Machine Learning 2024 Spring
HW3: Neural Network
109511286 蔡佩蓉

1. Data (preparing and loading)

```python
transform = transforms.Compose([transforms.ToTensor()])
trainset = torchvision.datasets.MNIST(
    root="./data", train=True, download=True, transform=transform
)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
testset = torchvision.datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)
```

2. Build model

Library used:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F  # activation function
from torch.utils.data import DataLoader, TensorDataset

import torchvision
import torchvision.transforms as transforms

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Initial DNN model:

```python
class DNN_1Layer(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(DNN_1Layer, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x


class DNN_2Layer(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(DNN_2Layer, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


class DNN_3Layer(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(DNN_3Layer, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.num_layers = num_layers

    def forward(self, x):
        x = F.relu(self.fc1(x))
        if self.num_layers > 1:
            for i in range(1, self.num_layers):
                x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Improvement (3 into 1 [add in Xavier and HE Weight Initialization]):

```python
class DNN(nn.Module):
    def __init__(self, input_size, output_size, neurons, num_layers):
        super(DNN, self).__init__()
        self.fc1 = nn.Linear(input_size, neurons)
        init.kaiming_uniform_(self.fc1.weight, nonlinearity="relu")
        self.fc2 = nn.Linear(neurons, neurons)
        init.kaiming_uniform_(self.fc2.weight, nonlinearity="relu")
        self.fc3 = nn.Linear(neurons, output_size)
        init.xavier_uniform_(self.fc3.weight)
        self.num_layers = num_layers

    def forward(self, x):
        x = F.relu(self.fc1(x))
        for _ in range(self.num_layers - 1):
            x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

DNN model for Part II (model with different number of neurons for different hidden layers):

```python
class DNNP2(nn.Module):
    def __init__(self, input_size, output_size, hidden_sizes):
        super(DNNP2, self).__init__()
        layers = []
        current_input_size = input_size

        for hidden_size in hidden_sizes:
            linear_layer = nn.Linear(current_input_size, hidden_size)
            layers.append(linear_layer)
            layers.append(nn.ReLU())
            current_input_size = hidden_size

        output_layer = nn.Linear(current_input_size, output_size)
        layers.append(output_layer)

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)
```

Explanation of the DNN Model

The DNN class defines a fully connected deep neural network with a variable number of hidden layers.

PyTorch module	It's function
torch.nn (nn)	Contains modules and classes for building neural networks.
torch.optim (optim)	Contains various optimization algorithms (these tell the model parameters how to best change to improve gradient descent and in turn reduce the loss).
torch.nn.functional (F)	Provides functions for various neural network operations, such as activation functions.
def forward()	All nn.Module subclasses require a forward() method, this defines the computation that will take place on the data passed to the particular nn.Module.

The DNN class inherits from nn.Module (subclass), which is the base class for all neural network modules in PyTorch.

The __init__ method initializes the network layers based on input parameters.

In the forward method (forward pass of the network), there is a loop that can add additional hidden layers. Each iteration adds a linear layer followed by a ReLU activation, creating a fully connected layer structure.

3. Train, Validate and Evaluate model

```python
def train(model, num_epochs, dataloader, learning_rate):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=10)

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for images, labels in dataloader:
            images = Variable(images.view(images.shape[0], -1))
            labels = Variable(labels)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        val_loss = validate(model, dataloader, criterion)

        scheduler.step(val_loss)
        print(
            f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(dataloader):.4f}"
        )
```

```python
def validate(model, val_loader, criterion):
    model.eval()
    val_loss = 0.0
    with torch.inference_mode():
        for images, labels in val_loader:
            images = Variable(images.view(images.shape[0], -1))
            labels = Variable(labels)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
    return val_loss / len(val_loader)
```

```python
def evaluate(model, dataloader):
    model.eval()
    correct = 0
    total = 0
    with torch.inference_mode():
        for images, labels in dataloader:
            images = Variable(images.view(images.shape[0], -1))
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)

            # compute accuracy
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return 100 * correct / total
```

