CSED/AIGS 526: Data Mining

Your name: Jaeyoon Sim

Student ID: 20222421

Use late days quota(Yes/No): No

All problems are configured to be performed in one main function.

```
def main():
    # Download MNIST datset
    train_data = datasets.MNIST(root = './data/',
                                train=True,
                                download=True,
                                transform=transforms.ToTensor())
    test_data = datasets.MNIST(root = './data/',
                                train=False,
                                download=True,
                                transform=transforms.ToTensor())

    p1(train_data, test_data) # Problem 1.1

    p2(train_data, test_data) # Problem 1.2

    p3_p4(train_data, test_data) # Problem 1.3, 1.4

    p5_p6(train_data, test_data) # Problem 1.5, 1.6

if __name__ == "__main__":
    main()
```

Listing 1: main function

This problem is a problem of designing a single layer natural network that classifies y based on x. A single layer natural network was implemented through the class as follows.

```
class SingleLayerNeuralNet(nn.Module): # Problem 1.1
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(28*28, 10)

    def forward(self, x):
        x = self.linear(x)
        out = F.softmax(x)
        return out
```

Listing 2: single layer neural network class

The following is the complete code that performs problem 1.1. Here, train and test are conducted using MNIST data, and at the end, loss function according to epoch can be drawn. Through the Early stopping technique, code was written to stop when loss increased more than a specific number of times. However, instead of proceeding 100 epochs to the end without stopping here, the message was output in the stopping section.

```
def p1(train_data, test_data):
    batch_size = 32 # Batch size
    learning_rate = 0.001 # Learning rate
    num_epochs = 100 # Epochs

    train_loader = DataLoader(dataset=train_data,
                              batch_size=batch_size,
                              shuffle=True)

    test_loader = DataLoader(dataset=test_data,
                              batch_size=batch_size,
                              shuffle=True)
```

```
13
14      model = SingleLayerNeuralNet().to(device) # Model(Single layer neural network)
15      loss_function = nn.CrossEntropyLoss().to(device) # Loss function(Cross entropy
        )
16      optimizer = optim.Adam(model.parameters(), lr=learning_rate) # Optimizer(Adam)
17
18      last_loss = 0
19      trigger_time = 0
20      patience = 2
21      train_loss =[]
22      num_train = len(train_loader.dataset)
23
24      # Training
25      model.train()
26      for epoch in range(1, num_epochs+1):
27          correct_train = 0
28          batch_loss = 0
29          for data, label in train_loader:
30              data = data.to(device)
31              label = label.to(device)
32
33              data = data.view(data.shape[0], -1) # [32, 28*28]
34
35              # Set the parameter gradients to zero for training
36              optimizer.zero_grad()
37
38              # Forward
39              output = model(data) # [32, 10]
40              pred_train = output.data.max(1)[1]
41              correct_train += pred_train.eq(label.data).sum().item()
42
43              loss = loss_function(output, label)
44              batch_loss += loss
45
46              # Backword
47              loss.backward()
48
49              # Update
50              optimizer.step()
51
52          current_loss = batch_loss / len(train_loader) # Current loss for mini-
        batch
53
54          # Early Stopping
55          if current_loss > last_loss:
56              trigger_time += 1
57              if trigger_time >= patience:
58                  print(f"[Epoch {epoch}] CONVERGE!")
59          else:
60              trigger_time = 0
61
62          last_loss = current_loss
63
64          accuracy_train = correct_train / num_train
65          train_loss.append(current_loss.item())
66
67          if epoch % 10 == 0:
68              print(f"[Epoch {epoch}] Train Loss : {current_loss:.3f} Accuracy : {
        accuracy_train:.3f}")
```

```
69
70      correct_test = 0
71      num_test = len(test_loader.dataset)
72
73      # Test
74      model.eval()
75      for data, label in test_loader:
76          data = data.to(device)
77          label = label.to(device)
78
79          data = data.view(data.shape[0], -1)
80          output = model(data)
81          pred_test = output.data.max(1)[1]
82          correct_test += pred_test.eq(label.data).sum()
83
84      accuracy_test = correct_test /num_test
85      print(f"Training Accuracy : {accuracy_train:.3f}, Test Accuracy : {
    accuracy_test:.3f}")
```

Listing 3: train and test code for problem 1.1

The train result of the model with 0.001 learning rate was shown every 10 epoch, and the train accuracy and test accuracy were output. It shows about 93% test accuracy, and I think this model can satisfy the convergence with more epochs.

```
[Epoch 10] Train Loss : 1.535 Accuracy : 0.934
[Epoch 20] Train Loss : 1.527 Accuracy : 0.941
[Epoch 30] Train Loss : 1.523 Accuracy : 0.944
[Epoch 40] Train Loss : 1.520 Accuracy : 0.947
[Epoch 50] Train Loss : 1.518 Accuracy : 0.949
[Epoch 60] Train Loss : 1.516 Accuracy : 0.950
[Epoch 70] Train Loss : 1.515 Accuracy : 0.951
[Epoch 80] Train Loss : 1.514 Accuracy : 0.952
[Epoch 90] Train Loss : 1.513 Accuracy : 0.953
[Epoch 100] Train Loss : 1.512 Accuracy : 0.953
Training Accuracy : 0.953, Test Accuracy : 0.933
```

