A4_Exercise_3

November 24, 2023

[5]:
```
import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import MNIST
```

0.0.1 Importing data

[ ]:
```
root = 'data'
classes = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

# load data
dataset = {'train': MNIST(root=root, train=True, download=True),
           'test': MNIST(root=root, train=False, download=True)}

# create dicts for storing sampled data
labels = ['train', 'test']
X = {'train': [], 'test': []}
Y = {'train': [], 'test': []}

# for the training and test datasets
for label in labels:
    # sample 600 points for each class
    for c in classes:
        subset_idx = torch.isin(dataset[label].targets, torch.as_tensor(c))
        X[label].append(dataset[label].data[subset_idx][:600].view(-1, 28*28).
 ↪float()) # flatten to 784x1
        Y[label].append(dataset[label].targets[subset_idx][:600].long())

    # concatenate along the first dimension
    X[label] = torch.cat(X[label], dim=0)
```

```
    Y[label] = torch.cat(Y[label], dim=0)

# print(X['train'].shape)
# print(X['test'].shape)
# print(Y['train'].shape)
# print(Y['test'].shape)
```

0.0.2 Question 1: MLP implementation

[11]:
```
# MLP model
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        # 1. Linear(784, 512) - ReLU
        self.hidden1 = nn.Linear(784, 512)
        self.act1 = nn.ReLU()

        # 2. Linear(512, 512) - BatchNorm(512) - ReLU
        self.hidden2 = nn.Linear(512, 512)
        self.batchNorm2 = nn.BatchNorm1d(512)
        self.act2 = nn.ReLU()

        # 3. Linear(512, 10)
        self.output = nn.Linear(512, 10)

    def forward(self, x):
        x = self.act1(self.hidden1(x))
        x = self.act2(self.batchNorm2(self.hidden2(x)))
        x = self.output(x)
        return x

# instantiate the model, loss function, and optimizer
model = MLP()
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

n_epochs = 20
batch_size = 64

# grabbing training data
x = X['train'].float()
y = Y['train']

accuracies = {'train': [], 'test': []}
losses = {'train': [], 'test': []}

# gradient descent to train
```

```python
for epoch in range(n_epochs):
    model.train()

    start = time.time()

    # generate a random permutation of indices for shuffling
    perm_indices = torch.randperm(len(x))

    for i in range(0, len(x), batch_size):
        # use shuffled indices to create shuffled mini-batches
        batch_indices = perm_indices[i:i + batch_size]
        x_batch = x[batch_indices]
        y_batch = y[batch_indices]

        y_pred = model(x_batch)
        loss = loss_fn(y_pred, y_batch)

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

    model.eval()
    for label in labels:
        with torch.no_grad():
            y_pred = model(X[label].float())
            _, predicted = torch.max(y_pred, dim=1) # returns index of class␣
↪with highest probability
            accuracies[label].append((predicted == Y[label]).float().mean())
            losses[label].append(loss_fn(y_pred, Y[label]).item())

    end = time.time()
    print(f'Epoch [{epoch + 1}/{n_epochs}], Time elapsed: {(end - start):.4f}s')
```

