Introduction to Deep Neural Networks (Spring 2021)

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

Student ID 2016312761

Name 여혁수

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.

(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.

[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
import matplotlib.pyplot as plt

'''
Please refer the models/Linear_regressor.py and the PyTorch tutorials in
report file.
'''
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.loss_function = nn.MSELoss()
        self.fc1 = nn.Linear(in_features=input_dim, out_features=50, bias=True)
```

```
        self.fc2 = nn.Linear(50, 1, bias=True)

        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 = torch.rrelu(self.fc1(x))
        out = self.fc2(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)
                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,
print_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,
valid_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()
```

[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 file.
'''
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.linear = nn.Linear(in_features=input_dim, out_features=200)
        self.linear2 = nn.Linear(200, 128)
        self.linear3 = nn.Linear(128, 10)
```

```
        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 = torch.relu(self.linear(x))
        x = torch.relu(self.linear2(x))
        out = self.linear3(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, dtype=torch.float32)
        data_loader = DataLoader(x_tenser, batch_size=self.batch_size)

        # predict y with mini-batch
        cnt = 0
        with torch.no_grad():
            for batch_data in data_loader:
                batch_x = batch_data
                batch_pred_y = self.forward(batch_x)
                pred_y[cnt:cnt+len(batch_pred_y)] = batch_pred_y.max(1)[1]
                cnt+=len(batch_pred_y)
        #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,
print_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
Accuracy = %.3f' % valid_acc)
```

```
            # best model save
            if best_acc < valid_acc:
                print('Best Accuracy updated (%.4f => %.4f)' % (best_acc,
valid_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.

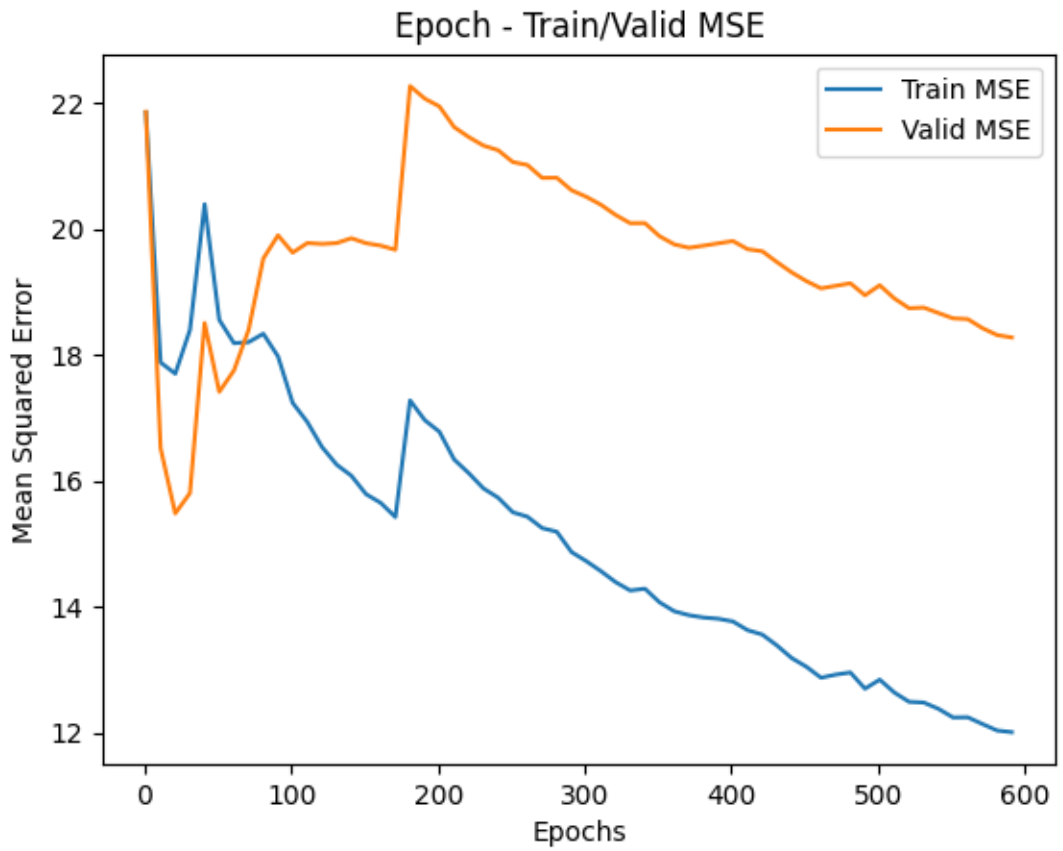
Answer: Fill the blank in the table. Show the plot of training & validation MSE through epochs.

	Model structure	# of epochs	Learning rate	Best Validation MSE	Final Test MSE

1st Best	**FC-1(14,30)** **RReLU** **FC-2(30,1)**	600	0.002	15.45	12.79
2nd Best	**FC-1(14,50)** **RReLU** **FC-2(50,50)** **RReLU** **FC-3(50,1)**	600	0.002	22.28	13.12
3rd Best	**FC-1(14,50)** **RReLU** **FC-2(50,30)** **RReLU** **FC-3(30,1)**	800	0.001	8.72	14.98

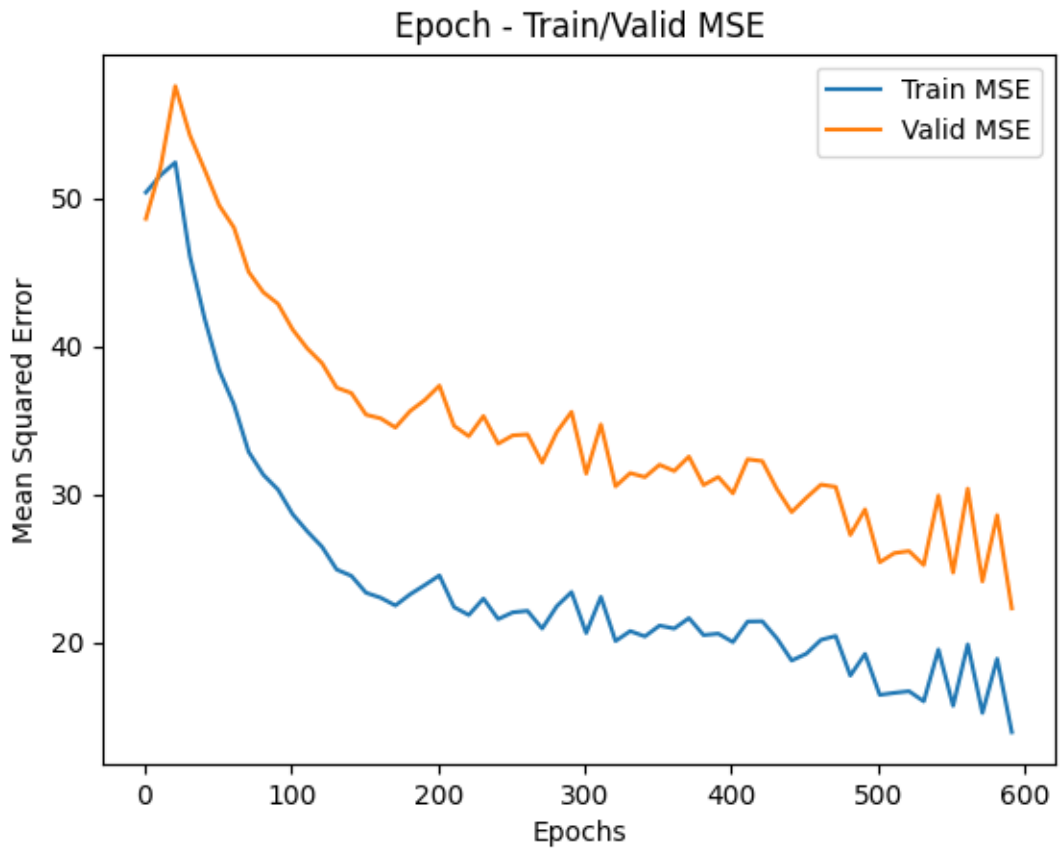