Figure 1: Training and testing results (learning rate = 0.001)

Next is the code that plots the loss.

```
1       # Plot the changes of loss
2       plt.subplots(figsize=(5,5))
3       plt.suptitle("Plot the changes of loss w.r.t. the # of epochs\n\n", fontsize
        =16, fontweight='bold')
4       plt.plot(train_loss, color='purple')
5       plt.xlabel('The # of epochs', fontweight='bold')
6       plt.ylabel('Loss', fontweight='bold')
7       plt.show()
```

Listing 4: plot the loss

Finally, this is the result of plotting the loss that occurred for each epoch in the train process.

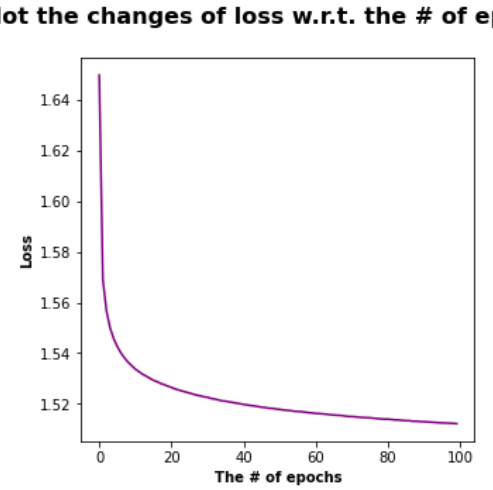

Figure 2: Loss function with respect to epoch (learning rate = 0.001)

And learning was conducted by changing the learning rate to 0.1 according to the conditions in question. As a result, due to the increased learning rate, there was a problem that it could not converge properly to the global minimum.

```
[Epoch 10] Train Loss : 1.565 Accuracy : 0.896
[Epoch 13] CONVERGE!
[Epoch 20] Train Loss : 1.553 Accuracy : 0.908
[Epoch 21] CONVERGE!
[Epoch 27] CONVERGE!
[Epoch 28] CONVERGE!
[Epoch 30] Train Loss : 1.549 Accuracy : 0.912
[Epoch 36] CONVERGE!
[Epoch 40] Train Loss : 1.548 Accuracy : 0.913
[Epoch 41] CONVERGE!
[Epoch 50] CONVERGE!
[Epoch 50] Train Loss : 1.548 Accuracy : 0.913
[Epoch 55] CONVERGE!
[Epoch 60] Train Loss : 1.543 Accuracy : 0.918
[Epoch 65] CONVERGE!
[Epoch 70] CONVERGE!
[Epoch 70] Train Loss : 1.541 Accuracy : 0.920
[Epoch 71] CONVERGE!
[Epoch 79] CONVERGE!
[Epoch 80] CONVERGE!
[Epoch 80] Train Loss : 1.542 Accuracy : 0.919
[Epoch 90] Train Loss : 1.536 Accuracy : 0.925
[Epoch 93] CONVERGE!
[Epoch 100] Train Loss : 1.544 Accuracy : 0.917
Training Accuracy : 0.917, Test Accuracy : 0.913
```

Figure 3: Training and testing results (learning rate = 0.1)

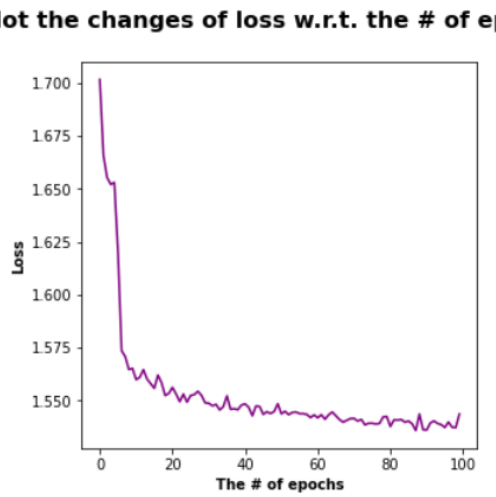

Figure 4: Loss function with respect to epoch (learning rate = 0.1)

Question 1.2, Homework 4, CSED/AIGS526

This problem is a problem of designing a multi layer natural network that classifies y based on x. A multi layer natural network was implemented through the class as follows. It has one hidden layer with 128 hidden nodes.

```
class MultiLayerNeuralNet1(nn.Module): # Problem 1.2
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = F.sigmoid(x)
        x = self.fc2(x)
        out = F.softmax(x)
        return out
```

