보고서

- MLP / CNN 모델 구현 -

과목명		인공지능
담당교수		박병준
제출일		2019년 12월 8일
소속		소프트웨어학부
학번		2015202065
이름		윤홍찬

광운대학교
KwangWoon University

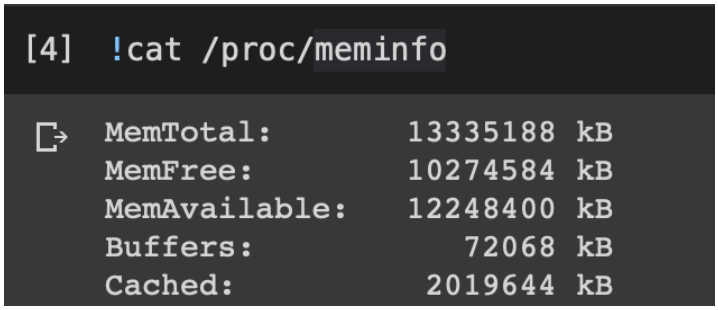
< 목 차 >

1. 코드 구동 환경 및 개발 환경 세팅

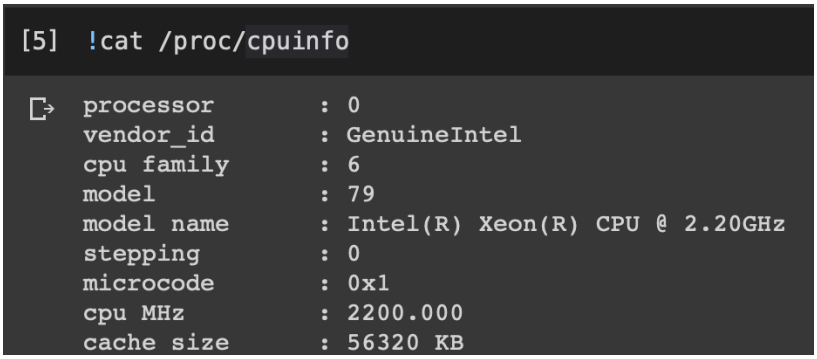
OS : macOS 10.15
IDE : Google Colab
Python Version : 3.6.9
TensorFlow Version : 2.0.0
Keras Version : 2.2.4

개발 환경을 구축하면서 **Google Colab** 을 사용하였다. Colab 은 제한적이지만 무료로 GPU 와 TPU 를 제공해 딥러닝 응용프로그램을 실행시킬 수 있는 IDE 이다. 또한 Colab 은 딥려닝 라이브러리(Keras, Tensorflow, Pytorch)가 기본으로 설치되어있어, 이번 과제를 수행하기 위한 환경을 설정하기 편하겠다고 생각하여 IDE 로 선택하였다.

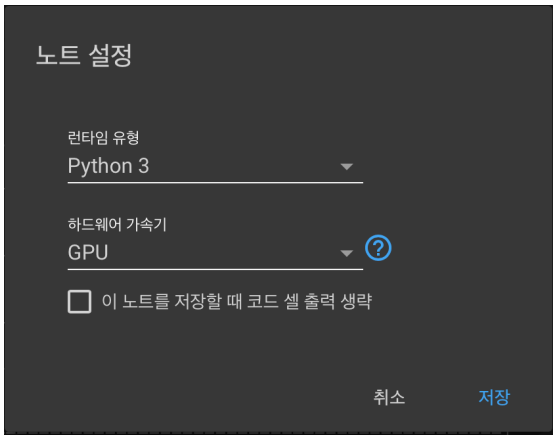
Colab 에서 사용한 클라우드 서버의 사양을 확인하기 위해 다음과 같은 명령어를 사용해 메모리, CPU, GPU 정보를 확인하였다.



' !cat /proc/meminfo ' 명령어를 입력해 메모리 정보를 확인하였다. 총 **13GB 의 메모리**를 사용하고 있는 것을 알 수 있다.



' !cat /proc/cpuinfo' 명령어를 입력해 CPU 정보를 확인해보았다. **Intel(R) Xeon(R) CPU @ 2.20GHz 듀얼코어**를 사용하는 것을 확인하였다.



Colab 은 제한적이지만 무료로 GPU 와 TPU 를 제공한다. 이번 과제를 수행하면서, 위의 사진과 같이 GPU 를 추가적으로 사용하였다.

```
[6]  !nvidia-smi

Fri Dec  6 13:04:37 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P0    32W / 250W |    265MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

‘ **!nvidia-smi** ’ 명령어를 입력해 현재 그래픽 카드에 대한 정보를 확인하였다. 고성능인 Tesla P100 그래픽 카드를 사용하는 것을 알 수 있다.

Google Colab 은 딥러닝 모델을 개발하기 위한 여러 라이브러리 들을 포함하고 있다. 이에 대해 사용하려는 라이브러리들을 import 하고, 버전들을 확인하였다.

