HW3 REPORT

110511010 楊育陞

1 Results

1.1 Part 1: DNN on MNIST

Training/Testing accuracy versus different configuration

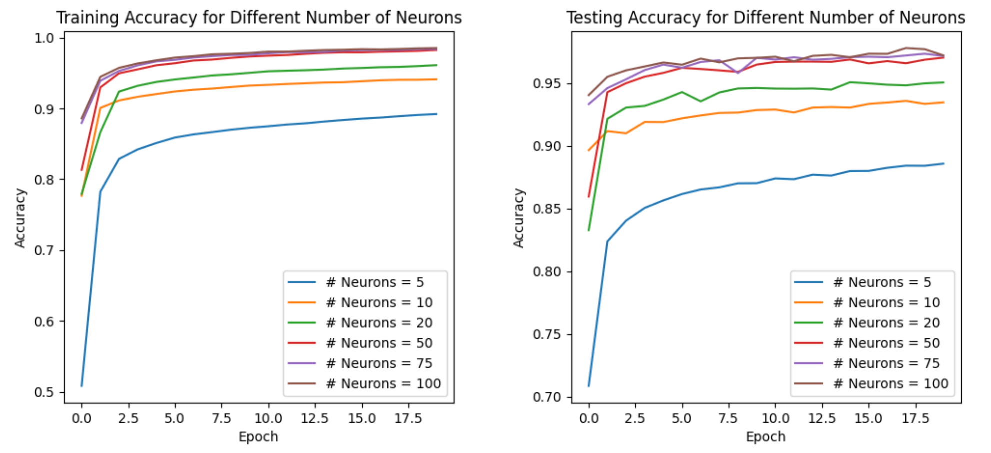

Figure 1: Training/Testing accuracy vs different **number of neurons**

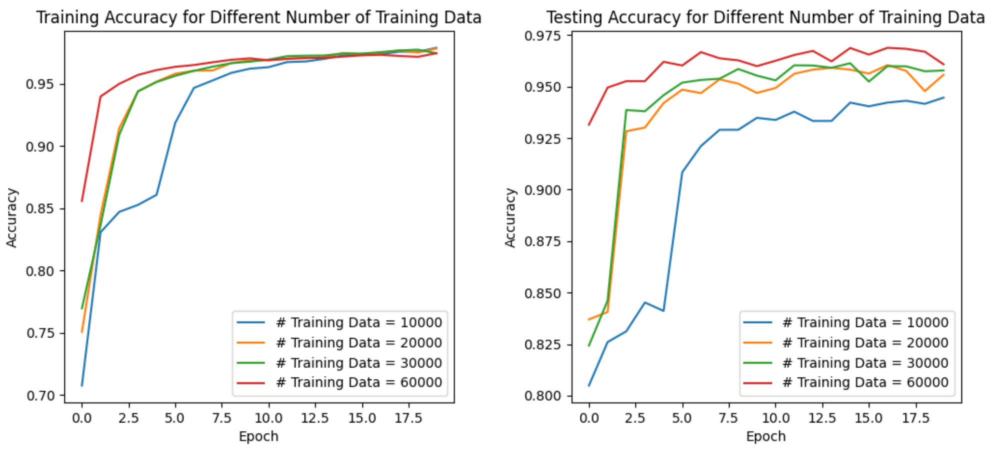

Figure 2: Training/Testing accuracy vs different **number of data**

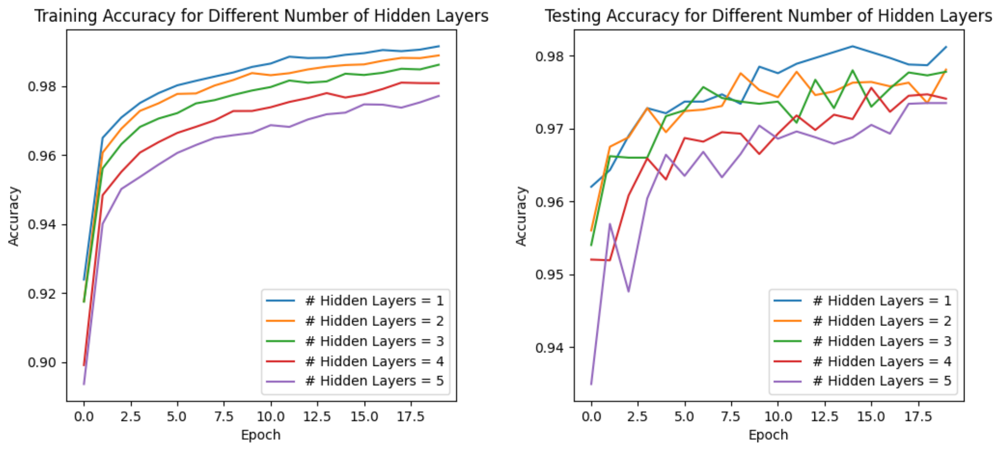

Figure 3: Training/Testing accuracy vs different **number of layers**

1.1.1 Different number of neurons in one hidden layer

The results of Figure 1 show that the training accuracy and the test accuracy increase as the number of neurons in one hidden layer increases. Also, the results of Figure 4 show that the training accuracy is usually a little higher than the test accuracy. This is because the model is optimized to fit the training data, so the training accuracy is usually higher than the test accuracy. On the other hand, the accuracy from the model of 5 neurons in one hidden layer is significantly lower than the other models. This is because the model is too simple to learn the complex relationship between the input and output.

1.1.2 Different number of training data

The results of Figure 2 show that the training accuracy and the test accuracy increase as the number of training data increases. This is much reasonable since the model can learn better with more training data, namely, the model can generalize better with more training data. Also, the results of Figure 5 show that the training accuracy converges quickly for all models, but the overall testing accuracy is lower for the model with fewer training data.

1.1.3 Different number of hidden layers

The results of Figure 3 show that the training accuracy and the test accuracy decrease as the number of hidden layers increases. This is because the model is too complex to learn the relationship between the input and output, which may lead to overfitting. Also, the testing accuracy has ripple patterns, which may be caused by the randomness and the overfitting of the model.

