인공지능 과제 리포트

과제 제목 : Image Classification

B611030 김민호 (1분반)

B611137 이강복 (2분반)

B611169 임정민 (2분반)

1 과제 개요

이전에 KNN 알고리즘을 사용하여 MNIST 데이터를 분류하는 과제가 있었는데, 이 과제로 인해 이미지 분류하는 작업에 흥미가 생겨 이 주제를 하게 되었다.

2 구현 환경

2.1 SVM

IDE : VS Code, Framework : Keras, Scikit-learn
Intel(R) Core(TM) i9-9880H CPU @2.30GHz, RAM 16.0GB, macOS Big Sur, Radeon Pro 560X 4GB 환경에서 실행

2.2 DNN

IDE : Visual Studio Code, Framework : Pytorch
Intel(R) Core™ i5-10400 CPU @ 2.90GHz, RAM 16.0GB, Windows 10-64bit, NVIDIA GeForce GTX 1660 환경에서 실행

2.3 CNN

IDE : Pycharm, Framework : Pytorch

Intel(R) Core(TM) i5-7500 CPU @3.40GHz, RAM 16.0GB, Windows 10-64bit, NVIDIA GeForce GTX 1060 3GB 환경에서 실행

3 알고리즘에 대한 설명

3.1 SVM

서포트 벡터 머신(support vector machine, SVM)은 기계 학습의 분야 중 하나로 패턴 인식, 자료 분석을 위한 지도 학습 모델이며, 주로 분류와 회귀 분석을 위해 사용한다. 두 카테고리 중 어느 하나에 속한 데이터의 집합이 주어졌을 때, SVM 알고리즘은 주어진 데이터 집합을 바탕으로 하여 새로운 데이터가 어느 카테고리에 속할지 판단하는 비확률적 이진 선형 분류 모델을 만든다. 만들어진 분류 모델은 데이터가 사상된 공간에서 경계로 표현되는데 SVM 알고리즘은 그 중 가장 큰 폭을 가진 경계를 찾는 알고리즘이다.

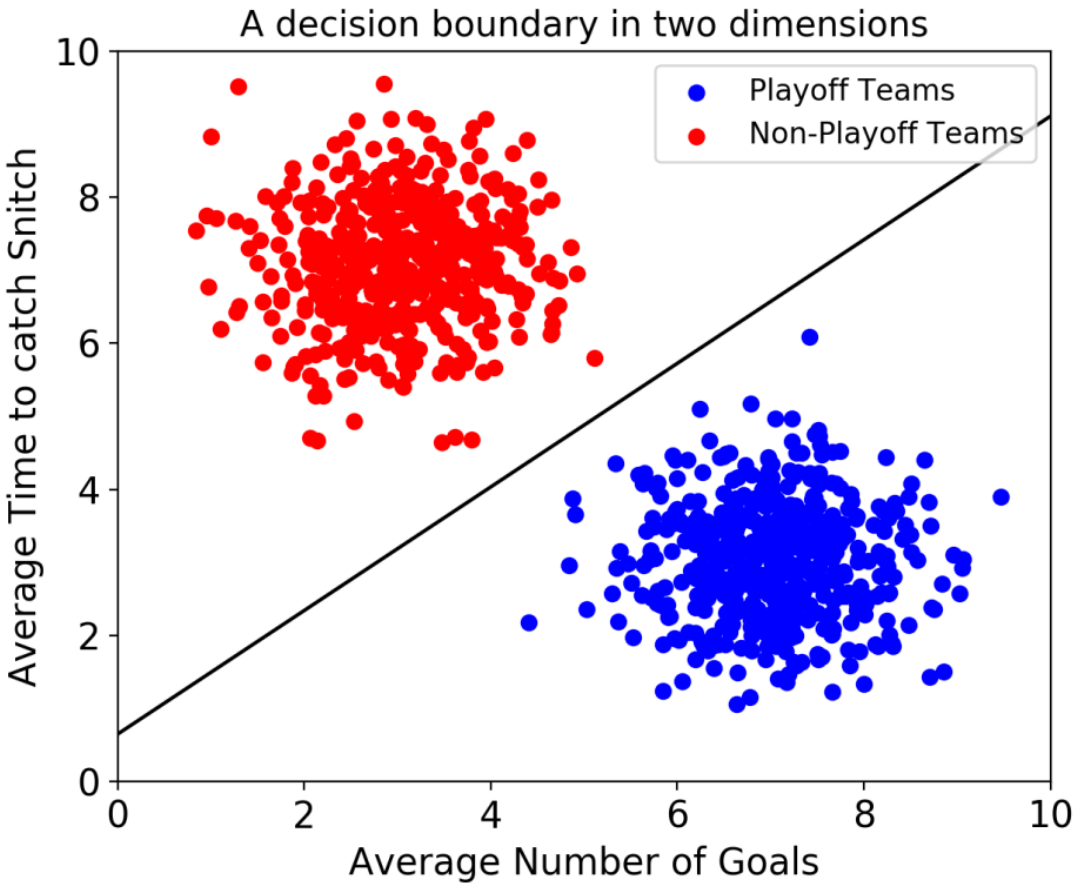


그림 1 : A decision boundary in two dimensions

3.1.1 최적의 결정 경계

SVM 이라는 이름에서 Support Vectors 는 결정 경계와 가장 가까이 있는 데이터 포인트들을 의미한다. 이 데이터들이 경계를 정의하는 결정적인 역할을 하는데, 아래 6개의 그래프에서는 F 그래프가 두 클래스(분류) 사이에서 거리가 가장 멀기 때문에 가장 적절하고, 이때 결정 경계는 데이터 군으로부터 최대한 멀리 떨어지는 게 좋다는 것을 확인할 수 있다.

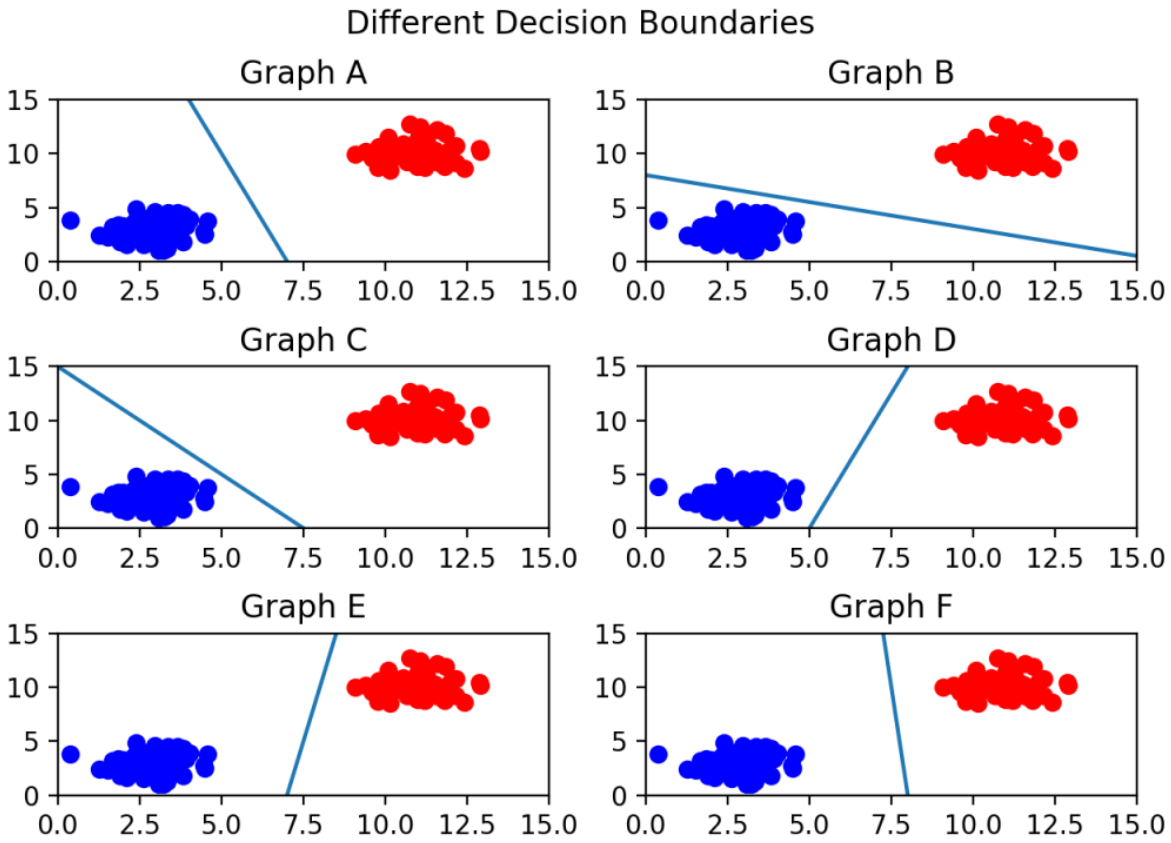


그림 2 : Different decision boundaries

3.1.2 마진(Margin)

마진(Margin)은 결정 경계와 서포트 벡터 사이의 거리를 의미한다. 아래 그림에서 가운데 실선이 결정 경계이고, 그 실선으로부터 검은 테두리가 있는 빨간점 1개, 파란점 2개까지 영역을 두고 점선이 그어져있는데, 이때 점선으로부터 결정 경계까지의 거리가 마진(Margin)이다.

여기서 최적의 결정 경계는 마진이 최대화된 경우이며, n개의 속성을 가진 데이터에는 최소 n+1개의 서포트 벡터가 존재한다는 것을 확인할 수 있다.

실제로 SVM에서는 결정 경계를 정의하는 서포트 벡터만 잘 골라내면 나머지 불필요한 데이터 포인트들을 무시할 수 있어 매우 빠르다.

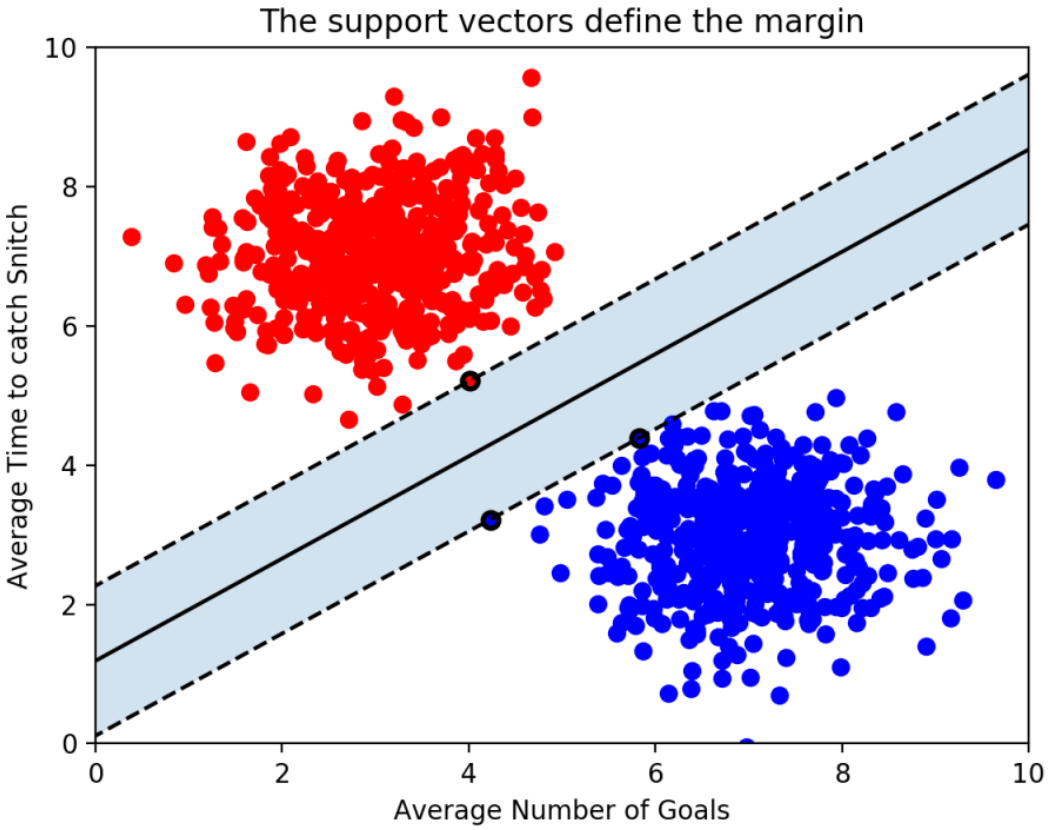


그림 3 : The support vectors define the margin

3.1.3 이상치(Outlier)

분류 과정에서 혼자 튀는 데이터 포인터들을 이상치(Outlier)라고 한다. 아래 그림 중 첫번째 그림은 이상치를 허용하지않고 기준을 까다롭게 세운 경우인데, 이는 하드 마진(Hard margin)으로 마진(Margin)이 매우 작아지고 오버피팅(overfitting)의 문제가 발생할 수 있다. 두번째 그림은 이상치를 마진 안에 어느정도 포함시키고 기준을 너그럽게 세운 경우인데, 이는 소프트 마진(Soft Margin)으로 마진(Margin)이 커지고 반대로 언더피팅(underfitting)의 문제가 발생할 수 있다.

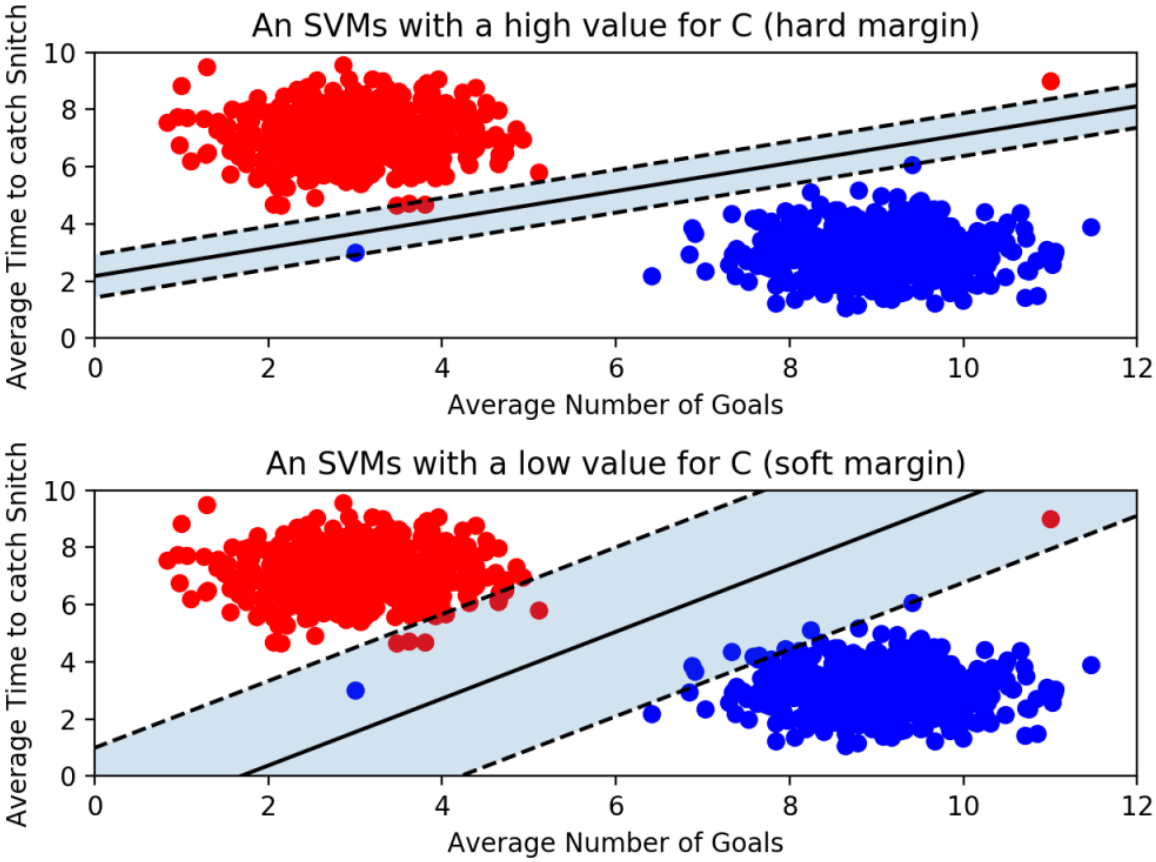


그림 4 : An SVMs with a high value for C

3.2 DNN

깊은 신경망(Deep Neural Network)은 퍼셉트론이나 단순한 Neural Network와 다르게 신경망의 층을 더욱 깊게 만든 방법이다. 컴퓨팅 기술의 발달로 활용 불가능하던 DNN을 사용하여 딥러닝의 침체기를 해결할 수 있었다. 이 DNN은 Error Function, Chain Rule, Gradient Descent, Error Back Propagation을 이용하여 컴퓨터가 스스로 가중치를 최적화 시켜주도록 되어있다.

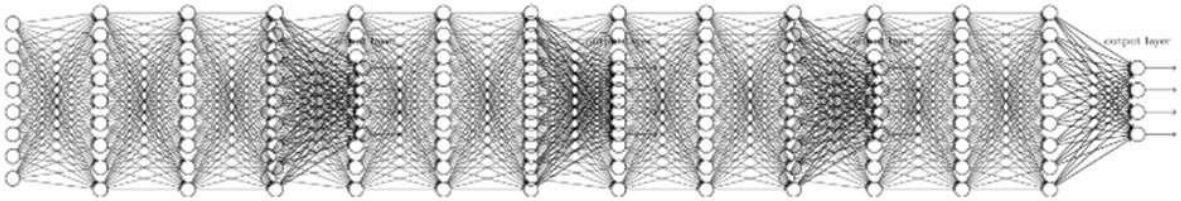


그림 5 : Deep Neural Network

3.2.1 ANN

인공 신경망은 생체 신경망의 신호전달 체제를 모방하여 만들어낸 수학적인 모델이다. 인공신경망의 구성요소는 입력층(Input Layer), 은닉층(Hidden Layer), 출력층(Output Layer)으로 구성되어 있다. 입력층과 출력층은 각각 1 개로 구성되어 있고, 은닉층을 1 개 이상을 사용하여 다층 신경망(Multi-Layered Neural Network)을 구성할 수 있다.

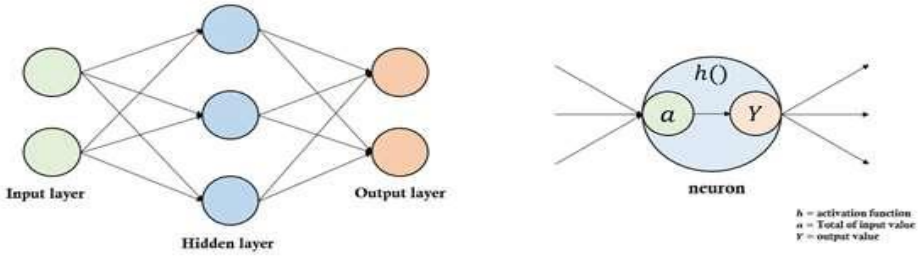


그림 6 : ANN 의 기본 구조(좌)와 Hidden Layer(우)

3.2.2 Backpropagation

역전파는 역방향으로 오차를 전파한다는 뜻으로 경사 하강법과 같은 최적화 방법과 함께 인공 신경망을 학습시킬 때 사용한다. 역전파 알고리즘은 출력층의 오차를 역전파 하여 은닉층을 학습함으로써 다층 퍼셉트론이 가진 문제점을 해결하였다. 또한, 다층 퍼셉트론뿐만 아니라 다양한 종류의 FNN 에서 학습 알고리즘으로 많이 이용된다.