I chose the activation function for all models to RReLU rather than ReLU because it is fast and it keeps the positive values so I think it is proper to regression models. Also If I used ReLU, there are some dying ReLU error in several cases. I just randomly decide most of the parameters but I decreased the learning rate of first and third model from 0.01 to 0.002 because I see the value of the plot of train/valid MSE goes up and down severely if learning rate is high. My batch size is 10, first I defined batch size to 100 but test MSE improved when I decreased the batch size.

1 st model



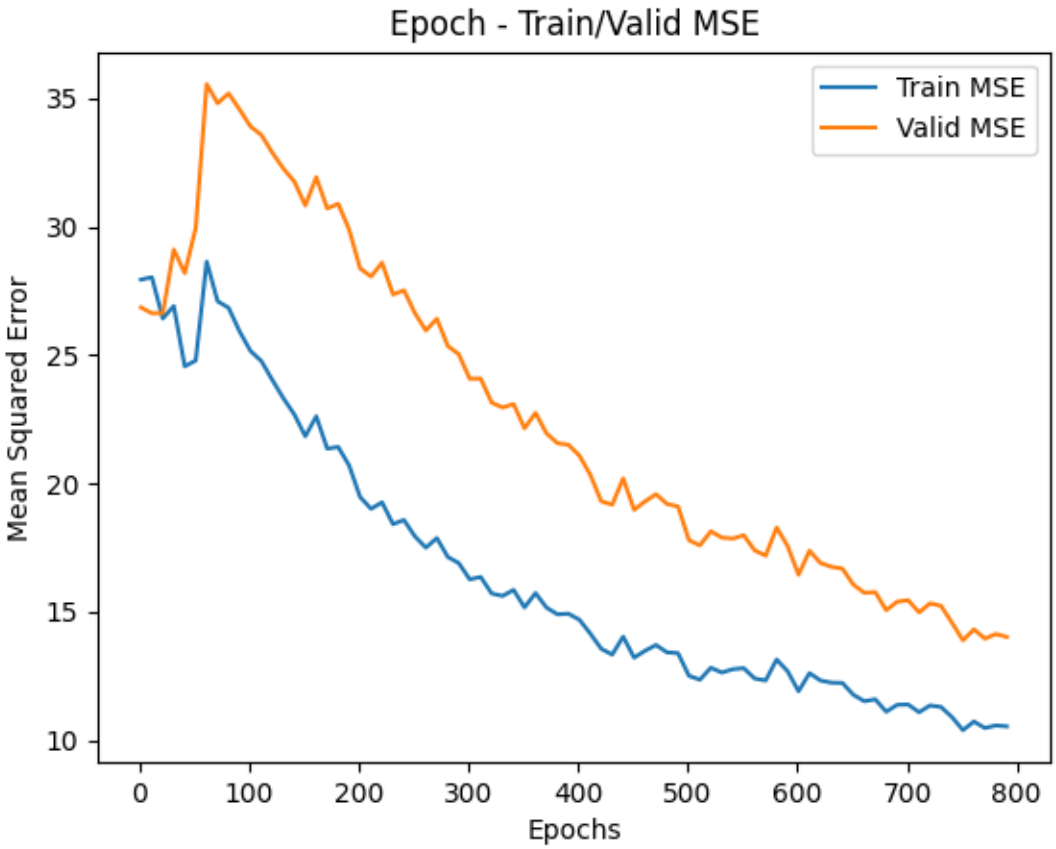
In this case, model has two hidden layers. I used fully connected layer for hidden layers and each hidden layer except last layer has RReLU activation function. First hidden layer increased hidden nodes 14 to 30. Then last hidden layer decreased nodes to 1. I set the number of epochs to 600. Then set the learning rate and batch size to 0.002 and 10 each.

2 nd model



In this case, model has three hidden layers. I used fully connected layer for hidden layers and each hidden layer except last layer has RReLU activation function. First hidden layer increased hidden nodes 14 to 50. Second hidden layer just kept the number of hidden nodes, and last hidden layer decreased nodes to 1. I set the number of epochs to 600. I set the learning rate and batch size to 0.002 and 10 each.