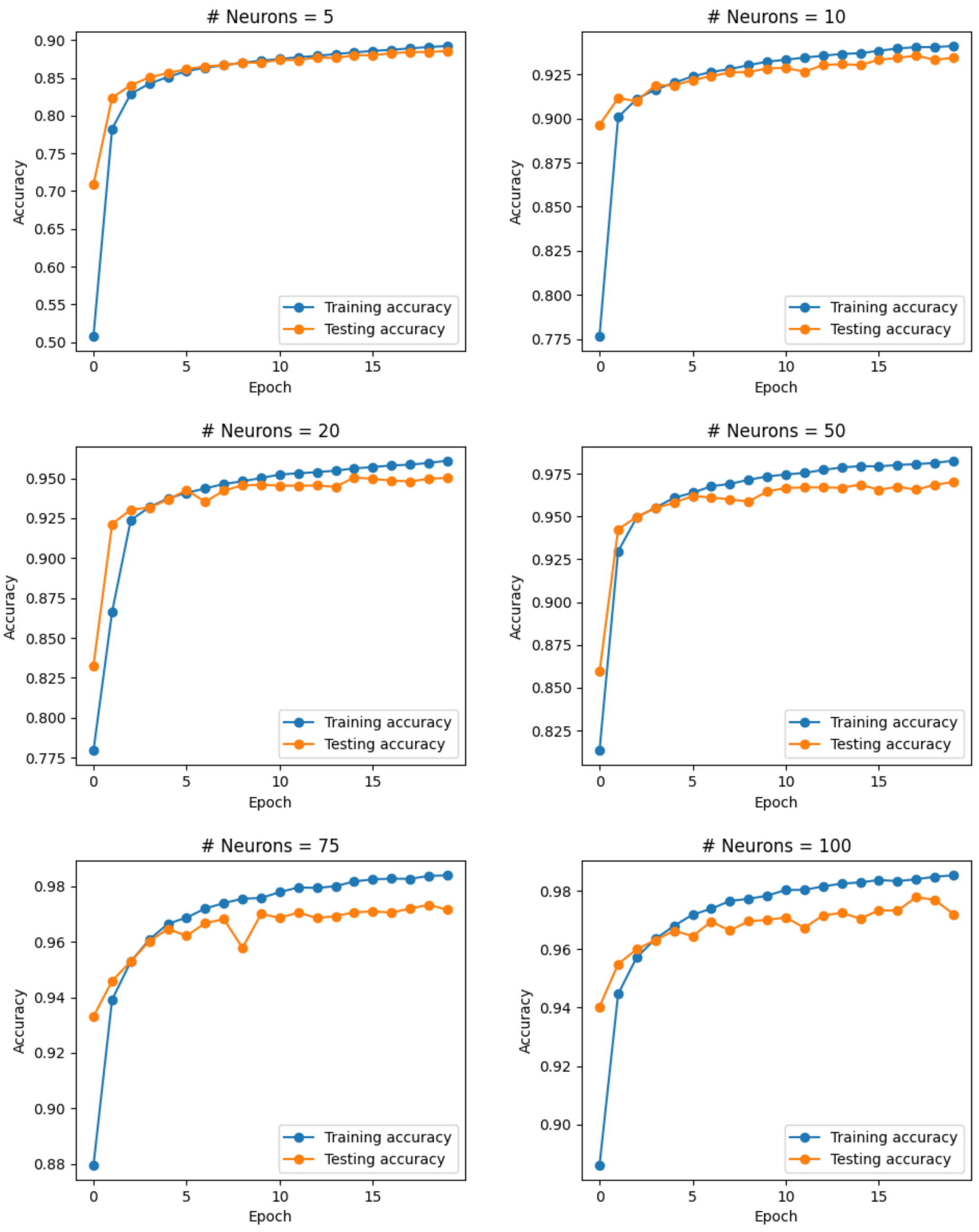

Figure 4: Training vs Testing accuracy among different **number of neurons**

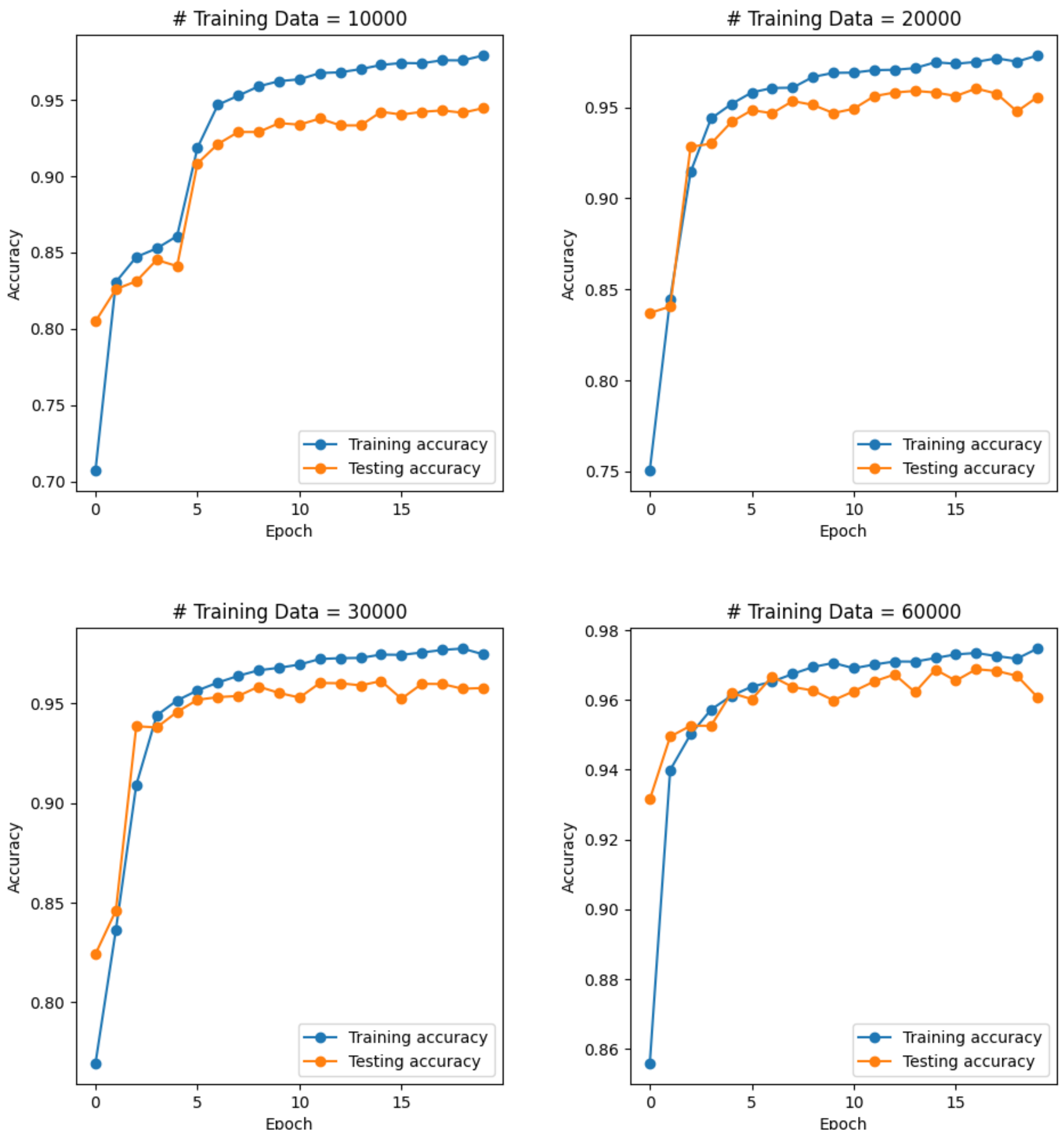

Figure 5: Training vs Testing accuracy among different **number of datas**

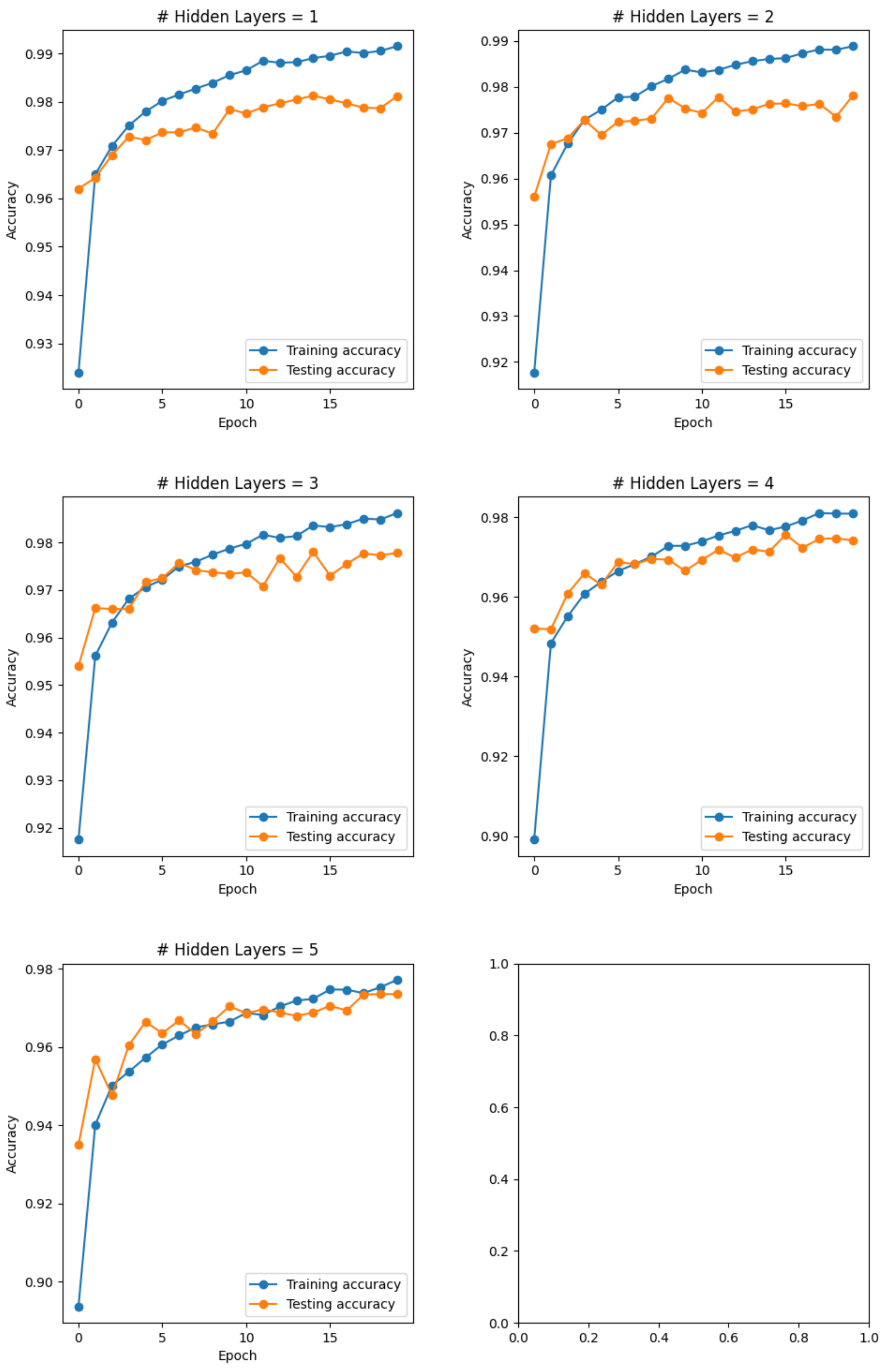