Listing 5: multi layer with one hidden layer neural network class

The following is the complete code that performs problem 1.2. Here, train and test are conducted using MNIST data, and at the end, loss function according to epoch can be drawn.

```
def p2(train_data, test_data):
    batch_size = 32 # Batch size
    learning_rate = 0.001 # Learning rate
    num_epochs = 100 # Epochs

    train_loader = DataLoader(dataset=train_data,
                              batch_size=batch_size,
                              shuffle=True)

    test_loader = DataLoader(dataset=test_data,
                              batch_size=batch_size,
                              shuffle=True)

    model = MultiLayerNeuralNet1().to(device) # Model(Multi layer neural network)
    loss_function = nn.CrossEntropyLoss().to(device) # Loss function(Cross entropy
    )
    optimizer = optim.Adam(model.parameters(), lr=learning_rate) # Optimizer(Adam)

    last_loss = 0
    trigger_time = 0
    patience = 2
    train_loss =[]
    num_train = len(train_loader.dataset)

    # Training
    model.train()
    for epoch in range(1, num_epochs+1):
        correct_train = 0
        batch_loss = 0
        for data, label in train_loader:
            data = data.to(device)
            label = label.to(device)

            data = data.view(data.shape[0], -1) # [32, 28*28]

            # Set the parameter gradients to zero for training
            optimizer.zero_grad()
```

```
37
38              # Forward
39              output = model(data) # [32, 10]
40              pred_train = output.data.max(1)[1]
41              correct_train += pred_train.eq(label.data).sum().item()
42
43              loss = loss_function(output, label)
44              batch_loss += loss
45
46              # Backword
47              loss.backward()
48
49              # Update
50              optimizer.step()
51
52          current_loss = batch_loss / len(train_loader) # Current loss for mini-
    batch
53
54          # Early Stopping
55          if current_loss > last_loss:
56              trigger_time += 1
57              if trigger_time >= patience:
58                  print(f"[Epoch {epoch}] CONVERGE!")
59          else:
60              trigger_time = 0
61
62          last_loss = current_loss
63
64          accuracy_train = correct_train / num_train
65          train_loss.append(current_loss.item())
66
67          if epoch % 10 == 0:
68              print(f"[Epoch {epoch}] Train Loss : {current_loss:.3f} Accuracy : {
    accuracy_train:.3f}")
69
70      correct_test = 0
71      num_test = len(test_loader.dataset)
72
73      # Test
74      model.eval()
75      for data, label in test_loader:
76          data = data.to(device)
77          label = label.to(device)
78
79          data = data.view(data.shape[0], -1)
80          output = model(data)
81          pred_test = output.data.max(1)[1]
82          correct_test += pred_test.eq(label.data).sum()
83
84      accuracy_test = correct_test /num_test
85      print(f"Training Accuracy : {accuracy_train:.3f}, Test Accuracy : {
    accuracy_test:.3f}")
```

Listing 6: train and test code for problem 1.2

The train result of the model with 0.001 learning rate was shown every 10 epoch, and the train accuracy and test accuracy were output. This model took about 50 epochs and 96 epochs to converge and after this epoch, the result get better slowly. It shows about 99% train accuracy and

97% test accuracy, and I think this model gets better than problem 1.1 model.

```
[Epoch 10] Train Loss : 1.484 Accuracy : 0.981
[Epoch 20] Train Loss : 1.472 Accuracy : 0.991
[Epoch 30] Train Loss : 1.468 Accuracy : 0.994
[Epoch 40] Train Loss : 1.466 Accuracy : 0.995
[Epoch 50] CONVERGE!
[Epoch 50] Train Loss : 1.466 Accuracy : 0.995
[Epoch 60] Train Loss : 1.465 Accuracy : 0.996
[Epoch 70] Train Loss : 1.465 Accuracy : 0.996
[Epoch 80] Train Loss : 1.465 Accuracy : 0.996
[Epoch 90] Train Loss : 1.465 Accuracy : 0.997
[Epoch 96] CONVERGE!
[Epoch 100] Train Loss : 1.464 Accuracy : 0.997
Training Accuracy : 0.997, Test Accuracy : 0.977
```

Figure 5: Training and testing results

Finally, this is the result of plotting the loss that occurred for each epoch in the train process.

```
    # Plot the changes of loss
    plt.subplots(figsize=(5,5))
    plt.suptitle("Plot the changes of loss w.r.t. the # of epochs\n\n", fontsize
    =16, fontweight='bold')
    plt.plot(train_loss, color='purple')
    plt.xlabel('The # of epochs', fontweight='bold')
    plt.ylabel('Loss', fontweight='bold')
    plt.show()
```

