Introduction to Deep Neural Networks (Spring 2021)

Homework #3 (50 Pts, Due Date: May 2)

Student ID 2019311195

Name 김지유

Instruction: We provide all codes and datasets in Python. Please write your code to complete models (MLP_calssifier.py, MLP_regressor.py). Submit two files as follows:

'DNN_HW3_YourName_ STUDENTID.zip': ./model/*.py and your document
'DNN_HW3_YourName_ STUDENTID.pdf': Your document should be converted into pdf.

NOTE: You should write your source code in the **'EDIT HERE' part** and do not edit other parts. You can check your code by executing the main code ('main_classification.py' and 'main_regression.py').

TIP 1. Refer to the PyTorch implementation for the linear regression (main_linear.py and models/Linear_regressor.py).

TIP 2. You can also refer to the PyTorch tutorials as below.

Kor 1. https://tutorials.pytorch.kr/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py
Kor 2. https://wikidocs.net/book/2788
Eng 1. https://pytorch.org/tutorials/beginner/basics/intro.html
Eng 2. https://www.analyticsvidhya.com/blog/2019/09/introduction-to-pytorch-from-scratch/

(1) [**30 pts**] Implement Multilayer Perceptron (MLP) models in 'MLP_classifier.py' and 'MLP_regressor.py.'

(a) [Regression] Implement __init__, forward, and predict method functions in 'MLP_regressor.py.'

(b) [Classification] Implement __init__, forward, and predict method functions in 'MLP_classifier.py.'

Answer: Fill your code here. You also have to submit your code to i-campus.