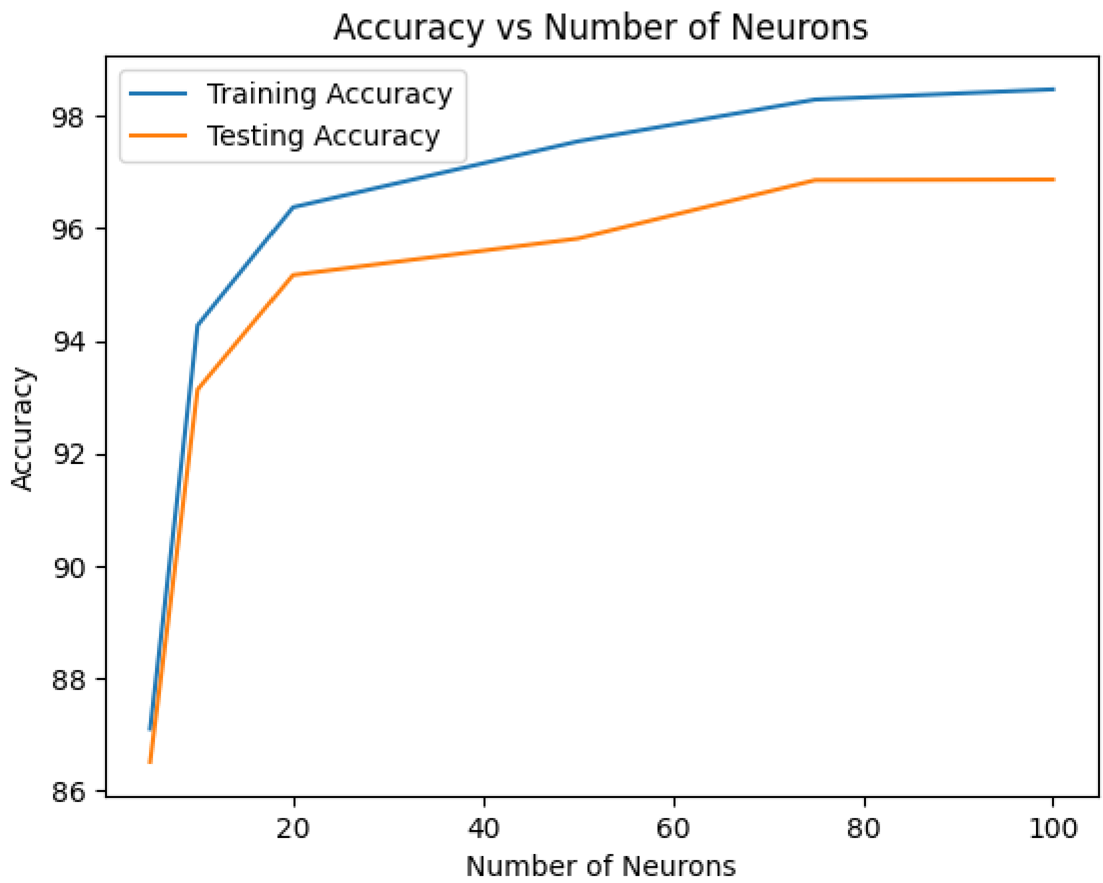
Function	What does it do?	Which is used?
Loss function (Criterion)	Measures how wrong your model predictions are compared to the truth labels. The lower the better.	nn.CrossEntropyLoss (This criterion computes the cross-entropy loss between input logits and target.)
Optimizer	Tells your model how to update its internal parameters to best lower the loss.	torch.optim.Adam (Implements Adam algorithm.)

Note: Further explanations on how the Adam optimizer is chosen can be seen in Discussion.

Training loop:
optimizer.zero_grad(): The optimizers gradients are set to zero (they are accumulated by default) so they can be recalculated for the specific training step.
model(): The model goes through all of the training data once, performing its forward() function calculations.
loss = criterion(outputs, labels): The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are.
loss.backward(): Computes the gradient of the loss with respect for every model parameter to be updated (each parameter with requires_grad=True). This is known as backpropagation, hence "backwards".
optimizer.step(): Update the parameters with requires_grad=True with respect to the loss gradients in order to improve them.

