DAT565/DIT407 Assignment 6

Giacomo Guidotto
gusguigi@student.gu.se

Leong Jia Yi, Janna
gusleoji@student.gu.se

2024-10-17

1 The dataset

Below, Figure 1 displays some of the MNIST dataset images in grayscale.

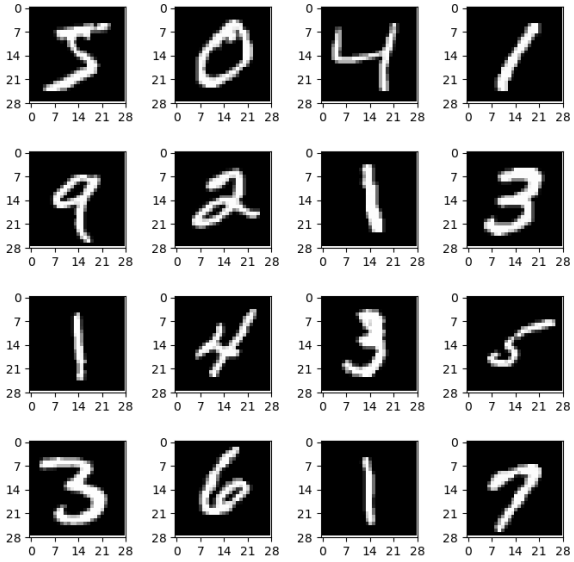

Figure 1: 16 MNIST dataset images in gray scale

2 Single hidden layer

The validation accuracy is shown in Table 1 and Figure 2. In training the network, we used a learning rate **1e-3** for a parameter in the stochastic gradient descent optimizer and trained the network for **10** epochs. The learning rate measures how far the neural network weights will change during optimization.

We chose the learning rate to be **1e-3** as this is a commonly used value that we thought would be a good starting point. We maintained this learning rate value after training the network and getting favorable results. As for the structure of the neural network, it consists of 1 hidden layer where input size = 784 to match the size of the flattened input vector and output size = 512 to map to 512 features. The final linear layer has an input size = 512 to match the output size of the previous layer and output size = 10 to map to the 10 different numbers.

Single Layer Neural Network Accuracy	
epoch	**accuracy (%)**
1	66.59
2	73.99
3	77.08
4	80.25
5	81.99
6	83.62
7	84.71
8	85.59
9	86.22
10	86.74