(a) [Regression]

```
from utils import MSE
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
import time
import os
import numpy as np
```

```python
import matplotlib.pyplot as plt

'''
Please refer the models/Linear_regressor.py and the PyTorch tutorials in report fi
le.
'''
class MLP_regressor(nn.Module):
    def __init__(self, input_dim, learning_rate):
        super(MLP_regressor, self).__init__()
        '''
        Please define layers, loss, and optimizer in here.
        You can use "nn.Linear" for MLP layers,
        "nn.MSELoss" for MSE Loss, and "torch.optim.SGD" for SGD optimizer.
        '''
        self.loss_function = None
        self.optimizer = None
        # ================================ EDIT HERE ================================
=
        self.linear1 = nn.Linear(in_features=input_dim, out_features=512, bias=Tru
e)
        self.linear2 = nn.Linear(in_features=512, out_features=256, bias=True)
        self.linear3 = nn.Linear(in_features=256, out_features=128, bias=True)
        self.linear4 = nn.Linear(in_features=128, out_features=64, bias=True)
        self.linear5 = nn.Linear(in_features=64, out_features=1, bias=True)
        self.relu = nn.ReLU()

        self.loss_function = nn.MSELoss()
        self.optimizer = torch.optim.SGD(self.parameters(), lr=learning_rate)
        # ============================================================================
=

        return

    def forward(self, x):
        '''
        Please define model in here.
        You can use "torch.sigmoid" for Sigmoid function.
        '''
        out = torch.zeros((x.shape[0], 1))
        # ================================ EDIT HERE ================================
=
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        x = self.relu(x)
        x = self.linear3(x)
        x = self.relu(x)
        x = self.linear4(x)
```

```
        x = self.relu(x)
        out = self.linear5(x)
        # ================================================================
=
        return out

    def predict(self, x):
        '''
        Please define model predict function.
        You have to use "torch.no_grad()" in order not to calculate the gradient.
        And, implement in mini-batch.
        '''
        pred_y = np.zeros((x.shape[0], 1))
        # ================================ EDIT HERE ================================
=
        # transfrom numpy data to torch data and make torch dataset
        x_tenser = torch.tensor(x)
        data_loader = DataLoader(x_tenser, batch_size=self.batch_size)

        # predict y with mini-batch
        pred_y = []
        with torch.no_grad():
            for batch_data in data_loader:
                batch_x = batch_data
                batch_pred_y = self.forward(batch_x) #np.ndarray
                pred_y.append(batch_pred_y.numpy())

        pred_y = np.concatenate(pred_y, axis=0)
        # ================================================================
=
        return pred_y

    def train(self, train_x, train_y, valid_x, valid_y, num_epochs, batch_size, pr
int_every=10):
        '''
        Calculate loss and update model using optimizer.
        You can easily use mini-batch using "TensorDataset" and "DataLoader".
        '''
        self.train_MSE = []
        self.valid_MSE = []
        best_epoch = -1
        best_mse = float('inf')
        self.num_epochs = num_epochs
        self.print_every = print_every

        # transfrom numpy data to torch data and make torch dataset
        x_tenser = torch.tensor(train_x).float()
        y_tenser = torch.tensor(train_y).float()
```

```
        dataset = TensorDataset(x_tenser, y_tenser)

        data_loader = DataLoader(dataset, batch_size=batch_size)
        self.batch_size = batch_size

        # train the model with mini-batch
        for epoch in range(1, num_epochs+1):
            start = time.time()
            epoch_loss = 0.0
            # model train
            for b, batch_data in enumerate(data_loader):
                batch_x, batch_y = batch_data
                pred_y = self.forward(batch_x)

                if self.loss_function:
                    # calcuate the loss
                    loss = self.loss_function(pred_y.reshape(-1), batch_y)
                    # model update
                    self.optimizer.zero_grad()
                    loss.backward()
                    self.optimizer.step()
                    epoch_loss += loss

            epoch_loss /= len(data_loader)
            end = time.time()
            lapsed_time = end - start
            print(f'Epoch {epoch} took {lapsed_time} seconds\n')

            # model validate
            if epoch % print_every == 0:
                # TRAIN ACCURACY
                pred = self.predict(train_x)
                train_mse = MSE(pred, train_y)
                self.train_MSE.append(train_mse)

                # VAL ACCURACY
                pred = self.predict(valid_x)
                valid_mse = MSE(pred, valid_y)
                self.valid_MSE.append(valid_mse)

                print('[EPOCH %d] Loss = %.5f' % (epoch, epoch_loss))
                print('Train MSE = %.3f' % train_mse + ' // ' + 'Valid MSE = %.3f'
 % valid_mse)

                # best model save
                if best_mse > valid_mse:
                    print('Best Accuracy updated (%.4f => %.4f)' % (best_mse, vali
d_mse))
```

```
                    best_mse = valid_mse
                    best_epoch = epoch
                    torch.save(self.state_dict(), './best_model/MLP_regressor.pt')
        print('Training Finished...!!')
        print('Best Valid mse : %.2f at epoch %d' % (best_mse, best_epoch))

    def restore(self):
        with open(os.path.join('./best_model/MLP_regressor.pt'), 'rb') as f:
            state_dict = torch.load(f)
        self.load_state_dict(state_dict)

    def plot_accuracy(self):
        """
            Draw a plot of train/valid accuracy.
            X-axis : Epoch
            Y-axis : train MSE & valid MSE
            Draw train MSE-epoch, valid MSE-epoch graph in 'one' plot.
        """
        epochs = list(np.arange(1, self.num_epochs+1, self.print_every))

        print(len(epochs), len(self.train_MSE))

        plt.plot(epochs, self.train_MSE, label='Train MSE')
        plt.plot(epochs, self.valid_MSE, label='Valid MSE')

        plt.title('Epoch - Train/Valid MSE')
        plt.xlabel('Epochs')
        plt.ylabel('Mean Squared Error')
        plt.legend()

        plt.show()
```