```
[1]  try:
       %tensorflow_version 2.x
     except Exception:
       pass

     TensorFlow 2.x selected.
```

Google Colab 은 아직 default 로 1.x 버전의 tensorflow 를 지원한다. 이에 위와 같은 코드를 입력해 2.x 버전을 사용하도록 하였다.

```
[2]  from __future__ import absolute_import, division, print_function, unicode_literals, unicode_literals

     import tensorflow as tf
     from tensorflow import keras

     import numpy as np
     import sys


     print('TensorFlow version : ', tf.__version__)
     print('Keras version : ', keras.__version__)
     print('Python version: ', sys.version)

     TensorFlow version :  2.0.0
     Keras version :  2.2.4-tf
     Python version:  3.6.9 (default, Nov  7 2019, 10:44:02)
     [GCC 8.3.0]
```

tensorflow, keras 를 import 하였다. keras 는 tensorflow 와 같은 여러 딥러닝 툴을 더 쉽게 사용하도록 지원하는 Hight level API 이다. 따라서 이번 과제에서 Keras 를 사용하였다. 이후 각 버전을 확인해보았으며, 각 버전은 위의 사진과 같다.

2. MNIST 데이터 셋

데이터 셋으로는 기본적으로 제공하는 MNIST 데이터 셋을 이용했다. MNIST 데이터 셋은 손 글씨 숫자의 이미지로 이루어져 있으며, 각 이미지는 28 x 28 로 낮은 해상도를 가지고 있다.

```
[3]  (train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.load_data()
```

위와 같이 load_data()를 호출하면, 4 개의 NumPy 배열이 반환된다. (train_images 와 train_labels)는 학습 데이터로 사용하였고, (test_images 와 test_labels)는 테스트 데이터로 사용하였다. 이제, 다음과 같이 데이터셋 구조를 살펴보았다.

```
[4]  train_images.shape

     (60000, 28, 28)
```

학습 이미지 데이터의 구조이다. 60000 개의 데이터가 있으며 28 x 28 픽셀로 표현되는 것을 알 수 있다.

```
[6]  test_images.shape

     (10000, 28, 28)
```

테스트 이미지 데이터의 구조이다. 10000 개의 데이터로 테스트를 진행하며 학습 이미지 데이터와 같이 28 x 28 픽셀로 표현되는 것을 확인하였다.

```
[5]  train_labels

     array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

학습 데이터의 Class(부류) 값을 확인해보았다. 0~9 숫자 이미지와 대응되어 0 ~ 9 까지 10 개의 Class 를 가지고 있다.

```
[7]  train_images = train_images / 255.0
     test_images = test_images / 255.0
```

마지막으로, 신경망 모델의 입력 데이터로 사용하기 위해 데이터 전처리 과정을 거쳤다. 각 픽셀 값의 범위는 0 ~ 255 이며, 이 값의 범위를 0 ~ 1 사이로 조정하기 위해서 학습 데이터와 테스트 데이터를 255 로 나누었다.

3. 다층 퍼셉트론 (MLP)

1) 기본 모델 구현

기본 모델을 구현하는데, 인터넷에 있는 오픈소스를 활용하지 않고 TensorFlow, Keras 튜토리얼과 개발 문서를 참고하였다. 작성한 코드 설명을 시작으로, 로그 캡쳐를 살펴보고, 보고서 맨 마지막에 TensorBoard 출력물을 살펴보겠다.

```
[4]  # First Model

     model1 = keras.Sequential([
         keras.layers.Flatten(input_shape=(28, 28)),
         keras.layers.Dense(64, activation='sigmoid'),
         keras.layers.Dense(10, activation='softmax')
     ])

     model1.compile(optimizer='sgd',
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])

     model1.summary()
```

위와 같이 3 개의 층을 가진 매우 간단한 다층 퍼셉트론 모델을 구현해보았다. **첫 번째 층은 Flatten** 으로, 인자로 입력 데이터의 shape 를 받는다. 이는 28 x 28 픽셀인 2 차원 배열의 이미지 형식을 784(28 x 28) 픽셀의 1 차원 배열로 변환한다. 즉, 이미지에 있는 픽셀의 행을 일렬로 늘린다.

두 번째, 세 번째 층은 Dense 층으로 FC(완전 연결)층이다. **두 번째 Dense 층**은 64 개의 노드를 가지도록 했다. 또한 **활성화 함수로 sigmoid 함수를 사용**하도록 하였다.