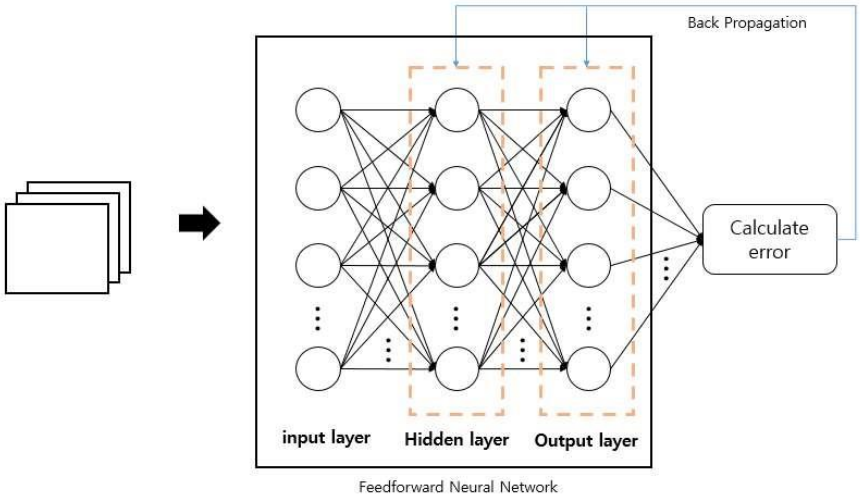


그림 7 : Backpropagation

역전파 알고리즘은 학습 데이터로부터 신경망의 출력 값과 목적 값과의 차이인 오차를 계산하여 각 층에 전달하는 전파단계와 전파된 오차를 이용하여 가중치를 수정하는 가중치 수정 단계로 구성되며 작동 방식은 다음과 같다.
역전파는 Feedforward 와 Backpropagation 총 2 단계로 구성된다. Feed forward 단계에서는 훈련 데이터를 신경망에 인가하고 출력단에서의 에러와 Cost function 을 구한다. 신경망이 충분히 학습되지 못하는 경우는 오차가 클 것이며, 이 큰 오차값을 Backpropagation 시키면서 가중치와 바이어스 값을 갱신한다. 훈련 데이터에 대해서 반복적으로 이 과정을 거치게 되면, 가중치와 바이어스는 훈련데이터에 최적화된 값으로 바뀌게 된다. 좋은 학습결과를 얻으려면 이 훈련 데이터가 한쪽으로 치우치지 않고, 범용성을 가져야 한다. 아래는 Backpropagation 의 한 예이다.

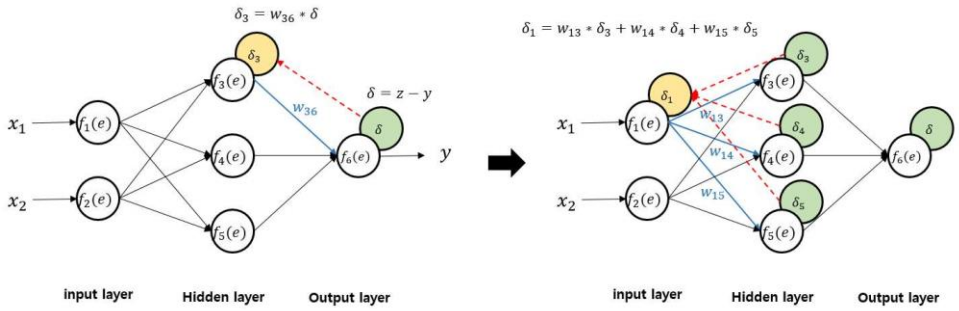


그림 8 : Backpropagation 기본 원리

왼쪽 그림의 최종 노드에서 z-y 인 오차값을 구해 이를 가중치를 곱한 후 그전 노드에 전달하여 최종 처음 노드로까지 이어지게 된다. 이렇게 반복적으로 오류를 전파시키며 가중치와 바이어스 값을 갱신한다.

3.2.3 Over-fitting

과적합 문제는 Training set 에만 지나치게 적응하여 Test set 에 제대로 반응하지 못하는 현상이다. 이는 훈련데이터가 적은 경우나 훈련을 과하게 하였을 때 생길 수 있다. 우리가 하는 프로젝트의 훈련데이터는 적절히 있으니 훈련을 과하게 하는 문제에 신경을 써야할 것 같다.

3.2.4 Dropout

Dropout 은 과적합을 방지하기 위한 방법 중 하나이다. Fully connected 되어 있는 Hidden unit 을 랜덤하게 없애 버린 상태에서 학습을 하는 것이다. 즉 파라미터를 줄여 과적합을 방지할 수 있다. 이렇게 해도 학습 정확도는 영향을 받지 않는데, 그 이유는 채널이 여러 개라면 2 차원적으로 layer 들이 연결되어 있을 뿐만 아니라 채널끼리도 연결되어 있다. 이 말은 Dropout 으로 몇가지의 뉴런을 임의로

삭제하더라도 채널 간의 Shared Weight 에 의해 정확도는 거의 영향을 받지 않기 때문이다.

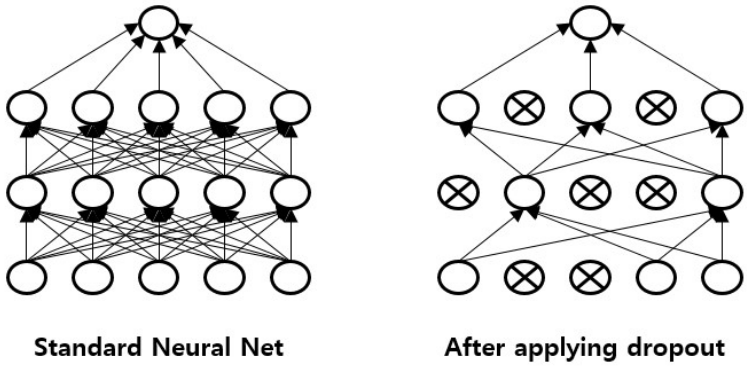


그림 9 : Backpropagation 기본 원리

3.3 CNN (Convolutional Neural Network)

기존 Neural Network(DNN)나 Machine Learning 에서는 이미지 픽셀을 Flatten 해서 1 차원으로 쭉 늘려서 사용해왔는데, 이렇게 되면 이미지 데이터에선 가까운 위치에 있지만 이러한 정보를 반영하지 못하게 되므로 정보 부족으로 인해 비효율적으로 학습을 하게 되고 이에 따라 정확도를 높이는데 한계가 생기게 된다. 하지만 CNN 은 이미지 처리에서 이미지의 공간 정보를 유지한 상태로 학습이 가능한 모델이다.
CNN 은 기본적으로 Region Feature 를 뽑아내는 Convolution Layer, Feature Dimension 을 줄이기 위한 Pooling Layer, 그리고 최종적인 분류를 위한 Fully-Connected-Layer(일반적인 MLP 구조)로 이루어져 있다.

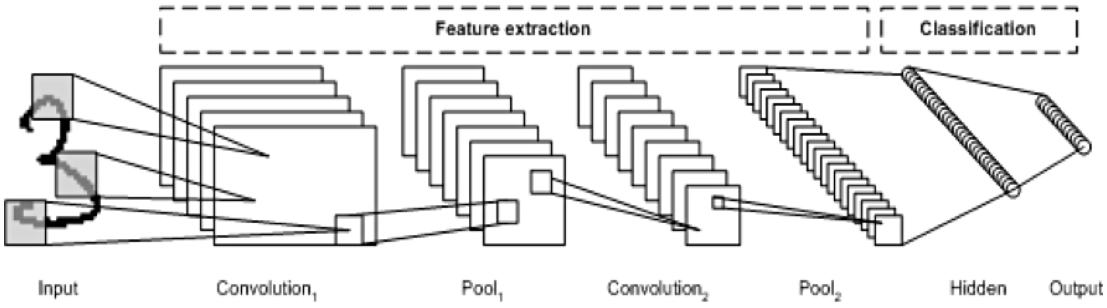


그림 10 : CNN 구조

3.3.1 Convolution Layer

Convolution Layer 는 Receptive Field 를 정의해 입력 층의 이미지의 Feature 를 추출하는 역할을 담당한다. Receptive Field 를 Filter 혹은 Kernel 이라고도 하며, 이미지 Input 과의 Convolution 연산을 통해 Feature Map 을 만들어낸다. 아래

그림은 채널이 1 개인 입력 데이터를 3*3 크기의 Filter 로 Convolution 연산하는 예시이다.

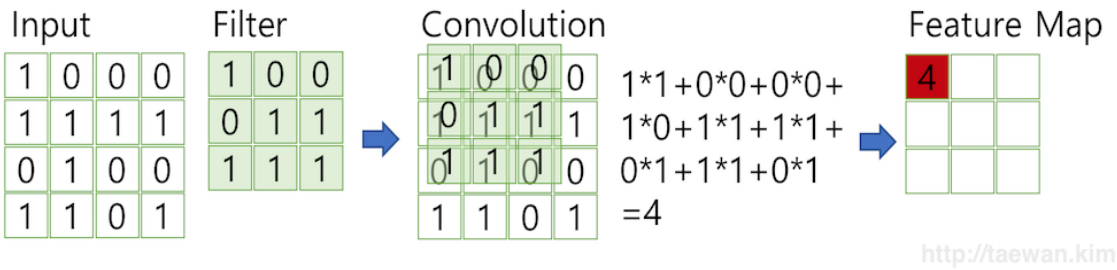


그림 11 : Convolution 연산

Filter 는 Input Data 를 지정한 간격으로 순회를 하면서 Convolution 연산을 실시한다. 이 때 Filter 가 이동하는 칸의 수를 Stride 라고 하며 아래 그림은 Stride = 1 로 Filter 를 Input Data 에 순회하는 과정이다.

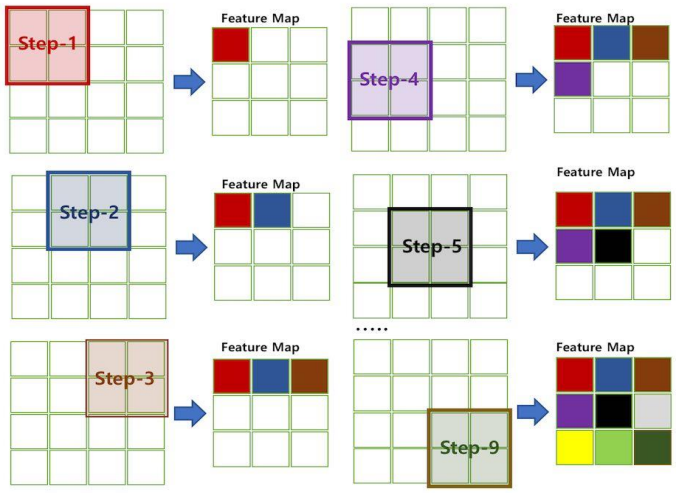


그림 12 : Convolution with Stride = 1

3.3.2 Padding

일반적인 Convolution 을 적용하면 다음 Image 또는 Feature 의 크기가 줄어들고, 가장자리에 있는 픽셀 값은 안쪽에 있는 픽셀 값보다 적게 Convolution 이 적용되는 단점이 있다. 모든 값에 Convolution 을 적용하기 위해 이미지의 외각에 지적된 픽셀만큼 특정 값으로 채우게 되는데, 이 기법을 Padding 이라 하고, 특정 값은 보통 0 을 많이 사용한다.

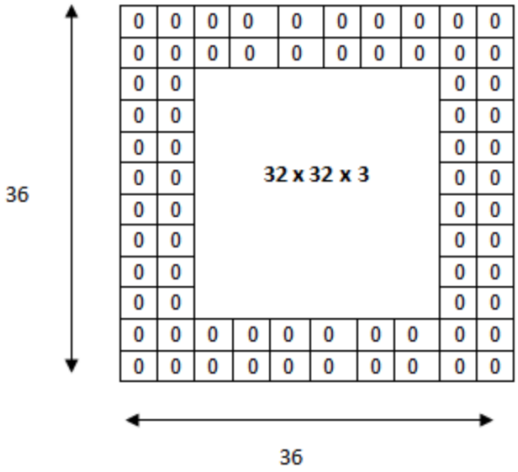


그림 13 : Padding

3.3.3 Pooling Layer

Pooling Layer 는 Convolution Layer 의 Output Data 를 Input 으로 받아 Output Data(Feature Map)의 크기를 줄이거나, 특정 데이터를 강조하는 용도로 사용한다. 다른 말로, CNN 의 Training 속도를 향상시키기 위해 Feature 의 Dimension 을 줄이는 개념이라고 볼 수 있다. 특정 영역 중 가장 큰 값을 뽑는 것을 Max Pooling, 특정 영역의 평균값을 뽑는 것을 Average Pooling 이라고 한다. 아래 그림은 2 * 2 Stride Max Pooling 과 Average Pooling 을 적용한 예이며, 2 * 2 의 사각형이 2 칸(Stride =2)씩 이동하며 값을 추출하는 과정이다.

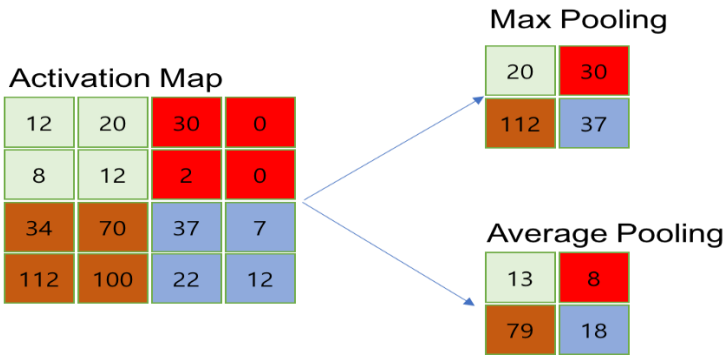


그림 14 : Pooling Example

3.3.4 Fully Connected Layer

Fully-Connected Layer 는 일반적인 MLP 구조와 동일하다. 이전 Pooling Layer 에서 나온 Feature 를 Flatten 시켜 MLP 의 Input 으로 놓고 학습을 진행한다.

3.3.5 Project 에서 사용하는 CNN 구조

이번 프로젝트에서 사용하는 CNN 의 구조는 아래 표와 그림으로 표현되어 있다. 이 자료는 논문 Fruit recognition from images using deep learning 에서 가져왔으며, 기본적인 구조는 아래의 구조를 사용하고, 실험을 진행할 때 이 구조를 기본으로 다양하게 변화시켜서 정확도를 높일 생각이다. 아래 구조와 실제가 다른 점은 Input Channel 의 수가 3 개라는 점이다.

표 1 : Project 에서 사용할 CNN 구조

Layer type	Dimensions	Output
Convolutional	5 x 5 x 4	16
Max pooling	2 x 2 — Stride: 2	-
Convolutional	5 x 5 x 16	32
Max pooling	2 x 2 — Stride: 2	-
Convolutional	5 x 5 x 32	64
Max pooling	2 x 2 — Stride: 2	-
Convolutional	5 x 5 x 64	128
Max pooling	2 x 2 — Stride: 2	-
Fully connected	5 x 5 x 128	1024
Fully connected	1024	256
Softmax	256	60

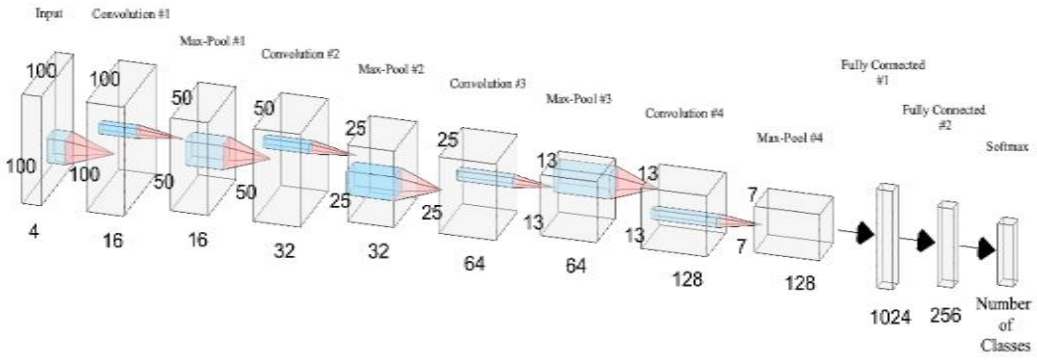


그림 15 : Model Architecture: CNN

4 데이터에 대한 설명

데이터셋의 이름은 Fruits365 이다. 총 90483 개의 이미지로 이루어져 있고, Training Set 이 67692 개의 이미지로 약 75%를 차지하고, Test Set 이 22688 개의 이미지로 약 25%를

포함한다. 실제로는 이 두 집합 말고 Multiple fruits 라는 여러 개의 과일이 같이 있는 사진이 있는데 훈련이나 테스트시 사용하지 않으므로 제외하면 총 90380 개의 이미지를 사용한다. SVM 을 실험할 때에는 직접 데이터 폴더에 접근하여 이미지 데이터를 numpy array 로 형식을 바꾸어 .npy 의 파일로 저장을 하였고, DNN 과 CNN 실험을 할 때에는 PyTorch 의 ImageFolder 라는 method 를 사용하여 데이터셋을 Load 하였다.

4.1 Input Feature

이 데이터셋의 이미지 종류는 총 131 개, 즉 131 개의 클래스를 가지고 있으며, 이미지의 크기는 전부 100 * 100 pixel 의 크기로 이루어져 있다. 그리고 RGB 값을 가지기 때문에 Input Feature 은 100 * 100 * 3 의 크기가 된다

4.2 Target Output

데이터셋에 총 존재하는 Class 의 개수는 131 개이며, 존재하는 클래스의 종류는 Apples (different varieties: Crimson Snow, Golden, Golden-Red, Granny Smith, Pink Lady, Red, Red Delicious), Apricot, Avocado, Avocado ripe, Banana (Yellow, Red, Lady Finger), Beetroot Red, Blueberry, Cactus fruit, Cantaloupe (2 varieties), Carambula, Cauliflower, Cherry (different varieties, Rainier), Cherry Wax (Yellow, Red, Black), Chestnut, Clementine, Cocos, Corn (with husk), Cucumber (ripened), Dates, Eggplant, Fig, Ginger Root, Granadilla, Grape (Blue, Pink, White (different varieties)), Grapefruit (Pink, White), Guava, Hazelnut, Huckleberry, Kiwi, Kaki, Kohlrabi, Kumsquats, Lemon (normal, Meyer), Lime, Lychee, Mandarine, Mango (Green, Red), Mangostan, Maracuja, Melon Piel de Sapo, Mulberry, Nectarine (Regular, Flat), Nut (Forest, Pecan), Onion (Red, White), Orange, Papaya, Passion fruit, Peach (different varieties), Pepino, Pear (different varieties, Abate, Forelle, Kaiser, Monster, Red, Stone, Williams), Pepper (Red, Green, Orange, Yellow), Physalis (normal, with Husk), Pineapple (normal, Mini), Pitahaya Red, Plum (different varieties), Pomegranate, Pomelo Sweetie, Potato (Red, Sweet, White), Quince, Rambutan, Raspberry, Redcurrant, Salak, Strawberry (normal, Wedge), Tamarillo, Tangelo, Tomato (different varieties, Maroon, Cherry Red, Yellow, not ripened, Heart), Walnut, Watermelon. 가 있다.

