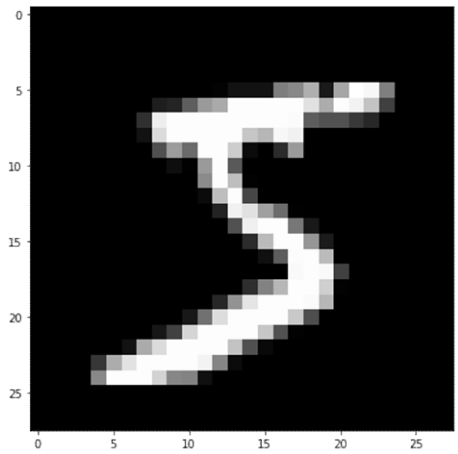
데이터를 불러온 후 가장 먼저 해야 할 일은 데이터를 분리해야 하는 작업이다. 이를 통해 train set과 test set을 분류하고 모델이 train set을 통해 학습한 이후 test set을 통해 제대로 학습되었는지 확인할 데이터를 준비하게 된다. 데이터가 준비되었다면 데이터의 shape를 확인해야 한다. 이를 통해 model의 input을 어떻게 설정해야 하는지, output의 shape는 어떻게 설정해야 하는지가 결정된다. 앞서 진행한 and 연산이나 xor 연산에 대한 학습이 이루어질 땐 단순한 형태의 data set이었지만 실제 적용해야 되는 데이터는 좀 더 높은 차원의 데이터들이 주를 이룬다. 따라서 데이터의 shape을 항상 확인해야 한다. MNIST Dataset 은 두 부분으로 나뉠 수 있다. 손으로 쓴 숫자와 그에 따른 label로 앞으로 x_train과 y_train으로 구분할 것이다.

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
print("X_train shape :", X_train.shape)
print("y_train shape :", y_train.shape)
```

shape에 관한 결과는 60000의 데이터가 존재하고 각 데이터는 28×28 크기의 이미지로 구성되어 있다. 실제 어떤 이미지가 저장되어 있는지 확인하기 위해 matplotlib의 pyplot을 이용해 표시해보면 첫 번째 데이터는 숫자 5인임을 확인할 수 있다.

```
X_train shape : (60000, 28, 28)
y_train shape : (60000,)
```

```
plt.imshow(X_train[0], cmap='gray')
```



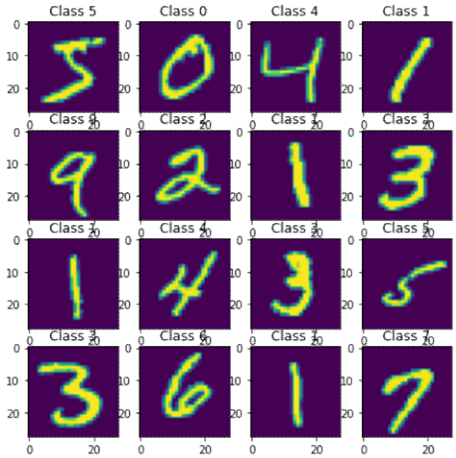
<MNIST 이미지>

실제 숫자 5 는 0~1 사이의 숫자로 이루어진 28차원의 벡터로 이루어져 있다. 28차원 벡터를 28개로 합쳐 이미지를 표현했다. 이미지로 변환하기 전 원본 데이터인 숫자를 통해 표현하면 다음과 같이 표현할 수 있다.

<MNIST 이미지의 vector>

나머지 데이터를 시각화하면 다음과 같이 0~9까지의 숫자들이 dataset으로 존재하는 것을 확인할 수 있다.

```
for i in range(16):
    plt.subplot(4,4,i+1)
#    plt.imshow(X_train[i], cmap='gray') 익숙한 색으로 표현
    plt.imshow(X_train[i])
    plt.title("Class {}".format(y_train[i]))
```



<MNIST 이미지>

훈련에 앞서 입력 데이터의 shape를 우리가 생성할 model에 맞는 크기로 변경해야 한다. model은 입력값을 28×28 크기의 matrix로 입력받는 게 아니라, 28×28=784 크기의 벡터로 입력받는다. 이를 평탄화 작업이라 한다.

```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
```

또한, label의 값도 수정이 필요하다. 현재 label의 값은 0, 1, 2, ... 와 같은 정수 형태의 원소인데 앞서 말했듯이 분류란 각 Class에 속할 확률을 나타내는 것이기 때문에 정수 형태로는 학습시킬 수 없다. 따라서 one-hot encoding을 수행하는 keras.utils.to_categorical() 함수를 사용해 각 숫자를 index number로 인식하여 index 위치에는 1을 부여하고, 다른 index에는 0을 부여하도록 변환해 준다. 이렇게 인덱스를 통해 class를 부여하면 학습 후 test dataset을 통해 확인할지 각 클래스에 속할 확률로 출력되게 된다.