세 번째 Dense 층은 분류를 위해 10 개의 노드를 가진(Class 수와 같음) **Softmax 층**으로 설정했다. Softmax 층 각 노드의 출력 값은 0 보다 크며, 총 합은 1 이 된다. 그리고 각 노드는 현재 이미지가 10 개의 Class 중 해당 Class 에 속할 확률이 된다.

이렇게 모델을 생성한 후에 모델을 훈련하기 전 필요한 몇 가지 설정이 Compile 단계에서 추가된다. 첫 번째, 매개변수로 전달한 **Optimizer** 는 데이터와 Loss Function(두 번째 매개변수)을 바탕으로 모델의 업데이트 방법을 결정한다. 이 모델에서는 경사하강법을 약간 개선한 **SGD(Stochastic Gradient Descent) 방법을 사용**하였다. 이는 기본 경사하강법(GD)과는 달리 모든 데이터 마다 업데이트를 하는 것이 아니라 Mini batch 크기만큼 업데이트를 진행한다. 따라서 일반적인 경사하강법보다 아주 정확하지는 않지만, 빠르게 동작하게 된다. 두 번째, 매개변수로 전달한 **loss** 는 위에서 언급한 Loss Function 이며, **cross entropy** 를 사용한다. 모델의 학습이 올바른 방향으로 가기 위해 이 함수를 최소화 하도록 가중치를 업데이트 하는 것이 목표이다. 마지막 세 번째 인자 **Metrics** 는 훈련 단계와 테스트 단계를 모니터링하기 위해 사용하며, **정확도를** 측도로 사용하였다.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 64)                50240
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

모델을 설정하고, summary() 함수를 호출한 결과이다. 위와 같이 각 층에 대한 정보를 시각화하여, 모델의 각 층을 보기 편하게 확인 할 수 있다.

```
[ ]  # Tensor Board Code
     %reload_ext tensorboard
     logdir1 = './logs/1'
     logdir2 = './logs/2'

     tbCallBack1 = keras.callbacks.TensorBoard(logdir1,
                                 histogram_freq=1,
                                 write_graph=True,
                                 write_images=True)
```

위의 코드는 Colab 에서 **TensorBoard 를 사용하기 위한 코드**이다. Keras 에서는 비교적 쉽게 TensorBoard 를 이용할 수 있다. 첫 번째 줄에서 텐서 보드를 사용하기 위해 로드를 하고, 로그를 저장할 위치를 지정하였다.(logs/1 에는 첫 번째 모델이, logs/2 에는 개

선된 두 번째 모델 저장) 그리고 로그 위치 등을 인자로 전달해 TensorBoard 콜백 함수를 만들었다.

```
[ ]  model1.fit(train_images, train_labels, epochs=20, callbacks=[tbCallBack1])

     Train on 60000 samples
     Epoch 1/20
     60000/60000 [==============================] - 5s 88us/sample - loss: 1.5139 - accuracy: 0.6847
     Epoch 2/20
     60000/60000 [==============================] - 4s 65us/sample - loss: 0.7669 - accuracy: 0.8437
     Epoch 3/20
     60000/60000 [==============================] - 4s 66us/sample - loss: 0.5612 - accuracy: 0.8684
     Epoch 4/20
     60000/60000 [==============================] - 4s 67us/sample - loss: 0.4724 - accuracy: 0.8813
```

이제 모델을 학습시키기 위해, 학습 데이터(train_images, train_labels)를 인자로 fit 함수를 호출하였다. 이때, 두 번째 인자인 **epochs 값**은 해당 학습 데이터로 n 번 학습을 진행하는 것이며, TensorBorad로 시각화하여 표현하기 위해, 위해서 설정한 TensorBoard 콜백 함수를 인자로 전달하였다.

```
[ ]  test_loss1, test_acc1 = model1.evaluate(test_images,  test_labels, verbose=2)

     print('\n첫 번째 모델 테스트 정확도:', test_acc1)

     10000/1 - 1s - loss: 0.2007 - accuracy: 0.9297

     첫 번째 모델 테스트 정확도: 0.9297
```

이후 테스트 데이터(test_images, test_labels)를 인자로 전달하여 evaluate 함수를 호출해, **학습한 모델을 평가**하였다. **첫 번째 모델의 정확도는 약 92%**가 나왔다.

2) 개선 모델 구현

기본 모델에서 **두 가지 개선할 점**을 발견하여, 개선 모델을 개발하였다. 이는 **활성화 함수와 최적화 기법이다.**

첫 번째로, 기본 모델에서는 활성화 함수로 **Sigmoid 함수를 사용**하였다. 하지만 이 함수는 입력 값이 조금만 커지거나 작아져도 곡선이 평평해져서 기울기가 0 에 가까워지는 기울기 소멸 문제가 발생하게 된다. 따라서 여러 층이 있는 딥러닝에서는 잘 쓰이지 않는다. 따라서 **이를 개선하기 위해 ReLU 함수를 활성화 함수로 사용**하였다. 이 함수는 입력 값이 0 보다 크면 그대로, 0 보다 작으면 0 을 내보내는 형태로, 기울기 소멸 문제를 완화할 수 있다.