Listing 7: plot the loss

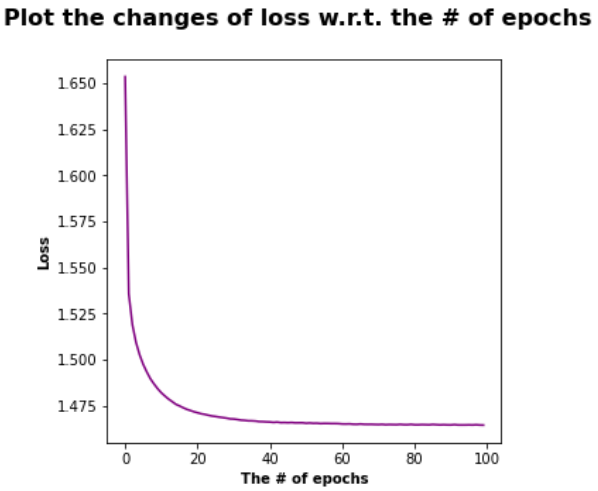

Figure 6: Loss function with respect to epoch

Question 1.3, Homework 4, CSED/AIGS526

This problem is a problem of designing a multi layer natural network that classifies y based on x. A multi layer natural network was implemented through the class as follows. It has two hidden layer with 128 and 4 hidden nodes. And the rest of the setting should be the same as above.

```
class MultiLayerNeuralNet2(nn.Module): # Problem 1.3, 1.4
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 4)
        self.fc3 = nn.Linear(4, 10)

    def forward(self, x):
        x1 = self.fc1(x)
        x1 = F.sigmoid(x1)
        x2 = self.fc2(x1)
        x2 = F.sigmoid(x2)
        x3 = self.fc3(x2)
        out = F.softmax(x3)
        return out, x2
```

Listing 8: multi layer with two hidden layers(128/4) neural network class

The following is the complete code that performs problem 1.3 and 1.4. Here, train and test are conducted using MNIST data, and at the end, we save the activated values from the 4 hidden nodes and take average of each node per class for testing data.

```
def p3_p4(train_data, test_data):
    batch_size = 32 # Batch size
    learning_rate = 0.001 # Learning rate
    num_epochs = 100 # Epochs

    train_loader = DataLoader(dataset=train_data,
                              batch_size=batch_size,
                              shuffle=True)

    test_loader = DataLoader(dataset=test_data,
                              batch_size=batch_size,
                              shuffle=True)

    model = MultiLayerNeuralNet2().to(device) # Model(Multi layer neural network)
    loss_function = nn.CrossEntropyLoss().to(device) # Loss function(Cross entropy
    )
    optimizer = optim.Adam(model.parameters(), lr=learning_rate) # Optimizer(Adam)

    last_loss = 0
    trigger_time = 0
    patience = 2
    train_loss =[]
    num_train = len(train_loader.dataset)

    # ================================== Training
    ==================================
    model.train()
    for epoch in range(1, num_epochs+1):
        correct_train = 0
        batch_loss = 0

        for data, label in train_loader:
            data = data.to(device)
```

```
            label = label.to(device)

            data = data.view(data.shape[0], -1) # [32, 28*28]

            # Set the parameter gradients to zero for training
            optimizer.zero_grad()

            # Forward
            output, _ = model(data) # [32, 10], [32, 4]
            pred_train = output.data.max(1)[1] # [32]
            correct_train += pred_train.eq(label.data).sum().item()

            loss = loss_function(output, label)
            batch_loss += loss

            # Backword
            loss.backward()

            # Update
            optimizer.step()

        current_loss = batch_loss / len(train_loader) # Current loss for mini-batch

        # Early Stopping
        if current_loss > last_loss:
            trigger_time += 1
            if trigger_time >= patience:
                print(f"[Epoch {epoch}] CONVERGE!")
        else:
            trigger_time = 0

        last_loss = current_loss

        accuracy_train = correct_train / num_train
        train_loss.append(current_loss.item())

        if epoch % 10 == 0:
            print(f"[Epoch {epoch}] Train Loss : {current_loss:.3f} Accuracy : {accuracy_train:.3f}")


    correct_test = 0
    num_test = len(test_loader.dataset)

    prediction_counters = np.zeros(10, dtype=np.int64)
    activated_values_information = np.zeros((10, 4), dtype=np.float64)

    # ================================== Test ==================================
    model.eval()
    for data, label in test_loader:
        data = data.to(device)
        label = label.to(device)
        data = data.view(data.shape[0], -1)

        # Forward
        output, activated_values = model(data)
        pred_test = output.data.max(1)[1]
        correct_test += pred_test.eq(label.data).sum()
```

```
89
90          # Build the information matrix of 4 activated values w.r.t. predicted
    labels
91          for i in range(data.shape[0]):
92              pred = pred_test[i].item() # Predicted label (0 ~ 9)
93              prediction_counters[pred] += 1
94
95              activated_values_list = activated_values[i].tolist()
96              activated_values_information[pred] = [x + y for x, y in zip(
    activated_values_information[pred], activated_values_list)]
97
98      # Calculate the average of 4 activated values
99      for i in range(10):
100         for j in range(4):
101             activated_values_information[i][j] = activated_values_information[i][j
    ] / prediction_counters[i]
102
103     # Print the result of activate values
104     print(f"The activated values from the 4 hidden nodes for testing samples")
105     for i in range(10):
106         print(f"[Number {i}] : {activated_values_information[i]} with {
    prediction_counters[i]} samples")
107
108     accuracy_test = correct_test / num_test
109     print(f"Training Accuracy : {accuracy_train:.3f}, Test Accuracy : {
    accuracy_test:.3f}")
```