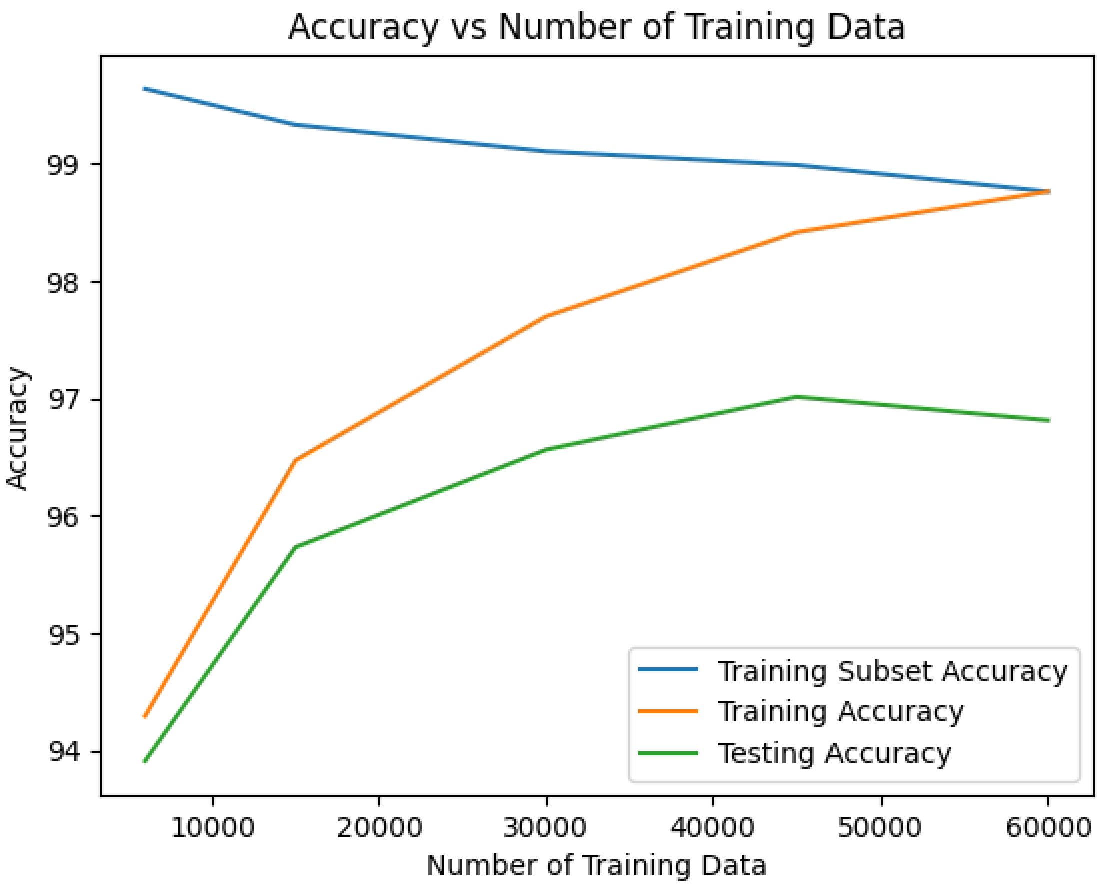
Validation and Evaluation loop:
model(): The model goes through all of the training data once, performing its forward() function calculations.
Then calculate the loss and accuracy on the dataset (train/test set).