```
Y_train = keras.utils.to_categorical(y_train)
Y_test = keras.utils.to_categorical(y_test)
```

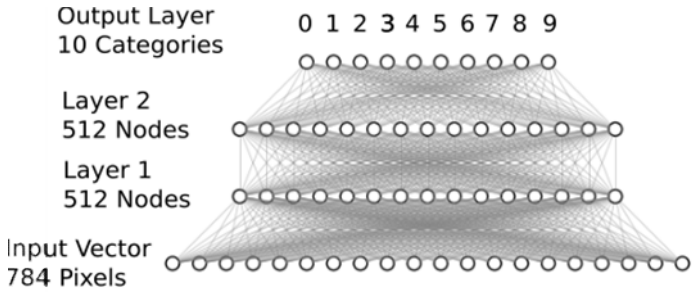
정수가 인덱스로 잘 변환됐는지 확인하자. 0번째 데이터는 숫자 5였으니 [0, 0, 0, 0, 0, 1, 0, 0, 0, 0] 으로 표현되어야 한다.

```
Y_train[0]
```

<결과>

```
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

우리가 구성할 모델을 미리 시각화하면 다음과 같다. 2개의 Hidden Layer를 가진 모델로 input shape로 784를 받는다. 위에서 자세히 설명하지 않았지만, input_shape를 (28, 28,)로 설정하면 학습이 되지 않는다. 이는 우리가 모델을 설계할 때 layer를 1차원적으로 구성했기 때문이다. 만약 (28, 28,) 의 형태를 입력받게끔 model을 설계하고 싶다면 functional API를 사용하여 구성하여 layer를 2차원적으로 구성해야 한다.



<MNIST 학습을 위한 model 구조>

hidden layer의 node 개수는 다양한 방법을 통해 최적의 결과를 나타내는 값을 선택해야 한다. 다수의 논문에서 추천하는 512개의 node를 구성하고 마지막 노드에서는 softmax function을 이용해 각 Class 에 속할 확률을 출력하도록 하겠다.

```
model = keras.Sequential()
model.add(keras.layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(keras.layers.Dense(512, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))
```

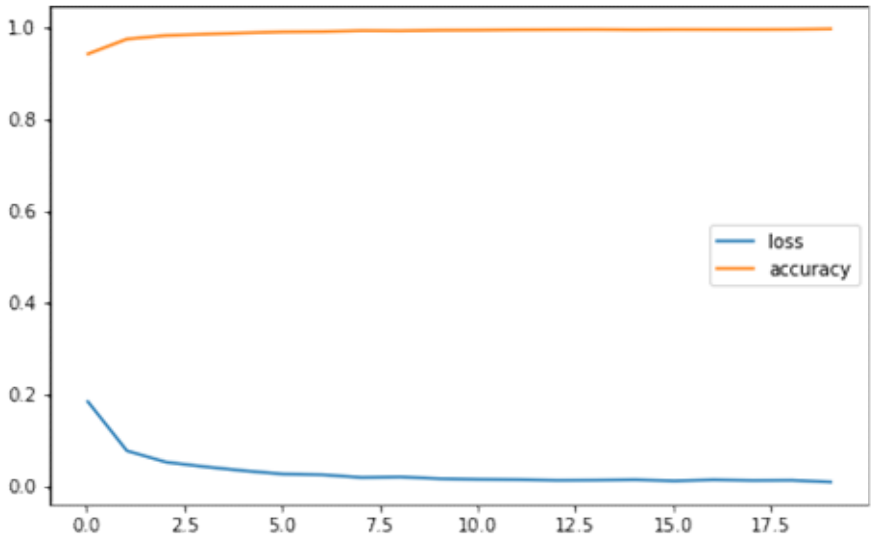
이제 학습요소를 지정하고 model을 학습시키도록 하겠다. 앞서 우리는 손실함수로 'binary_crossentropy'를 사용했지만, 이번에는 이진 분류가 아닌 다중 분류 문제이므로 'categorical_crossentropy'를 사용하겠다. 또한, 옵티마이저는 'adam'을 사용하도록 하겠다.

```
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
history=model.fit(X_train, Y_train, batch_size=32, epochs=20)
```

학습된 과정과 결과는 다음과 같다.

```
Epoch 1/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1824 - accuracy: 0.9439
Epoch 2/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0798 - accuracy: 0.9753
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0574 - accuracy: 0.9815
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0418 - accuracy: 0.9865
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0325 - accuracy: 0.9897
```

```
import pandas as pd
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.show()
```



<학습후 loss값과 accuracy값>

학습곡선을 분석했을 때, 꽤 좋은 결과가 나옴을 알 수 있다. 이는 loss 값은 점차 줄어들었는데 이는 model이 숫자를 빠르게 분류하기 시작했음을 보여주고 있다.

이제 신경망을 평가하는 단계이다. 이를 통해 model이 test set에 대해 얼마

나 제대로 된 분류를 진행할 수 있는지 확인할 수 있다. model.evaluate() 함수는 model을 compile 할 때 정의된 손실 값과 metric 값을 계산하여 보여준다.

```
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

모델 평가 시 다음과 같은 결괏값이 나온다.