(b) [Classification]

```
import time
import os
import torch
import torch.nn as nn

import numpy as np
import matplotlib.pyplot as plt

from torch.utils.data import TensorDataset, DataLoader

'''
```

```
Please refer the models/Linear_regressor.py and the PyTorch tutorials in report fi
le.
'''
class MLP_classifier(nn.Module):
    def __init__(self, input_dim, output_dim, learning_rate):
        super(MLP_classifier, self).__init__()
        '''
        Please define layers, loss, and optimizer in here.
        You can use "nn.Linear" for MLP layers,
        "nn.CrossEntropyLoss" for CE Loss, and "torch.optim.SGD" for SGD optimizer
.
        '''
        self.output_dim = output_dim
        self.loss_function = None
        self.optimizer = None
        # ============================== EDIT HERE ==============================
=
        self.linear1 = nn.Linear(in_features = input_dim, out_features = 512, bias
 = True)
        self.linear2 = nn.Linear(in_features = 512, out_features = 256, bias = Tru
e)
        self.linear3 = nn.Linear(in_features = 256, out_features = 128, bias = Tru
e)
        self.linear4 = nn.Linear(in_features = 128, out_features = output_dim, bia
s = True)
        self.relu = nn.ReLU()

        self.loss_function = nn.CrossEntropyLoss()
        self.optimizer = torch.optim.SGD(self.parameters(), lr=learning_rate)
        # ========================================================================
=

    def forward(self, x):
        '''
        Please define model in here.
        You can use "torch.sigmoid" for Sigmoid function.
        '''
        out = torch.zeros((x.shape[0], self.output_dim))
        # ============================== EDIT HERE ==============================
=
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        x = self.relu(x)
        x = self.linear3(x)
        x = self.relu(x)
        x = self.linear4(x)
        out = torch.sigmoid(x)
```

```
        # ================================================================================
=
        return out

    def predict(self, x):
        '''
        Please define model predict function.
        You have to use "torch.no_grad()" in order not to calculate the gradient.
        And, implement in mini-batch.
        '''
        pred_y = np.zeros((x.shape[0], ))
        # ============================== EDIT HERE ==============================
=
        # transfrom numpy data to torch data and make torch dataset
        x_tenser = torch.tensor(x).float()
        data_loader = DataLoader(x_tenser, batch_size=self.batch_size)

        # predict y with mini-batch
        pred_y = []
        with torch.no_grad():
            for batch_data in data_loader:
                batch_x = batch_data
                batch_pred_y = self.forward(batch_x)
                pred_y.append(np.argmax(batch_pred_y.numpy(), axis=1))
        pred_y = np.concatenate(pred_y, axis=0)
        # ================================================================================
=
        return pred_y

    def train(self, train_x, train_y, valid_x, valid_y, num_epochs, batch_size, pr
int_every=10):
        '''
        Calculate loss and update model using optimizer.
        You can easily use mini-batch using "TensorDataset" and "DataLoader".
        '''
        self.train_accuracy = []
        self.valid_accuracy = []
        best_epoch = -1
        best_acc = -1
        self.num_epochs = num_epochs
        self.print_every = print_every

        # transfrom numpy data to torch data and make torch dataset
        x_tenser = torch.tensor(train_x).float()
        y_tenser = torch.tensor(train_y).float()
        dataset = TensorDataset(x_tenser, y_tenser)

        data_loader = DataLoader(dataset, batch_size=batch_size)
```

```
        self.batch_size = batch_size

        # train the model with mini-batch
        for epoch in range(1, num_epochs+1):
            start = time.time()
            epoch_loss = 0.0
            # model train
            for b, batch_data in enumerate(data_loader):
                batch_x, batch_y = batch_data
                pred_y = self.forward(batch_x)

                if self.loss_function:
                    # calcuate the loss
                    loss = self.loss_function(pred_y, torch.argmax(batch_y, -1))

                    # model update
                    self.optimizer.zero_grad()
                    loss.backward()
                    self.optimizer.step()

                    epoch_loss += loss

            epoch_loss /= len(data_loader)
            end = time.time()
            lapsed_time = end - start
            print(f'Epoch {epoch} took {lapsed_time} seconds\n')

            # model validate
            if epoch % print_every == 0:
                # TRAIN ACCURACY
                pred = self.predict(train_x)
                true = np.argmax(train_y, -1).astype(int)

                correct = len(np.where(pred == true)[0])
                total = len(true)
                train_acc = correct / total
                self.train_accuracy.append(train_acc)

                # VAL ACCURACY
                pred = self.predict(valid_x)
                true = np.argmax(valid_y, -1).astype(int)

                correct = len(np.where(pred == true)[0])
                total = len(true)
                valid_acc = correct / total
                self.valid_accuracy.append(valid_acc)

                print('[EPOCH %d] Loss = %.5f' % (epoch, epoch_loss))
```

```
                print('Train Accuracy = %.3f' % train_acc + ' // ' + 'Valid Accura
cy = %.3f' % valid_acc)

                # best model save
                if best_acc < valid_acc:
                    print('Best Accuracy updated (%.4f => %.4f)' % (best_acc, vali
d_acc))
                    best_acc = valid_acc
                    best_epoch = epoch
                    torch.save(self.state_dict(), './best_model/MLP_classifier.pt'
)
        print('Training Finished...!!')
        print('Best Valid acc : %.2f at epoch %d' % (best_acc, best_epoch))

    def restore(self):
        with open(os.path.join('./best_model/MLP_classifier.pt'), 'rb') as f:
            state_dict = torch.load(f)
        self.load_state_dict(state_dict)


    def plot_accuracy(self):
        """
            Draw a plot of train/valid accuracy.
            X-axis : Epoch
            Y-axis : train_accuracy & valid_accuracy
            Draw train_acc-epoch, valid_acc-epoch graph in 'one' plot.
        """
        epochs = list(np.arange(1, self.num_epochs+1, self.print_every))

        print(len(epochs), len(self.train_accuracy))

        plt.plot(epochs, self.train_accuracy, label='Train Acc.')
        plt.plot(epochs, self.valid_accuracy, label='Valid Acc.')

        plt.title('Epoch - Train/Valid Acc.')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.legend()

        plt.show()
```