Part I

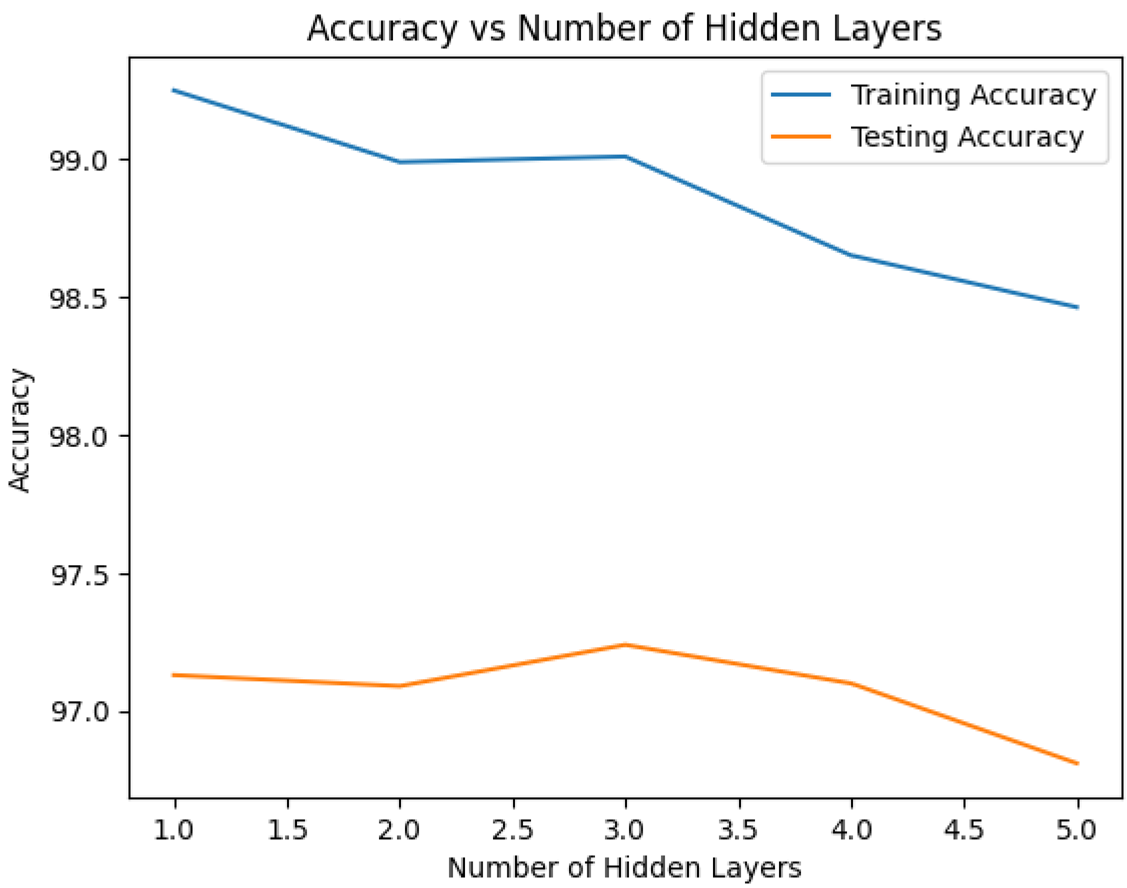


From the plot above, observed that as the number of neurons increases, the accuracy increases for this model.

Noted that the number of neurons in the hidden layer of a DNN model is a critical hyperparameter that affects the model's ability to learn and generalize. Too few neurons can lead to underfitting, while too many can cause overfitting (need optimal neurons).