Figure 6: Training vs Testing accuracy among different **number of layers**

1.2 Part 2: DNN on HW2 Dataset

1.2.1 Result of training

To avoid gradient vanishing, I use the technique of batch normalization.
Trained with different number of hidden layers with 100 neurons per layer.
Best model is trained with 5 hidden layers.
Accuracy on Training Data: **0.927** - Testing Data: **0.931**

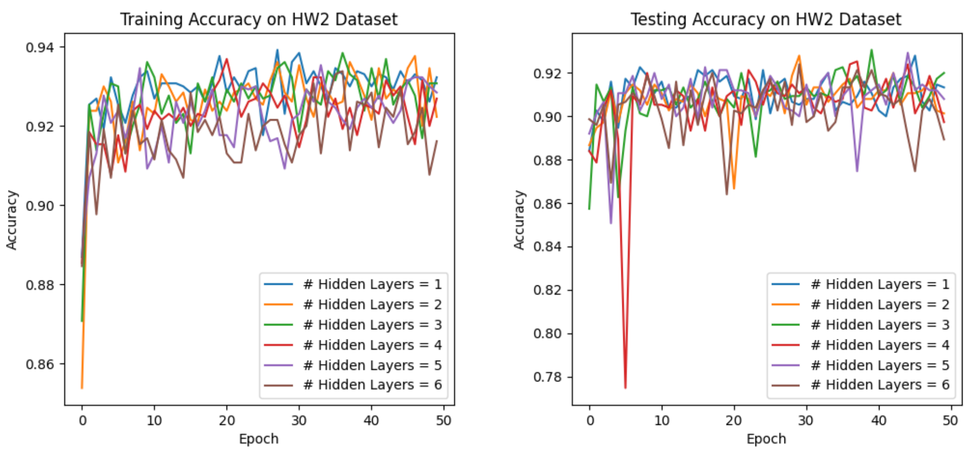

Figure 7: Training/Testing accuracy vs different number of neurons **on HW2 dataset**