두 번째로, 기본 모델에서는 최적화 기법으로 경사하강법을 약간 개선한 **SGD(Stochastic Gradient Descent) 방법을 사용**하였다. 하지만 이 방법을 개선한 여러 가지 최적화 기법이 존재하였다. 대표적으로 **Adam 을 사용**하는데, 이는 또 다른 개선된

기법인 RMSprop 에 모멘텀의 개념을 도입한 최적화 기법이다. 따라서 Adam 방식을 적용하여 모델을 개선해보았다.

```
[ ]  # Second Model

    model2 = keras.Sequential([
        keras.layers.Flatten(input_shape=(28, 28)),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(10, activation='softmax')
    ])

    model2.compile(optimizer='adam',
                   loss='sparse_categorical_crossentropy',
                   metrics=['accuracy'])

    model2.summary()
```

모델을 개선한 소스코드이다. 두 번째 Dense 층에서, **활성화 함수로 ReLU 함수를 사용**한 것을 볼 수 있으며, compile 단계에서 최적화 기법으로 **Adam 방식을 적용**하였다.

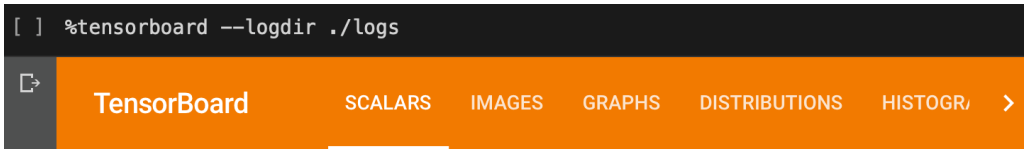
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

summary() 함수를 호출한 결과이다. 전체적인 층의 구조를 바꾸지는 않았기 때문에, 큰 차이는 없어 보인다.

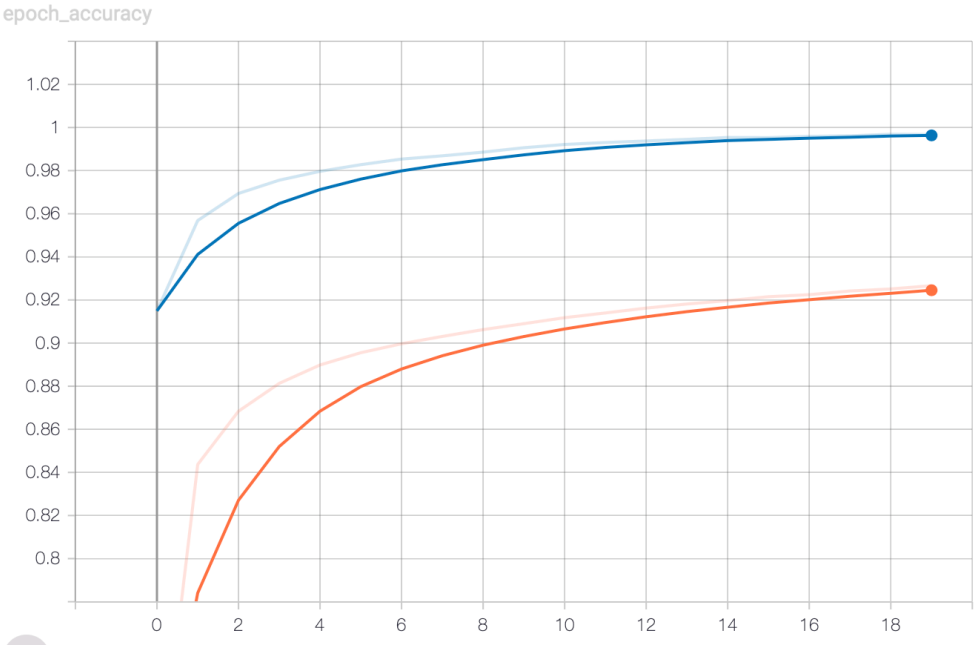
```
[ ]  test_loss2, test_acc2 = model2.evaluate(test_images,  test_labels, verbose=2)

     print('\n개선된 모델 테스트 정확도:', test_acc2)

     10000/1 - 1s - loss: 0.0550 - accuracy: 0.9751

     개선된 모델 테스트 정확도: 0.9751
```

이후, 기본 모델과 같이 학습 단계를 거쳐 테스트 데이터로 평가해보았다. 개선된 두 번째 모델의 **정확도는 약 97%**가 나왔다.