Table 1: Single Layer Neural Network accuracy during training

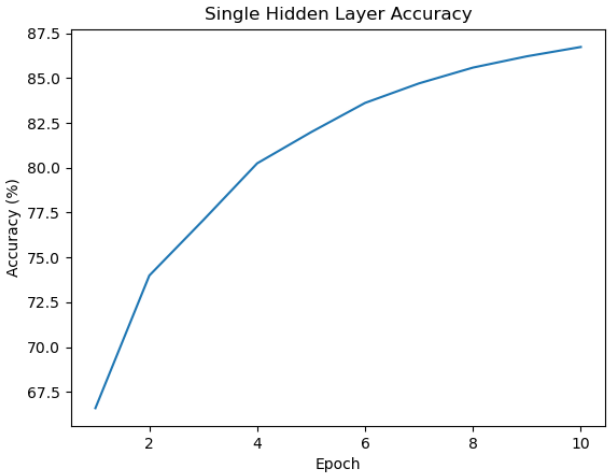

Figure 2: Single Hidden Layer Neural Network accuracy plot during training

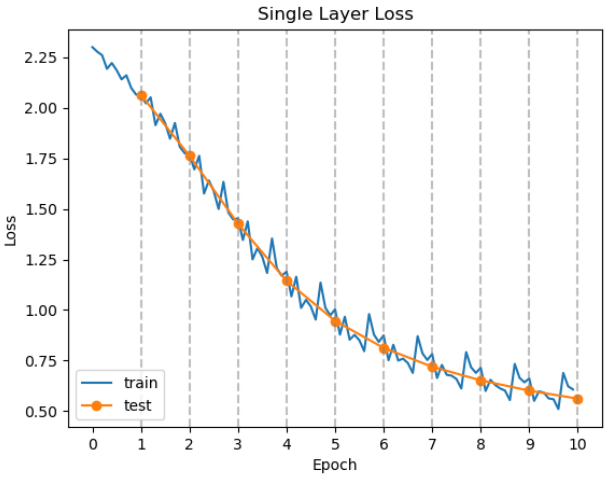

Figure 3: Single Hidden Layer Neural Network loss during training and testing

3 Two hidden layer

The validation accuracy is shown in Table 2 and Figure 4

Two Layer Neural Network Accuracy	
epoch	**accuracy (%)**
1	93.55
2	95.57
3	96.57
4	96.97
5	97.26
6	97.41
7	97.6
8	97.73
9	97.75
10	97.74
11	97.79
12	97.84
13	97.87
14	97.88
15	97.89
16	97.89
17	97.94
18	97.92
19	98.0
20	98.03
21	98.03
22	98.07
23	98.1
24	98.13
25	98.16
26	98.18
27	98.19
28	98.19
29	98.19
30	98.18
31	98.19
32	98.2
33	98.2
34	98.21
35	98.21
36	98.21
37	98.21
38	98.22
39	98.21
40	98.22

Table 2: Two Layer Neural Network accuracy during training

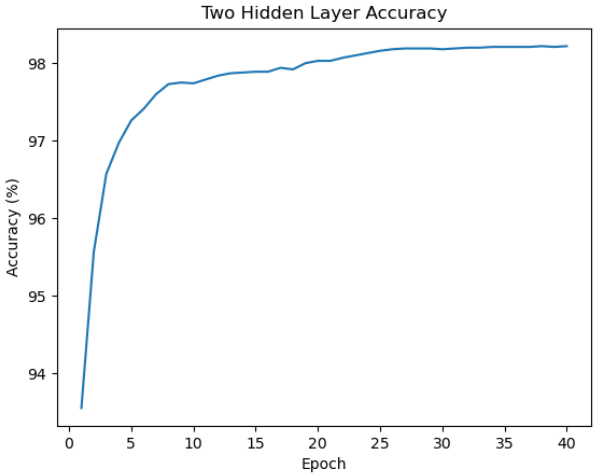

Figure 4: Two Hidden Layer Neural Network accuracy plot during training

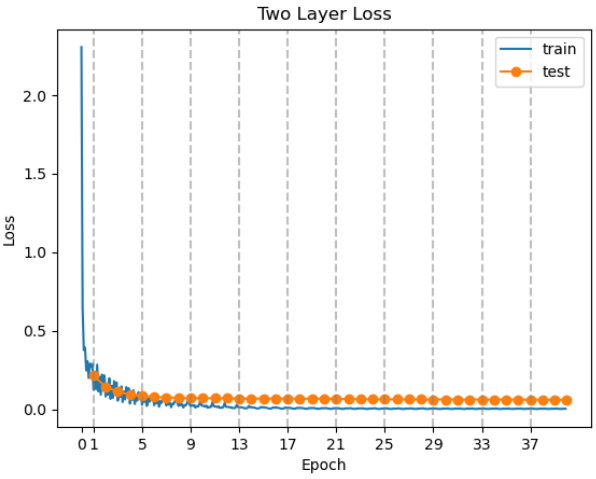

Figure 5: Two Hidden Layer Neural Network loss during training and testing

4 Convolutional neural network

The validation accuracy is shown in Table 3 and Figure 6. After the 40th epoch, we attained an accuracy of 99.5%.

For the network structure, we found a source that obtained an extremely high accuracy after performing a similar task. We have taken reference from this source to design our network[1]. It consists of:

- 2 convolutional layers
- 1 pooling layer
- 1 convolutional layer
- 1 pooling layer
- 3 fully connected layers

to classify the inputs into the 10 numbers.