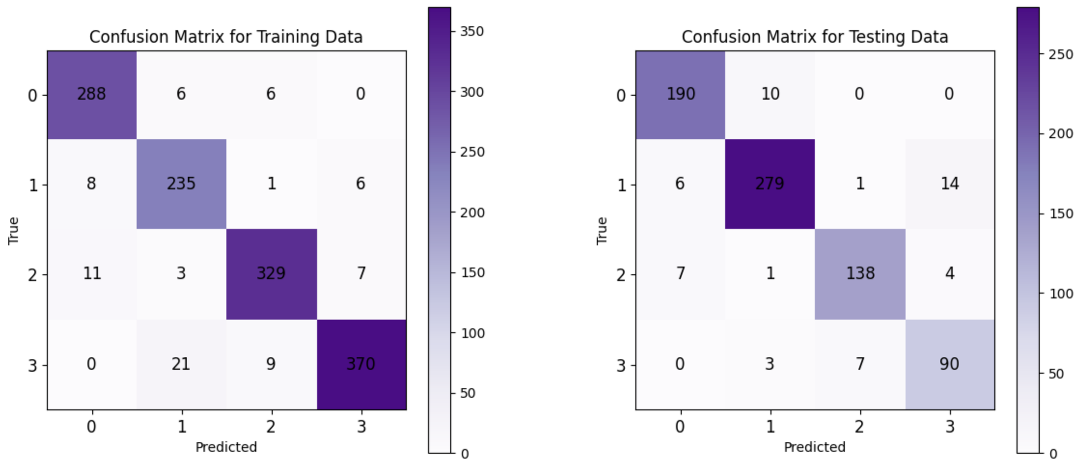

Figure 8: Confusion Matrix for model with 5 layer

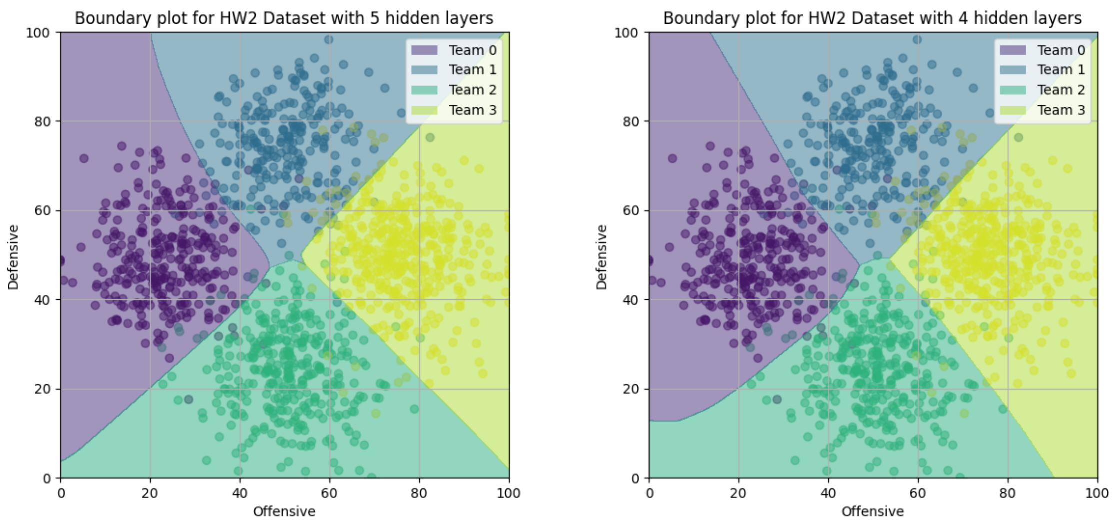

Figure 9: Boundary plots for model with 5(left)/4(right) layer

1.2.2 Comparison with Generative and Discriminative Model

Accuracy	DNN	Generative	Discriminative
Training	0.927	0.943	0.943
Testing	0.931	0.912	0.909

Table 1: Accuracy among different models on HW2 dataset

The testing accuracy of the DNN model is higher than the traditional method. This is because the DNN model can learn the non-linear relationship between the input and output, which may lead to better performance. Also, the decision boundary of the DNN model is more complex than the traditional method as shown in Figure 9, which can fit different data distribution better.

2 Implementation

I implemented the DNN model with **PyTorch**. The plotting is done with **matplotlib**.