Listing 9: train and test code for problem 1.3 and 1.4

The train result of the model was shown every 10 epoch, and the train accuracy and test accuracy were output. This model took about 75 epochs to converge and after this epoch, the result get better slowly. It shows about 99% train accuracy and 96% test accuracy, and I think this model seems to perform about 1% less than the previous results. I thought that the accuracy could be reduced in the process of compressing information into 4 hidden nodes.

```
[Epoch 10] Train Loss : 1.499 Accuracy : 0.971
[Epoch 20] Train Loss : 1.478 Accuracy : 0.985
[Epoch 30] Train Loss : 1.473 Accuracy : 0.989
[Epoch 40] Train Loss : 1.472 Accuracy : 0.990
[Epoch 50] Train Loss : 1.470 Accuracy : 0.991
[Epoch 60] Train Loss : 1.469 Accuracy : 0.992
[Epoch 70] Train Loss : 1.469 Accuracy : 0.992
[Epoch 75] CONVERGE!
[Epoch 80] Train Loss : 1.469 Accuracy : 0.993
[Epoch 90] Train Loss : 1.468 Accuracy : 0.993
[Epoch 100] Train Loss : 1.468 Accuracy : 0.993
Training Accuracy : 0.993, Test Accuracy : 0.969
```

Figure 7: Training and testing results

In this problem, when we do the testing, we have to save the activated values from 4 hidden nodes and take average of each node per class. The following is the result of averaging the values for each digit number output by four hidden nodes. Looking at the results of the test, it can be seen that in the case of 0, the fourth node is greatly activated. 1 is the first and second nodes, 2 is the second and fourth nodes, and 3 is the third and fourth nodes are greatly activated. In this way, it can be seen that different classes are classified according to the activated values among the four nodes.

```
The activated values from the 4 hidden nodes for testing samples
[Number 0] : [0.01098533 0.00916396 0.00516801 0.99609632] with 995 samples
[Number 1] : [0.98215972 0.96132253 0.01527625 0.00283346] with 1134 samples
[Number 2] : [0.64237966 0.98120008 0.05477197 0.98972222] with 1013 samples
[Number 3] : [0.01893861 0.08967653 0.99187905 0.98741669] with 1035 samples
[Number 4] : [0.01072293 0.01896514 0.01739268 0.00508281] with 978 samples
[Number 5] : [0.97706166 0.00810065 0.98807196 0.98676917] with 879 samples
[Number 6] : [0.99079883 0.01159389 0.00761131 0.97357056] with 962 samples
[Number 7] : [0.01110013 0.97900144 0.97216692 0.00883626] with 1039 samples
[Number 8] : [0.97823346 0.0615209  0.98103784 0.02147384] with 958 samples
[Number 9] : [0.00724922 0.01487238 0.98146378 0.00738715] with 1007 samples
```

Figure 8: Activated values from the 4 hidden nodes with testing samples

Question 1.5, Homework 4, CSED/AIGS526

This problem is a problem of designing a multi layer natural network that classifies y based on x. A multi layer natural network was implemented through the class as follows. It has two hidden layer with 128 and 2 hidden nodes. And the rest of the setting should be the same as above.

```
class MultiLayerNeuralNet3(nn.Module): # Problem 1.5, 1.6
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 2)
        self.fc3 = nn.Linear(2, 10)

    def forward(self, x):
        x1 = self.fc1(x)
        x1 = F.sigmoid(x1)
        x2 = self.fc2(x1)
        x2 = F.sigmoid(x2)
        x3 = self.fc3(x2)
        out = F.softmax(x3)
        return out, x2
```