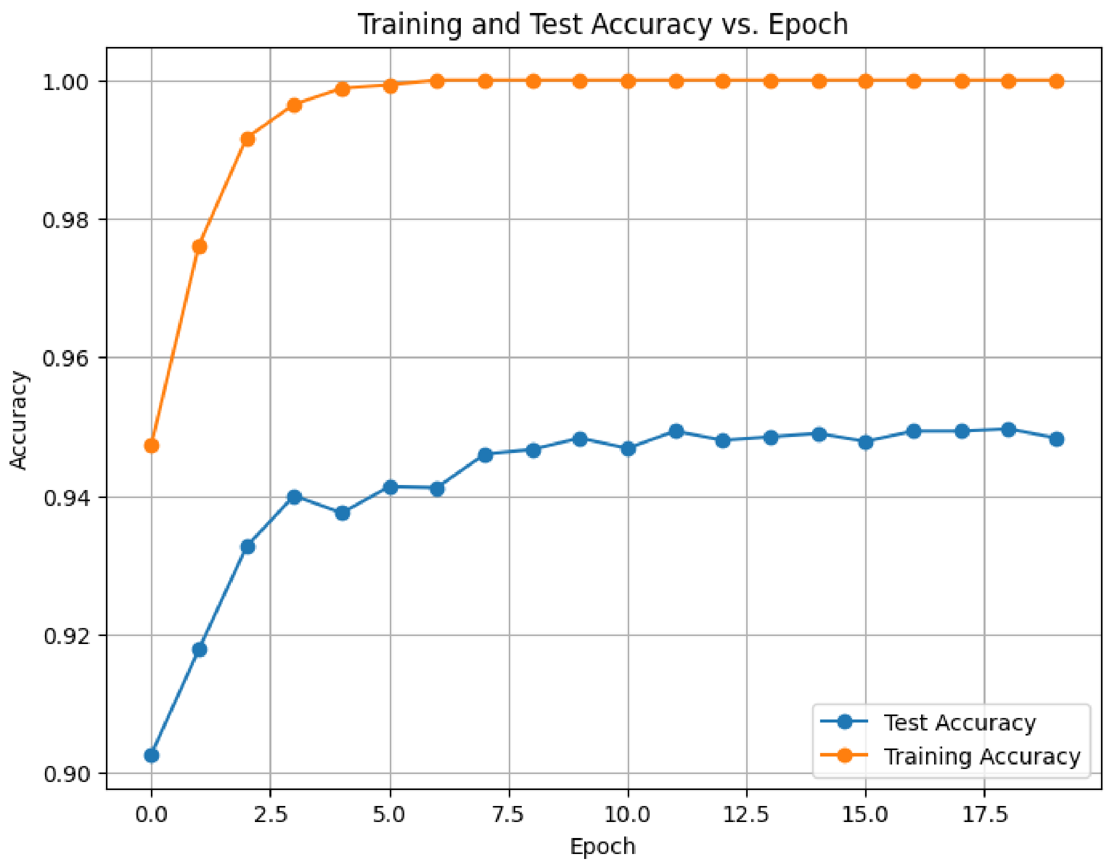
```
Epoch [1/20], Time elapsed: 0.8930s
Epoch [2/20], Time elapsed: 0.8917s
Epoch [3/20], Time elapsed: 0.8447s
Epoch [4/20], Time elapsed: 0.8596s
Epoch [5/20], Time elapsed: 0.9729s
Epoch [6/20], Time elapsed: 1.2832s
Epoch [7/20], Time elapsed: 1.2884s
Epoch [8/20], Time elapsed: 1.2295s
Epoch [9/20], Time elapsed: 0.8895s
Epoch [10/20], Time elapsed: 0.8923s
Epoch [11/20], Time elapsed: 0.8881s
Epoch [12/20], Time elapsed: 0.8909s
Epoch [13/20], Time elapsed: 0.8972s
Epoch [14/20], Time elapsed: 0.8890s
Epoch [15/20], Time elapsed: 0.9088s
```

```
Epoch [16/20], Time elapsed: 0.9056s
Epoch [17/20], Time elapsed: 0.8961s
Epoch [18/20], Time elapsed: 0.8950s
Epoch [19/20], Time elapsed: 0.9238s
Epoch [20/20], Time elapsed: 1.3909s
```