```
Test score: 0.1194722056388855
Test accuracy: 0.9801999926567078
```

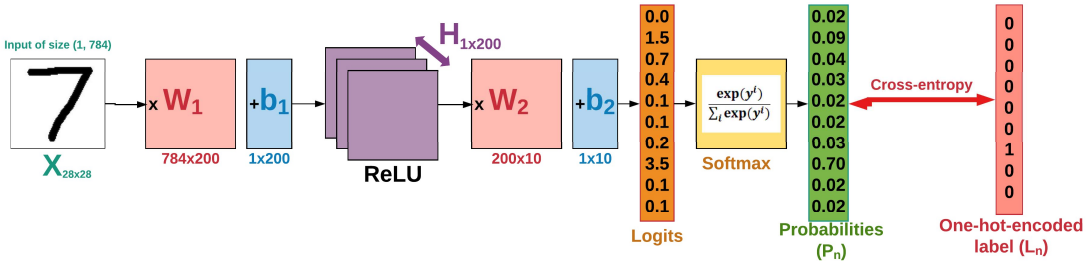
모델 학습 시 마지막 정확도가 'accuracy: 0.9897' 이 나왔으며, 'Test accuracy: 0.9801999926567078' 가 나온 것을 확인할 수 있다. test set과 train set의 정확도 비교 시 큰 차이가 존재하지 않는 건 과대 적합이 일어나지 않았음을 의미한다. 이렇게 간단한 단층 퍼셉트론에서 복잡한 다층 퍼셉트론까지 구성해 보았다. 기초 학습을 마쳤으니 다양한 신경망을 구성해 보도록 하자.

3.2 CNN

이세돌과 알파고의 바둑대결은 세간의 관심을 집중시키며 딥러닝을 세상에 알렸다. 하지만 그보다도 전에 큰 주목을 받게 된 계기가 있었는데, CNN이 세상에 공개되면서부터이다. 2012년 이미지 인식 대회인 ILSVRC(ImageNet Large Scale Visual Recognition Challenge)에서는 AlexNet 딥러닝 알고리즘이 기존의 알고리즘 성능을 큰 폭으로 개선하면서 우승하게 되었다. 이를 기점으로 딥러닝 알고리즘은 그동안 어려움을 겪던 컴퓨터 비전 분야에 대세로 떠오르게 되었다. AlexNet 알고리즘은 CNN을 기반으로 한 딥러닝 알고리즘으로 기존의 Shallow architecture 구조를 가진 알고리즘과는 차이가 있었다. 이번 절에서는 AlexNet이 발표된 이후로 컴퓨터 비전의 대세가 된 CNN 알고리즘을 이해하고 구현해 보도록 하겠다.

(1) CNN의 구조

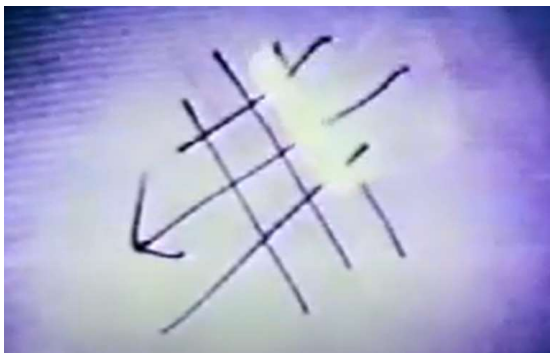
앞서 다룬 MLP(Multi-Layer Perceptron, Multi-layer NN, MLNN) 의 문제점을 생각해보자. 기계가 숫자 이미지를 인식하기 위해선 픽셀 단위로 쪼개서 인식하게 되는데 MLP를 이용해 구현한 경우 이미지의 위상과는 관계없이 입력 벡터에 따라 학습된다. 이럴 때 글자가 이동하거나 회전하는 경우 또는 글자의 크기가 달라짐에 따라 서로 다른 데이터가 되어 다른 결과를 나타내게 된다. 이러한 이유에는 앞서 말했듯이 글자 자체의 위상적 형태는 고려하지 않고 Raw Data를 직접 처리하기 때문이다.



<MLP 가 이미지를 인식하는 과정>

이런 문제점을 고안하기 위해 도입한 알고리즘이 Convolution이다. 데이비드 허블과 토르스텐 비셀은 고양이를 통한 대뇌 시각피질 실험을 진행하였다. 이 실험은 고양이 또는 원숭이 뇌에 있는 시각세포가 특정상태에 활성화 됨을 기록한 실험이다. 이 실험을 통해 신경을 구성하는 세포는 기울어지고 크고 작은 edge 요소들의 Convolution 과정을 통해 이미지를 구성하며, 활성화되는 뉴

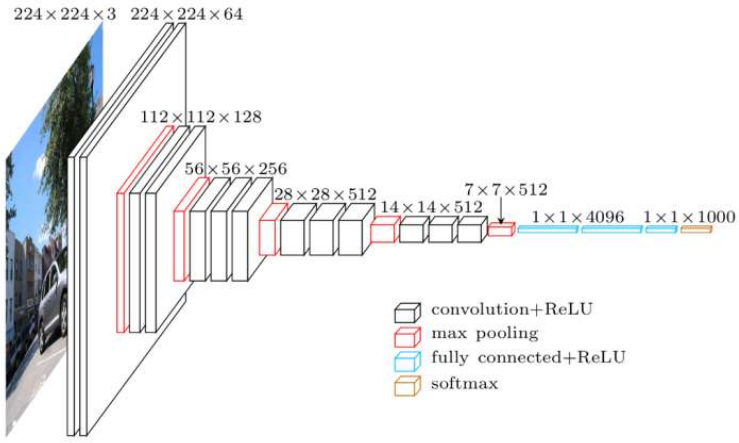
런에 따라 인식하는 모양이 달라짐을 결론 내렸다.



<David Hubel & Torsten Wiesel의 실험>

CNN은 이러한 합성 연산을 딥러닝에 적용한 것으로 이미지를 인식하기 위해 이미지의 윤곽을 찾아내고 윤곽의 특징에 따라 사물을 분류할 수 있도록 만들었다.

CNN의 전체 구조를 살펴보자.



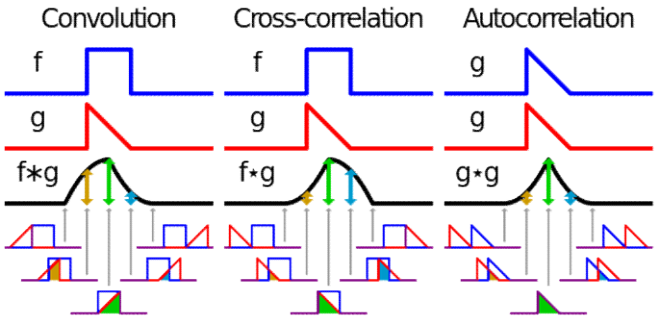
<CNN을 사용하는 대표 알고리즘인 VGG-16>

CNN은 크게 3가지의 계층으로 구성되어 있고 각각의 계층은 다음과 같은 특징을 지니고 있다.

- Convlution 계층 : 입력 데이터의 특징을 추출하는 역할 수행
- Relu 계층 : 입력 정보를 0보다 크면 값을 그대로 내보내고, 0보다 작으면 0을 출력하는 역할 수행
- Pooling 계층 : 입력으로 주어진 정보를 최대/최소/평균값으로 압축하여 데이터 연산량을 줄여주는 역할 수행(대푯값 추출)

(2) 합성곱(Convolution) 연산

Convolution 연산은 하나의 함수와 또 다른 함수를 반전 이동한 값을 곱한 다음, 구간에 대해 적분하여 새로운 함수를 구하는 수학 연산자이다. (wiki 백과) 결국 두 개의 신호를 합성해 출력하는 연산으로 이미지 처리에서는 필터연산에 해당한다.



<Convolution 연산>

필터연산은 커널이라는 값과 이미지 데이터를 합성곱 연산을 이용해 출력하는 것으로 이미지를 벡터화시킨 다음 Convolution 연산을 적용한다.

실제 이미지에 Convolution 연산을 적용해보자. PIL 은 Python Image Library로 이미지 처리를 위한 라이브러리 이다.