Listing 10: multi layer with two hidden layers(128/2) neural network class

The following is the complete code that performs problem 1.5 and 1.6. Here, train and test are conducted using MNIST data, and at the end, we save the activated values from the 2 hidden nodes and take average of each node per class for training data.

```
def p5_p6(train_data, test_data):
    batch_size = 32 # Batch size
    learning_rate = 0.001 # Learning rate
    num_epochs = 100 # Epochs

    train_loader = DataLoader(dataset=train_data,
                              batch_size=batch_size,
                              shuffle=True)

    test_loader = DataLoader(dataset=test_data,
                              batch_size=batch_size,
                              shuffle=True)

    model = MultiLayerNeuralNet3().to(device) # Model(Multi layer neural network)
    loss_function = nn.CrossEntropyLoss().to(device) # Loss function(Cross entropy
    )
    optimizer = optim.Adam(model.parameters(), lr=learning_rate) # Optimizer(Adam)

    last_loss = 0
    trigger_time = 0
    patience = 2
    train_loss =[]
    num_train = len(train_loader.dataset)

    # ================================== Training
    ==================================
    model.train()
    for epoch in range(1, num_epochs+1):
        correct_train = 0
        batch_loss = 0

        prediction_counters = np.zeros(10, dtype=np.int64)
        activated_values_information = np.zeros((10, 2), dtype=np.float64)
```

```
32
33          # Initialization dictionary for 2D scatter plot
34          activated_values_dict = {}
35          for digit in range(10):
36              activated_values_dict.setdefault(digit)
37
38          for data, label in train_loader:
39              data = data.to(device)
40              label = label.to(device)
41
42              data = data.view(data.shape[0], -1) # [32, 28*28]
43
44              # Set the parameter gradients to zero for training
45              optimizer.zero_grad()
46
47              # Forward
48              output, activated_values = model(data) # [32, 10], [32, 2]
49              pred_train = output.data.max(1)[1] # [32]
50              correct_train += pred_train.eq(label.data).sum().item()
51
52              loss = loss_function(output, label)
53              batch_loss += loss
54
55              # Backword
56              loss.backward()
57
58              # Update
59              optimizer.step()
60
61              # Build the information matrix of 2 activated values w.r.t. predicted
    labels
62              if epoch == num_epochs:
63                  for i in range(data.shape[0]):
64                      pred = pred_train[i].item() # Predicted label (0 ~ 9)
65                      prediction_counters[pred] += 1
66
67                      activated_values_list = activated_values[i].tolist()
68                      activated_values_information[pred] = [x + y for x, y in zip(
    activated_values_information[pred], activated_values_list)]
69                      activated_values_list = np.reshape(activated_values_list, (1,
    2))
70
71                      # Make dictionary for 2 activated values w.r.t. digit to save
    the values
72                      if activated_values_dict[pred] is None:
73                          activated_values_dict[pred] = activated_values_list
74                      else:
75                          activated_values_dict[pred] = np.concatenate((
    activated_values_dict[pred], activated_values_list), axis=0)
76
77          if epoch == num_epochs:
78              # Calculate the average of 2 activated values
79              for i in range(10):
80                  for j in range(2):
81                      activated_values_information[i][j] =
    activated_values_information[i][j] / prediction_counters[i]
82
83              # Print the result of activated values
84              print(f"The activated values from the 2 hidden nodes for training
```

```
    samples")
85              for i in range(10):
86                  print(f"[Number {i}] : {activated_values_information[i]} with {
    prediction_counters[i]} samples")
87
88          current_loss = batch_loss / len(train_loader) # Current loss for mini-
    batch
89
90          # Early Stopping
91          if current_loss > last_loss:
92              trigger_time += 1
93              if trigger_time >= patience:
94                  print(f"[Epoch {epoch}] CONVERGE!")
95          else:
96              trigger_time = 0
97
98          last_loss = current_loss
99
100         accuracy_train = correct_train / num_train
101         train_loss.append(current_loss.item())
102
103         if epoch % 10 == 0:
104             print(f"[Epoch {epoch}] Train Loss : {current_loss:.3f} Accuracy : {
    accuracy_train:.3f}")
105
106     correct_test = 0
107     num_test = len(test_loader.dataset)
108
109     # ================================== Test ==================================
110     model.eval()
111     for data, label in test_loader:
112         data = data.to(device)
113         label = label.to(device)
114
115         data = data.view(data.shape[0], -1)
116         output, _ = model(data)
117         pred_test = output.data.max(1)[1]
118         correct_test += pred_test.eq(label.data).sum()
119
120     accuracy_test = correct_test / num_test
121     print(f"Training Accuracy : {accuracy_train:.3f}, Test Accuracy : {
    accuracy_test:.3f}")
```

Listing 11: train and test code for problem 1.5 and 1.6

The train result of the model was shown every 10 epoch, and the train accuracy and test accuracy were output. This model took about 91 epochs to converge and after this epoch, the result get better slowly. It shows about 99% train accuracy and 95% test accuracy, and I think this model seems to perform about 1% less than the previous results which have 4 hidden nodes. I thought that the accuracy could be reduced in the process of compressing information into 2 hidden nodes compared to 4 hidden nodes. It may be difficult to classify 10 numbers in two dimensions.

```
[Epoch 10] Train Loss : 1.780 Accuracy : 0.864
[Epoch 20] Train Loss : 1.569 Accuracy : 0.975
[Epoch 30] Train Loss : 1.509 Accuracy : 0.983
[Epoch 40] Train Loss : 1.489 Accuracy : 0.986
[Epoch 50] Train Loss : 1.481 Accuracy : 0.987
[Epoch 60] Train Loss : 1.478 Accuracy : 0.988
[Epoch 70] Train Loss : 1.476 Accuracy : 0.989
[Epoch 80] Train Loss : 1.474 Accuracy : 0.989
[Epoch 90] Train Loss : 1.473 Accuracy : 0.990
[Epoch 91] CONVERGE!
[Epoch 100] Train Loss : 1.472 Accuracy : 0.990
Training Accuracy : 0.990, Test Accuracy : 0.955
```