(2) [20 pts] Report the experiment results for each dataset.

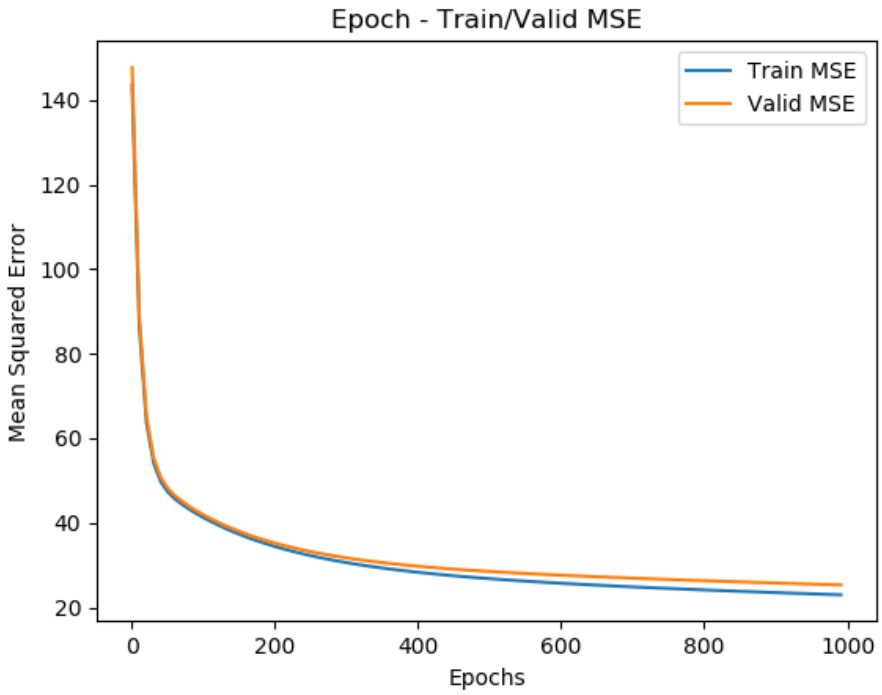
(a) [Regression with different architectures] Adjust the model settings (number of hidden layers, number of hidden nodes, number of epochs, learning rate, etc.) to obtain the best results over the **House dataset** using 'main_classification.py.' Report your top 3 best test accuracy with your fine-tuned hyperparameters. Show the plot of training and validation **MSE** every epoch on each case. Also, describe how you determined the model

structure or parameters in 4~5 lines.

Answer: Fill the blank in the table. Show the plot of training & validation MSE through epochs.