3) 실험 결과 분석 (로그 캡쳐, TensorBoard 출력)



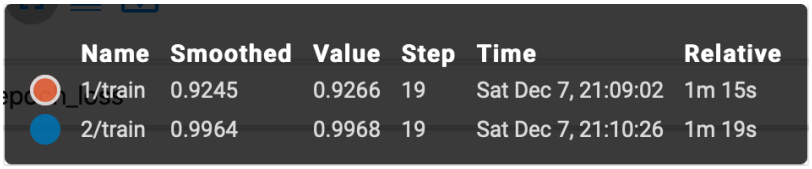
위와 같은 코드를 입력해 Colab 에서 편하게 TensorBoard 를 확인할 수 있다.



[TensorBoard 로 학습 과정을 시각화]

위의 사진은 학습 과정을 TensorBoard 로 시각화 한 것이다. x 축은 Epoch 횟수를 나타내며, y 축은 정확도를 나타낸다. 또한, 주황색 곡선은 첫 번째 기본 모델이고 파란색 곡선은 개선한 두 번째 모델을 표현한다.

위의 사진에서 보듯이 개선된 두 번째 모델이 기본 모델에 비해 첫 학습 주기부터 **높은 정확도**를 보이며, **꾸준히 정확도가 증가**한다. 이에 비해, 첫 번째 기본 모델은 비교적 **낮은 정확도**에서 시작해 **큰 폭으로 정확도가 증가**한다.



Name	Smoothed	Value	Step	Time	Relative
1/train	0.9245	0.9266	19	Sat Dec 7, 21:09:02	1m 15s
2/train	0.9964	0.9968	19	Sat Dec 7, 21:10:26	1m 19s

최종적으로 두 모델은 학습하는데 **1 분을 약간 넘긴 시간이 소요**되었으며, 기본 모델은 약 **92%의 정확도로 학습을 마치고, 개선된 모델은 약 99%의 정확도로 학습을 마친다**.

```
[ ]  test_loss1, test_acc1 = model1.evaluate(test_images,  test_labels, verbose=2)

     print('\n첫 번째 모델 테스트 정확도:', test_acc1)

     10000/1 - 1s - loss: 0.2007 - accuracy: 0.9297

     첫 번째 모델 테스트 정확도: 0.9297
```

[첫 번째 모델의 정확도]

```
[ ]  test_loss2, test_acc2 = model2.evaluate(test_images,  test_labels, verbose=2)

     print('\n개선된 모델 테스트 정확도:', test_acc2)

     10000/1 - 1s - loss: 0.0550 - accuracy: 0.9751

     개선된 모델 테스트 정확도: 0.9751
```

[두 번째 모델의 정확도]

위에서 각 모델 구현 마지막에 제시한 테스트 데이터를 이용한 정확도 로그이다. **첫 번째 모델은 약 92%의 정확도**를 보이고, **두 번째 모델은 이보다 높은 약 97%의 정확도**를 보인다. 따라서 **개선된 모델이 첫 번째 모델보다 더 정확하게 이미지를 분류하는 것을 알 수 있다.**

3. Convolution Neural Network (CNN)

1) 기본 모델 구현

CNN 기본 모델을 구현하는데, MLP 와 마찬가지로 인터넷에 있는 오픈소스를 활용하지 않고 TensorFlow, Keras 튜토리얼과 개발 문서를 참고하였다. 작성한 코드 설명을 시작으로, 로그 캡쳐를 살펴보고, 보고서 맨 마지막에 TensorBoard 출력물을 살펴보겠다.

```
[5]  from tensorflow.keras.models import Sequential
     from tensorflow.keras import optimizers
     from tensorflow.keras.layers import Dense, Activation, Flatten, Conv2D, MaxPooling2D
     from tensorflow.keras.layers import BatchNormalization, Dropout
```

먼저, 다음과 같이 필요한 모듈들을 import 하였다.

```
[6]  # First Model

     model1 = Sequential()
     model1.add(Conv2D(input_shape = (train_images.shape[1], train_images.shape[2], train_images.shape[3]),
                       filters = 30, kernel_size = (3,3), strides = (1,1), padding = 'same'))
     model1.add(Activation('relu'))
     model1.add(Conv2D(filters = 30, kernel_size = (3,3), strides = (1,1), padding = 'same'))
     model1.add(Activation('relu'))
     model1.add(MaxPooling2D(pool_size = (2,2)))
     model1.add(Flatten())
     model1.add(Dense(64, activation = 'relu'))
     model1.add(Dense(10, activation = 'softmax'))

     adam = optimizers.Adam(lr = 0.001)
     model1.compile(loss = 'categorical_crossentropy', optimizer = adam, metrics = ['accuracy'])
     model1.summary()
```