3 rd model



In this case, model has three hidden layers. I used fully connected layer for hidden layers and each hidden layer except last layer has RReLU activation function. First hidden layer increased hidden nodes 14 to 50. Second hidden layer decreased the number of hidden nodes to 30, and last hidden layer decreased nodes to 1. I set the number of epochs to 800. Then set the learning rate and batch size to 0.001 and 10 each.

(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.

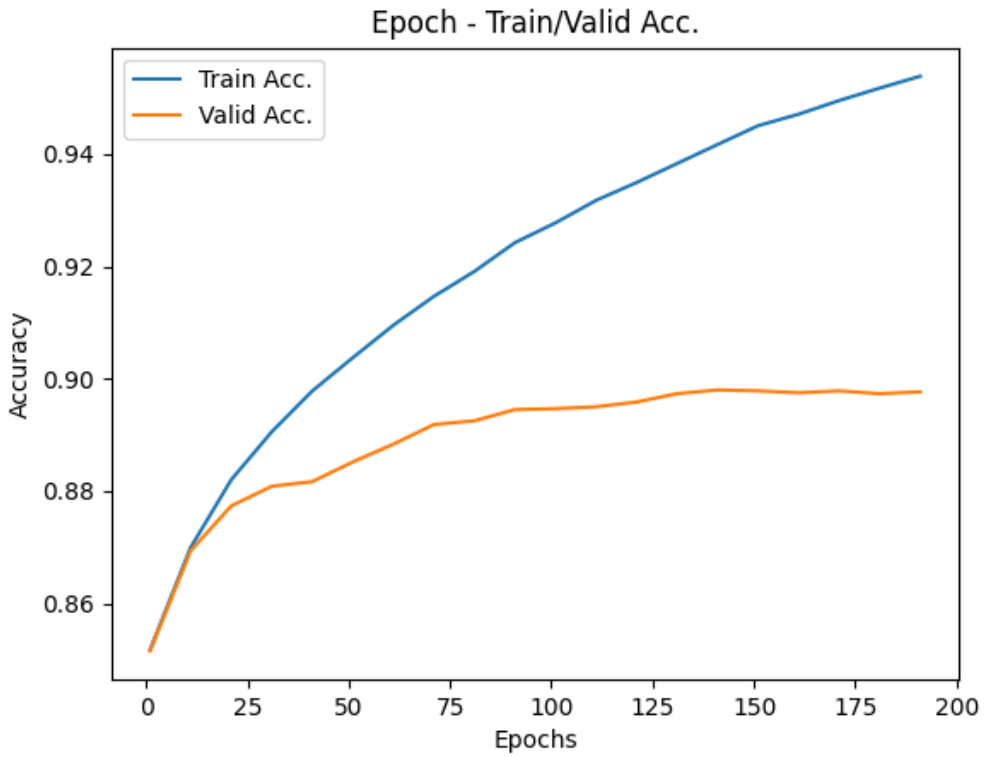
Answer: Fill the blank in the table. Show the plot of training & validation accuracy through epochs.

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

1st Best	**FC-1(784,200)** **ReLU** **FC-2(200,10)**	190	0.02	0.90	0.89
2nd Best	**FC-1(784,200)** **ReLU** **FC-2(200,128)** **ReLU** **FC-3(128,10)**	130	0.01	0.90	0.88
3rd Best	**FC-1(784,200)** **ReLU** **FC-2(200,128)** **ReLU** **FC-3(128,64)** **ReLU** **FC-4(64,10)**	90	0.01	0.89	0.88

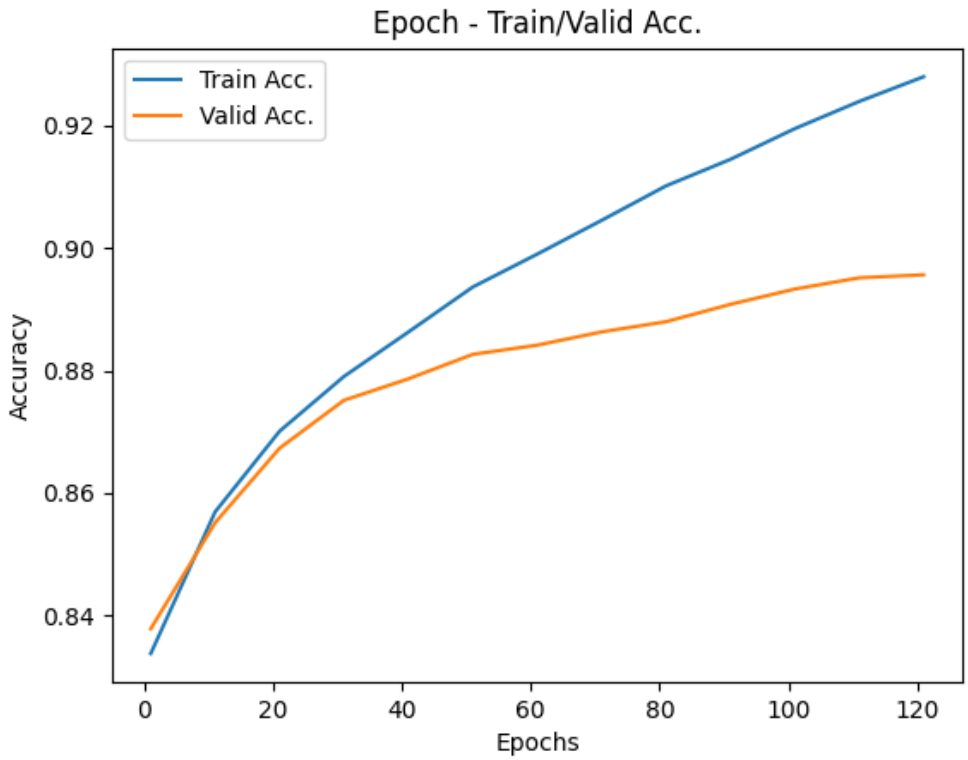
I chose the activation function for all models to ReLU because it is faster than Tanh, sigmoid and it keeps the positive values so I think it is proper to hidden layers. I just randomly decide most of the parameters because I saw that whatever I decide the value of parameters, accuracy can improve until certain level near 0.89 when I modulate # of epochs. In case of reg_lambda, I used default value 0.01 to all models.