```
from PIL import Image, ImageFilter
```

적용할 이미지 객체 생성 체를 생성하고 적용할 커널을 정의하겠다. kernel Edge는 이미지의 가장자리를 출력하는 커널이며, kernelgaussian 커널은 이미지에 blur 처리를 해주는 필터이다. 이미지 처리에 관한 다양한 필터는 https://en.wikipedia.org/wiki/Kernel_(image_processing) 에 나와있으니 참고하도록 하자.

```
originImg = Image.open("03.20402175.1.jpg")

kernelIdentity = [
              0,0,0,
              0,1,0,
              0,0,0
]
kernelEdge = [
          -1,-1,-1,
          -1,8,-1,
          -1,-1,-1
]

kernelgaussian=[1/16,2/16,1/16,
                2/16,4/16,2/16,
                1/16,2/16,1/16
]
```

이제 이미지에 커널을 적용할 차례이다. 생성한 이미지 객체의 함수인 filter 함수를 사용하면 지정한 커널이 Convolution 연산을 통해 정의된다. 이때 커널의 크기, 적용할 커널 등은 필수적으로 입력해 줘야 한다.

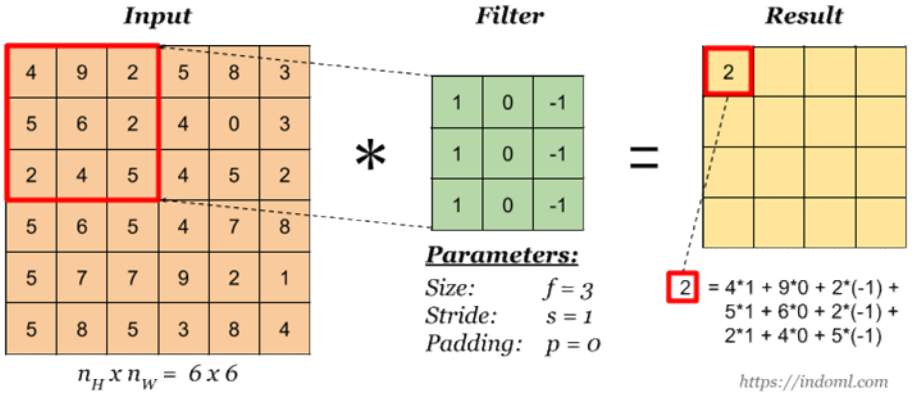
```
convol=originImg.filter(ImageFilter.Kernel((3,3),kernelEdge,1,0))
```

적용된 이미지 출력해보자. 다음과 같이 가장자리 필터를 적용한 경우 이미지의 가장자리만 추출된 걸 확인할 수 있다. 실제 CNN을 통해 학습되는 가중치는 Convolution을 통해 적용되는 커널 값이라고 할 수 있다.