그림 16 : Data1 (Class: Banana)

그림 17 : Data2 (Class: Apple Red1)

5 코드 설명

5.1 SVM

```
print('--------------------Loading Dataset--------------------')

base_dir = '/Users/seni/Desktop/2021-01/AI/PJ'
img_dir_tr = '/Users/seni/Desktop/2021-01/AI/PJ/fruits-360/Training'
img_dir_ts = '/Users/seni/Desktop/2021-01/AI/PJ/fruits-360/Test'
MODEL_SAVE_FOLDER_PATH = './model/'

'''
if not os.path.exists(MODEL_SAVE_FOLDER_PATH):
    os.mkdir(MODEL_SAVE_FOLDER_PATH)

model_path = MODEL_SAVE_FOLDER_PATH + 'fruit-360-' + '{epoch:02d}-{val_loss:.4f}.hdf5'

cb_checkpoint = ModelCheckpoint(filepath=model_path, monitor='val_loss',
                                verbose=1, save_best_only=True)

cb_early_stopping = EarlyStopping(monitor='val_loss', patience=10)
'''
```

데이터를 불러오는 경로를 설정하고, 모델 학습 과정에서 체크포인트와 모델 저장 경로를 설정한다. 실제로 모델은 './model/' 경로에 설정한 이름 형식에 맞춰 저장된다. 실험 과정에서는 따로 저장할 필요가 없어 주석처리 해둔 상황이다.

```
'''
Set HyperParameter
'''
BATCH_SIZE = 32
EPOCHS = 10
val_spllit = 0.0
```

실험 과정에서 주로 바꾸는 파라미터들을 따로 설정하도록 하였다.

```
'''
# Get Training Data
path = img_dir_tr
train_x, train_y = imgToArray(path)

print("Start Saving Train Data")
np.save('train_X.npy', train_x)
np.save('train_y.npy', train_y)
print("Save Train Data to .npy")

path = img_dir_ts
test_x, test_y = imgToArray(path)

print("Start Saving Test Data")
np.save('test_X.npy', test_x)
np.save('test_y.npy', test_y)
print("Save Test Data to .npy")
'''


train_x = np.load('train_X.npy')
train_y = np.load('train_y.npy')
test_x = np.load('test_X.npy')
test_y = np.load('test_y.npy')
```

학습에 사용되는 이미지들을 imgToArray() 메소드를 사용하여 불러온 뒤 넘파이 형태로 바꿔 디렉토리에 저장한다. 매 실험 마다 똑같은 데이터를 또 불러올 필요가 없기 때문에 이후에는 주석처리 후 load() 메소드를 통해 불러온다.

```
model = keras.Sequential(
    [
        keras.Input(shape=(30000,)),
        RandomFourierFeatures(
            output_dim=4096, scale=10.0, kernel_initializer="gaussian"
        ),
        layers.Dense(units=131),
    ]
)

model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-2),
    loss=keras.losses.hinge,
    metrics=[keras.metrics.CategoricalAccuracy(name="acc")],
)
```

SVM 모델의 기본 아키텍처는 keras 의 "SVM을 이용한 MNIST 분류 모델"을 참고하였다. Shape은 100 * 100 픽셀의 이미지에 RGB 값 3이 들어가있는 데이터를 flatten 처리하여 (30000, 1)로 구성하였고, RandomFourierFeatures 에 값들은 앞서 참고한 document 의 sample을 유지하였다. Dense()는 나올 수 있는 클래스의 수에 맞춰 131로 설정하였다.

컴파일 과정에서 optimizer는 Adam을 사용했고 Learning_rate는 0.001에서 유동적으로 조정하였다. SVM 모델에서 사용하는 loss 방식에 맞춰 losses.hinge로 설정하였다.

```
# Preprocess the data by flattening & scaling it
train_x = train_x.reshape(-1, 30000).astype("float32") / 255
test_x = test_x.reshape(-1, 30000).astype("float32") / 255

# Categorical (one hot) encoding of the labels
train_y = keras.utils.to_categorical test_y: None
test_y = keras.utils.to_categorical(test_y)
```

전처리 과정에서 flatten 처리를 위해 기존의 데이터를 (30000,)으로 reshape() 처리했고, 정규화를 위해 255로 나눠준 상황이다. 대응하는 y 데이터에 대해서는 one hot encoding 처리했다.

```
print('--------------------Training--------------------')

#history = model.fit(train_x, train_y, epochs=EPOCHS, batch_size=BATCH_SIZE, validation_split=None, callbacks=[cb_checkpoint, cb_early_stopping])
history = model.fit(train_x, train_y, epochs=EPOCHS, batch_size=BATCH_SIZE, validation_split=None)

print('\nAccuracy: {:.4f}'.format(model.evaluate(test_x, test_y)[1]))

print('--------------------Plotting--------------------')


y_loss = history.history['loss']
x_len = np.arange(len(y_loss))
plt.plot(x_len, y_loss, marker='.', c='blue', label="Train-set Loss")

plt.legend(loc='upper right')
plt.grid()
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()

y_acc = history.history['acc']
x_len = np.arange(len(y_acc))
plt.plot(x_len, y_acc, marker='.', c='red', label="Train-set Accuracy")

plt.legend(loc='upper right')
plt.grid()
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```

설정 값으로 학습 한 후에는 테스트 정확도를 출력하고, matplotlib 을 이용하여 정확도와

loss를 시각화 할 수 있도록 하였다.

5.2 DNN

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3 * 100 * 100, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 256)
        self.fc4 = nn.Linear(256, 131)
        self.dropout = 0

    def forward(self, x):
        x = x.view(-1, 3 * 100 * 100)
        x = F.relu(self.fc1(x))
        if self.dropout > 0:
            x = F.dropout(x, training=self.training, p=self.dropout)
        x = F.relu(self.fc2(x))
        if self.dropout > 0:
            x = F.dropout(x, training=self.training, p=self.dropout)
        x = F.relu(self.fc3(x))
        if self.dropout > 0:
            x = F.dropout(x, training=self.training, p=self.dropout)
        x = self.fc4(x)
        x = F.log_softmax(x, dim=1)
```

```
        return x
```

이 네트워크는 4 개의 Fully Connected Layer 로 구성되어 있으며 fc1, fc2, fc3 는 ReLU activation function 까지 포함하는 형태이다. 마지막 fc4 layer 가 131 개의 클래스 중 가장 높은 값인 하나의 값으로 출력 한다. Fc1, fc2, fc3 의 역할은 입력 데이터를 받아 레이어마다 weight 를 행렬곱하고 bias 를 더하는 일반적인 Dense layer 의 선형 결합 연산을 수행하기 때문에 역전파 과정에서 발생할 수 있는 vanishing gradient 문제를 해결하기 위해 다음과 같은 ReLu 활성화 함수를 사용하게 된다.
Forward 함수는 처음 입력되는 이미지(x)는 색, 높이 너비 3*100*100 로 받고 view()를 통해 1 차원으로 변경해주게 된다. 이는 fc1 을 거치기 위함이고, batch size 를 32 로 설정하였기 때문에 실제로 파이썬 디버거를 사용하여 데이터의 크기를 체크해보면 [32, 3, 100, 100]으로 정의 되어있다.
Dropout 은 F.dropout 을 이용하여 진행하였다.
이후 마지막으로 F.log_softmax 를 통해 확률을 구한 후 최종 값을 return 한다.

5.3 CNN

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(
            in_channels=3,
            out_channels=16,
            kernel_size=5,
            padding=1)
```

CNN class 의 __init__ 부분이다.
우선 Pytorch Module 내에 딥러닝 모델 관련 기본 함수를 포함하고 있는 nn.Module 클래스를 상속받는 클래스를 정의한다. 그리고 nn.Conv2d Method 를 통해 Convolution Layer 의 Filter 를 정의한다.
초기 Input 의 채널 수는 Red, Green, Blue 총 3 개이므로 첫번째 layer 의 in_channels 값은 3 으로 설정하였다.
out_channels 의 경우는 Convolution 연산을 진행하는 Filter 의 개수를 설정한다. 여기서 설정한 값만큼 output 의 depth 가 설정이 된다.
Kernel_size 의 경우는 Filter 의 크기를 설정하는 부분이다. 이 경우에 5 로 설정하였으므로 5 * 5 크기의 Filter 를 사용하게 된다.
Padding 의 경우는 이미지의 구석 부분이 상대적으로 덜 연산하는 것을 방지하기 위해 설정하는 값인데, 1 로 설정하였을 때에는 이미지의 상하좌우 1 층으로 0 을 채워 준다는 것을 의미한다.

```
self.conv2 = nn.Conv2d(
    in_channels=16,
    out_channels=32,
    kernel_size=5,
    padding=1)
self.conv3 = nn.Conv2d(
    in_channels=32,
    out_channels=64,
    kernel_size=5,
    padding=1)
```

두번째, 세번째 Convolution Layer 을 보면, 두번째 Convolution 의 in_channels 의 수는 16 이고 이는 첫번째 Convolution Layer 의 out_channels 수와 같다. 세번째 Convolution Layer 의 경우에도 마찬가지이다.

```
self.pool = nn.MaxPool2d(
    kernel_size=2,
    stride=2)
```

Convolution Layer 을 통해 Feature Map 이 생겼을 때, 그 Feature Map 을 전부 다 이용하는 것이 아니라 Pooling Layer 을 통해 부분적으로 이용이 된다. 이 때 nn.MaxPool2d 를 이용하여 Pooling Layer 을 생성해 주었다.
Kernel_size 는, 이 경우에서는 2 * 2 크기의 Filter 가 돌아다니면서 가장 큰 Feature Map 값을 추출한다는 의미이다.
Stride 는, 2 * 2 크기의 Filter 가 움직일 때 Feature Map 에서 2 칸 단위로 움직인다는 것을 의미한다.

```
self.fc1 = nn.Linear(6400, 1024)
self.fc2 = nn.Linear(1024, 256)
self.fc3 = nn.Linear(256, 131)
```

Convolution 연산을 통해 Feature 들을 추출하고 난 이후에는, 그 추출한 Feature 들을 MLP 모형의 Input 으로 이용하여 여러 층의 Neural Network 를 통과시켜 Classification 하는 과정을 거쳐야 한다.
첫번째 Fully Connected Layer 에서 Input 의 개수가 6400 이라고 나와 있는데, 이에 대해서는 이후 Forward 함수 부분에서 더 자세히 설명을 할 예정이다.
위의 코드는 Hidden Layer 의 수가 2 개이며, 각 Layer 의 노드 수는 1024, 256 인 Fully-Connected Model 이다.

```
def forward(self, x):
    x = self.conv1(x)
    x = F.relu(x)
    x = self.pool(x)
    x = self.conv2(x)
    x = F.relu(x)
    x = self.pool(x)
    x = self.conv3(x)
    x = F.relu(x)
    x = self.pool(x)
```

CNN class 의 forward propagation 중 Convolution 연산 부분이다.
Input x 가 들어오게 되면, Convolution 연산을 통해 Feature Map 을 생성하고, Non-linear Function 인 ReLU()를 적용한다. 그리고 MaxPooling Layer 를 진행하여 Feature 수를 줄이면 첫번째 Convolution 연산이 끝나게 되고, 이것을 두번째 세번째 Convolution Layer 에도 똑같이 적용이 된다.
그 이후를 보기 전에 위의 Convolution 연산이 진행되는 과정에서 Input 의 Feature 변화를 눈 여겨 볼 필요가 있다.
우선 Input 으로는 100 * 100 크기에 RGB 값이 포함이 되어 있으므로 총 100 * 100 * 3 개의 Feature 이 Input 으로 들어오게 된다. 이 Input 이 Filter 의 크기가 5 * 5, Stride = 1, Padding = 1 인 Convolution 을 지나게 되면,

$$Output\ height = \frac{(Input\ height + 2 * paddingsize \quad - filter\ height)}{Stride} + 1$$

$$Output\ width = \frac{(Input\ width + 2 * paddingsize \quad - filter\ width)}{Stride} + 1$$

에 의해, 한 필터당 (100 + 2*1 – 5) + 1 * (100 + 2*1 – 5) +1, 즉 98 * 98 의 크기가 되고 이 때 kernel 수가 16 개 이므로 98 * 98 * 16 의 크기가 된다.
그리고 이 값이 MaxPooling 을 지나게 되면,

$$Output\ hegiht = \frac{Input\ height\ Size}{PoolingSize}$$

$$Output\ width = \frac{Input\ width\ Size}{PoolingSize}$$

에 의해 (98/2) * (98/2) * 16, 즉 49 * 49 * 16 이 된다.

이후 두번째 Convolution Layer 을 지나게 되면, (49 + 2*1 -5) + 1 * (49 + 2*1 -5) + 1 * 32, 즉 47 * 47 * 32 가 되고, MaxPooling Layer 을 지나게 되면, 23 * 23 * 32 가 된다.(나누었을 때 몫을 값으로 한다.)
마지막 세번째 Convolution Layer 을 지나게 되면, (23 + 2*1 -5) + 1 * (23 + 2*1 -5) + 1 * 64, 즉 21 * 21 * 64 의 크기가 되고 MaxPooling Layer 을 지나게 되면 10 * 10 * 64 의 크기가 된다.
따라서 Convolution 연산을 모두 마쳤을 때의 Feature 크기는 10 * 10 * 64 가 된다.

```
x = x.view(-1, 6400)
x = self.fc1(x)
x = F.relu(x)
if self.dropout > 0:
    x = F.dropout(x, training=self.training, p=self.dropout)
x = self.fc2(x)
x = F.relu(x)
if self.dropout > 0:
    x = F.dropout(x, training=self.training, p=self.dropout)
x = self.fc3(x)
x = F.log_softmax(x, dim=1)
return x
```

Convolution 연산 이후 Fully Connected Layer 의 Forward Propagation 부분이다.
Convolution 연산의 결과로 10 * 10 * 64 의 Feature 이 되었으므로, Fully Connected Layer 에 적용하기 위해서는 view 함수를 통해 1 차원으로 데이터를 펴 줘야 한다.
이후에 이 값을 각 Fully Connected Layer 에 적용하고, dropout 값이 0 보다 크면 dropout 도 적용시키고 최종적으로 SoftMax 함수를 통해 확률 값을 계산한다.
log_softmax 함수를 사용하면 Back Propagation 알고리즘을 이용해 학습을 진행할 때 Gradient 값을 좀더 원활하게 계산할 수 있다고 하여 log_softmax(SoftMax 함수에 log 취한 것) 함수를 사용하였다

5.4 Train

DNN 과 CNN 에서 훈련을 할 때 사용하는 함수이다.

```
def train(model, train_loader, optimizer, log_interval, criterion, epoch):
    model.train()
    for batch_idx, (image, label) in enumerate(train_loader):
        image = image.to(DEVICE)
        label = label.to(DEVICE)
```

```
        optimizer.zero_grad()
        output = model(image)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        if batch_idx == len(train_loader) - 1:
            last_loss = loss

        if batch_idx % log_interval == 0:
            print("Train Epoch: {} [{}/{} ({:.0f}%)]₩tTrain Loss: {:.6f}".format(
                epoch, batch_idx * len(image),
                len(train_loader.dataset), 100. * batch_idx / len(train_loader),
                loss.item()))

    return last_loss.cpu().detach().numpy()
```

model.train()을 통해 기존에 정의하였던 Neural Network 모델을 Train 상태로 지정한다.
train_loader 는 image 와 label 데이터가 Mini-Batch 단위로 묶여 저장되어 있어 Mini-Batch 단위로 지정된 데이터를 순서대로 이용해 모델을 학습한다.
그리고 image 와 label 을 사전에 정의하였던 DEVICE(CPU or GPU)에 할당 시킨다.
이후, 과거에 이용한 Mini-Batch 내에 있는 Image 데이터와 label 을 바탕으로 계산된 Loss 의 Gradient 값이 Optimizer 에 할당되어 있으므로 zero_grad()를 통해 Gradient 를 초기화 한다.
이후 데이터를 Model 에 적용시켜 Output 을 계산하고 기존에 정의한 Criterion(여기선 nn.CrossEntropy)를 이용하여 Loss 값을 계산하고, Back Propagation 을 진행하여 계산된 Gradient 를 각 파라미터에 할당하고 optimizer.step()을 통해 파라미터를 업데이트한다.
그리고 마지막 Mini-Batch 인 경우 최종 Loss 값을 저장해 나중에 반환하고, 특정 batch 때마다 계산된 Loss 를 출력한다.

5.5 Evaluate

DNN 과 CNN 에서 epoch 마다 test_data 를 이용해 결과를 반환하는 함수이다.

```
def evaluate(model, test_loader, criterion):
    model.eval()
    test_loss = 0
    correct = 0
```

```
with torch.no_grad():
    for image, label in test_loader:
        image = image.to(DEVICE)
        label = label.to(DEVICE)
        output = model(image)
        test_loss += criterion(output, label).item()
        prediction = output.max(1, keepdim=True)[1]
        correct += prediction.eq(label.view_as(prediction)).sum().item()