위와 같이 8 개의 층을 가진 CNN 구조 치고는 간단한 CNN 모델을 구현해보았다. 첫 번째 층은 **Conv2D 층으로, 컨볼루션 연산을 한다**. **input_shape** 는 입력 데이터의 형태를 인자로 전달하였다. 두 번째로, **filters 인자는 필터의 개수를 의미**하며, 필터의 개수는 다음 레이어의 깊이를 결정하게 된다. 세 번째 인자는 **kernel_size** 로, 합성곱 연산을 진행할 **필터의 사이즈를 의미**한다. 네 번째 인자인 **strides** 는 합성곱 연산을 수행할 때, **한 번에 움직이는 정도**를 의미한다. 마지막 인자인 **padding** 은 주위에 0 으로 이루어진 패딩을 추가해 **차원을 유지할 수 있는데**, 값으로 same 을 전달하여, **출력의 크기가 입력의 크기와 같도록 설정**하였다.

두 번째 층은 MLP 의 개선한 모델에서 사용한 활성화 함수인 **ReLU 를 사용**하였다. 세 번째 층, 네 번째 층은 다시 첫 번째 층과 동일하게 컨볼루션 연산을 수행하고, 다시 활성화 함수를 사용하였다.

다섯 번째 층에서는 **최대 값 풀링을 진행**하였다. 이는 2x2 영역에서 해당 영역을 묘사하는 최대 값을 뽑아 데이터의 크기를 줄여주는 역할을 한다.

여섯 번째 층에서 FC 층으로 연결하기 전에 3 차원의 데이터 차원을 줄이기 위해 **Flatten 층을 추가**하였다.

일곱 번째, 여덟 번째 층에서는 **Dense** 층으로, 위의 MLP 모델과 동일한 코드를 넣었다. 즉, 마지막 층은 분류를 하기 위해 **softmax 층으로 구성**하였다.

이렇게 모델을 생성한 후에 모델을 훈련하기 전 필요한 몇 가지 설정이 Compile 단계에서 추가된다. 이 과정은 개선된 MLP 모델과 동일하게 적용하였으므로, 설명을 생략하겠다.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 30)        300
_________________________________________________________________
activation (Activation)      (None, 28, 28, 30)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 30)        8130
_________________________________________________________________
activation_1 (Activation)    (None, 28, 28, 30)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 30)        0
_________________________________________________________________
flatten (Flatten)            (None, 5880)              0
_________________________________________________________________
dense (Dense)                (None, 64)                376384
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 385,464
Trainable params: 385,464
Non-trainable params: 0
_________________________________________________________________
```

모델을 설정하고, summary() 함수를 호출한 결과이다. 위와 같이 각 층에 대한 정보를 볼 수 있는데, MLP 보다 조금 더 복잡해 진 것을 알 수 있다.

```
[9] model1.fit(train_images, train_labels, epochs = 10, callbacks=[tbCallBack1])

Train on 60000 samples
Epoch 1/10
60000/60000 [==============================] - 18s 308us/sample - loss: 0.2743 - accuracy: 0.9464
Epoch 2/10
60000/60000 [==============================] - 15s 242us/sample - loss: 0.0667 - accuracy: 0.9801
Epoch 3/10
60000/60000 [==============================] - 15s 246us/sample - loss: 0.0466 - accuracy: 0.9853
```

이제 모델을 학습시키기 위해, 학습 데이터(train_images, train_labels)를 인자로 fit 함수를 호출하였다. 이것 또한 MLP 에서 설명했으므로, 자세한 설명은 생략한다.

```
[12] test_loss1, test_acc1 = model1.evaluate(test_images,  test_labels, verbose=2)

     print('\n첫 번째 모델 테스트 정확도:', test_acc1)

10000/1 - 1s - loss: 0.0661 - accuracy: 0.9802

첫 번째 모델 테스트 정확도: 0.9802
```

이후 테스트 데이터(test_images, test_labels)를 인자로 전달하여 evaluate 함수를 호출해, **학습한 모델을 평가**하였다. **첫 번째 모델의 정확도는 약 98%**가 나왔다.

2) 개선 모델 구현

기본 모델에서 **두 가지 개선할 점**을 발견하여, 개선 모델을 개발하였다. 이는 **가중치 초기화 와 Dropout 이다.**

신경망 학습은 가중치를 학습 해나간다. 이때, 가중치를 어떻게 초기화 하느냐에 따라 모델의 성능이 달라지게 된다. 가중치 초기화 방법을 따로 설정해 주지 않으면 기본적으로 Keras 는 일정 구간 내에서 랜덤하게 선택하는 random_uniform 방식을 사용한다. 따라서 가중치를 초기화 하는 여러 방법들이 있는데, 이번 모델에서는 대표적으로 사용되는 **He 초기화** 방법을 사용하였다

두 번째로, MLP 와 같이 간단한 모델은 드랍아웃이 의미가 없을 수 있지만, CNN 과 같이 복잡한 모델을 만들 때 드랍아웃은 유용하다. 따라서 모델의 과적합을 방지하기 위해 **드랍아웃 기법을 사용**하였다.

추가적으로, 동일한 컨볼루션-ReLU-컨볼루션-ReLU-Pool 층을 FC 층 전에 추가하였다.

