**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.’
2. **[Regression]** Implement \_\_init\_\_, forward, and predict method functions in ‘MLP\_regressor.py.’
3. **[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() |

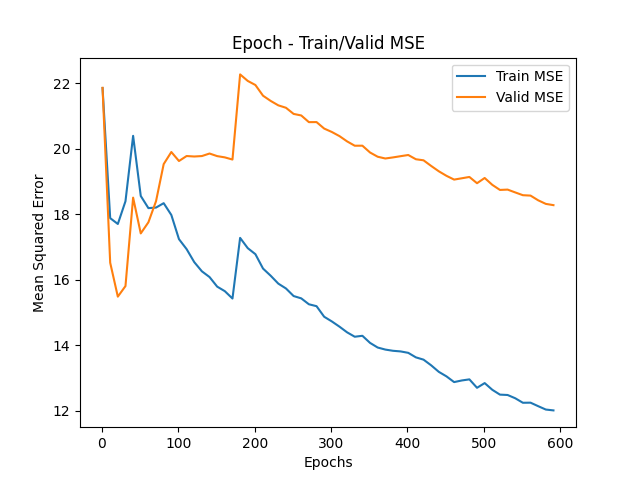
1. **[20 pts]** Report the experiment results for each dataset.
2. **[Regression with different architectures]** Adjust the model settings (number of hidden layers, number of hidden nodes, number of epochs, learning rate, etc.) to obtain the best results over the **House dataset** using ‘main\_classification.py.’ 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 |

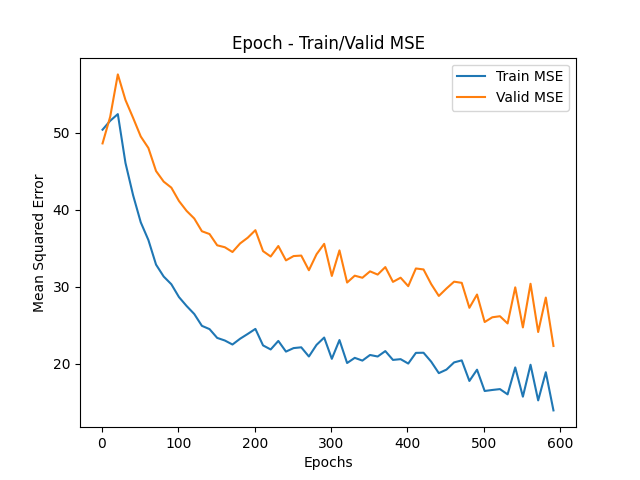
I chose the activation function for all models to RReLU rather than ReLU because it is fast and it keeps the positive values so I think it is proper to regression models. Also If I used ReLU, there are some dying ReLU error in several cases. I just randomly decide most of the parameters but I decreased 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.

1st model



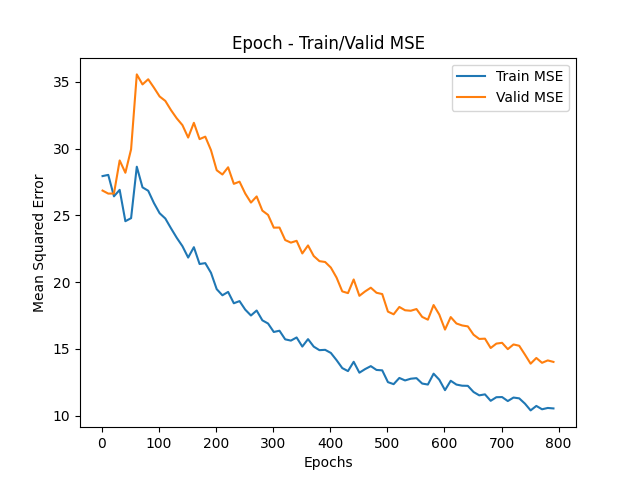
In this case, model has two hidden layers. I used fully connected layer for hidden layers and each hidden layer except last layer has RReLU activation function. First hidden layer increased hidden nodes 14 to 30. Then last hidden layer decreased nodes to 1. I set the number of epochs to 600. Then set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

3rd 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.