test_loss /= len(test_loader.dataset)
test_accuracy = 100 * correct / len(test_loader.dataset)
return test_loss, test_accuracy
```

model.eval()을 통해 모델을 평가 상태로 지정한다.
Test data 의 Loss 를 계산하기 위해 0 으로 초기화를 하고, Accuracy 계산을 위한 올바르게 맞은 데이터의 개수를 세기 위한 correct 도 0 으로 초기화를 해준다.
그리고 평가 과정에서는 파라미터가 업데이트 되는 것을 방지하기 위해 no_grad() method 를 이용한다.
Test_loader 의 경우도 Mini-Batch 단위로 되어 있으므로, 반복문을 이용해 이미지 데이터와 레이블을 Train 때와 마찬가지로 DEVICE 에 할당하고, 모델에 적용시키고 Loss 를 계산한다. 그 다음 나온 결과(총 131 개의 Output)중 확률이 제일 높은 것을 출력하여 이를 실제 데이터와 비교해 올바르게 예측한 횟수를 저장하게 된다.
이후에 지금까지 더한 test_loss 를 Mini-Batch 의 개수로 나누어 평균을 구하고, 정확도도 계산하여 반환을 하게 된다.

6 학습 과정

6.1 SVM

Data normalization, Batch-Size 조절, Learning rate 조절, Scikit learn-SVM과의 비교를 진행하였다

6.2 DNN

Hidden Layer 수를 조절 및 비교, Batch Size 의 크기 조절, Learning rate 조절, Hidden layer Dropout 조절, 총 4 가지의 실험을 통해 비교할 예정이다.

6.3 CNN

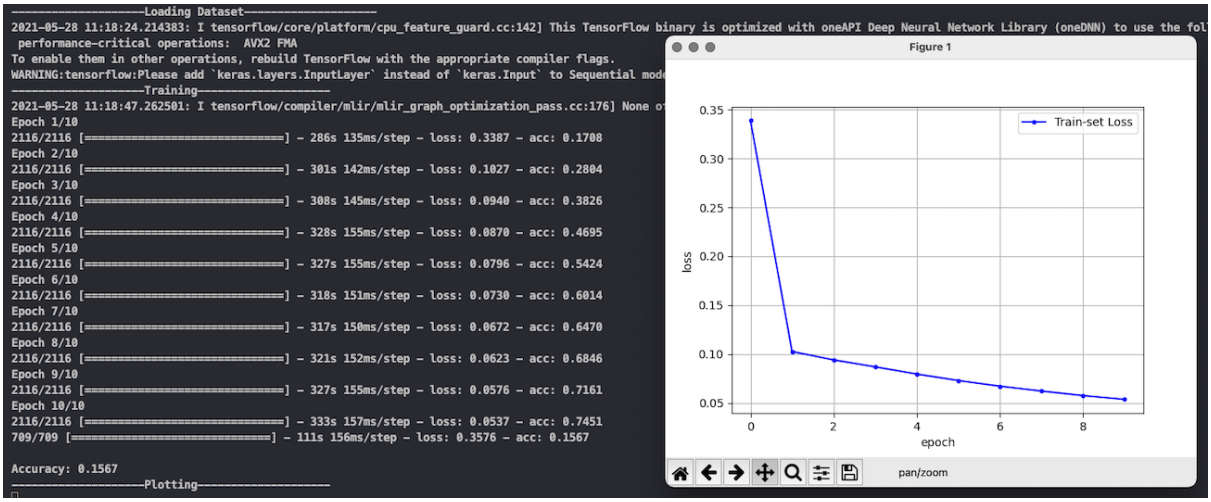
실험은 총 5 가지로 진행이 된다. 먼저 Convolution Layer 의 Channel 수를 조절하여 비교하고, 두번째로는 Convolution Layer 의 수를 조절해서 비교하고, 세번째는 Fully-Connected Layer 의 Dropout 을 조절하고, 네번째는 BATCH_SIZE 를 조절하고, 마지막으로는 Learning rate 를 조절하여 비교할 예정이다.

7 결과 및 분석

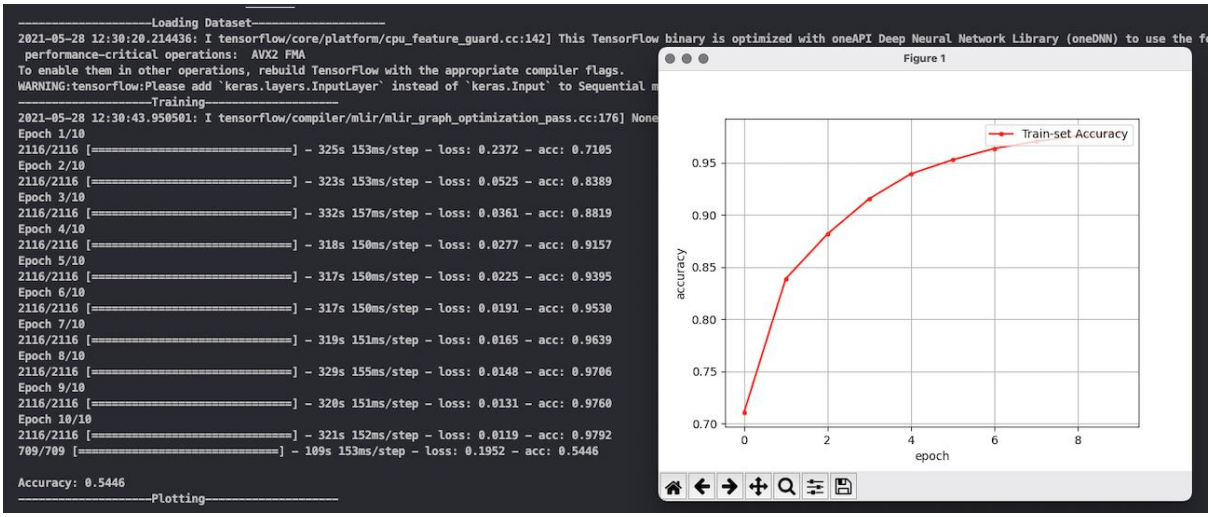
7.1 SVM

7.1.1 Data Normalization

기존의 데이터는 0-255 사이의 값들로 구성되어있어 계산의 효율을 위해 정규화가 요구된다. 정규화는 두가지 방식으로 진행하였다. 먼저 데이터들을 [0,1] 사이의 값으로 조정 후에 학습하고, 이후에는 [-1, 1] 사이의 값으로 조정 후에 학습했다. 학습 결과 [0,1] 사이의 값으로 조정한 경우에 더 높은 정확도를 보였다.



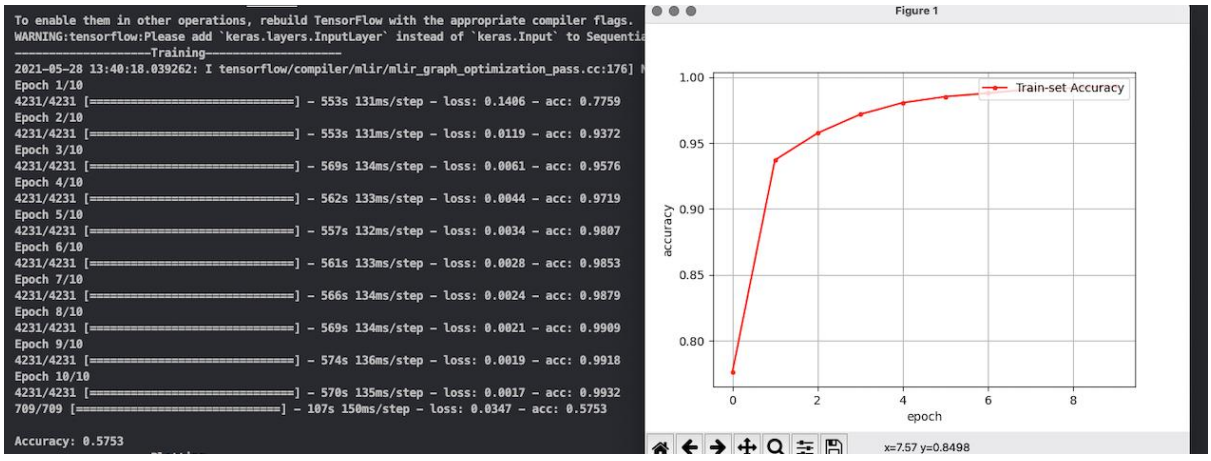
결과 1 : Data Scailing [-1,1] / (32,0.01,[-1,1]) (Acc : 15%)



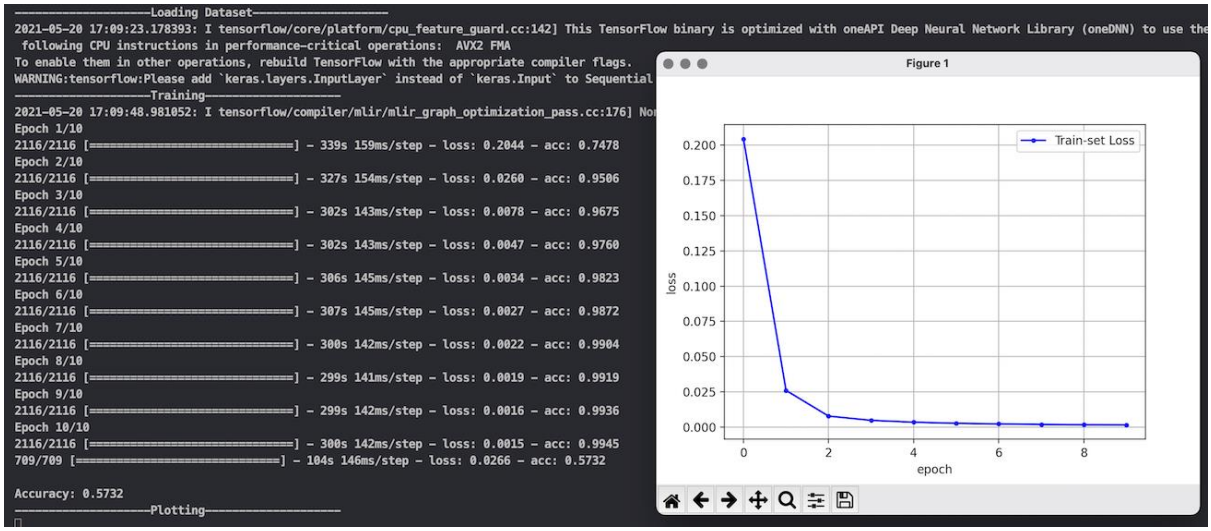
결과 2 : Data Scailing [0,1] / (32,0.01,[0,1]) (Acc : 54%)

7.1.2 Batch-Size

Batch-Size 는 한 step 당 처리하는 데이터의 양으로 32가 적절한 값인지 판단하기 위해 16 으로 학습했다. 그 결과 비슷한 수준의 정확도를 보였다. 이미 충분히 적절한 사이즈의 Batch-size 가 설정되었다고 판단할 수 있었다.



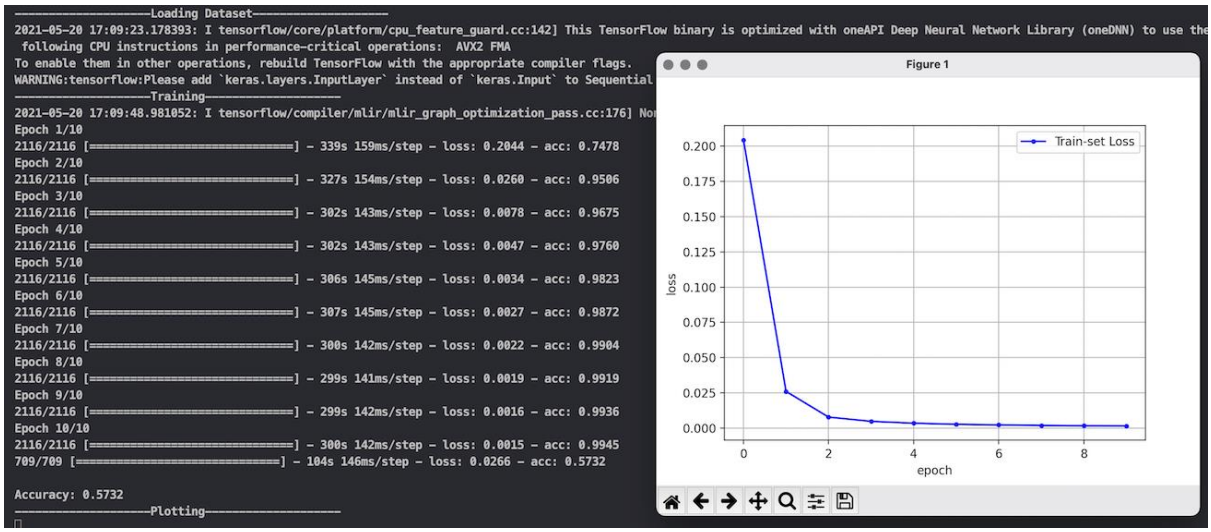
결과 1 : Batch-size 32 / (32,0.001,[0,1]) (Acc : 57%)



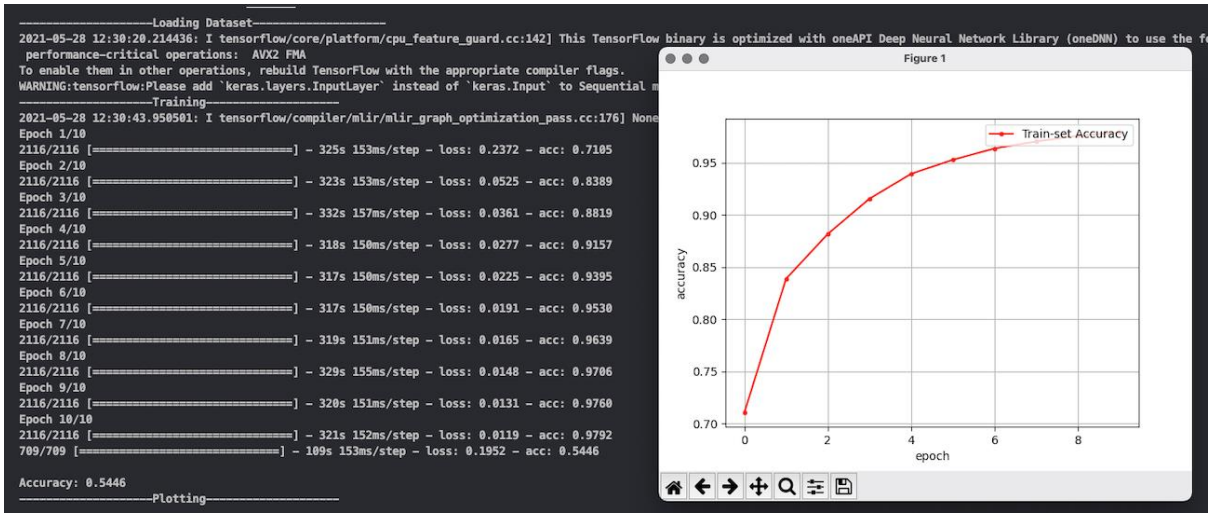
결과 2 : Batch-size 16 / (16,0.001,[0,1]) (Acc : 57%)

7.1.3 Learning rate

학습률(Learning rate)은 최소 손실 함수를 향해 이동하면서 각 step에서 그 크기를 결정하는 최적화 알고리즘의 튜닝 매개변수다. 따라서 적절한 학습률 설정을 통해 최저점에 최적의 시간 안에 도달하도록 해야한다. 기존에 0.01에서 0.001로 학습률을 수정했을 때, 정확도가3% 정도 소폭 증가한 것을 확인했다.



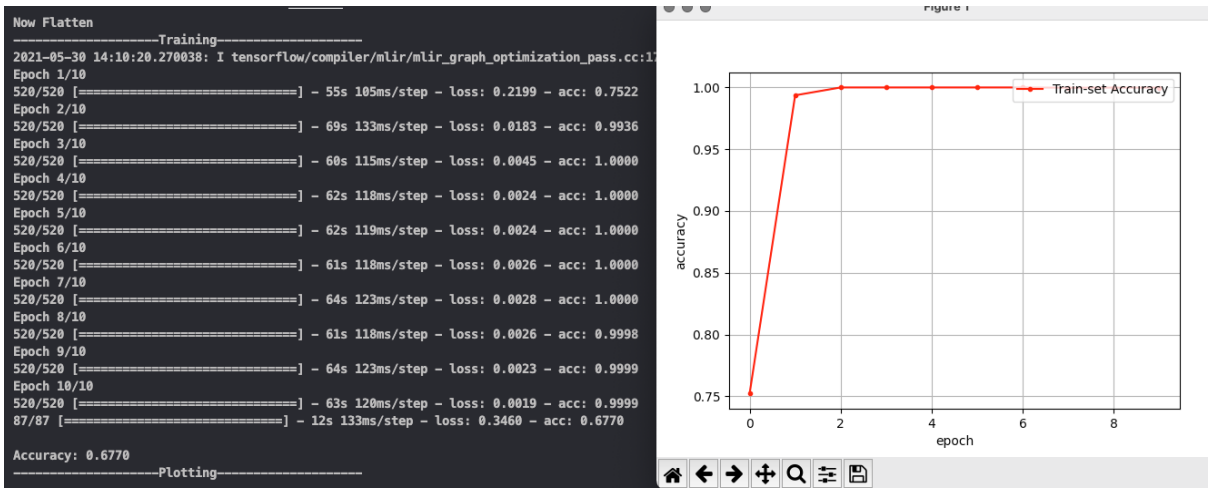
결과 1 : Learning rate 0.001 / (32,0.001,[0,1]) (Acc : 57%)



결과 2 : Learning rate 0.01 / (32,0.01,[0,1]) (Acc : 54%)

7.1.4 ScikitLearn SVM

마지막으로 Scikit-learn 에서 제공하는 SVM 라이브러리를 이용한 모델과 비교해보았다. Scikit-learn 에서 직접 제공하는 SVM 라이브러리는 GPU 를 사용할 수 없어 주어진 시간 내에 모델 학습이 불가능했다. 따라서 상위 18개의 클래스에 대한 데이터만 가지고 분류를 실험하였다. 실험 결과 18개의 클래스로 기존 Keras 코드를 이용해 학습했을때 정확도 67%로 소폭 상승했고, scikit-learn SVM 코드는 98%의 높은 정확도를 보였다.



결과 1 : SVM Model by Keras / 18 classes (Acc : 67%)

```
23   model = SVC(gamma='auto', kernel='rbf')
24
25   print("Model fit")
26   # Fit the model to training data
27   model.fit(train_x[:8316], train_y[:8316])
28   print("Model Predict")
29   # Check test set accuracy
30   accuracy = model.score(test_x[:2777], test_y[:2777])
31
32   print('Accuracy: {}'.format(accuracy))
```