```
[7]  # Second Model

     model2 = Sequential()
     model2.add(Conv2D(input_shape = (train_images.shape[1], train_images.shape[2], train_images.shape[3]),
                       filters = 30, kernel_size = (3,3), strides = (1,1), padding = 'same', kernel_initializer='he_normal'))
     model2.add(Activation('relu'))
     model2.add(Conv2D(filters = 30, kernel_size = (3,3), strides = (1,1), padding = 'same', kernel_initializer='he_normal'))
     model2.add(Activation('relu'))
     model2.add(MaxPooling2D(pool_size = (2,2)))
     model2.add(Conv2D(filters = 30, kernel_size = (3,3), strides = (1,1), padding = 'same', kernel_initializer='he_normal'))
     model2.add(Activation('relu'))
     model2.add(Conv2D(filters = 30, kernel_size = (3,3), strides = (1,1), padding = 'same', kernel_initializer='he_normal'))
     model2.add(Activation('relu'))
     model2.add(MaxPooling2D(pool_size = (2,2)))
     model2.add(Flatten())
     model2.add(Dense(64, activation = 'relu'))
     model2.add(Dropout(0.3))
     model2.add(Dense(10, activation = 'softmax'))

     adam = optimizers.Adam(lr = 0.001)
     model2.compile(loss = 'categorical_crossentropy', optimizer = adam, metrics = ['accuracy'])
     model2.summary()
```

모델을 개선한 소스코드이다. 각 컨볼루션 층에서 **kernel_initializer 인자 값으로 he_normal 을 전달 해 He 초기화 방법을 사용**하고, 맨 마지막 Softmax 층 이전에 **Dropout 을 적용**한 것을 볼 수 있다. 또한 추가적으로 **동일한 층을 늘린 것**을 알 수 있다.

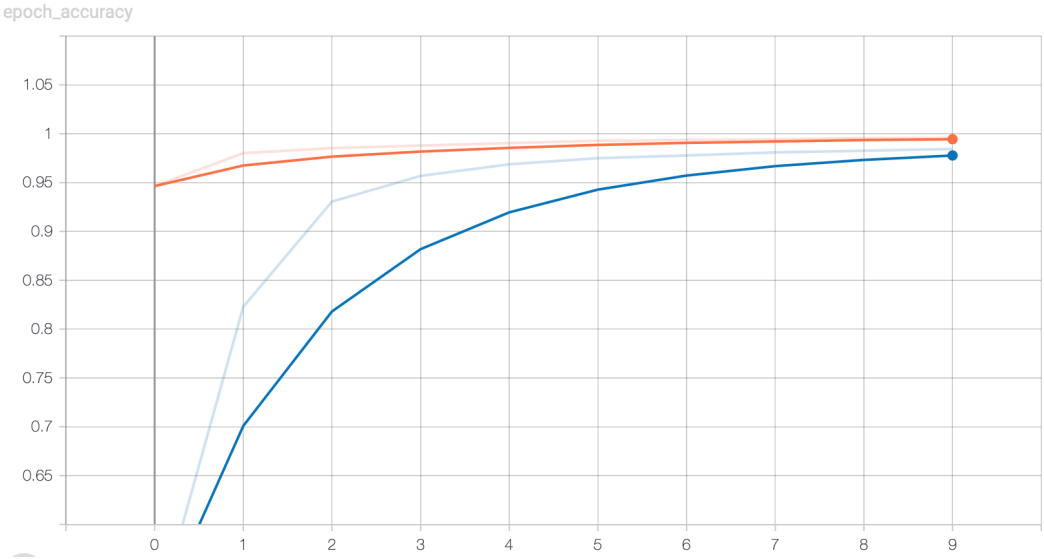
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 28, 28, 30)        300
_________________________________________________________________
activation_2 (Activation)    (None, 28, 28, 30)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 30)        8130
_________________________________________________________________
activation_3 (Activation)    (None, 28, 28, 30)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 30)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 30)        8130
_________________________________________________________________
activation_4 (Activation)    (None, 14, 14, 30)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 14, 14, 30)        8130
_________________________________________________________________
activation_5 (Activation)    (None, 14, 14, 30)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 30)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1470)              0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                94144
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 119,484
Trainable params: 119,484
Non-trainable params: 0
_________________________________________________________________
```