1 st model



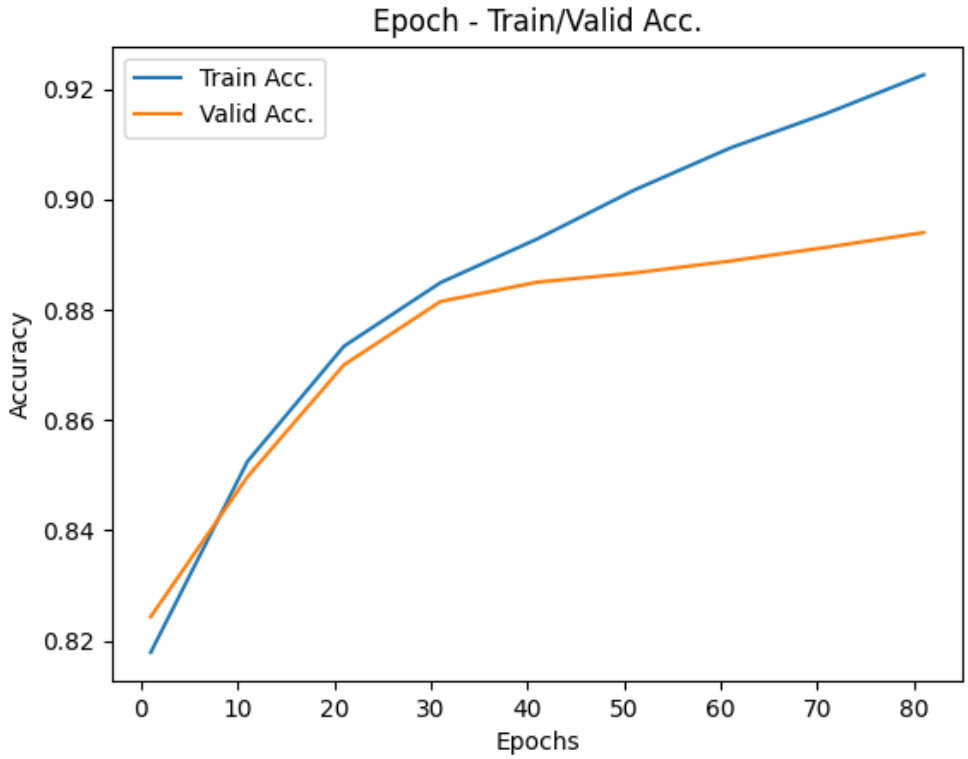
In this case, model has two hidden layers. I used fully connected layer for hidden layers and each hidden layer except last layer has ReLU activation function. First hidden layer decreased hidden nodes 784 to 200. Then second hidden layer directly decreased the number of hidden nodes to 10. I set the number of epochs to 200. Then set the learning rate and batch size to 0.02 and 100 each.

2 nd model



In this case, model has three hidden layers. I used fully connected layer for hidden layers and each hidden layer except last layer has ReLU activation function. First hidden layer decreased hidden nodes 784 to 200. Second hidden layer decreased the number of hidden nodes to 128, and last hidden layer decreased nodes to 10. I set the number of epochs to 130. Then set the learning rate and batch size to 0.01 and 64 each.

3 rd model



In this case, model has four hidden layers. I used fully connected layer for hidden layers and each hidden layer except last layer has ReLU activation function. First hidden layer decreased hidden nodes 784 to 200. Second and third hidden layer make the number of hidden nodes to half each, and last hidden layer decreased nodes to 10. I set the number of epochs to 90. Then set the learning rate and batch size to 0.01 and 100 each.