Figure 9: Training and testing results

In this problem, when we do the training, we have to save the activated values from 2 hidden nodes and take average of each node per class. The following is the result of averaging the values for each digit number output by two hidden nodes.

```
The activated values from the 2 hidden nodes for training samples
[Number 0] : [0.99690076 0.00373109] with 5923 samples
[Number 1] : [0.99739452 0.99742706] with 6741 samples
[Number 2] : [0.39192034 0.94181534] with 5949 samples
[Number 3] : [0.05071764 0.00224304] with 6092 samples
[Number 4] : [0.00321623 0.99476174] with 5863 samples
[Number 5] : [0.47251868 0.00394139] with 5391 samples
[Number 6] : [0.96434649 0.40605588] with 5937 samples
[Number 7] : [0.00123758 0.18322207] with 6237 samples
[Number 8] : [0.3085985  0.30186517] with 5918 samples
[Number 9] : [0.01058073 0.56433624] with 5949 samples
```

Figure 10: Activated values from the 2 hidden nodes with training samples

The following is a code that visualizes each sample in the 2D feature space using the values activated from the two nodes.

```
    # Visualize each training sample in 2D feature space
    plt.subplots(figsize=(10,10))
    plt.suptitle("Visualization of each training samples\n", fontsize=16,
    fontweight='bold')
    for i in range(10):
        if activated_values_dict[i][:,0] is None:
            continue
        plt.scatter(activated_values_dict[i][:,0], activated_values_dict[i][:,1],
    label=i)
    plt.legend(loc=(1.0, 1.0))
    plt.xlabel('x1', fontweight='bold')
    plt.ylabel('x2', fontweight='bold')
    plt.show()
```

Listing 12: train and test code for problem 1.5 and 1.6

Then, it can be seen that the 2D feature space is painted differently for each sample corresponding to each class as follows. Based on the values activated from the two nodes, most samples form clusters for each class.

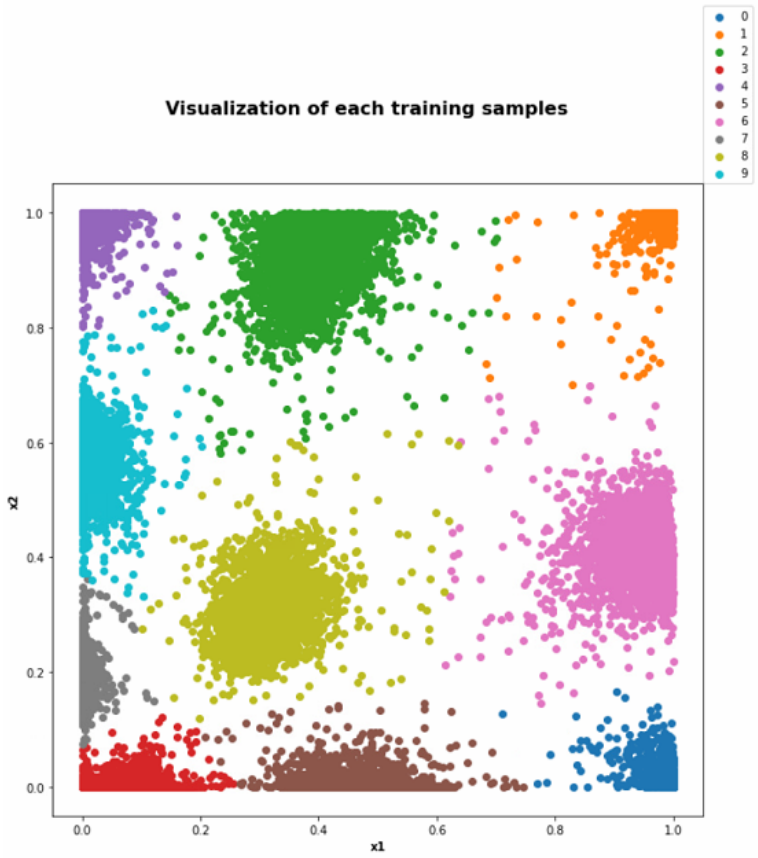

Figure 11: Visualization of each sample in trained 2D feature space

This experiment has been reviewed several times, and it can be seen that most of the well-learned cases show 94 to 95%.

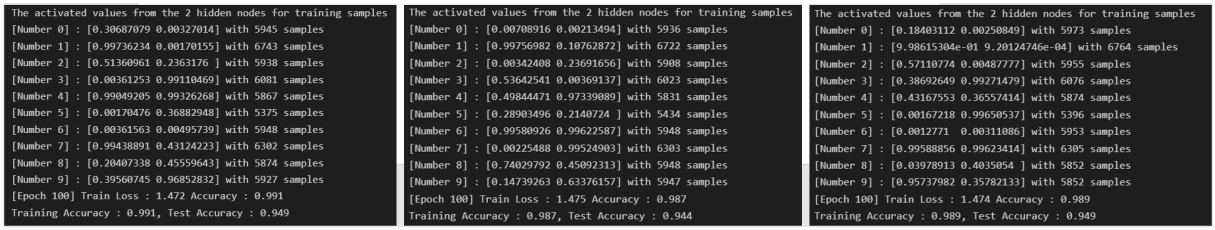

Figure 12: Activated values from the 2 hidden nodes with training samples (Good Case)