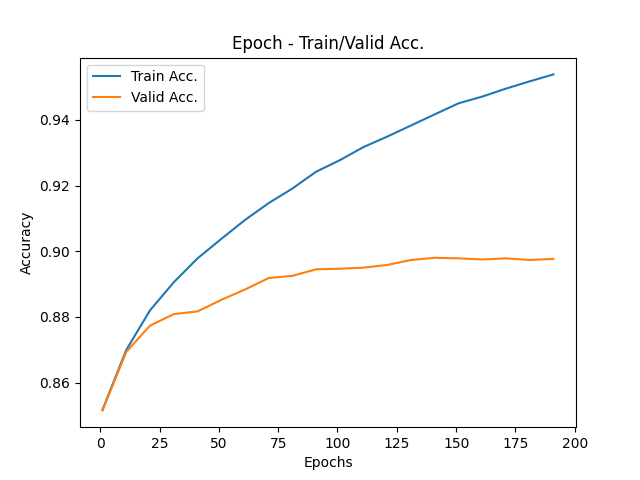
1. **[Classification with different architectures]** Adjust the model settings (number of hidden layers, number of hidden nodes, number of epochs, learning rate, etc.) to get the best results over **FashionMNIST** using ‘main\_classification.py.’ Report your top 3 best test accuracy with your fine-tuned 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 |

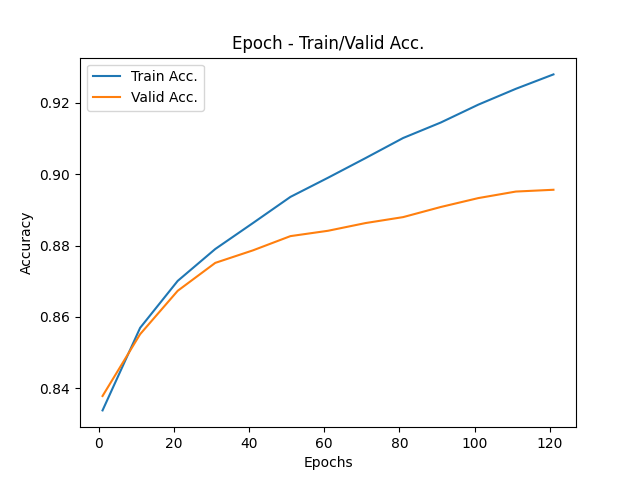
I chose the activation function for all models to ReLU because it is faster than Tanh, sigmoid and it keeps the positive values so I think it is proper to hidden layers. I just randomly decide most of the parameters because I saw that whatever I decide the value of parameters, accuracy can improve until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

1st model



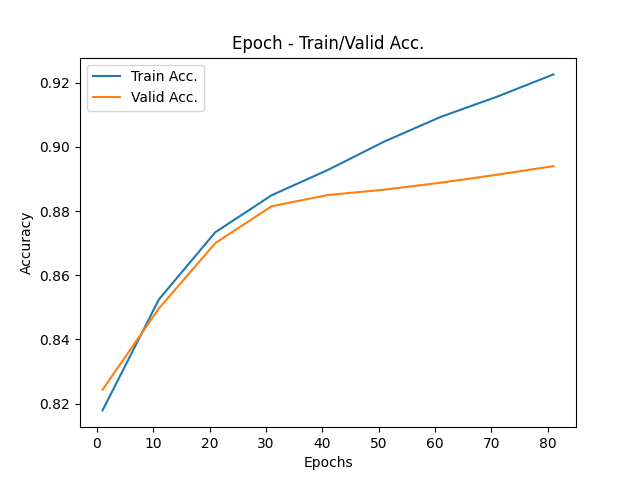
In this case, model has two hidden layers. I used fully connected layer for hidden layers and each hidden layer except last layer has ReLU activation function. First hidden layer decreased hidden nodes 784 to 200. Then second hidden layer directly decreased the number of hidden nodes to 10. I set the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

2nd 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.

3rd 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.