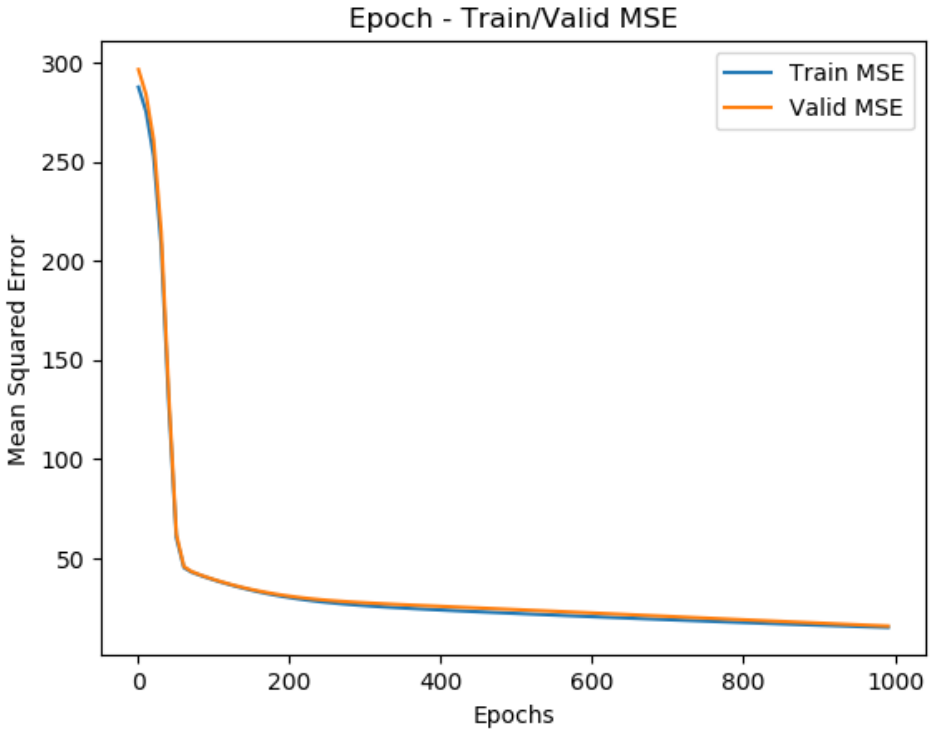
	Model structure	# of epochs	Learning rate	Best Validation MSE	Final Test MSE
1st Best	**FC-1(14,512)** **ReLU-1** **FC-2(512, 256)** **ReLU-2** **FC-3(256, 128)** **ReLU-3** **FC-4(128, 64)** **ReLU-4** **FC-5(64, 1)**	1000	0.001	7.78	16.72
2nd Best	**FC-1(14, 128)** **ReLU-1** **FC-2(128, 64)** **ReLU-2** **FC-3(64, 1)**	1000	0.0001	15.80	18.23
3rd Best	**FC-1(14,1)**	1000	0.001	25.30	22.26

[3rd Best]



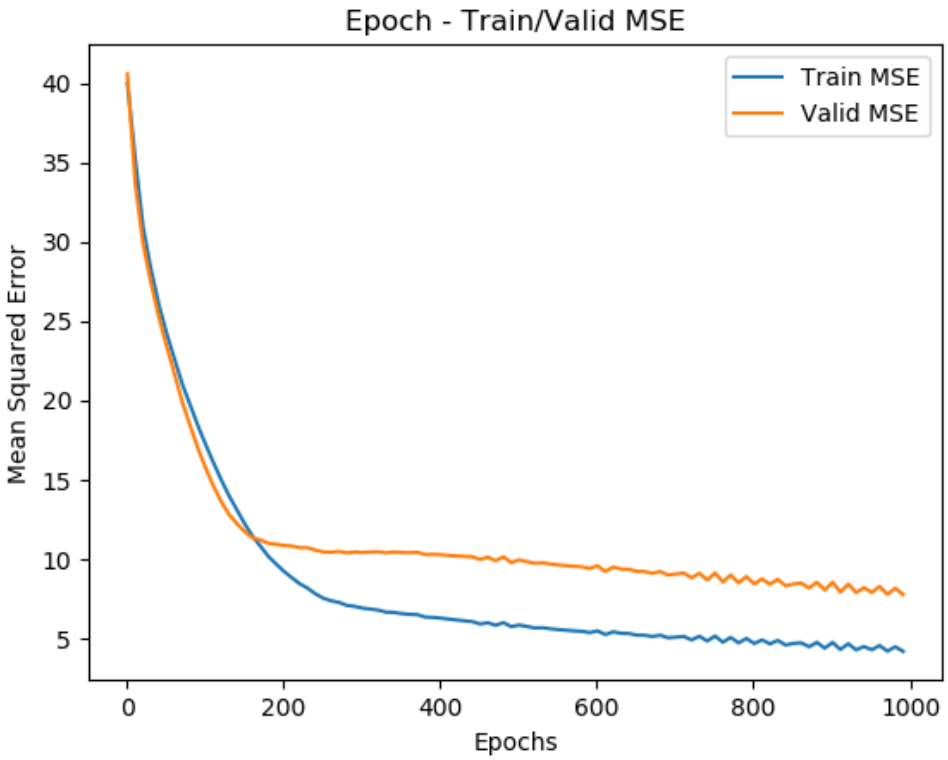
세번째로 성능이 좋은 모델의 파라미터는 num_epoch = 1000, learning_rate = 0.0001, batch_size = 64 이다. 이 모델의 아키텍처는 FC layer 1 개로만 이루어져 있다. 파라미터 중 batch size 가 모델의 정확도에 큰 영향을 미쳤다. 해당 모델의 아키텍처와 다른 파라미터들은 유지한 상황에서 batch size 가 작아질수록 MSE 또한 함께 작아지다가 batch size 가 32 가 되면 다시 커지는 모습을 보였다. 그래서 batch size 는 비교적 작은 값인 64 로 하여 모델을 학습시켰다. 모델의 파라미터 중 learning rate 도 batch size 다음으로 모델의 정확도에 큰 영향을 미쳤다. learning rate 가 0.1 일 경우, MSE 가 가장 크게 나오고, 그 이하에서부터는 비교적 비슷한 MSE 가 나왔다. 그래서 learning rate 는 0.01 로 하여 모델이 빠르게 학습할 수 있도록 하였다.