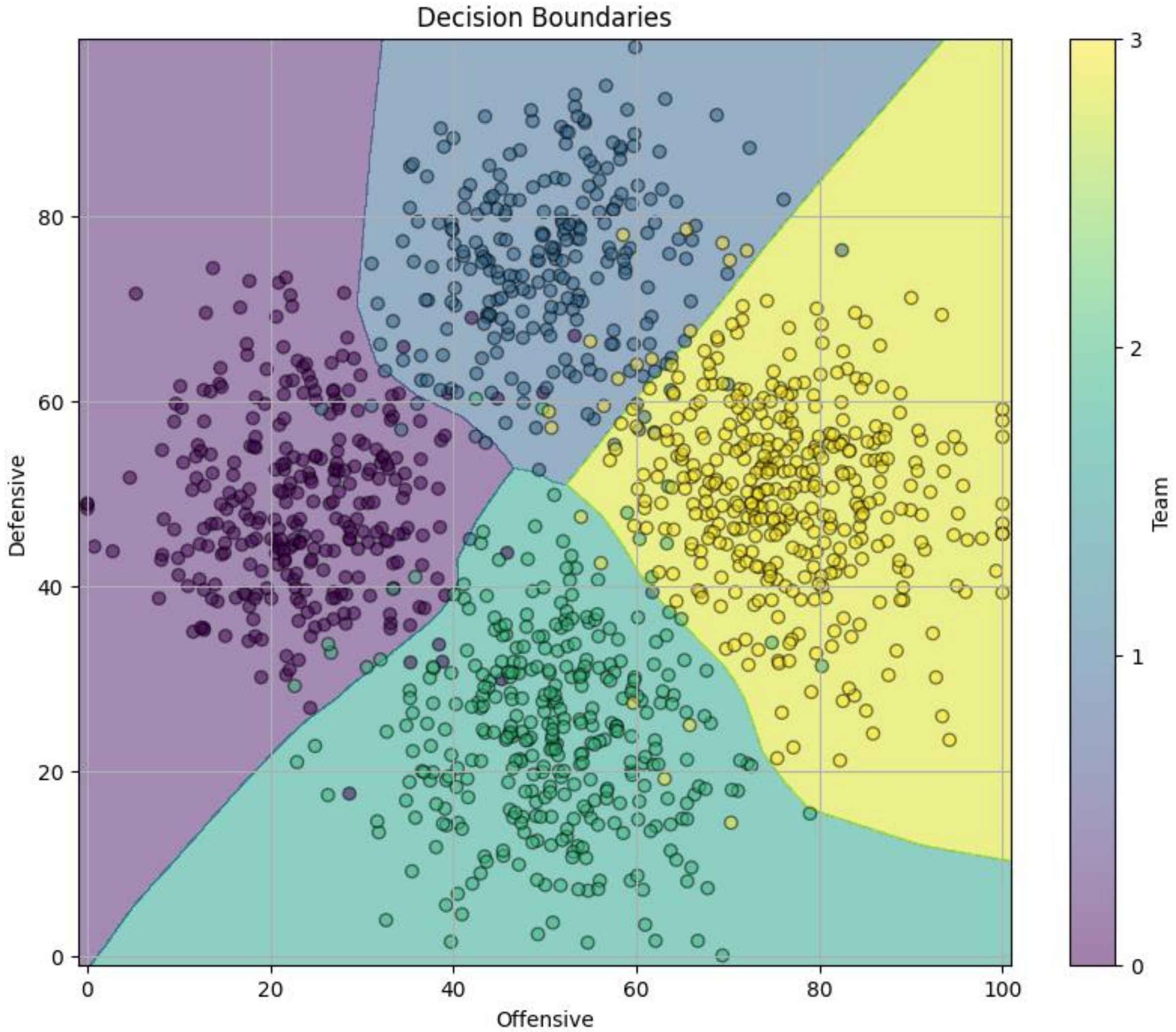


We observed that as the number of training data increases, the accuracy increases.
A larger training dataset generally leads to better generalization and higher test accuracy, as the model can learn more robust and representative patterns from the data. However, with a small dataset, the model is prone to overfitting, leading to high training accuracy but poor test accuracy. Balancing dataset size and model complexity is key to achieving optimal performance.



The number of hidden layers in a DNN, when combined with a fixed number of neurons per hidden layer (in this case is 100 neurons), profoundly influences the model's capacity to learn and represent complex patterns in the data. Deeper architectures enable hierarchical feature learning, leading to improved performance on many tasks. However, deeper networks also pose challenges related to training and overfitting (may affect accuracy).

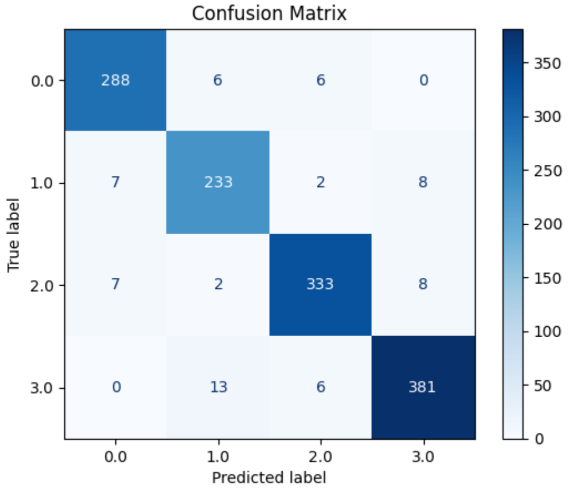
Part II



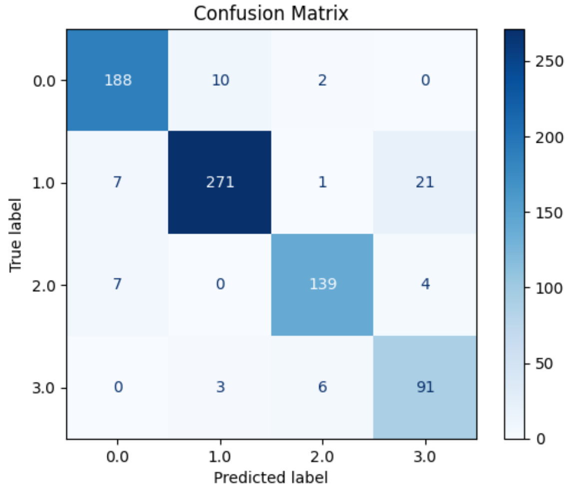

```
Train Accuracy: 95.00, Test Accuracy: 91.87
```