```
display(convol)
```

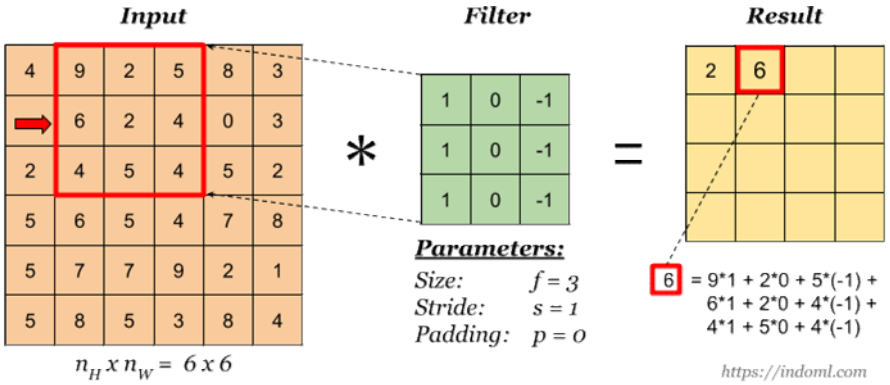
<Convolution 연산이 적용된 결과>

Convolution 연산은 다음과 같은 과정으로 이루어진다. input 값과 kernel(filter)이 kernel의 크기 만큼 요소 곱을 진행한다.



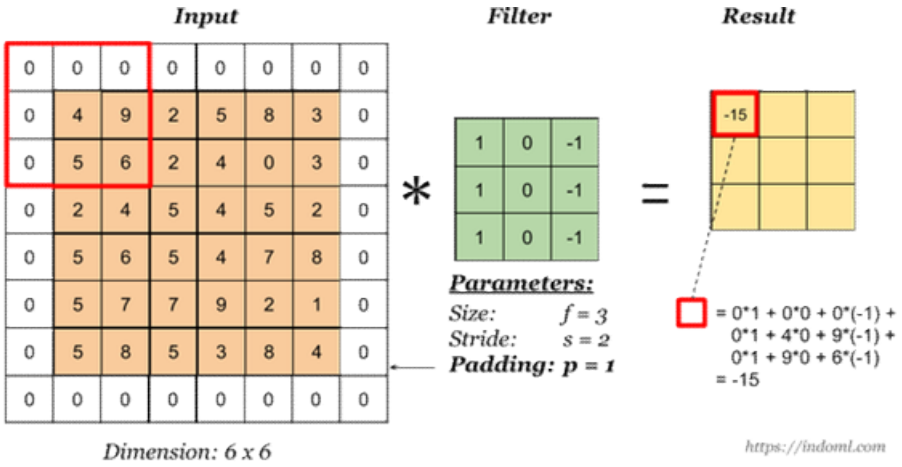
<image 의 Convolution 연산>

이때, kernel과 Convolution 연산이 이루어지는 입력값의 단위를 수용장이라고 한다. 한 수용장과 다음 수용장 사이의 간격을 스트라이드라 한다. 스트라이드의 값은 출력값에 영향을 준다.



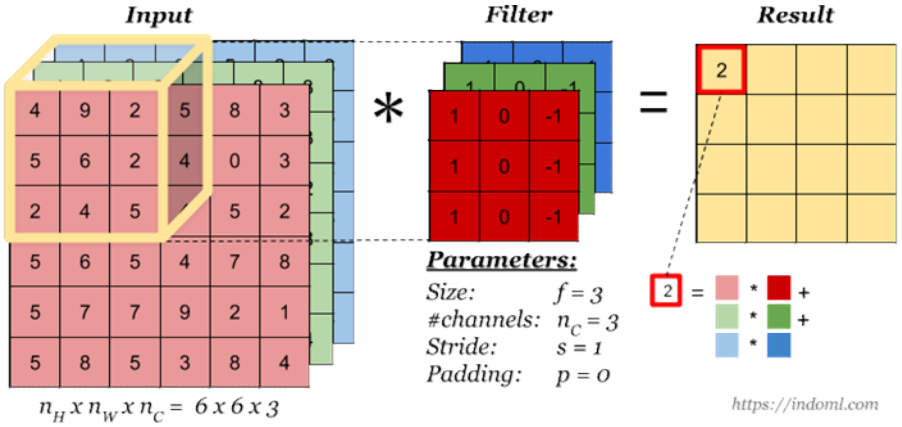
<stride=1인 연산>

Convolution 연산이 반복될수록 stride의 값에 영향을 받아 이미지의 크기는 줄어든다. 이러한 단점을 보정하기 위해 padding이 존재한다. padding은 이미지의 경계에 대한 정보를 유지하기 위한값으로 가장자리의 값을 특정 값(0)으로 채우는 것을 의미한다.



<padding=1인 연산>

Convolution 연산은 이미지의 채널, kernel의 채널 등에 따라 다양한 방법으로 연산된다. 지금까지 보여준 Convolution 연산은 1채널 이미지에 대한 Convolution 설명이다. 일반적인 이미지는 RGB 3개의 채널이 존재한다. 이에 대한 Convolution 연산은 각각에 대한 kernel이 존재하여 요소 곱을 진행한 후 모든 값을 합한다. 다음 예제를 끝으로 설명을 마치도록 하겠다.



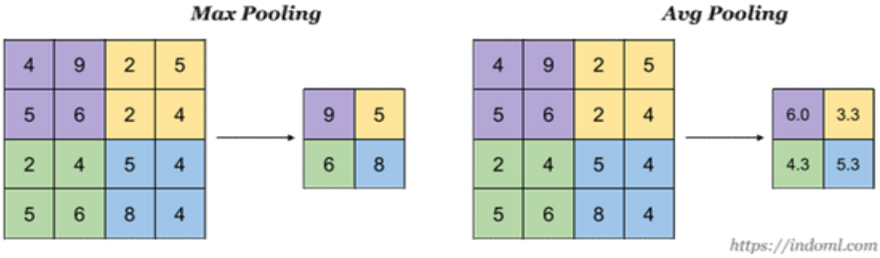
<Volume 에 대한 Convolution 연산>

Convolution에 대해 이해했으니 CNN model을 생성하고 Dataset 을 이용해 학습시켜 보자. 다음은 기본적인 CNN의 모습이다. Conv2D 와 MaxPooling2D를 쌓아 올린 형태를 지니고 있으며, 앞서 진행한 MLP와는 다르게 flatten Layer가 존재하고 있다.

(3) 풀링(Pooling) 연산

네트워크를 거치면서 연산해야 하는 값들은 점차 많아진다. CNN의 목표는 이

미지의 특징을 추출하고 추상화된 특징을 학습하는 것으로, 이미지가 추상화되는 과정에서 과도하게 커다란 이미지는 연산 시 비효율을 갖고 있다. 따라서 Convolution 연산 이후 Pooling 연산을 통해 이미지의 크기를 줄여준다. pooling 연산은 각수용장에서 최댓값을 추출하는 Maxpooling, 최솟값을 추출하는 MinPooling, 평균값을 추출하는 Avaragepooling 등 다양하게 존재한다.



<size=2인 pooling 연산>

(4) CNN model 생성

이제 실제로 이미지를 load 하고 MNIST Dataset을 model에 훈련시켜 보자. 앞서 진행한 MLP에서 사용한 코드와 거의 비슷하지만 input_shape에 채널 추가, Layer의 구성이 다르다는 점을 유의하자.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 11, 11, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 3200)              0
_________________________________________________________________
dense (Dense)                (None, 128)               409728
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 493,130
Trainable params: 493,130
Non-trainable params: 0
_________________________________________________________________
```