[2nd Best]



두번째로 성능이 좋은 모델의 파라미터는 num_epoch = 1000, learning_rate = 0.0001, batch_size = 128 이다. 3rd Best 모델의 MSE 가 epoch 을 늘려도 큰 값에서 줄어들지 않아서 layer 를 좀 더 쌓은 모델을 만들어 실험했다. 이 모델의 아키텍처는 FC layer(14, 128), ReLU, FC layer(128, 64), ReLU, FC layer(64, 1)로 이루어져 있다. 3rd Best 모델보다 layer 개수가 많아서 learning rate 가 크면 불안정하게 학습되는 모습을 보이고, batch size 가 작으면 MSE 가 커지는 모습을 보였다. 그래서 learning rate 를 상대적으로 작은 0.0001 로, batch size 는 상대적으로 큰 128 로 하여 학습을 진행하였다.

[1st Best]



가장 성능이 좋은 모델의 파라미터는 num_epoch = 1000, learning_rate = 0.001, batch_size = 128 이다. 2nd Best 모델에서 layer 를 더 쌓아서 모델의 복잡성을 증가시켰다. 이 모델의 아키텍처는 FC layer(14, 512), ReLU, FC layer(512, 256), ReLU, FC layer(256, 128) FC layer(128, 64), ReLU, FC layer(64, 1)로 이루어져 있다. 이 이상의 layer 를 쌓아 실험을 진행하여 본 결과, 유의미한 MSE 의 감소가 일어나지 않았다. learning rate 가 작으면 안정적으로 학습되는 모습을 보였지만, MSE 는 상대적으로 큰 값이 나와서 learning rate 를 0.001 로 하여 학습시켰다. 또한 batch size 는 128 로 하여 학습시켰다.