2.1 Model

```python
class DNN(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim, num_hidden_layers, batch_norm):
        super(DNN, self).__init__()

        self.input_dim = input_dim

        seq = []
        seq.append(nn.Linear(input_dim, hidden_dim))
        seq.append(nn.ReLU())
        for _ in range(num_hidden_layers):
            seq.append(nn.Linear(hidden_dim, hidden_dim))
            if batch_norm:
                seq.append(nn.BatchNorm1d(hidden_dim))
            seq.append(nn.ReLU())
        seq.append(nn.Linear(hidden_dim, output_dim))

        self.seq = nn.Sequential(*seq)


    def forward(self, x):
        x = x.view(-1, self.input_dim)
        x = self.seq(x)
        x = F.softmax(x, dim=1)
        return x
```

I implemented the DNN model with numbers of Linear layers followed by ReLU activation function. The number of hidden layers and the number of neurons in each hidden layer can be set by parameters for convenience among different experiments. The output layer is followed by a softmax function. The input data is first reshaped to a 1D tensor. Also, I added the batch normalization layer as an option for part 2.

2.2 Training and Testing

```python
def train(model, trainloader, testloader, criterion, optimizer, n_epochs, device, n_class, save_path=None):
    history = []
    best_accuracy = 0
    for epoch in range(n_epochs):
        result = {}
        model.train()
        total = trainloader.dataset.__len__()
        correct = 0
        for inputs, labels in tqdm(trainloader):
            inputs, labels = inputs.to(device), labels.to(device)
            labels_onehot = torch.zeros(labels.size(0), n_class).to(device)
            labels_onehot.scatter_(1, labels.view(-1, 1), 1)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels_onehot)
            loss.backward()
            optimizer.step()
            correct += (outputs.argmax(1) == labels).sum().item()
        result["train_accuracy"] = correct / total
        model.eval()
        total = testloader.dataset.__len__()
        correct = 0
        with torch.no_grad():
            for inputs, labels in testloader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                correct += (outputs.argmax(1) == labels).sum().item()
        result["test_accuracy"] = correct / total
        history.append(result)
        # Save the model if the test accuracy is the best
        if result["test_accuracy"] > best_accuracy:
            best_accuracy = result["test_accuracy"]
            if save_path is not None:
                torch.save(model.state_dict(), save_path)
    return history
```

The major training and testing process is implemented in the above function. The model is trained with the given trainloader and tested with the given testloader. The criterion and optimizer are also given as parameters. The training process is repeated for the given number of epochs. The model is saved if the test accuracy is the best. As follows, I use the CrossEntropyLoss and Adam optimizer for the criterion and optimizer, respectively.

```
def train(model, trainloader, testloader, criterion, optimizer, n_epochs, device, n_class, save_path=None):
criterion = torch.nn.CrossEntropyLoss()
model = DNN(784, 10, hidden_dim, 1, False).to(device)
optimizer = torch.optim.Adam(model.parameters())
history = train(model, trainloader, testloader, criterion, optimizer, n_epochs, device, 10)
```

2.3 Plotting Decision Boundary

For the HW2 dataset model, I plotted the decision boundary by first generating the mesh data and then predicting the class of each mesh point. The mesh data is genrated from 0 to 100 with resolution 0.1.

3 Further Discussion

3.1 Difference between DNN and Traditional Methods (Generative Models and Discriminative Models)

By seeing the decision boundary of the DNN model, we can see that the decision boundary is not linear and can be more complex. This is because the DNN model can learn the non-linear relationship between the input and output. Although DNN models are also discriminative models, they can learn the complex relationship between the input and output since they have multiple layers and non-linear activation functions.

3.2 MSE Loss and Cross Entropy loss on Classification

The cross entropy loss is more suitable for multi-class classification problems than the MSE loss. The reason is that the cross entropy loss performs better when the output is a probability distribution and the target is a one-hot vector. On the other hand, the MSE loss is more suitable for regression problems. The cross entropy loss is also more robust to the scale of the output since it is calculated by the log of the output. The MSE loss is more sensitive to the scale of the output since it is calculated by the square of the difference between the output and the target, which may lead to the gradient problem.

3.3 Sigmoid and ReLU

The ReLU activation function is more suitable for the hidden layers of the DNN model than the sigmoid activation function. The reason is that the ReLU activation function is more robust to the vanishing gradient problem than the sigmoid activation function. The ReLU activation function is also more computationally efficient than the sigmoid activation function since it is a simple threshold function. The ReLU activation function is also more biologically plausible than the sigmoid activation function since it is more similar to the firing of a neuron.