MNIST 데이터를 불러와 train set과 test set으로 분리하자. 분리 후 데이터를 학습시키기 전 학습시킬 데이터의 shape를 확인하자.

```
(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
X_train_full[0].shape
```

<출력>

```
(28,28)
```

Conv2D Layer는 (이미지 높이, 이미지 너비, 이미지 채널) 인 3D 크기의 텐서를 입력값으로 사용하는 특징이 있다. 이는 Convolution 연산의 특징으로 우리가 다루고 있는 예제와는 별도로 Convolution 연산은 다차원 다채널의

연산이 가능하므로 연산 시 차원을 지정해줘야 한다. 이 조건에 따라 MNIST는 크기 28x28, 1채널의 이미지이기 때문에 첫번째 conv2D Layer의 입력값을 input_shape=(28, 28, 1)로 지정해준다. 출력 역시 (높이, 너비, 채널)의 형태로 출력된다.

X_train_full의 데이터 타입은 (28, 28)의 data shape에 채널을 추가한 (28, 28, 1)의 형태로 변경 해줘야 한다. 이를 위해 np.expand_dim 함수를 통해 이를 적용하도록 하자.

```
# CNN에 적용 가능한 데이터 타입으로 변경해준다.
X_train_full = np.expand_dims(X_train_full, -1)/255
X_test = np.expand_dims(X_test,-1)/255

print("X_train shape :", X_train.shape)
print("y_train shape :", y_train.shape)
```

<결과>

```
X_train shape : (55000, 28, 28, 1)
y_train shape : (55000,)
```

단계의 마지막에는 flatten 계층이 존재해 이전 출력층에서 생성한 3D 텐서를 1D 텐서로 펼쳐준다. 이 과정을 통해 Convolution 연산을 종료한 뒤 각각의 특징이 추출된 이미지들을 정답 label에 맞게 분류하는 과정이 이루어진다. 이 과정을 마지막으로 softmax function을 사용해 10개의 클래스 중 각 클래스에 속할 확률을 나타내게 된다. 모델 생성에서 학습까지의 전체 코드는 다음과 같다.