For the first convolutional layer, input channel = 1 as the input data are grayscale images with only 1 intensity value to indicate the shade of gray. Next, the number of output channels = 32 to learn 32 different features from the input. kernel_size = 3 allows the layer to capture significant patterns while managing costs to train the network. Stride = 1 to move the filter 1 pixel at a time while padding = 1 to ensure the output size of the feature map is the same as the input size. Next, we add a second convolutional layer. In this layer, we set the number of input channels = 32 to match the output channels in the previous layer. We also, increase the number of output layers to 64 to capture more intricate patterns. Our next layer is a pooling layer which aims to downscale the feature maps and reduce the overfitting rate. Thus we lower the kernel_size to 2 and increase stride to 2 such that the filters move 2 pixels at a time. We then add another convolutional layer and set the input and output channels to match the previous layer. Finally, we end with our last pooling layer with the same parameters as the first.

After building a convolutional neural network that detects high-level features, we use these features as input to the perceptron layers of fully connected layers to finally classify these numbers. The first dropout layer has been set to probability = 0.5 which will randomly set 50% of features to 0. This will prevent overfitting while still retaining significant information from previous training. Next, the parameters of the first linear layer were set to account for the output number of the final convolutional layer, which was connected to a layer of 128 neurons. Finally, after 2 dropout and 2 linear layers, we end with the last linear layer where the output size = 10 to map to the 10 classes of numbers.

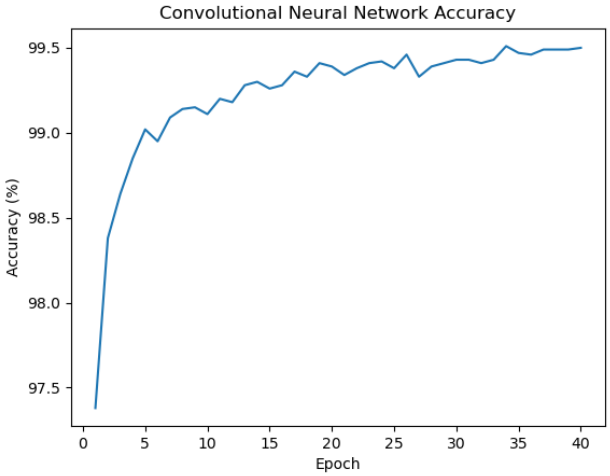

Figure 6: Convolutional Neural Network accuracy plot during training

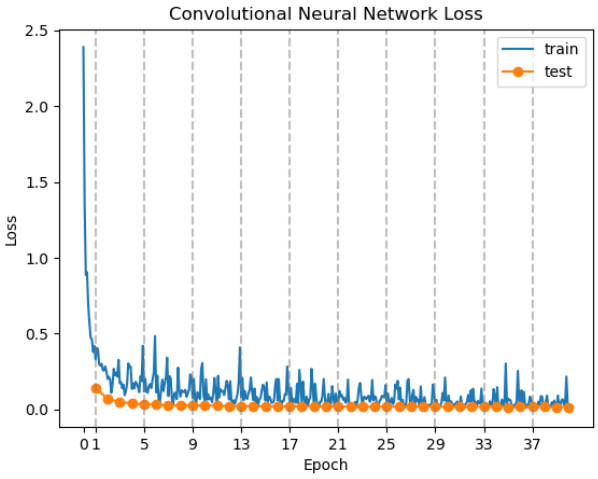

Figure 7: Convolutional Neural Network loss during training and testing

CNN Accuracy	
epoch	**accuracy (%)**
1	97.38
2	98.38
3	98.64
4	98.85
5	99.02
6	98.95
7	99.09
8	99.14
9	99.15
10	99.11
11	99.2
12	99.18
13	99.28
14	99.3
15	99.26
16	99.28
17	99.36
18	99.33
19	99.41
20	99.39
21	99.34
22	99.38
23	99.41
24	99.42
25	99.38
26	99.46
27	99.33
28	99.39
29	99.41
30	99.43
31	99.43
32	99.41
33	99.43
34	99.51
35	99.47
36	99.46
37	99.49
38	99.49
39	99.49
40	99.5

Table 3: Convolutional Neural Network accuracy during training

References

[1] Kaggle. *MNIST with Pytorch + CNN*. Retrieved 2023-10-13. 2019. URL: https://www.kaggle.com/code/cgurkan/mnist-with-pytorch-cnn.