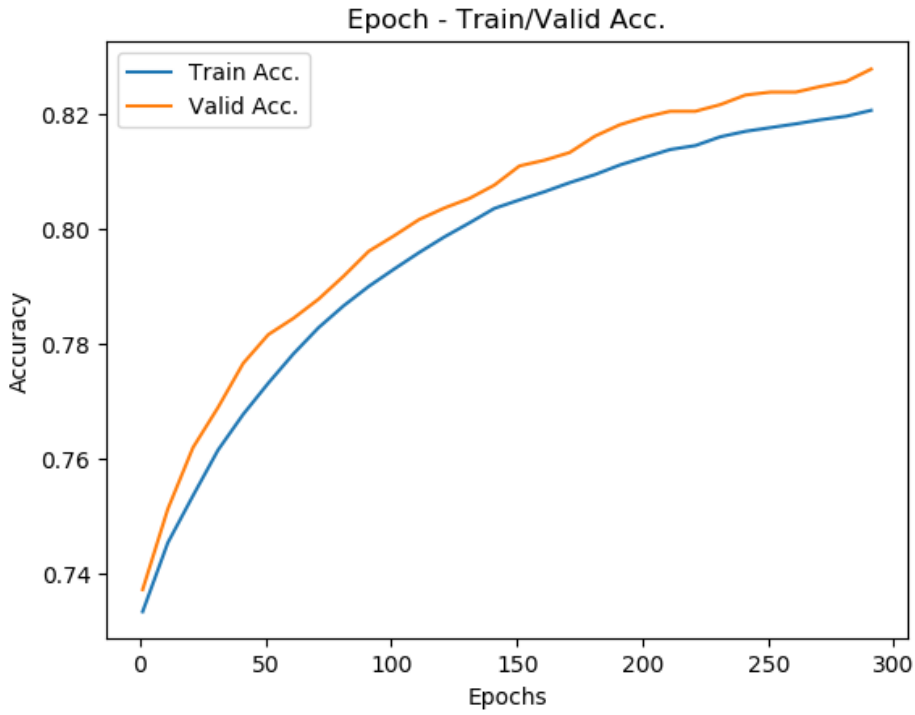
(b) **[Classification with different architectures]** Adjust the model settings (number of hidden layers, number of hidden nodes, number of epochs, learning rate, etc.) to get the best results over **FashionMNIST** using 'main_classification.py.' Report your top 3 best test accuracy with your fine-tuned hyperparameters. Show the plot of training and validation accuracy every epoch on each case. Also, describe how you determined the model structure or parameters in 4~5 lines.

Answer: Fill the blank in the table. Show the plot of training & validation accuracy through epochs.

	Model structure	# of epochs	Learning rate	Best Validation Acc.	Final Test Acc.

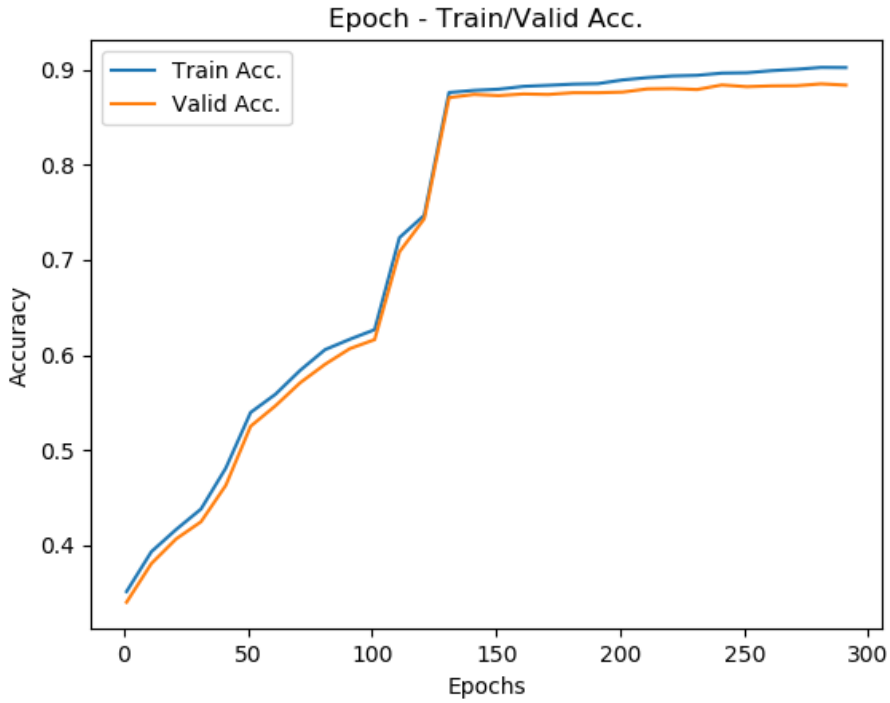
1st Best	**FC-1(784, 512)** **ReLU-1** **FC-2(512, 256)** **ReLU-2** **FC-3(256, 128)** **ReLU-3** **FC-4(128, 10)** **Sigmoid**	300	0.1	0.91	0.89
2nd Best	**FC-1(784, 256)** **ReLU-1** **FC-2(256, 64)** **ReLU-2** **FC-3(64, 10)** **Sigmoid**	300	0.1	0.89	0.87
3rd Best	**FC-1(784, 10)** **Sigmoid**	300	0.1	0.83	0.81

[3rd Best]