DNN:

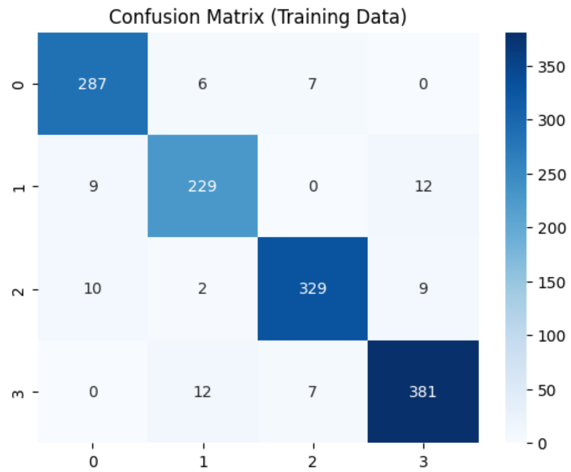
Train Confusion Matrix

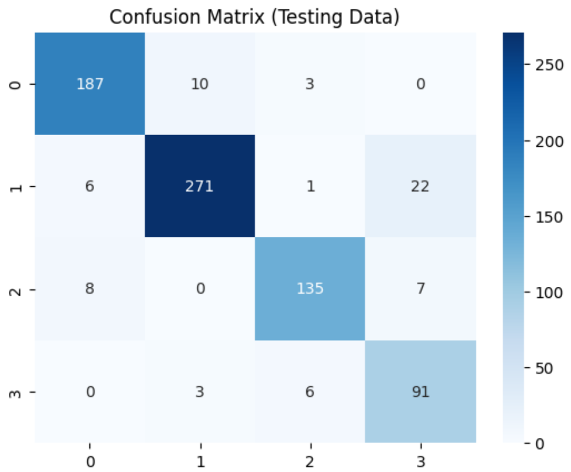


Test Confusion Matrix

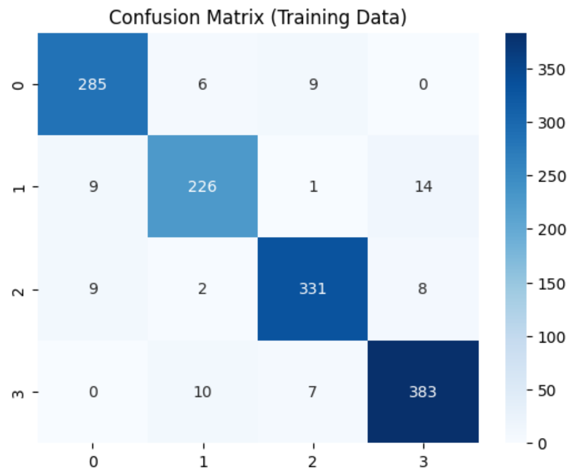


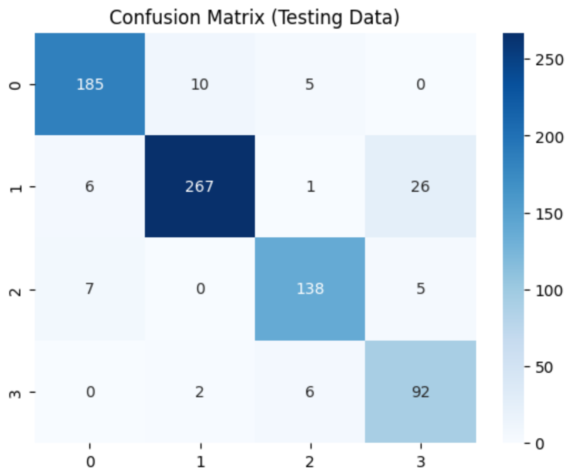
Generative (HW2):