A Code

Following is the Python code that we used to extract and visualize the information presented in this report:

A.1 Problem 1: The dataset

Loading the MNIST dataset from ‘torchvision.datasets‘:

```
import csv

import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

train_data = datasets.MNIST(
        root='data',
        train=True,
        download=True,
        transform=ToTensor()
)

test_data = datasets.MNIST(
        root='data',
        train=False,
        download=True,
        transform=ToTensor()
)
```

Wrapping the dataset in a ‘DataLoader‘:

```
train_loader = DataLoader(
        train_data,
        batch_size=64,
)

test_loader = DataLoader(
        test_data,
        batch_size=64,
)
```

Plotting the first images from the training set:

```
import matplotlib.pyplot as plt

```

```
images, _ = next(iter(train_loader))

ppr = 4  # plots per row

fig, ax = plt.subplots(
        ppr, ppr,
        figsize=(ppr * 2, ppr * 2)
)
plt.subplots_adjust(wspace=0.5, hspace=0.5)
for i in range(ppr ** 2):
        ax[i // ppr, i % ppr].imshow(
                images[i].squeeze(),
                cmap='gray'
        )
        ax[i // ppr, i % ppr].set_xticks(
                range(0, 29, 7))
        ax[i // ppr, i % ppr].set_yticks(
                range(0, 29, 7))

plt.savefig('figures/mnist.png')
```

A.2 Problem 2: Single hidden layer

A.2.1 Designing the model

Creating a neural network model with a single hidden layer with ReLU activation.

Using SGD as optimizer and CrossEntropy as loss function.

Also, defining the device to be used:

```
device = (
        "cuda"
        if torch.cuda.is_available()
        else "mps"
        if torch.backends.mps.is_available()
        else "cpu"
)
print(f"Using device: {device}")
```

```
class FNN(nn.Module):
        def __init__(self):
                super(FNN, self).__init__()
                self.flatten = nn.Flatten()
                self.linear_stack = nn.Sequential(
                        nn.Linear(28 * 28, 512),
                        nn.ReLU(),
                        nn.Linear(512, 10),
                )

        def forward(self, x):
```

```
                x = self.flatten(x)
                x = self.linear_stack(x)
                return x


model = FNN().to(device)
```

```
loss_fn = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(
        model.parameters(),
        lr=1e-3
)
```

A.2.2 Defining the training, testing, and evaluation functions

Defining the training of the model:

```
def train(dataloader, model, loss_fn, optimizer,
          logging=False):
        losses = []
        size = len(dataloader.dataset)
        model.train()
        for batch, (X, y) in enumerate(dataloader):
                X, y = X.to(device), y.to(device)

                pred = model(X)
                loss = loss_fn(pred, y)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                if batch % 100 == 0:
                        losses.append(loss.item())
                        if logging:
                                current = batch * len(X)
                                print(
                                        f"loss: {loss.item():>7f}  [{curren
                                )
        return losses
```

Defining the testing of the model performance against the ‘test_data‘:

```
def test(dataloader, model, loss_fn,
         logging=True):
        size = len(dataloader.dataset)
        num_batches = len(dataloader)
        model.eval()
        test_loss, correct = 0, 0
        with (torch.no_grad()):
```

```
                for X, y in dataloader:
                        X, y = X.to(device), y.to(device)
                        pred = model(X)
                        test_loss += loss_fn(pred, y).item()
                        correct += (
                                        pred.argmax(1) == y
                        ).type(torch.float).sum().item()
        test_loss /= num_batches
        accuracy = round(correct / size * 100, 4)
        if logging:
                print(f"\n accuracy: {accuracy:>0.1f}%")
                print(f" test loss: {test_loss:>8f}\n")
        return accuracy, test_loss
```