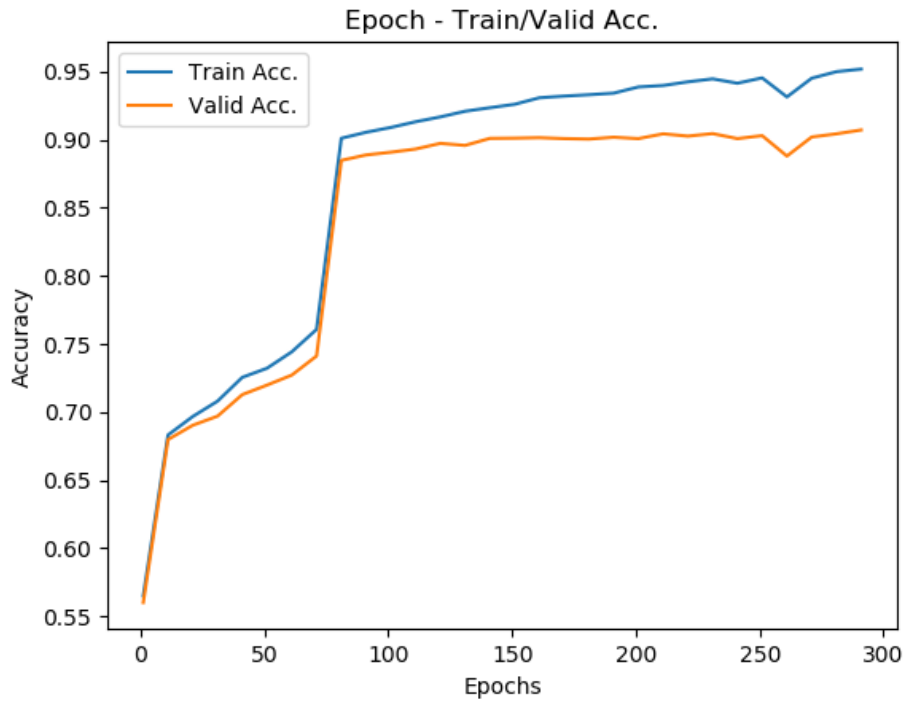
세번째로 성능이 좋은 모델의 파라미터는 num_epochs = 300, learning_rate = 0.1, batch_size = 512이다. 이 모델의 아키텍처는 FC layer(784, 10), sigmoid이다. num_epochs를 300 이상으로 증가시켜도 정확도가 크게 나아지지 않았다. 또한 마지막 layer가 sigmoid여서 learning rate가 작으면 정확도가 떨어지는 모습을 보였다. batch size는 초기값인 1000보다 512로 하였을 때 더 높은 정확도가 나왔다.

[2nd Best]



두번째로 성능이 좋은 모델의 파라미터는 num_epochs = 300, learning_rate = 0.1, batch_size = 126이다. 이 모델의 아키텍처는 FC-1(784, 256), ReLU, FC-2(256, 64), ReLU, FC-3(128,10), Sigmoid이다. Dataset이 이미지인 만큼 좀 더 복잡한 모델이 적절할 것 같아서 layer를 더 쌓았고, 빠른 학습을 위해 node 수는 2의 제곱수로 맞춰주었다. num_epoch을 300 이상으로 증가시켜도 정확도가 더 나아지지 않았다. 또한 분류 문제를 수행하기 위해 마지막 layer로 sigmoid를 사용하여 learning rate가 작으면 정확도가 떨어지는 모습을 보였다. batch_size는 128로 비교적 작은 값으로 설정하여야 학습이 잘 되는 모습을 보였다.

[1st Best]



가장 성능이 좋은 모델의 파라미터는 num_epochs = 300, learning_rate = 0.1, batch_size = 64이다. 이 모델의 아키텍처는 FC-1(784, 512), ReLU, FC-2(512, 256), ReLU, FC-3(256,128), ReLU, FC-4(128, 10), Sigmoid이다. 2nd Best 모델보다 hidden layer 개수를 증가시켜 복잡한 모델을 만들었다. layer 개수가 많아서 activation function으로는 sigmoid를 쓰지 않고 ReLU를 사용하였다. 이 모델도 2nd Best 모델, 3rd Best 모델들과 마찬가지로 분류 문제를 수행하기 위해 마지막 layer에 sigmoid를 사용한만큼 learning_rate를 0.1로 크게 잡았고, batch_size는 64로 작게 잡았다.