```
PROBLEMS 14   OUTPUT   DEBUG CONSOLE   TERMINAL

~/Desktop/2021-01/AI/PJ
venv › /Users/seni/Desktop/2021-01/AI/PJ/venv/bin/python /Users/seni/Desktop/2021-01/AI/PJ/svm_clf.py
Now Flatten
Model fit
Model Predict
Accuracy: 0.9823550594166367
```

결과 2 : SVM by scikit-learn / 18 classes (Acc : 98%)

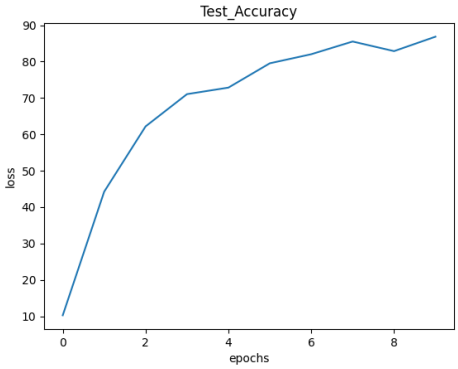
7.2 DNN

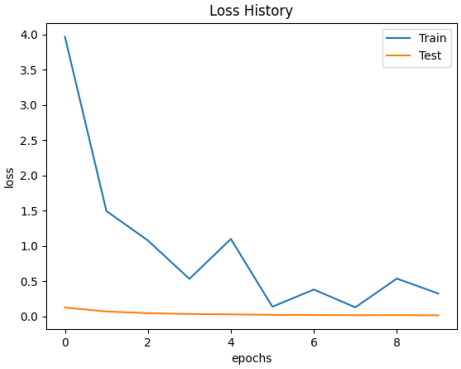
기본 파라미터 상태에서 특정한 파라미터 값들을 변화시켜 가장 좋은 결과값들이 나오는 것들을 마지막에 합하는 형식으로 진행하였다. CNN 과 마찬가지로 EPOCH 마다 Test Data 를 적용시켜 정확도가 제일 높게 나오는 값을 표시하였다.

7.2.1 Hidden Layer 조절

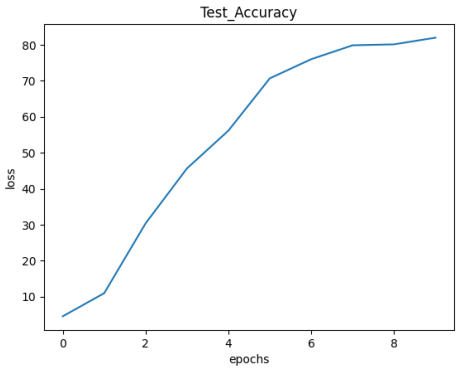
BATCH_SIZE = 32, EPOCHS = 10, Learning Rate = 0.001 Optimizer = SGD로 하고, Hidden Layer의 수만 조절을 하기로 하였다. 처음은 가장 기본으로 설정한 Hidden Layer 3개에 Node의 수가 (1024, 512, 256)인 것부터 시작하였다.

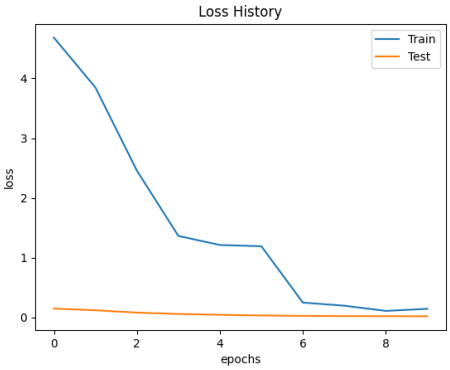
7.2.1.1 (1024, 512, 256)



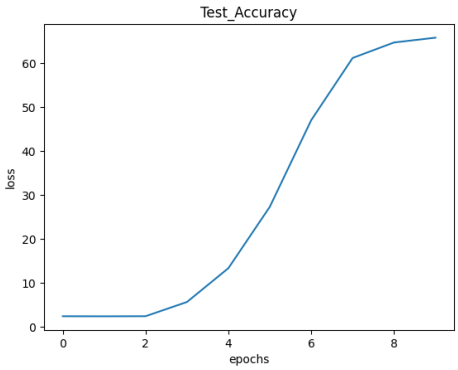


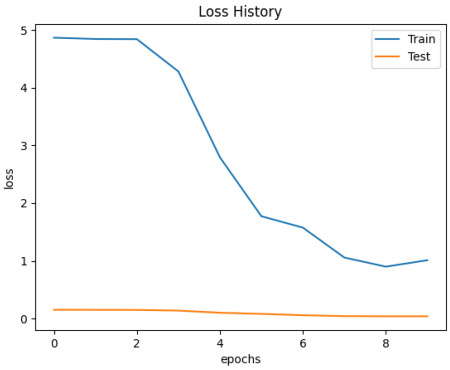
7.2.1.2 (2048, 1024, 512, 256)





7.2.1.3 (4096, 2048, 1024, 512, 256)





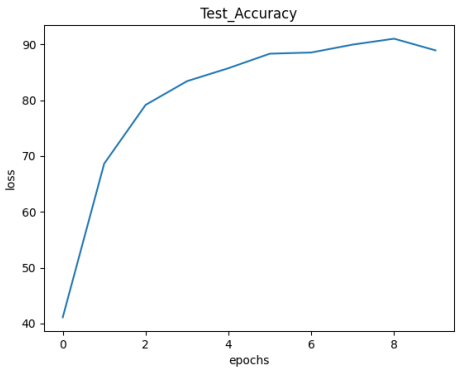
이론적으로 Hidden Layer 의 개수가 많을수록 Deep 한 신경망이 생성되어 성능이 높아지는 반면 Hidden Layer 가 너무 많으면 의사 결정을 못하게 되는 문제가 발생한다. 실험 결과 Hidden layer 의 개수는 4 개, 5 개로 넘어갈수록 의사결정을 못하는 현상을 경험하였으며, 3 개가 가장 적절한 것으로 결과가 나왔다.

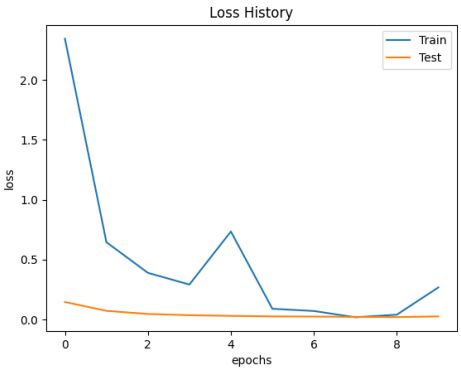
No	Configuration		Epochs	Test Loss	Test Accuracy
1	Hidden Layer	1024	10	0.0159	86.83 %
	Hidden Layer	512			
	Hidden Layer	256			
2	Hidden Layer	2048	10	0.0204	82.00 %
	Hidden Layer	1024			
	Hidden Layer	512			
	Hidden Layer	256			
3	Hidden Layer	4096	10	0.0395	65.79 %
	Hidden Layer	2048			
	Hidden Layer	1024			
	Hidden Layer	512			
	Hidden Layer	256			

7.2.2 BATCH SIZE

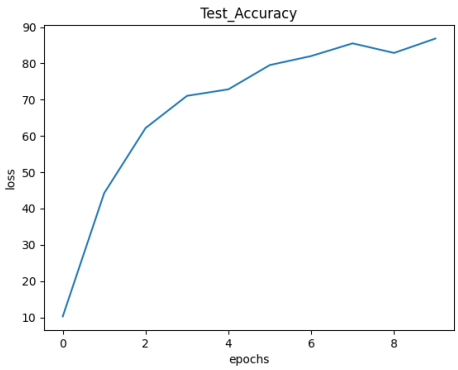
Hidden Layer 를 3 개(1024, 512, 256)으로 하고, EPOCHS = 10, Learning Rate = 0.001 Optimizer = SGD 로 하고, Batch size 만 다르게 해서 진행하였다. 처음은 16 부터 두 배씩 늘리면서 진행하였다.

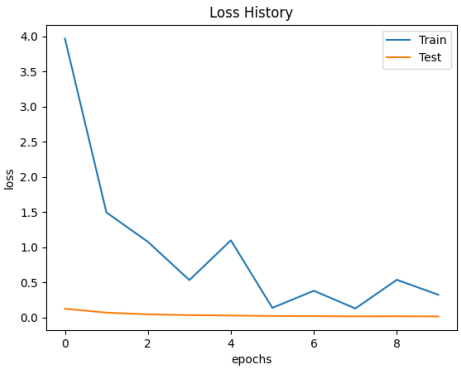
7.2.2.1 BATCH SIZE = 16



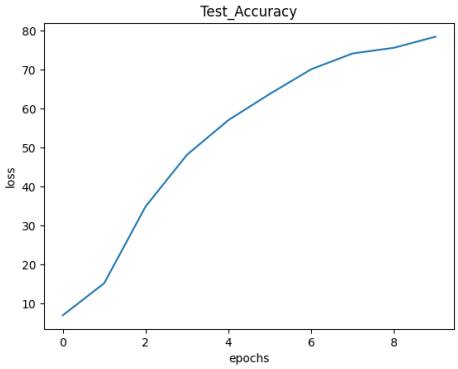


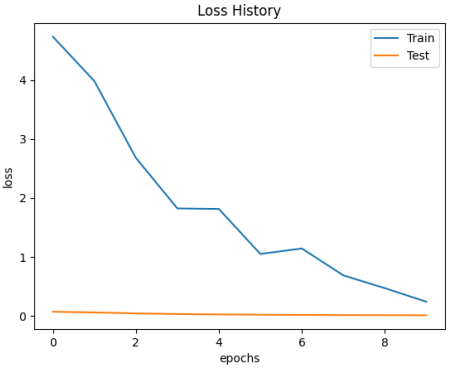
7.2.2.2 BATCH SIZE = 32



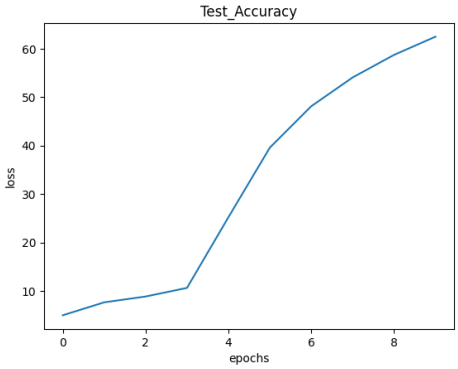


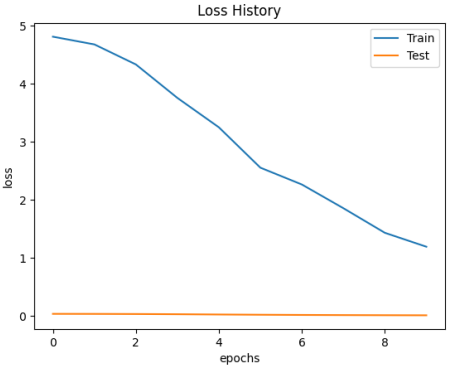
7.2.2.3 BATCH SIZE = 64





7.2.2.4 BATCH SIZE = 128





Batch size 를 어떻게 결정하느냐에 따라서 학습과정에서 차이가 발생하는데, Batch size 가 크면 경사도의 정확한 추정을 할 수 있지만 반복할 때마다 높은 계산비용이 들고, 병렬 처리의 고가용성이 발생할 수 있다. 반대로 Batch size 가 작으면 경사도의 정확한 추청을 할 수 없지만 반복하는데

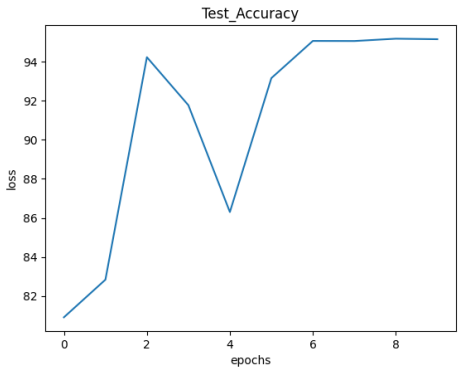
적은 계산비용이 들고 slow training 을 한다. 따라서 결과를 보면 Batch size 가 16 인 것이 가장 적절한 값으로 볼 수 있다.

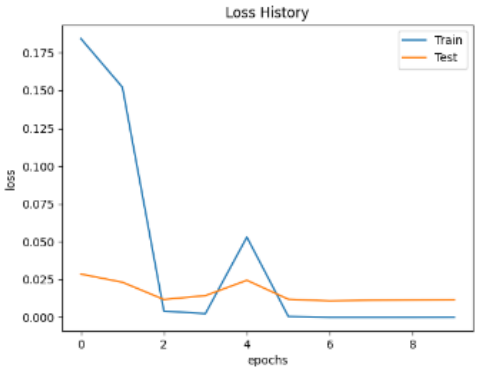
No	Batch Size	Epoch	Test Loss	Test Accuracy
1	16	9	0.0220	91.04 %
2	32	10	0.0159	86.83 %
3	64	10	0.0127	78.52 %
4	128	10	0.0129	62.52 %

7.2.3 Learning Rate

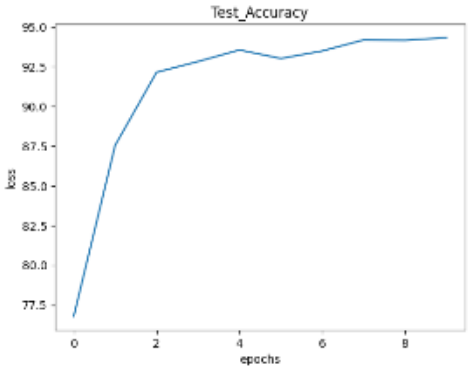
Hidden Layer 를 3 개(1024, 512, 256)으로 하고, EPOCHS = 10, Batch size = 32, Optimizer = SGD 로 하고, 처음은 0.1 부터 숫자를 줄이면서 진행하였다.

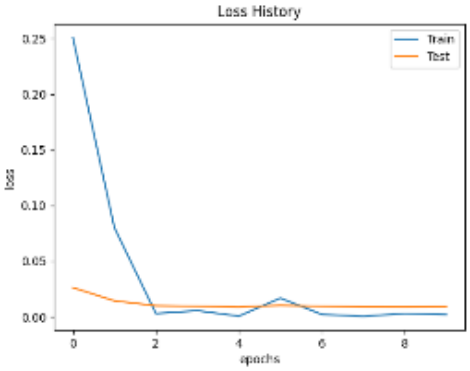
7.2.3.1 Learning Rate = 0.1



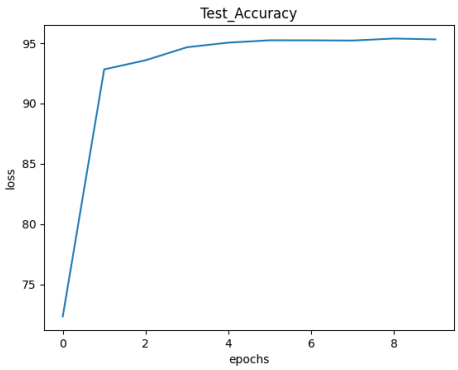


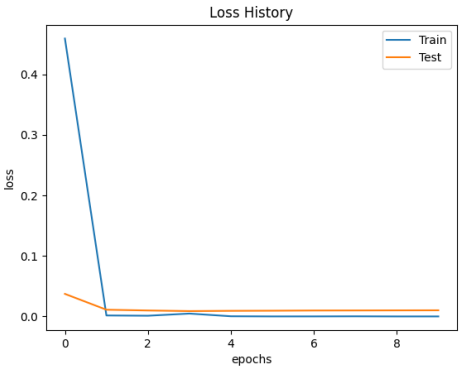
7.2.3.2 Learning Rate = 0.01



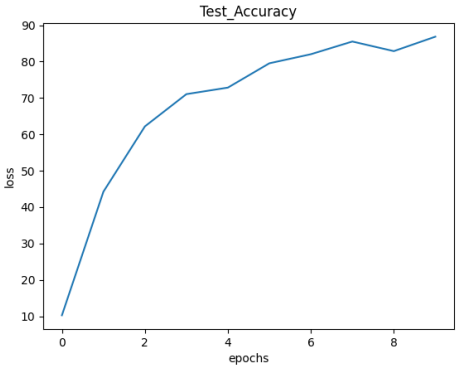


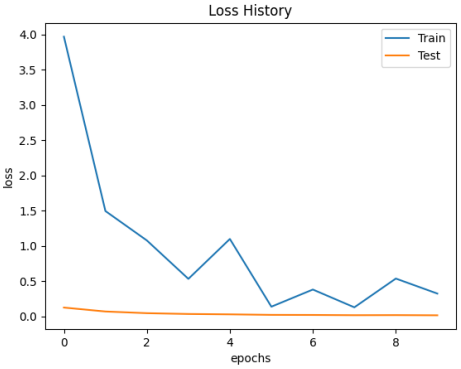
7.2.3.3 Learning Rate = 0.05



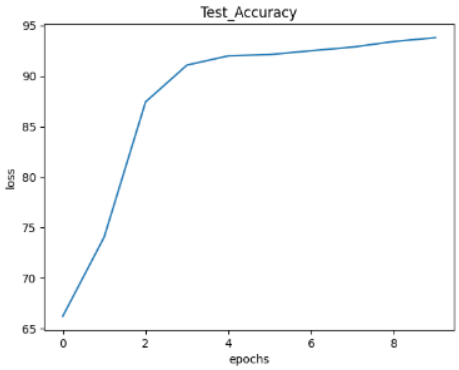


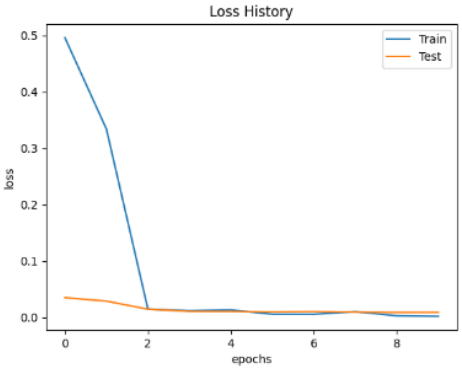
7.2.3.4 Learning Rate = 0.001



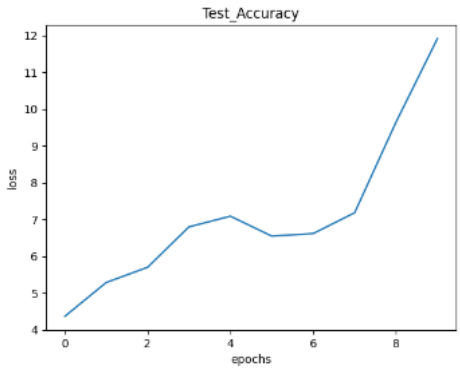


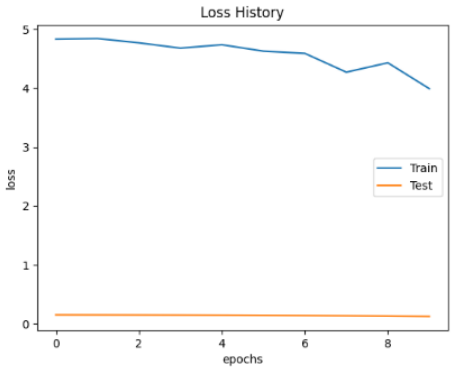
7.2.3.5 Learning Rate = 0.005



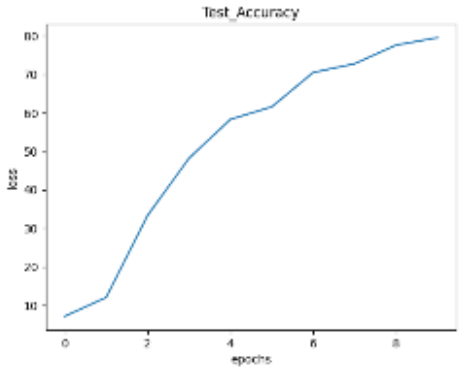


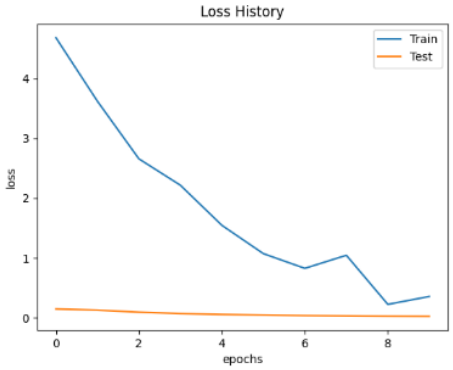
7.2.3.6 Learning Rate = 0.0001





7.2.3.7 Learning Rate = 0.0005





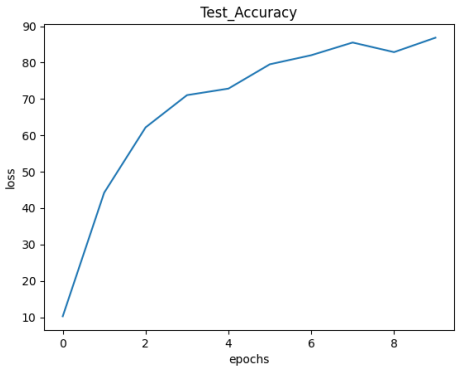
Learning Rate 는 매우 큰 값으로 잡게 될수록 그래프에서 하강하는 폭이 크다는 것을 알 수 있다. 또한 학습이 제대로 이루어지지 않을 뿐만 아니라 최저점에 도달하는 것을 넘어 그 반대 방향의 그래프에 도달할 만큼 overshooting 이 발생할 수 있다. 반대로 Learning Rate 를 작게 잡으면 최저점에 도달하는 데 소요되는 시간이 길어지고 global minimum 이 아닌 local minimum 을 그래프의 최저점으로 인식하는 문제가 발생할 수 있다. 가장 크지도 않고 작지도 않은 적절한 값은 95.39 %로 0.05 가 된다고 볼 수 있다.

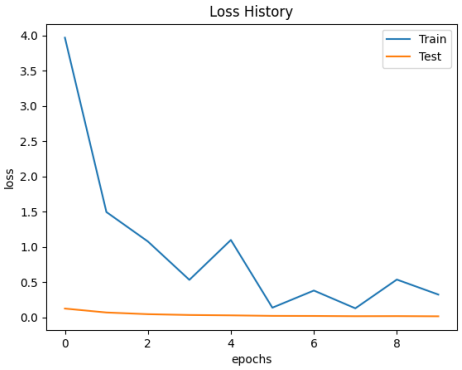
No	Learning Rate	Epochs	Test Loss	Test Accuracy
1	0.1	9	0.0115	95.18 %
2	0.05	9	0.0100	95.39 %
3	0.01	8	0.0090	94.19 %
4	0.005	10	0.0092	93.80 %
5	0.001	10	0.0159	86.83 %
6	0.0005	10	0.0247	79.51 %
7	0.0001	10	0.1248	11.91 %

7.2.4 Dropout

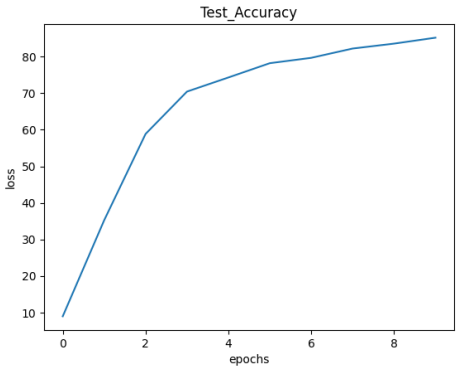
Hidden Layer를 3개(1024, 512, 256)으로 하고, EPOCHS = 10, Batch size = 32, Learning Rate = 0.001, Optimizer = SGD로 하고, Dropout 0부터 0.8까지 진행하였다.

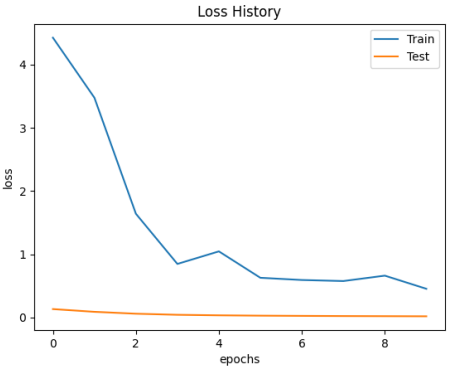
7.2.4.1 Dropout = 0



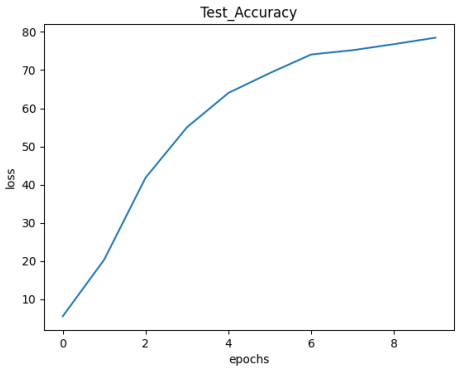


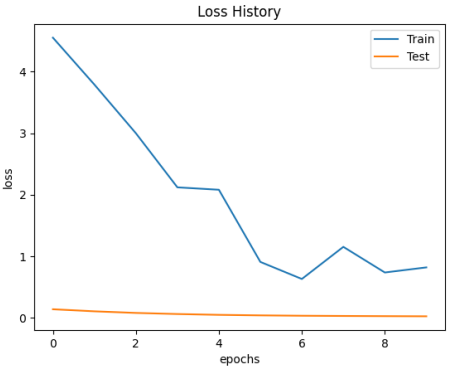
7.2.4.2 Dropout = 0.2



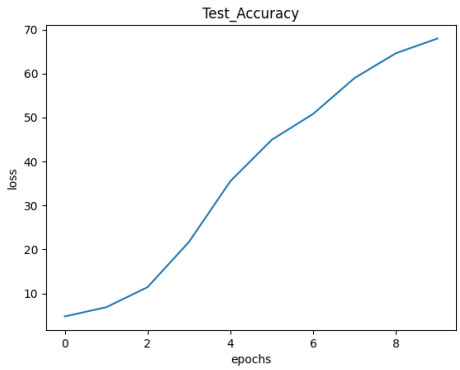