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
#CNN에 적용 가능한 데이터 타입으로 변경.
X_train_full = np.expand_dims(X_train_full, -1)/255
X_test = np.expand_dims(X_test,-1)/255

Y_train = keras.utils.to_categorical(y_train)
Y_test = keras.utils.to_categorical(y_test)
Y_vaild = keras.utils.to_categorical(y_vaild)

model = keras.Sequential()
model.add(keras.layers.Conv2D(64, kernel_size=(3, 3),
                              activation='relu',input_shape=[28, 28, 1]))
model.add(keras.layers.MaxPooling2D(pool_size=2))
model.add(keras.layers.Conv2D(128, (3, 3), activation='relu'))
model.add(keras.layers.MaxPooling2D(pool_size=2))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(units=128, activation='relu'))
model.add(keras.layers.Dense(units=64, activation='relu'))
model.add(keras.layers.Dense(units=10, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
model.fit(X_train, Y_train, batch_size=128, epochs=4)

score = model.evaluate(X_test, Y_test, verbose=2)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

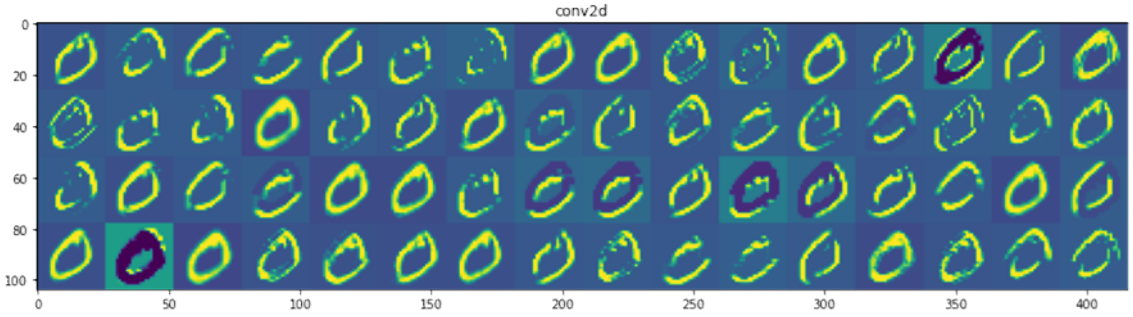
<결과>

```
313/313 - 1s - loss: 0.0308 - accuracy: 0.9897
Test score: 0.030774587765336037
Test accuracy: 0.9897000193595886
```

(4) 결과 분석

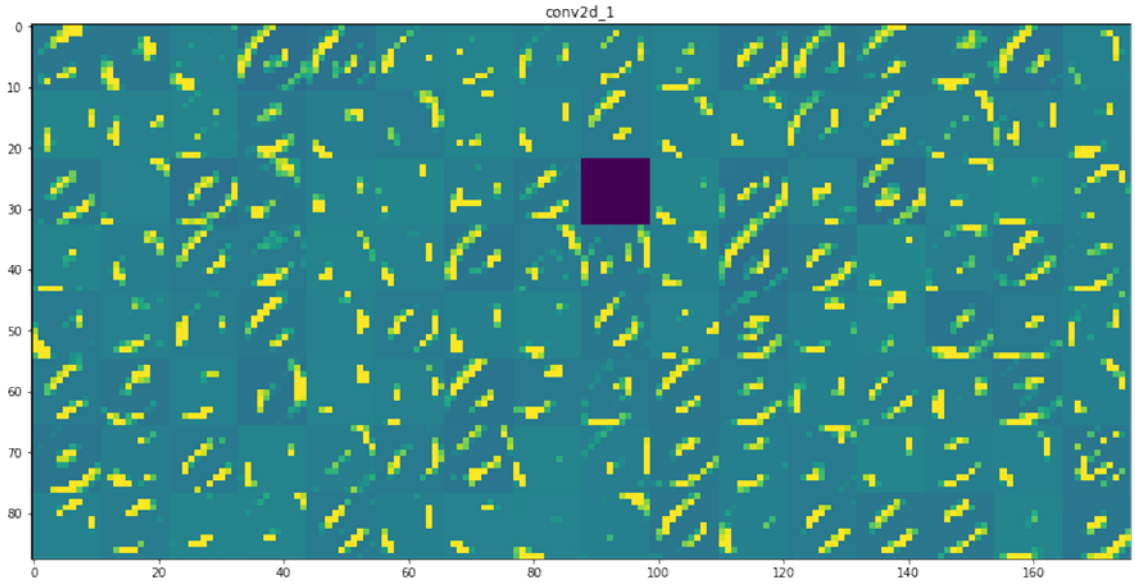
모델을 생성하고 학습시키면 크게 2가지 궁금증이 발생한다.
1. CNN 이 학습하는 가중치(weight)는 무엇인가?
2. Kernel의 개수와 특징 지도의 개수는 어떤 연관성이 있을까?

학습된 후 kernel Convolution 연산된 이미지를 확인해 보자. 첫 번째 Conv2D layer를 통과한 feature map이다. 같은 이미지 같지만 64개의 kernel이 형성되고 이에 따라 비슷해 보이지만 서로 다른 이미지들이 생성되었음을 확인할 수 있다. Convolution 연산을 거칠수록 특징들이 추출되는데 첫 번째 Layer만으로는 어떤 특징들이 추출됐는지 알 수 없다. 현재로서는 처음 입력된 이미지와 같은 형태를 나타내고 있다.