Defining the export of the accuracy data to a csv file:

```
def save_accuracy_data(
                filename, accuracy_list
):
        with open(filename, mode='w') as file:
                writer = csv.writer(file)
                writer.writerow(["epoch", "accuracy"])
                for epoch, accuracy in enumerate(
                                accuracy_list):
                        writer.writerow([epoch + 1, accuracy])
```

Defining the plotting of the training and test loss:

```
def plot_loss(
                train_loss, test_loss, epochs, title, filename
):
        epoch_length = len(train_loss) // len(test_loss)
        max_lines = 10
        epoch_step = max(1, len(epochs) // max_lines)
        plt.plot(train_loss, label="train")
        plt.plot(
                range(
                        epoch_length,
                        len(train_loss) + epoch_length,
                        epoch_length
                ),
                test_loss, label="test", marker='o'
        )
        selected_epochs = epochs[::epoch_step]
        for epoch in selected_epochs:
                plt.axvline(
                        epoch * epoch_length,
                        color='gray', linestyle='--', alpha=0.5
                )
        plt.xticks(
                [0] + list(range(
                        epoch_length,
```

```
                        len(train_loss) + 1,
                        epoch_length * epoch_step
                )),
                [0] + list(selected_epochs)
        )

        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.legend()
        plt.title(title)
        plt.savefig(f"figures/{filename}")
```

A.2.3 Training and evaluating the model

Training the model for 10 epochs and evaluating its performance:

```
epochs = range(1, 11)

train_loss = []
test_loss = []
accuracy_list = []
for t in epochs:
        print(f"training epoch {t}...")

        epoch_train_loss = train(
                train_loader, model, loss_fn, optimizer,
        )
        accuracy, epoch_test_loss = test(
                test_loader, model, loss_fn
        )

        train_loss.extend(epoch_train_loss)
        test_loss.append(epoch_test_loss)
        accuracy_list.append(accuracy)

        print(f"--------------------------------")

torch.save(
        model.state_dict(),
        "models/single_layer.pth"
)
print("training complete, model saved")
```

Saving and plotting the accuracy data:

```
save_accuracy_data(
        "models/single_layer_accuracy.csv",
        accuracy_list
)

plt.plot(epochs, accuracy_list)
```

```
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.title("Single Hidden Layer Accuracy")
plt.savefig(
        "figures/single_layer_accuracy.png")
```

Plotting the training and test loss:

```
plot_loss(
        train_loss, test_loss, epochs,
        "Single Layer Loss",
        "single_layer_loss.png"
)
```

A.3 Problem 3: Two hidden layers

A.3.1 Designing the model

Creating a neural network model with two hidden layers with ReLU activation.

Reusing the CrossEntropy as loss function and using SGD with L2 regularization as optimizer.

Reusing the device definition:

```
class FNN2(nn.Module):
        def __init__(self):
                super(FNN2, self).__init__()
                self.flatten = nn.Flatten()
                self.linear_stack = nn.Sequential(
                        nn.Linear(28 * 28, 500),
                        nn.ReLU(),
                        nn.Linear(500, 300),
                        nn.ReLU(),
                        nn.Linear(300, 10),
                )

        def forward(self, x):
                x = self.flatten(x)
                x = self.linear_stack(x)
                return x


model = FNN2().to(device)
```

```
optimizer = torch.optim.SGD(
        model.parameters(),
        lr=0.1,
        weight_decay=1e-4
)
```

A.3.2 Training and evaluating the model

Train the model for 40 epochs and evaluate its performance:

```
epochs = range(1, 41)

train_loss = []
test_loss = []
accuracy_list = []
for t in epochs:
        print(f"training epoch {t}...")

        epoch_train_loss = train(
                train_loader, model, loss_fn, optimizer,
        )
        accuracy, epoch_test_loss = test(
                test_loader, model, loss_fn
        )

        train_loss.extend(epoch_train_loss)
        test_loss.append(epoch_test_loss)
        accuracy_list.append(accuracy)

        print(f"--------------------------------")

torch.save(
        model.state_dict(),
        "models/two_layer.pth"
)
print("training complete, model saved")
```