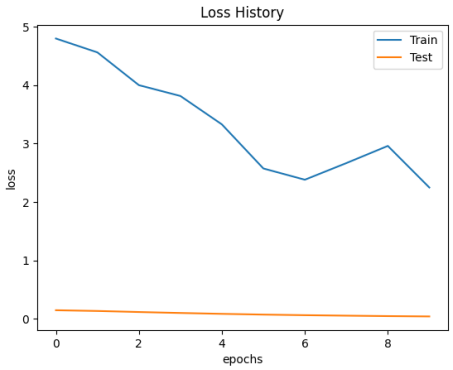
7.2.4.3 Dropout = 0.4



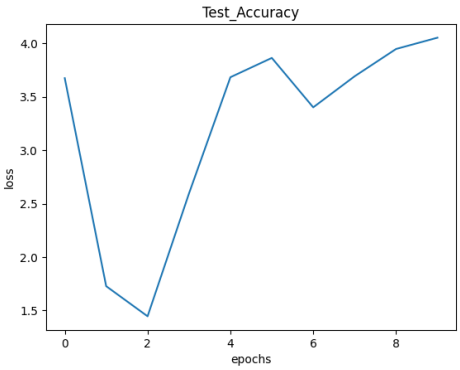


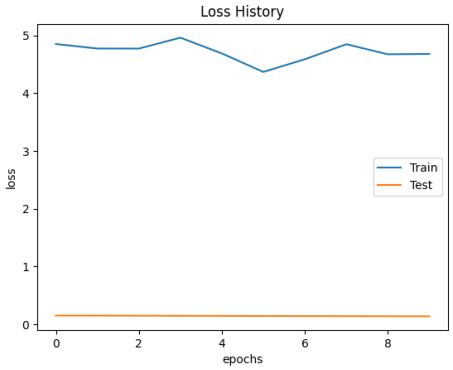
7.2.4.4 Dropout = 0.6





7.2.4.5 Dropout = 0.8





결과는 Dropout 을 사용하지 않은 것에 가장 정확도가 높게 나타났다. 이는 과적합이 이루어지지 않았다는 것이라고 볼 수 있는데 Epoch 의 개수가 적어서 과적합이 이루어지지 않은 것으로 보인다.

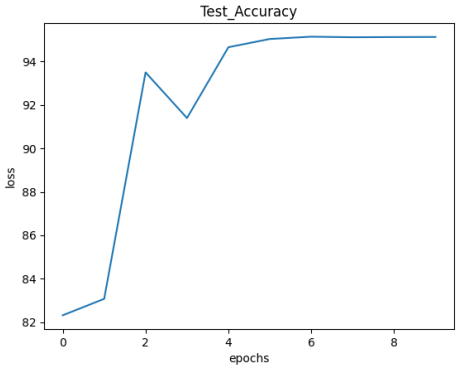
No	Dropout	Epoch	Train Loss	Train accuracy
1	0	10	0.0159	86.83 %
2	0.2	10	0.0176	85.17 %
3	0.4	10	0.0253	78.50 %
4	0.6	10	0.0418	67.96 %
5	0.8	10	0.1393	4.06 %

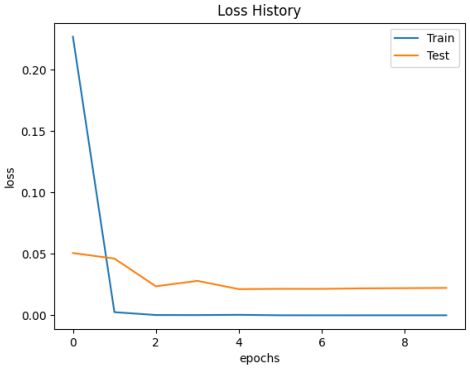
7.2.5 Conclusion

지금까지 여러가지 Hyperparameter 을 조정하면서 가장 높은 accuracy 가 나오는 값들을 찾았다. Hidden Layer 는 3 개일 때 가장 높은 정확도를 보였고, Batch Size 는 16 개일 때 가장 높게 나타났으며, Learning Rate 는 0.05 일 때 가장

높았고, Dropout 은 사용하지 않을 때 가장 높게 나타났다. 따라서 가장 높은 정확도를 가지는 Hyperparameter 로 모델을 설계하였을 때 다음과 같다.

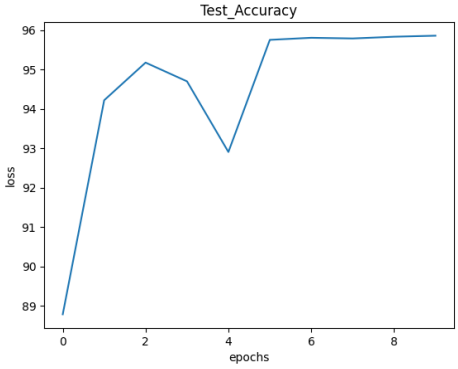
Hidden Layer	Batch size	Learning Rate	Dropout	Epoch	Train Loss	Accuracy
(1024, 512, 256)	16	0.05	0	7	0.0215	95.15 %

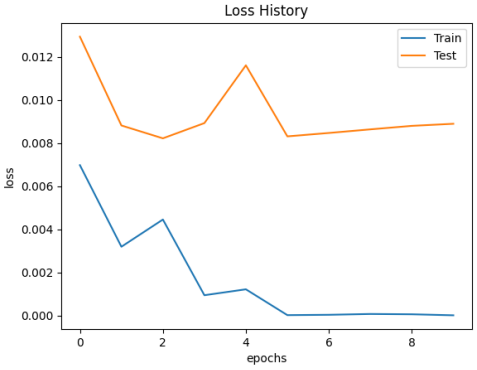




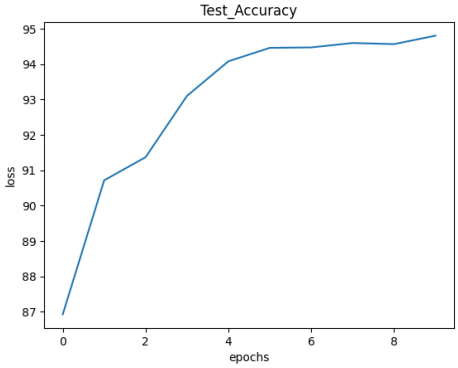
오히려 초기값에서 learning rate 를 0.05 로 한 것보다 정확도가 떨어지는 결과값이 나왔다. 그 이유를 분석해보니 너무 작은 batch size 와 높은 learning rate 의 조합은 적은 데이터로 많은 학습을 바라기 때문에 잘 수렴하기 힘들다. 혹은 큰 batch size 와 낮은 learning rate 는 많은 데이터로 조금씩 학습하기 때문에 과적합이 쉽게된다. 그래서 가장 적절한 batch size 와 learning rate 와의 상관관계는 큰 batch size 와 높은 learning rate, 혹은 작은 batch size 와 낮은 learning rate 가 가장 적절하다는 것을 알게 되었고, 따라서 batch size 를 증가하면서 더 높은 정확도를 찾아보았다.

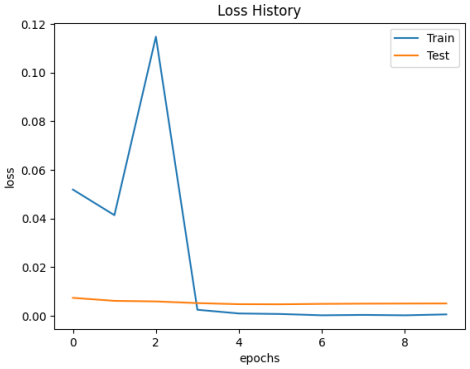
Hidden Layer	Batch size	Learning Rate	Dropout	Epoch	Train Loss	Accuracy
(1024, 512, 256)	32	0.05	0	10	0.0089	95.86 %





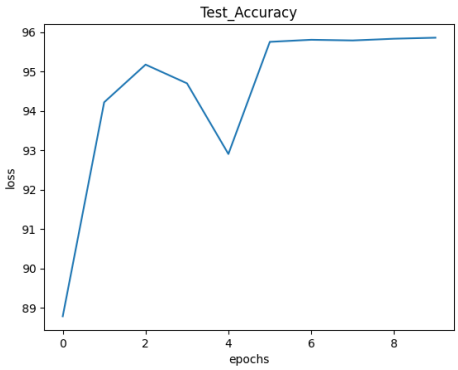
Hidden Layer	Batch size	Learning Rate	Dropout	Epoch	Train Loss	Accuracy
(1024, 512, 256)	64	0.05	0	10	0.0051	94.80 %

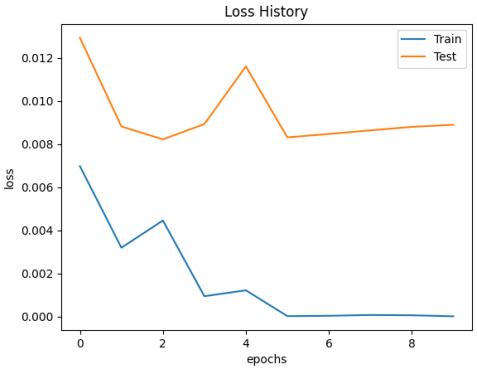




결론적으로 batch size 가 32 일 때 가장 높을 정확도가 나오는 것을 알 수 있다. 이 모델을 설계하면서 아쉬웠던 점은 Hidden Layer 의 개수는 동일하게 하되 node 의 수를 조정해보면서 보다 적절한 값을 찾았으면 더욱 성능이 좋았을 것 같고, 대부분의 결과에서 Epoch 가 10 일 때 가장 높은 정확도가 나타나서 Epoch 값을 조금 더 크게 설정하고 실험하였으면 더 좋은 결과를 도출해낼 수 있었지만 그러지 못했다는 아쉬움이 있다. 최종 모델설계는 다음과 같다.

Hidden Layer	Batch size	Learning Rate	Dropout	Epoch	Train Loss	Accuracy
(1024, 512, 256)	32	0.05	0	10	0.0089	95.86 %





7.3 CNN

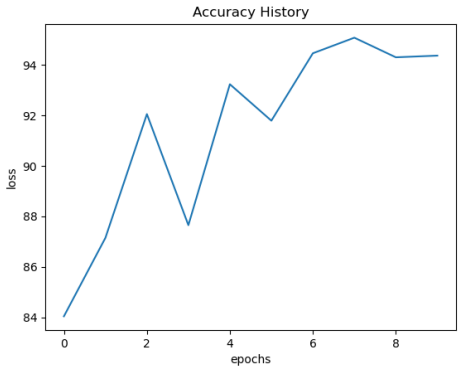
전체적인 실험은 특정 파라미터를 제외한 나머지를 고정시키고 특정 파라미터의 값만 변화를 시켜 실험을 진행하였다. 전체 실험에서의 Loss 는 nn.CrossEntropy 를

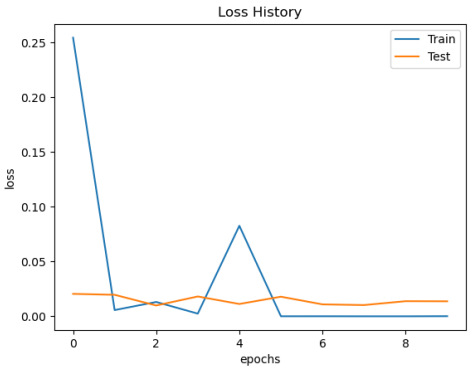
사용하였고, 기존 Training 과 다르게 EPOCH 마다 Test Data 를 적용시켜 정확도가 제일 높게 나왔을 때의 값을 표시하였다.

7.3.1 Convolution Layer Channel 조절

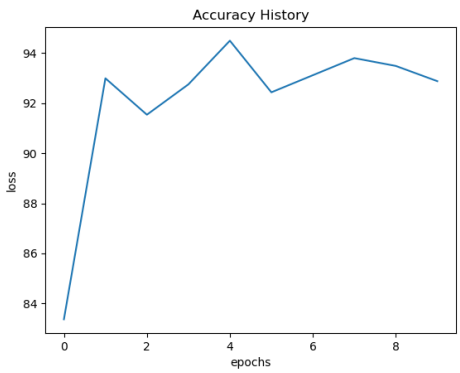
첫번째 실험의 경우에는 BATCH_SIZE = 32, EPOCHS=10, Learning Rate = 0.001, Optimizer = Adam 으로 고정시키고, 4 개의 Convolution Layer 와 2 개의 Fully-Connected Layer(Hidden Layer 의 Node 수는 1024, 256)로 구성되어 있으며, Convolution Layer 의 채널 수를 변화시키면서 실험을 진행하였다. 첫번째 실험의 경우에는 Fruit recognition from images using deep learning 논문에서 시도하였던 파라미터들을 참고하여 동일하게 실험을 진행하였다.
일부 Loss History Plotting 함수 중 Label 이 Validation 으로 나타나 있는 경우가 있는데, 코드 상에서 Validation Test 를 실시하다가 미처 바꾸지 못한 부분이다. Validation Data 를 따로 사용하지 않고 Test Data 를 매 Epochs 마다 적용하여 나온 결과임을 다시 한번 강조한다.

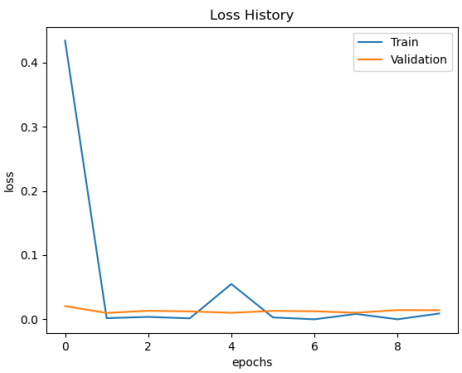
7.3.1.1 (8, 32, 64, 128)



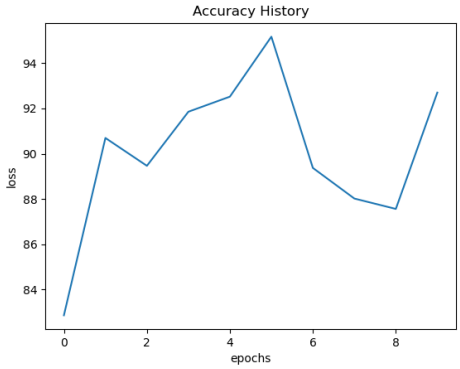


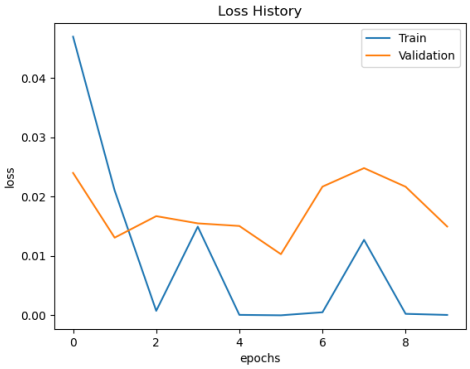
7.3.1.2 (16, 32, 64, 128)



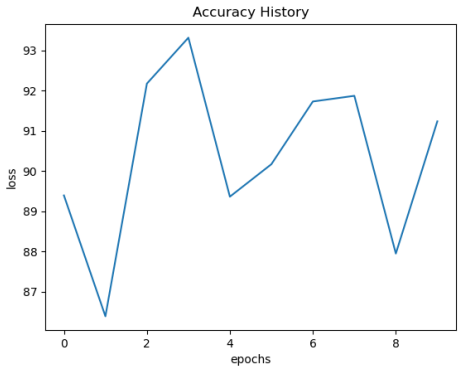


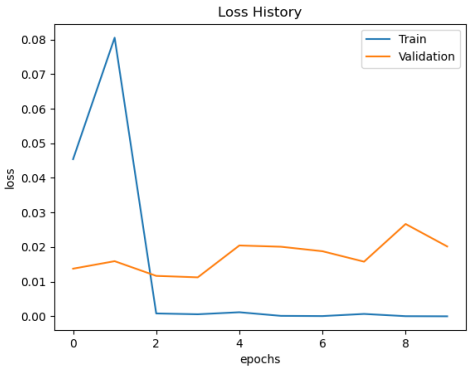
7.3.1.3 (16, 32, 128, 128)





7.3.1.4 (16, 64, 64, 128)





Convolution Layer의 채널 변화를 통해 나온 Test Accuracy는 대체로 비슷하게 나왔고 이 중에 (16, 32, 128, 128)일 때 Accuracy 값이 제일 높게 나왔다. 생각보다 Accuracy의 값들이 비슷하여 채널의 수를 조금 더 Dynamic하게 설정을 하여 실험을 하였다면 눈에 띄는 변화를 발견할 수 있었겠다는 생각이 들어 아쉬웠다.

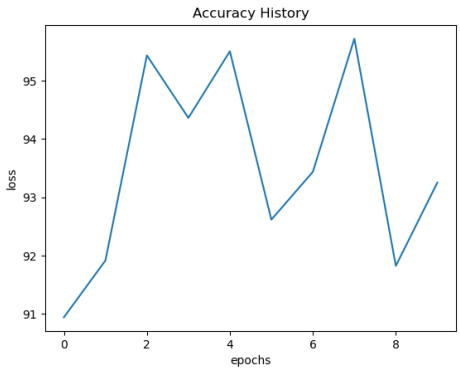
No.	Configuration			Epochs	Test Loss	Test Accuracy
1	Convolutional	5 x 5	8	8	0.0103	95.07%
	Convolutional	5 x 5	32			
	Convolutional	5 x 5	64			
	Convolutional	5 x 5	128			
	Fully Connected		1024			
	Fully Connected		256			

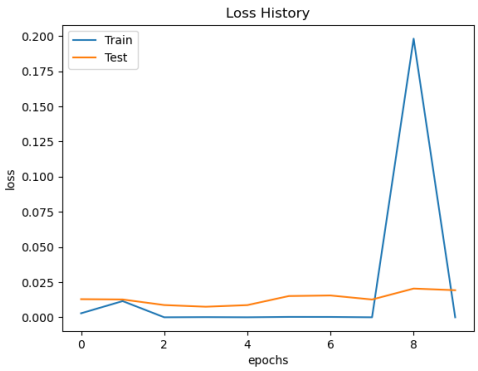
2	Convolutional	5 x 5	16	5	0.0101	94.5%
	Convolutional	5 x 5	32			
	Convolutional	5 x 5	64			
	Convolutional	5 x 5	128			
	Fully Connected		1024			
	Fully Connected		256			
3	Convolutional	5 x 5	16	6	0.0103	95.17%
	Convolutional	5 x 5	32			
	Convolutional	5 x 5	128			
	Convolutional	5 x 5	128			
	Fully Connected		1024			
	Fully Connected		256			
4	Convolutional	5 x 5	16	4	0.0113	93.32%
	Convolutional	5 x 5	64			
	Convolutional	5 x 5	64			
	Convolutional	5 x 5	128			
	Fully Connected		1024			
	Fully Connected		256			

7.3.2 Convolution Layer 수 조절

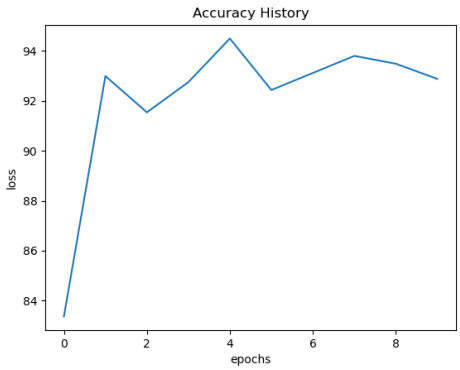