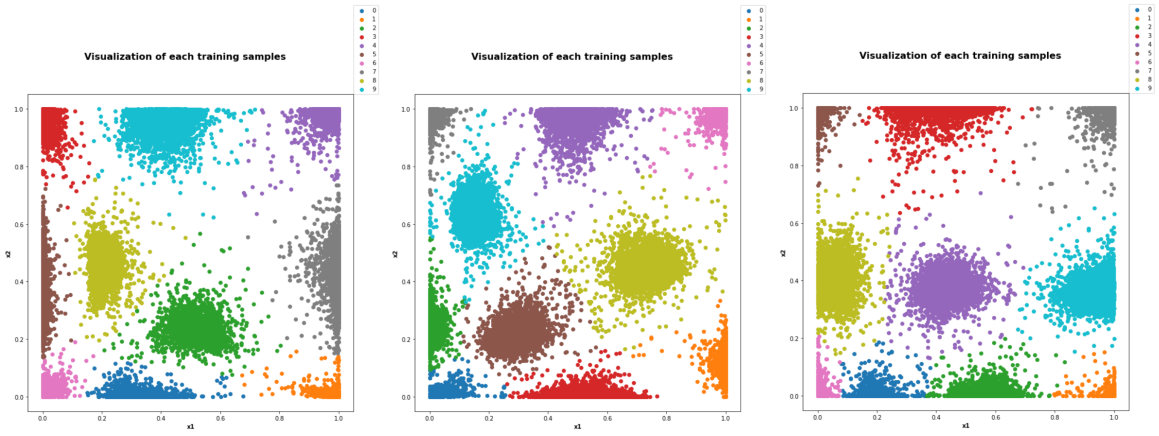

Figure 13: Visualization of each sample in trained 2D feature space (Good Case)

However, if the learning is not well done, it shows about 87% of accuracy, and when looking at the activated values, it was confirmed that the letter 5 was not learned well.

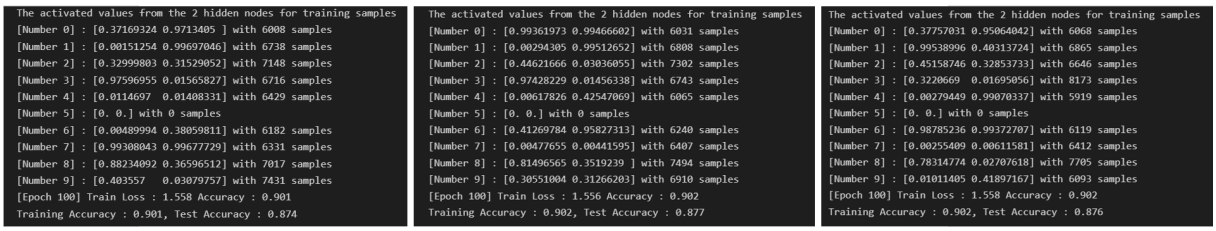

Figure 14: Activated values from the 2 hidden nodes with training samples (Bad Case)

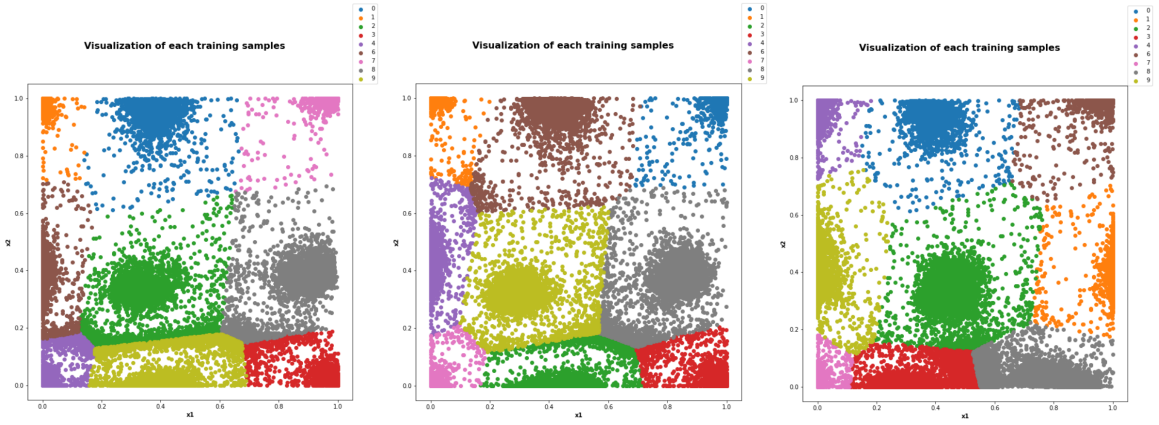

Figure 15: Visualization of each sample in trained 2D feature space (Bad Case)

In my opinion, the letter ”5” looks like many similar numbers, and there are many data that are difficult to distinguish accurately when looking at MNIST dataset, so it does not work properly when learning.