Saving and plotting the accuracy data:

```
save_accuracy_data(
        "models/two_layer_accuracy.csv",
        accuracy_list
)
plt.plot(epochs, accuracy_list)
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.title("Two Hidden Layer Accuracy")
plt.savefig(
        "figures/two_layer_accuracy.png")
```

Plotting the training and test loss:

```
plot_loss(
        train_loss, test_loss, epochs,
        "Two Layer Loss",
        "two_layer_loss.png"
)
```

A.4 Problem 4: Convolutional Neural Network

A.4.1 Designing the model

Creating a convolutional neural network model with two convolutional layers and two fully connected layers.

Reusing the CrossEntropy as loss function and reusing the SGD with L2 regularization as optimizer.

Reusing the device definition:

```
class CNN(nn.Module):
        def __init__(self):
                super(CNN, self).__init__()
                self.conv_block = nn.Sequential(
                        nn.Conv2d(
                                1, 32,
                                kernel_size=3, stride=1, padding=1
                        ),
                        nn.BatchNorm2d(32),
                        nn.ReLU(inplace=True),
                        nn.Conv2d(
                                32, 64,
                                kernel_size=3, stride=1, padding=1
                        ),
                        nn.BatchNorm2d(64),
                        nn.ReLU(inplace=True),
                        nn.MaxPool2d(kernel_size=2, stride=2),
                        nn.Conv2d(
                                64, 128,
                                kernel_size=3, stride=1, padding=1
                        ),
                        nn.BatchNorm2d(128),
                        nn.ReLU(inplace=True),
                        nn.MaxPool2d(kernel_size=2, stride=2)
                )

                self.linear_block = nn.Sequential(
                        nn.Dropout(p=0.5),
                        nn.Linear(128 * 7 * 7, 128),
                        nn.BatchNorm1d(128),
                        nn.ReLU(inplace=True),
                        nn.Dropout(0.5),
                        nn.Linear(128, 64),
                        nn.BatchNorm1d(64),
                        nn.ReLU(inplace=True),
                        nn.Dropout(0.5),
                        nn.Linear(64, 10)
                )

        def forward(self, x):
                x = self.conv_block(x)
```

```
                x = x.view(x.size(0), -1)
                x = self.linear_block(x)
                return x


model = CNN().to(device)
```

```
optimizer = torch.optim.SGD(
        model.parameters(),
        lr=0.01,
        weight_decay=1e-4
)
```

A.4.2 Training and evaluating the model

Train the model for 40 epochs and evaluate its performance:

```
epochs = range(1, 41)

train_loss = []
test_loss = []
accuracy_list = []
for t in epochs:
        print(f"training␣epoch␣{t}...")

        epoch_train_loss = train(
                train_loader, model, loss_fn, optimizer,
        )
        accuracy, epoch_test_loss = test(
                test_loader, model, loss_fn
        )

        train_loss.extend(epoch_train_loss)
        test_loss.append(epoch_test_loss)
        accuracy_list.append(accuracy)

        print(f"--------------------------------")

torch.save(
        model.state_dict(),
        "models/cnn.pth"
)
print("training␣complete,␣model␣saved")
```

Saving and plotting the accuracy data:

```
save_accuracy_data("models/cnn_accuracy.csv",
                   accuracy_list)
plt.plot(epochs, accuracy_list)
plt.xlabel("Epoch")
plt.ylabel("Accuracy␣(%)")
```

```
plt.title("Convolutional Neural Network Accuracy")
plt.savefig("figures/cnn_accuracy.png")
```

Plotting the training and test loss:

```
plot_loss(
        train_loss, test_loss, epochs,
        "Convolutional Neural Network Loss",
        "cnn_loss.png"
)
```