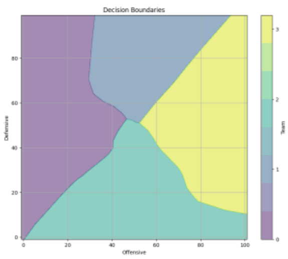
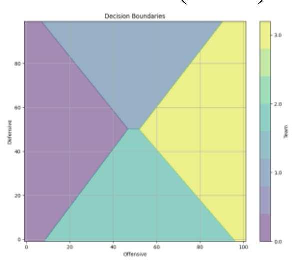
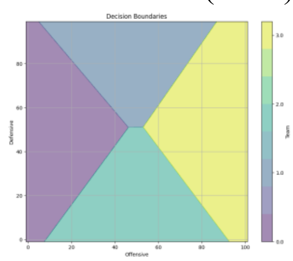
Discriminative (HW2):





Discussion

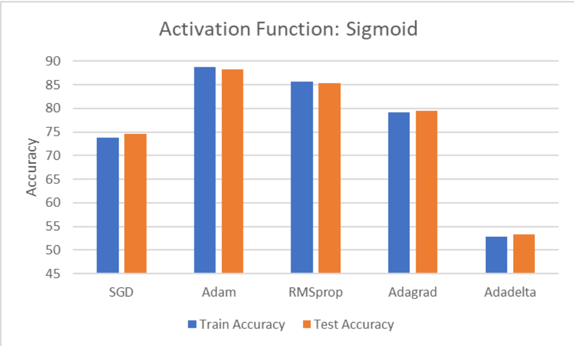
What is the difference between DNN and traditional methods (generative model, and discriminative model)

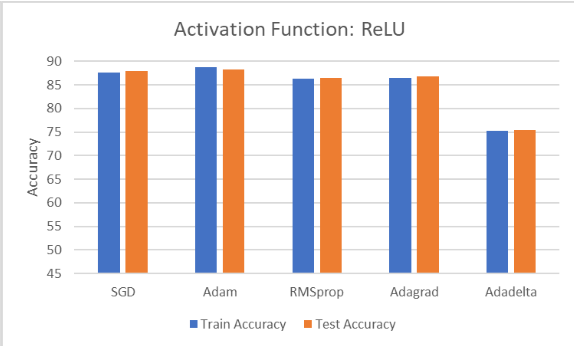
	DNN	Generative (HW2)	Discriminative (HW2)
Model			
Train Accuracy	95.00	94.31	94.23
Test Accuracy	91.87	91.20	90.93

Suppose that DNN have higher accuracy than traditional models, as they can learn from more data and capture more features and patterns. However, for my DNN model, the accuracy is almost the same (not much difference) with the traditional models. The only difference that can be seen clearly is that the decision boundary of DNN is more complex due to it being a nonlinear combination (via activation functions, ReLU) of individual decision boundaries. A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

Compare activation function and optimizer:
hidden layer = 1, neuron = 5 and batch size = 64

Activation Function	Optimizer	Train Accuracy	Test Accuracy
Sigmoid	SGD	73.76	74.59
Sigmoid	Adam	88.77	88.31
Sigmoid	RMSprop	85.64	85.29
Sigmoid	Adagrad	79.24	79.47
Sigmoid	Adadelta	52.83	53.35
ReLU	SGD	87.6	87.91
ReLU	Adam	88.69	88.27
ReLU	RMSprop	86.39	86.56
ReLU	Adagrad	86.43	86.75
ReLU	Adadelta	75.19	75.46





Observed that the optimizer, Adam has the highest accuracy for both activation function, Sigmoid and ReLU. Hence, I choose Adam as the optimizer for the model.
Observed that for optimizer, Adam, the accuracy for activation function, Sigmoid is higher than the accuracy for activation function, ReLu. However, I choose ReLU as the activation function because the accuracy is generally higher than Sigmoid.

Compare batch size:
hidden layer = 1, neuron = 5, activation function = ReLU and optimizer = Adam

Batch Size	Train Accuracy	Test Accuracy
32	84.91	84.39
64	86.27	85.72
128	90.42	89.58
256	89.92	89.39
512	90.04	89.6

Observed that batch size = 128 has highest accuracy. Therefore, I choose batch size = 128 for the model