두번째 실험의 경우에는 BATCH_SIZE = 32, EPOCHS=10, Learning Rate = 0.001, Optimizer = Adam 으로 고정시키고, Convolution Layer 의 수를 각각 3 개,4 개,5 개로 설정을 하고 2 개의 동일한 Fully-Connected Layer(Hidden Layer 의 Node 수는 1024, 256)로 구성하여 실험을 진행하였다.

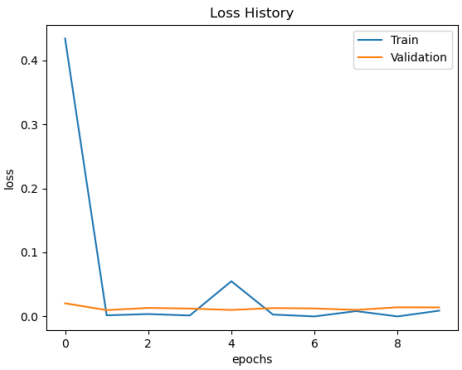
7.3.2.1 (16, 32, 64)



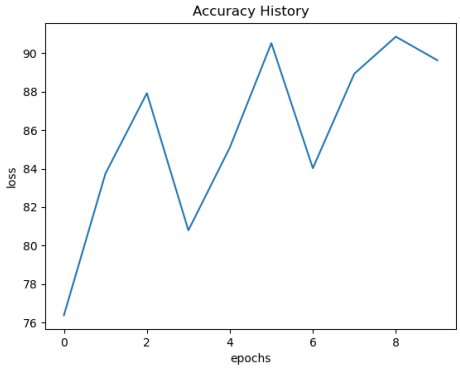


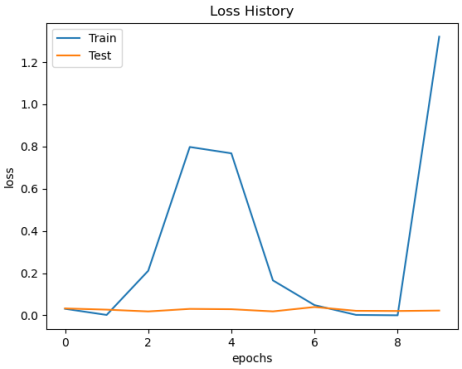
7.3.2.2 (16, 32, 64, 128)





7.3.2.3 (16, 32, 64, 128, 256)





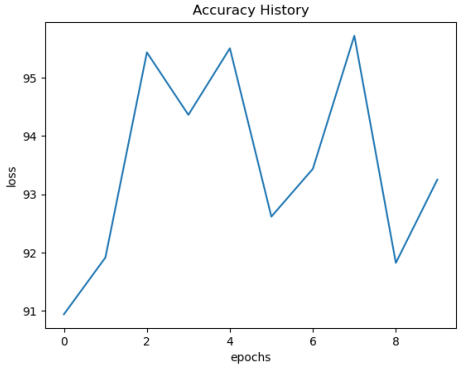
두번째 실험에서, Convolution Layer 의 수를 늘리면 조금 더 정확한 결과가 나올 거라고 예상을 했는데, 예상과는 다르게 Convolution Layer 가 3 개로 줄었을 때 정확도가 조금 더 잘 나왔다. 이 결과는 이미지 데이터가 단순한 과일 그림이라 Filter 을 더 많이 만드는 것이 훈련에 좋지 못한 영향을 끼친 것 같다.

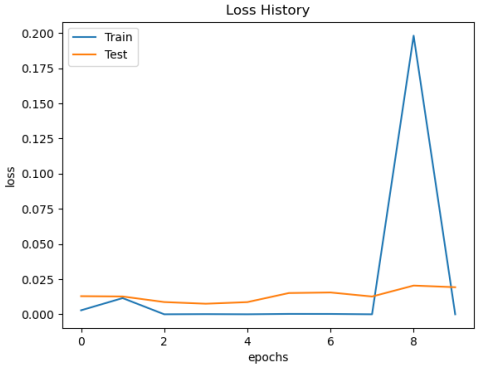
No.	Configuration			Epochs	Test Loss	Test Accuracy
1	Convolutional	5 x 5	16	8	0.0126	95.72%
	Convolutional	5 x 5	32			
	Convolutional	5 x 5	64			
	Fully Connected		1024			
	Fully Connected		256			
2	Convolutional	5 x 5	16	5	0.0101	94.5%
	Convolutional	5 x 5	32			
	Convolutional	5 x 5	64			
	Convolutional	5 x 5	128			
	Fully Connected		1024			
	Fully Connected		256			
3	Convolutional	5 x 5	16	9	0.0205	90.86%
	Convolutional	5 x 5	32			
	Convolutional	5 x 5	64			
	Convolutional	5 x 5	128			
	Convolutional	5 x 5	256			
	Fully Connected		1024			
	Fully Connected		256			

7.3.3 Dropout 설정

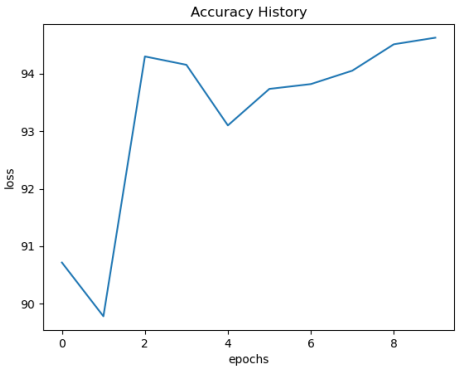
세번째 실험의 경우에는 BATCH_SIZE = 32, EPOCHS=10, Learning Rate = 0.001, Optimizer = Adam 으로 고정시키고, Convolution Layer 의 수를 각각 3 개(16,32,64) 로 설정을 하고 2 개의 동일한 Fully-Connected Layer(Hidden Layer 의 Node 수는 1024, 256)로 구성하여 실험을 진행하였다. 그리고 Fully-Connected Layer 에 Dropout 을 적용하여 그때의 결과를 비교하였다.

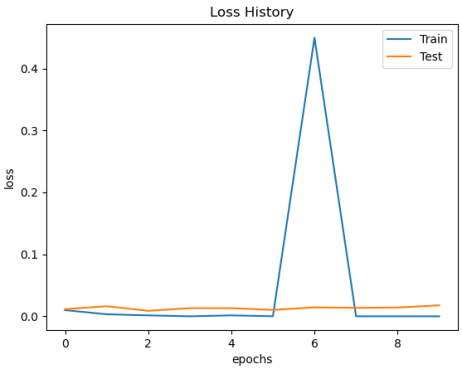
7.3.3.1 Dropout rate = 0



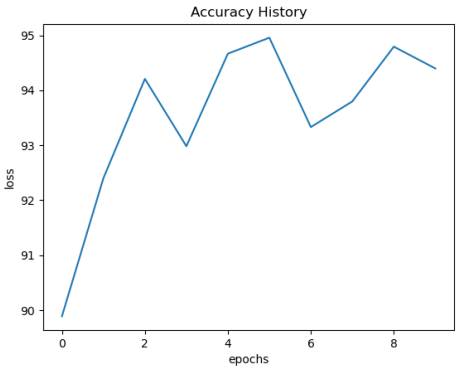


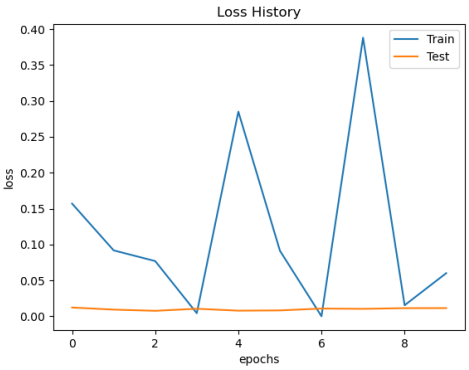
7.3.3.2 Dropout rate = 0.2



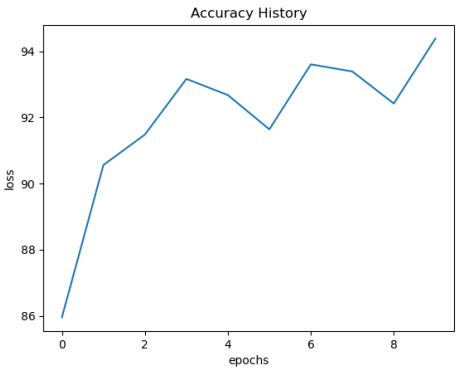


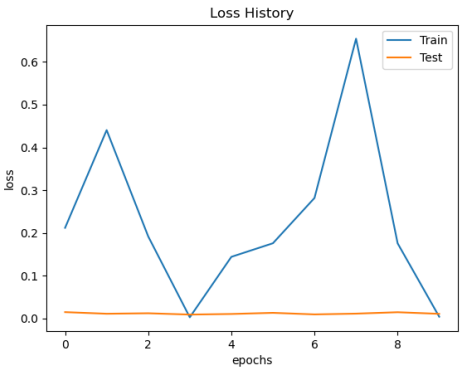
7.3.3.3 Dropout rate = 0.4



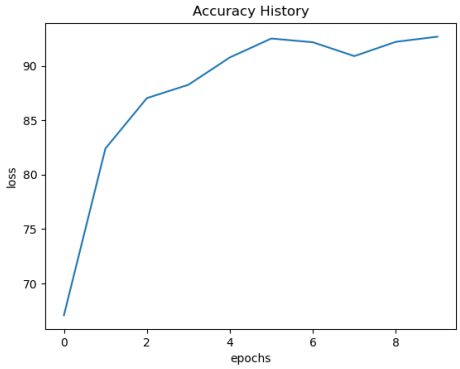


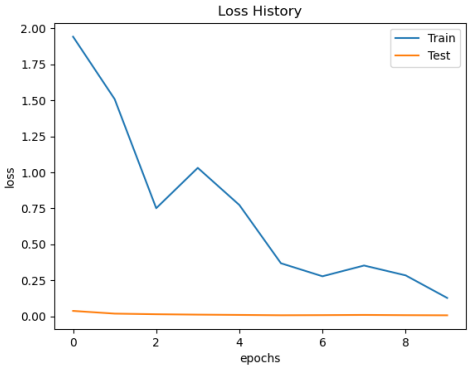
7.3.3.4 Dropout rate = 0.6





7.3.3.5 Dropout rate = 0.8





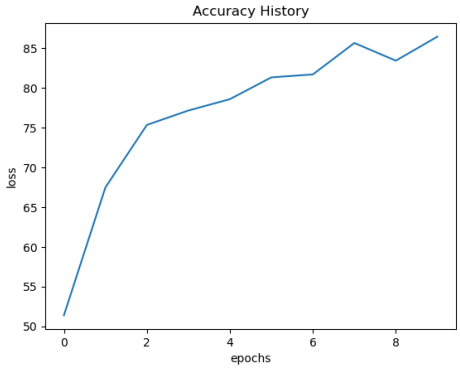
세번째 실험에선 Dropout 을 적용해 보았는데, Dropout 을 적용하여도 Accuracy 가 나아지지는 않았다. 이는 10 개의 epoch 으로 학습을 하는 동안 과적합(Overfitting)이 크게 발생하지 않았기 때문에 Dropout 을 적용한 것이 결과가 조금 더 안 좋게 나오지 않았나 생각이 든다.

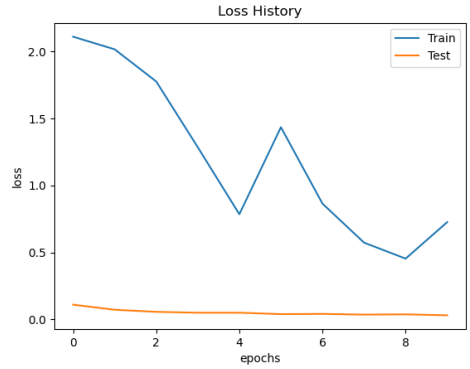
No.	Dropout Rate	Epoch	Test Loss	Test Accuracy
1	X	8	0.0126	95.72%
2	0.2	10	0.0176	94.63%
3	0.4	6	0.0083	94.96%
4	0.6	10	0.0109	94.38%
5	0.8	10	0.0082	92.69%

7.3.4 BATCH_SIZE 조절

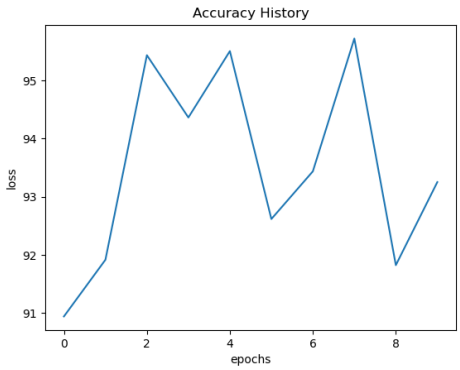
네번째 실험의 경우에는 EPOCHS=10, Learning Rate = 0.001, Optimizer = Adam으로 고정시키고, Convolution Layer의 수를 각각 3개(16,32,64) 로 설정을 하고 2개의 동일한 Fully-Connected Layer(Hidden Layer의 Node수는 1024, 256)로 구성하고, BATCH_SIZE만 다르게 하여 실험을 진행하였다.

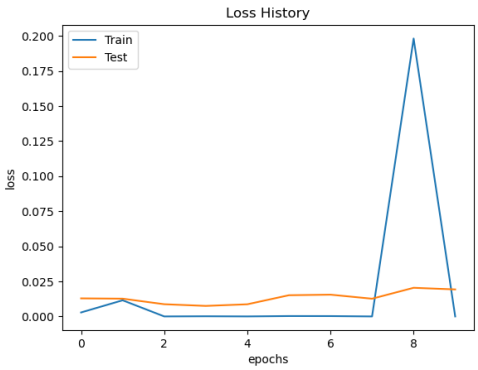
7.3.4.1 BATCH_SIZE = 16



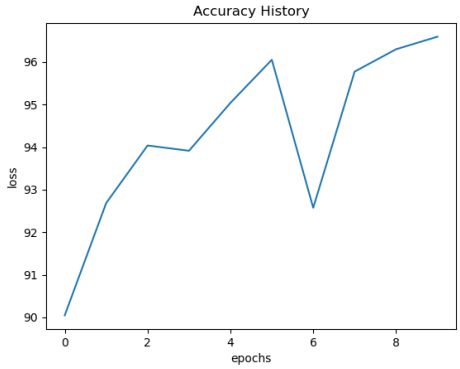


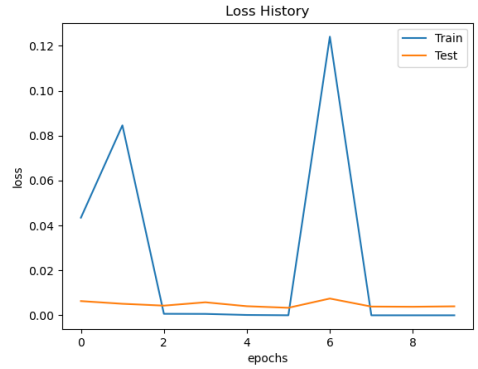
7.3.4.2 BATCH_SIZE = 32



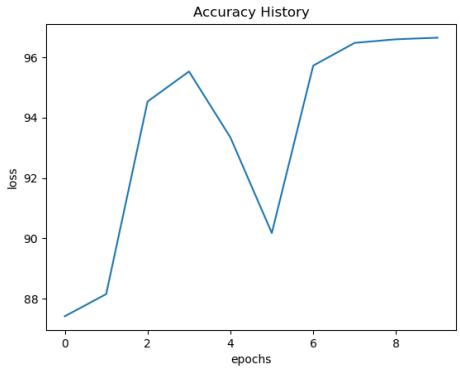


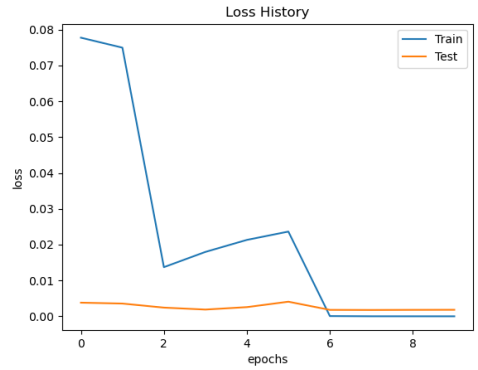
7.3.4.3 BATCH_SIZE = 64





7.3.4.4 BATCH_SIZE = 128





앞에 세번의 실험에서는 모두 Batch Size 를 32 로 놓고 실험을 진행하여 이와 비교하기위해 우선 32 보다 작은 16 으로 배치 사이즈를 정했는데, 이 때 눈에 띄게 Loss 가 증가하고 Accuracy 가 낮아진 것을 확인할 수 있었다. Batch Size 가 작으면 가중치 업데이트를 더 자주 하게 되므로 전체적인 데이터셋의 Trend 를 찾지 못하고 이리 저리 방황하기 때문에 이러한 결과가 나온 것이라고 생각한다. Batch Size 를 64, 128 로 늘렸을 때에는, Loss 도 줄고 Accuracy 도 증가한 모습을

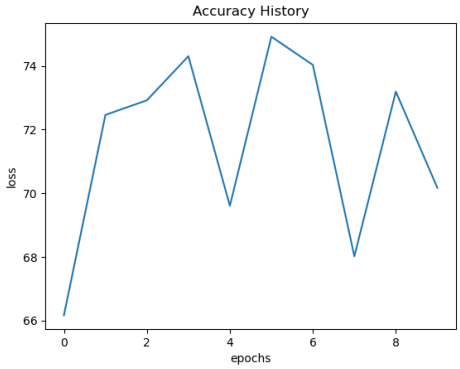
볼 수 있다. 이는 좀 더 데이터의 전체적인 Trend 가 반영이 되어 최적화가 좀 더 잘 된 것 같다.

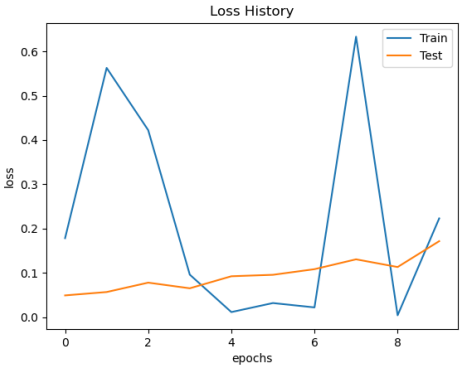
No.	Batch Size	Epoch	Test Loss	Test Accuracy
1	16	10	0.0305	86.47%
2	32	8	0.0126	95.72%
3	64	10	0.0040	96.59%
4	128	10	0.0018	96.65%

7.3.5 Learning Rate 조절

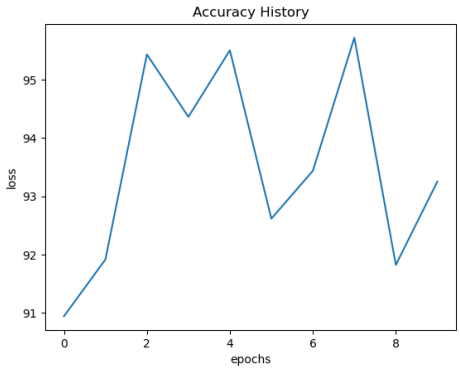
마지막 실험에서는 EPOCHS=10, BATCH_SIZE = 32, Optimizer = Adam 으로 고정시키고, Convolution Layer 의 수를 각각 3 개(16,32,64) 로 설정을 하고 2 개의 동일한 Fully-Connected Layer(Hidden Layer 의 Node 수는 1024, 256)로 구성하고, Learning Rate 만 다르게 하여 실험을 진행하였다.

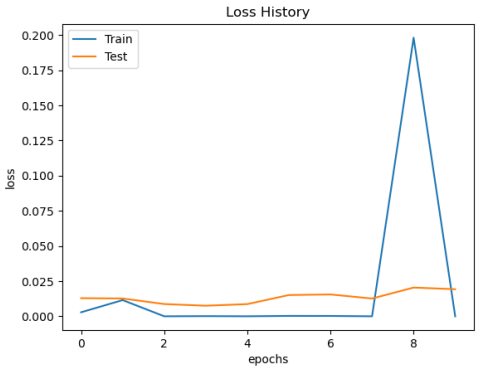
7.3.5.1 Learning Rate = 0.005



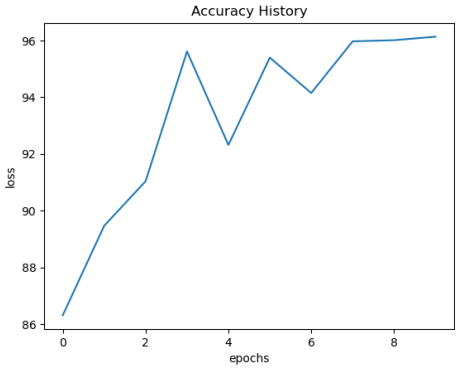


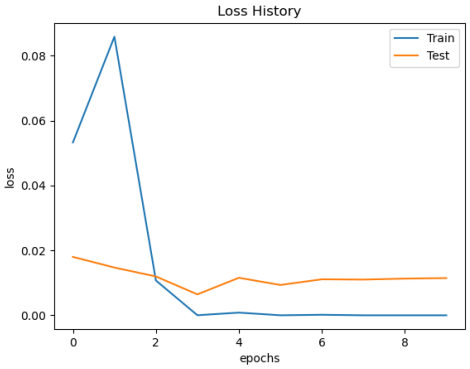
7.3.5.2 Learning Rate = 0.001



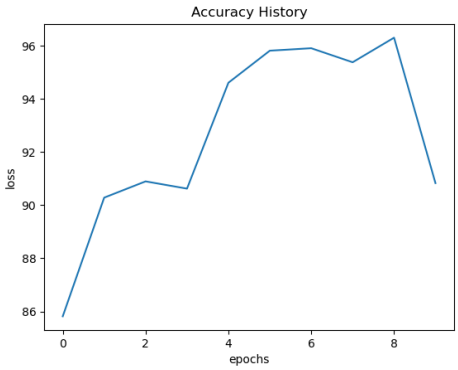


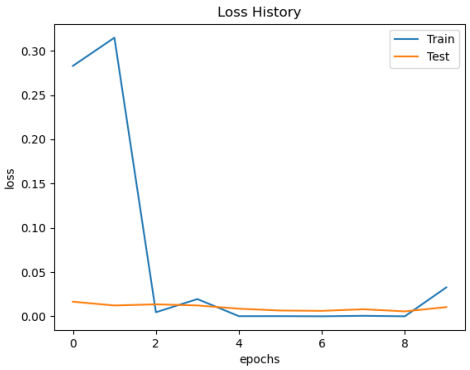
7.3.5.3 Learning Rate = 0.0005



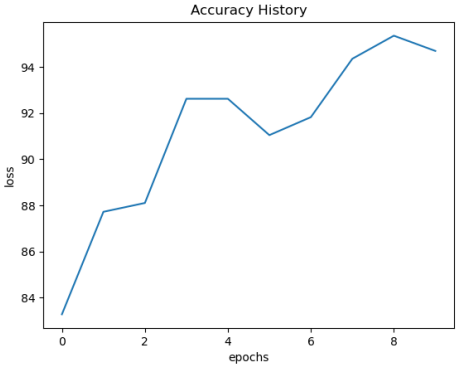


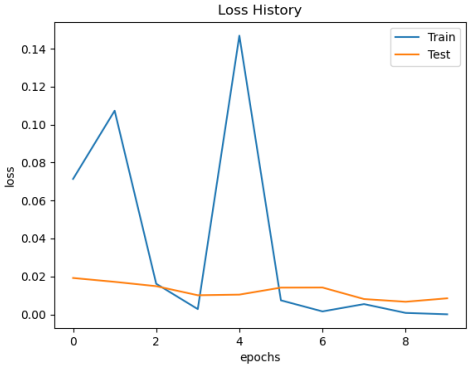
7.3.5.4 Learning Rate = 0.0001





7.3.5.5 Learning Rate = 0.00005





앞선 실험 들에선 Learning Rate = 0.001 로 놓고 실험을 진행하여 우선은 이 값보다 큰 값인 0.005 를 Learning Rate 로 하였을 땐 Loss 와 Accuracy 가 기존 결과들보다 상당히 나쁘게 나온 것을 확인할 수 있다. 학습률이 지나치게 커서 최소값에 수렴하지 못하여 발생할 결과인 것 같다. 그래서 이후 실험에서는 기존 Learning Rate 보다 더 작게 값을 하여 실험을 진행하였는데 0.0005 와 0.0001 에서 기존의 Learning Rate 보다 Loss 도 줄어들고 정확도도 늘어난 것을 확인할 수 있다. 하지만 Learning Rate 를 0.00005 로 설정하였을 때는 Loss 가 다시 증가하고