<첫번째 Layer를 거친 이미지의 특징맵>

두 번째 Conv2D layer를 통과한 feature map이다. 64개의 feature map이 128개의 kernel을 통과한 이미지들로 서로 다른 특징을 추출한 결과를 확인할 수 있다. Convolution 연산을 거칠수록 이미지는 점차 추상화되어 원본의 이미지의 형태를 잃어가지만 서로 다른 특징들이 추출되는 것을 확인 할수 있다.



<두번째 Layer를 거친 이미지의 특징맵>

https://www.youtube.com/watch?v=RNnKtNrsrmg 이 동영상은 과일을 종류를 인식하고 class를 분류할 수 있도록 VGG16 알고리즘이 학습하는 과정으로 위의 이미지로 이해가 잘 안 된다면 참고하길 바란다.

CNN은 입력 이미지에서 특징 맵(feature map)을 추출하고 이 특징들을 학습하는 방법 구성되었다. 위에 작성한 모델을 기준으로 우리는 초반에 1개 채널을 가진 이미지를 conv2D layer에 전달하였다. Kernel은 1채널이지만 이를 이용해 총 64개의 특징을 추출한다. 추출된 64개의 특징을 maxplooling2D를 이용해 이미지의 크기를 줄이고, 또다시 64개의 특징을 한번에 취합해 128개의 특징을 추출한다. 독자중에는 가끔 64의 특징 각각의 이미지마다 128개의 특징을 추출한다고 생각하는데, 이는 잘못된 생각이다. Conv 연산은 입력 채널의 개수와는 관계없이 연산이 이루어지며 우리가 지정한 filter의 개수만큼 특징이 추출된다.

학습된 모델의 성능을 평가 하고 어떤 이미지를 제대로 분류하지 못했는지 확인해 보자.

Confusion Matrix는 label의 원래 calss 와 model이 예측한 class가 일치하는 것을 계산한 matrix로 정답 class는 행, 예측한 class 는 열로 나타낸다.

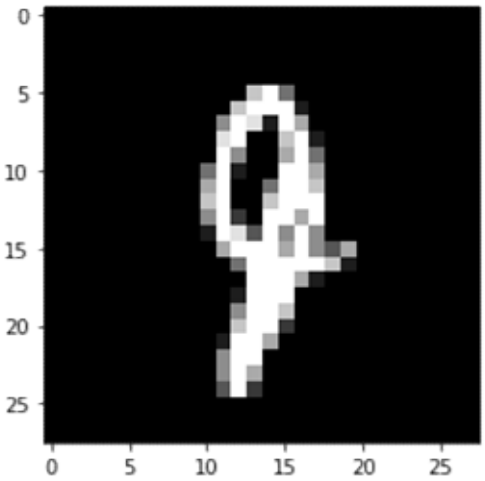
```
model.predict(X_test).argmax(axis=1)#softmax 로 나온 결과를 원래 결과로 되돌려주기 위한 방법
```

```
# Plot confusion matrix 를 통해 맞는값들과 틀린값들을 비교해 보자.
from sklearn.metrics import confusion_matrix
P_test = model.predict(X_test).argmax(axis=1)
confusion_matrix(y_test, P_test)
```

```
array([[ 976,    0,    0,    0,    0,    0,    1,    1,    0,    2],
       [   0, 1132,    1,    0,    0,    0,    1,    1,    0,    0],
       [   2,    1, 1022,    0,    1,    0,    0,    6,    0,    0],
       [   0,    0,    3, 1005,    0,    1,    0,    1,    0,    0],
       [   0,    0,    0,    0,  975,    0,    3,    0,    0,    4],
       [   3,    0,    1,    8,    0,  873,    3,    1,    0,    3],
       [   6,    2,    1,    0,    2,    4,  943,    0,    0,    0],
       [   0,    2,    3,    0,    0,    0,    0, 1020,    0,    3],
       [   6,    0,    6,    1,    1,    1,    1,    2,  953,    3],
       [   0,    1,    3,    0,    2,    2,    0,    2,    1,  998]],
      dtype=int64)
```

정방행렬에 나타난 데이터는 제대로 예측한 데이터를 나타내고 이외의 데이터는 오답을 나타내고 있다. 실제 label과 예측한 class가 서로 다른 데이터를 무작위로 추출해 확인해 보자. 아래의 이미지는 숫자 '9'로 작성했지만, model은 숫자 '4'로 예측했다. 충분히 오해의 소지가 있는 데이터라 할 수 있다.

```
miss_id = np.where(P_test != y_test)[0]
i = np.random.choice(miss_id)
X_test = X_test.reshape(10000, 28,28)
plt.imshow(X_test[i], cmap='gray')
plt.title("True : {} Predict: {}".format(y_test[i], P_test[i]));
```



<Label : 9, Predict : 4>

CNN 모델을 구성하고 학습시킨 뒤 다른 Dataset 에도 적용할 수 없겠냐는 생각을 할 수 있다. 앞서 말했듯이 학습이랑 특징을 추출하는 kernel을 학습시키는 과정으로 같은 model이라도 학습되는 Dataset 의 종류가 변경된다면 새롭게 학습하는 데이터를 분류할 수 있다.(물론 기존의 데이터는 분류하지 ht 한다.) MNIST 에서 제공하는 Fashion_mnist dataset을 이용해 model을 학습시켜 보자.

```
fashion_mnist = tf.keras.datasets.fashion_mnist
```