summary() 함수를 호출한 결과이다. 전체적인 층의 구조가 바뀌었기 때문에 조금 더 복잡한 CNN 모델이 완성되었다.

```
[13] test_loss2, test_acc2 = model2.evaluate(test_images,  test_labels, verbose=2)

     print('\n개선된 모델 테스트 정확도:', test_acc2)

     10000/1 - 2s - loss: 0.0243 - accuracy: 0.9892

     개선된 모델 테스트 정확도: 0.9892
```

이후, 기본 모델과 같이 학습 단계를 거쳐 테스트 데이터로 평가해보았다. 개선된 두 번째 모델의 **정확도는 약 99%**가 나왔다.

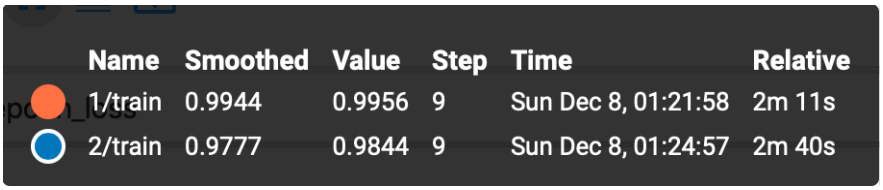
3) 실험 결과 분석 (로그 캡쳐, TensorBoard 출력)



[TensorBoard 로 학습 과정을 시각화]

위의 사진은 학습 과정을 TensorBoard 로 시각화 한 것이다. MLP 와 마찬가지로 x 축은 Epoch 횟수를 나타내며, y 축은 정확도를 나타낸다. 또한, 주황색 곡선은 첫 번째 기본 모델이고 파란색 곡선은 개선한 두 번째 모델을 표현한다.

위의 사진에서 보듯이 첫 번째 모델은 처음부터 개선된 모델에 비해 첫 학습 주기부터 **높은 정확도**를 보이며, **꾸준히 정확도가 증가**한다. 이에 비해, 두 번째 개선된 모델은 비교적 **낮은 정확도**에서 시작해 **큰 폭으로 정확도가 증가**하다가, **최종 학습 주기에 첫 번째 모델과 비슷한 정확도(약 99%)**를 보여준다. 이는 **두 번째 모델에 Dropout 을 적용**하여 이렇게 결과가 나타난 것이다. 이렇게 두 번째 모델은 초기에 학습에서는 정확도가 좋지 않다가 학습을 진행할 수록 큰 폭으로 정확도가 좋아지게 된다.



Name	Smoothed	Value	Step	Time	Relative
1/train	0.9944	0.9956	9	Sun Dec 8, 01:21:58	2m 11s
2/train	0.9777	0.9844	9	Sun Dec 8, 01:24:57	2m 40s

최종적으로 첫 번째 모델은 학습하는데 **2 분 11 초**가, 두 번째 개선된 모델은 **2 분 40 초**가 소요되었다. 기본적으로 두 모델은 MLP 모델보다 복잡해 학습 시간이 오래 걸렸으며, 두 번째 모델은 층을 추가적으로 추가했음에도 불구하고 큰 폭으로 시간이 늘어나지 않았는데, 이 또한 **Dropdout 을 적용해 학습 시간이 단축되었음**을 알 수 있다.

```
[12] test_loss1, test_acc1 = model1.evaluate(test_images,  test_labels, verbose=2)

     print('\n첫 번째 모델 테스트 정확도:', test_acc1)

     10000/1 - 1s - loss: 0.0661 - accuracy: 0.9802

     첫 번째 모델 테스트 정확도: 0.9802
```

[첫 번째 모델의 정확도]

```
[13] test_loss2, test_acc2 = model2.evaluate(test_images,  test_labels, verbose=2)

     print('\n개선된 모델 테스트 정확도:', test_acc2)

     10000/1 - 2s - loss: 0.0243 - accuracy: 0.9892

     개선된 모델 테스트 정확도: 0.9892
```

[두 번째 모델의 정확도]

위에서 각 모델 구현 마지막에 제시한 테스트 데이터를 이용한 정확도 로그이다. **첫 번째 모델은 약 98%의 정확도**를 보이고, **두 번째 모델은 이보다 약간 높은 약 99%의 정확도**를 보인다. 따라서 **개선된 모델이 첫 번째 모델보다 더 정확하게 이미지를 분류하는 것을 알 수 있으며,** 모델 개선이 성공적으로 이루어졌음을 알 수 있다.