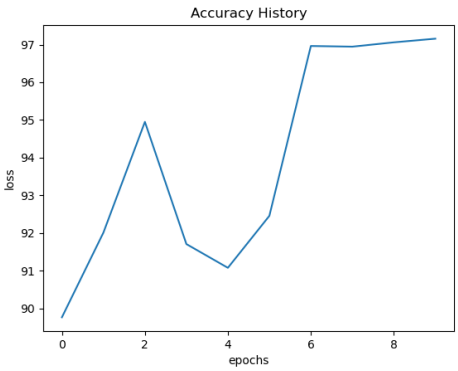
정확도도 떨어졌는데 이는 최소값까지 수렴하는 속도가 느려서 나타나는 결과라고 생각한다.

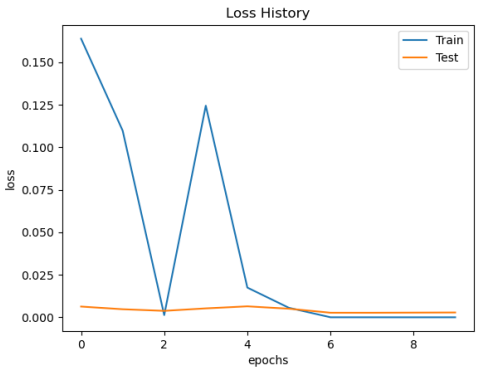
No.	Learning Rate	Epoch	Test Loss	Test Accuracy
1	0.005	6	0.0956	74.92%
2	0.001	8	0.0126	95.72%
3	0.0005	10	0.0115	96.13%
4	0.0001	9	0.0056	96.31%
5	0.00005	9	0.0067	95.36%

7.3.6 결론

위의 실험들을 통해서 어떤 Parameter 를 넣었을 때 가장 좋은 결과가 나올 지 생각을 해 보았다. Convolution Layer 의 수는 3 개일 때 가장 효과가 좋았고, Dropout 은 설정하지 않았을 때 결과가 좋았으며, BATCH_SIZE 는 클수록, Learning Rate 는 (BATCH_SIZE=32 기준) 0.0001 에서 제일 좋았다. 일반적으로 BATCH_SIZE 를 키우는 것과 Learning Rate 를 줄이는 것이 동일한 효과를 보이므로, 극단적으로 BATCH_SIZE 를 키우고 Learning Rate 를 줄이기 보다는, 적절하다고 생각하는 파라미터를 적용하여 실험하였다. 최종 모델 설계와 결과는 다음과 같다.

Configuration			Epochs	Test Loss	Test Accuracy
Convolutional	5 x 5	16	10	0.0028	97.16%
Convolutional	5 x 5	32			
Convolutional	5 x 5	64			
Fully Connected		1024			
Fully Connected		256			
Dropout	X				
Learning Rate	0.0005				
Batch Size	64				





물론 이 결과보다 더 좋은 결과가 나올 수 있다고 생각한다. 이번 실험에서 Padding Size 에 대한 Parameter 조절, MaxPooling Layer 의 Parameter 조절 혹은 AvgPooling Layer 과의 비교, Fully Connected Layer 의 수 조절, 그리고 기존 실험에서도 좀 더 복합적으로 (예를 들면 (BATCH_SIZE, Learning Rate)를 같이 변화시키면서 실험) 실험을 하였다면 더 나은 결과를 얻을 수 있지 않을 까 하는 아쉬움이 남았다.

7.4 최종 결과

지금까지 총 3 개의 모델을 활용하여 다양한 실험을 진행하였다. 각 모델로 실험하여 나온 결과 중 최고로 성능이 좋았던 결과들을 비교하면 다음과 같다.

Models	**Configuration**			**Epochs**	**Test Loss**	**Test Accuracy**
SVM	Batch Size	32		10	0.0347	57.33%
	Learning Rate	0.001				
	Data Scope	[0, 1]				
	Optimizer	Adam				
DNN	Fully Connected	1024		10	0.0089	95.86%
	Fully Connected	512				
	Fully Connected	256				
	Batch Size	32				
	Learning Rate	0.05				
	Data Scope	[-1, 1]				
	Optimizer	SGD				
	Dropout	x				
CNN	Convolutional	5 x 5	16	10	0.0028	97.16%
	Convolutional	5 x 5	32			
	Convolutional	5 x 5	64			
	Fully Connected	1024				
	Fully Connected	256				
	Batch Size	64				
	Learning Rate	0.0005				
	Data Scope	[-1, 1]				
	Optimizer	Adam				
	Dropout	X				

8 실행 방법

팀원들 마다 개발 환경이 다 다른 곳에서 실험을 하였기 때문에, 각자 실행하였던 코드와 라이브러리를 모아서 앞으로 설명할 개발 환경은 CNN 을 개발하였던 환경(PyCharm, 및 Anaconda venv)을 기준으로 설명을 할 것이다.

8.1 Anaconda 및 PyCharm 설치

우선 Anaconda를 설치하고 가상환경을 만들어 주어야 한다.

설치 사이트 : https://www.anaconda.com/products/individual

위 사이트에서 다운로드 후 Anaconda를 실행하고 프로젝트가 있는 폴더로 cd를 이용하여 이동한다. 그러면 프로젝트 폴더에 ai_project.yml 파일이 있는데, 이 파일을 통하여 가상환경을 생성한다. 생성 명령어는 다음과 같다.

conda env create -n ai_project --file ai_project.yml

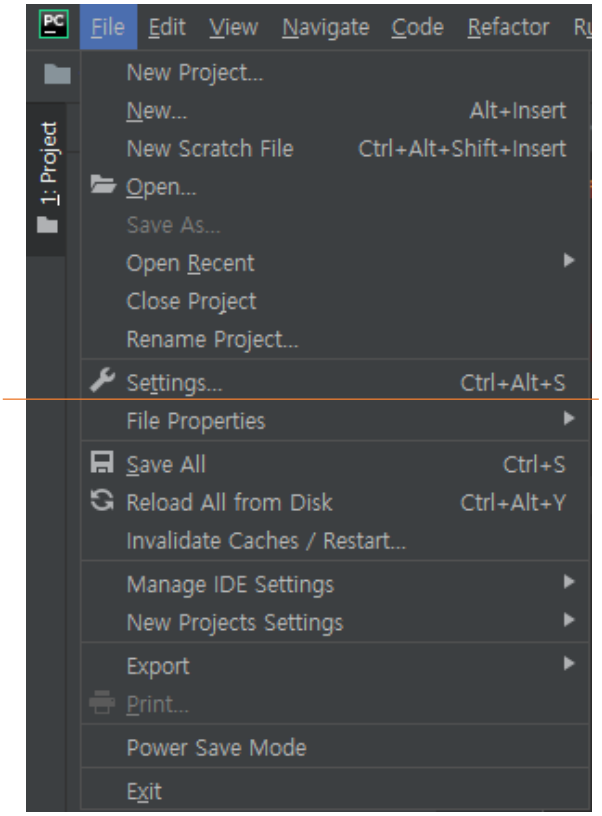
그 다음 절차로는 PyCharm을 설치해야 한다.

설치 사이트 : https://www.jetbrains.com/ko-kr/pycharm/download/#section=windows

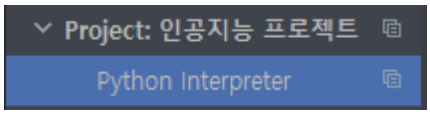
설치를 마친 후 PyCharm을 실행시킨다.

8.2 PyCharm에서 가상환경 설정

왼쪽 위 File Tab 의 Settings 를 누른다.

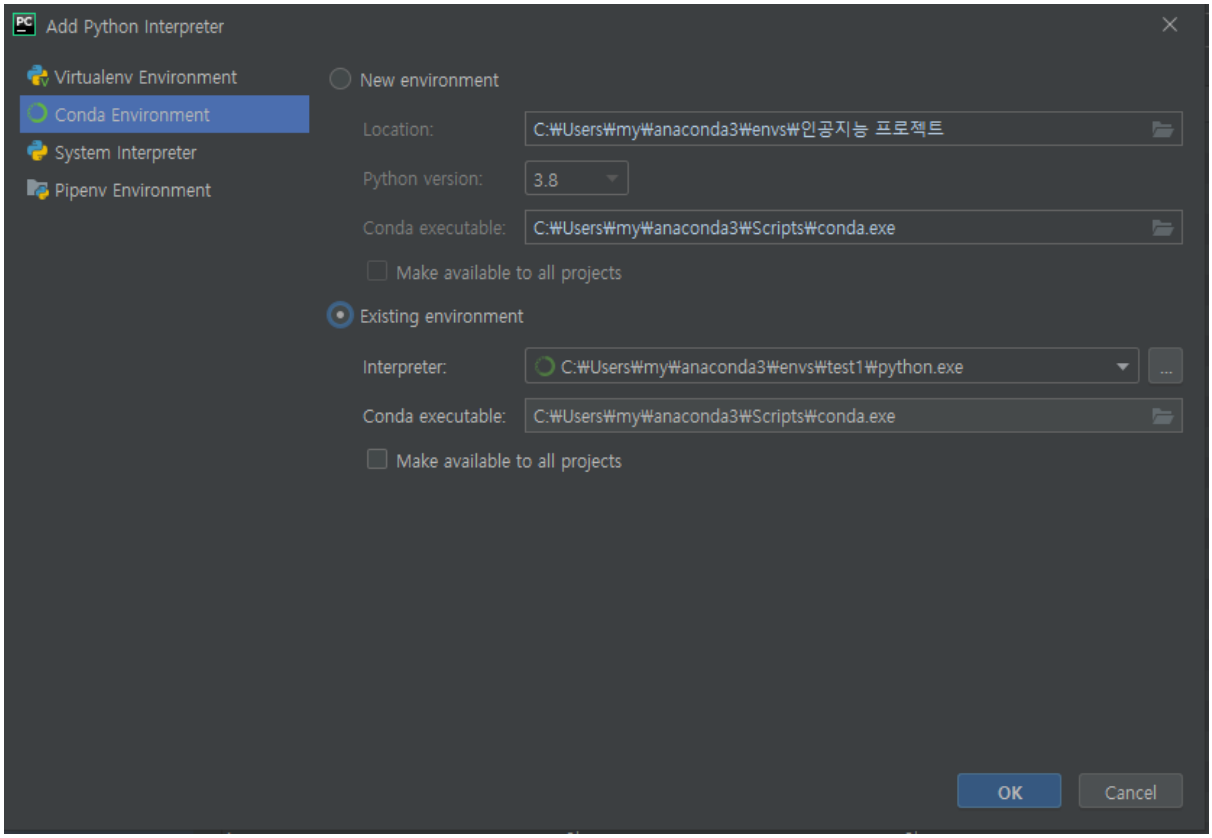


나오는 창의 왼쪽에 있는 Project 클릭 후 Python Interpreter 을 클릭한다.



아래 그림을 참고하여 오른쪽에 있는 톱니바퀴를 누른 후 Add 를 누른다.

그 다음에 나타나는 창에서 Conda Environment 를 누르고 Existing environment 를 누르게 되면 자동으로 이전에 만들었던 가상환경이 등장하게 될 것이다. Ok 를 누른 후에 창의 오른쪽 아래 Apply 를 누르고 Ok 를 누르게 되면 가상환경 설정이 완료된다.



8.3 실행

먼저 Dataset.zip 압축을 해제해야 한다. 압축을 풀 때 선택된 폴더 하위에 압축파일명으로 폴더 생성 버튼을 해제해야 한다. 즉 압축을 풀었을 때 나오는 폴더의 이름은 fruits-360 이어야 한다.

8.3.1 main.py

main.py 는 DNN 과 CNN 결과를 확인할 수 있다. Batch Size, Learning Rate, 훈련시킬 Model, Epochs 등을 설정할 수 있다. 모델에 대한 파라미터는 각 모델의 파일인 DNN.py, CNN.py 에서 원하는 파라미터를 직접 입력하고 실행을 해야 한다.

8.3.2 saveData.py

SVM 모델을 실행하기 위하여 사전에 실행해야 하는 파일이다. 만약 .npy 파일이 존재한다면 실행하지 않아도 된다. 데이터를 .npy 로 만드는 파일은 svm_keras.py 파일에도 있으나 다른 개발환경에서 진행하였기 때문에, 윈도우, PyCharm 에서 데이터를 얻고 싶을 때 이 파일을 실행하면 된다.

8.3.3 svm_keras.py

Keras 로 만들어진 SVM 모델을 실행하기 위한 파일이다. .npy 파일 형식의 데이터가 반드시 필요하며, 실행하면 Keras 로 만들어진 SVM 모델의 결과를 확인할 수 있다. 주의사항으로는 데이터 전처리 과정에서 메모리를 상당히 많이 잡아먹기 때문에 메모리가 충분하지 않다면 실행에 지장이 있을 수도 있다.

8.3.4 svm_scikit-learn.py

Scikit-learn 으로 만들어진 SVM 모델을 실행하기 위한 파일이다. .npy 파일 형식의 데이터가 반드시 필요하며, 실행시 Scikit-learn 으로 만들어진 SVM 모델의 결과를 확인할 수 있다.

출처

고속화도로 자율주행을 위한 영상 기반 딥러닝(DNN) 활용 '차로 위치 예측 및 변경 시스템'에 관한 연구 p30-42

DNN learning rate와 batch size의 상관관계 : https://inhovation97.tistory.com/32

Dataset : https://www.kaggle.com/moltean/fruits

Horea MureÈZan and Mihai Oltean. Fruit recognition from images using deep learning. Acta Universitatis Sapientiae, Informatica, 10:26–42, 06 2018.

CNN :http://taewan.kim/post/cnn/

CNN, DNN, : 이경택 외 2명,『파이썬 딥러닝 파이토치』, 정보문화사(2020), p135-136, p141-145

SVM :https://ko.wikipedia.org/wiki/%EC%84%9C%ED%8F%AC%ED%8A%B8_%EB%B2%A1%ED%84%B0_%EB%A8%B8%EC%8B%A0

SVM 사진 : http://hleecaster.com/ml-svm-concept/