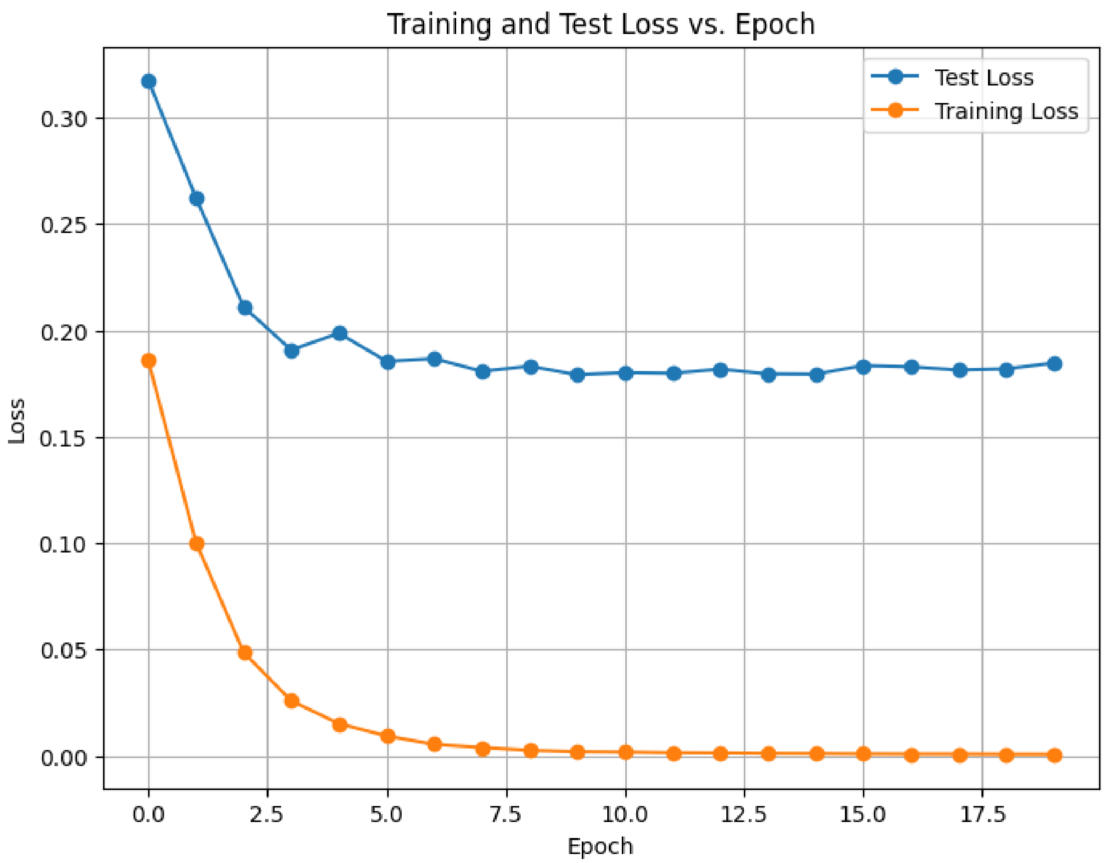
0.0.3 Question 2: Inspecting the training process

[12]:
```python
# accuracy vs number of epochs
import matplotlib.pyplot as plt
epochs = [i for i in range(0, n_epochs)]

plt.figure(figsize=(8, 6))
plt.plot(epochs, accuracies['test'], marker='o', linestyle='-', label='Test␣
↪Accuracy')
plt.plot(epochs, accuracies['train'], marker='o', linestyle='-', label='Training␣
↪Accuracy')
plt.title('Training and Test Accuracy vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```



[13]:
```
# loss vs number of epochs
plt.figure(figsize=(8, 6))
plt.plot(epochs, losses['test'], marker='o', linestyle='-', label='Test Loss')
plt.plot(epochs, losses['train'], marker='o', linestyle='-', label='Training␣
 ↪Loss')
plt.title('Training and Test Loss vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



0.0.4 Question 3: Comparing MLP and VGG11

The MLP model performs much better than the VGG11 model for MNIST classification, as it converges relatively quickly and is very stable over 20 epochs. Its stable test accuracy is only about 1% lower than that of the VGG11 model, which is not a significant difference. This suggests that the VGG11 model overcomplicates the task of classifying the MNIST dataset, by employing a deeper network and hierarchical feature extraction.

0.0.5 Question 4: Adding another layer

[14]:
```
# MLP model
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        # 1. Linear(784, 512) - ReLU
        self.hidden1 = nn.Linear(784, 512)
        self.act1 = nn.ReLU()

        # 2. Linear(512, 512) - BatchNorm(512) - ReLU
```

```python
        self.hidden2 = nn.Linear(512, 512)
        self.batchNorm2 = nn.BatchNorm1d(512)
        self.act2 = nn.ReLU()

        # 3. Linear(512, 512) - BatchNorm(512) - ReLU
        self.hidden3 = nn.Linear(512, 512)
        self.batchNorm3 = nn.BatchNorm1d(512)
        self.act3 = nn.ReLU()

        # 4. Linear(512, 10)
        self.output = nn.Linear(512, 10)

    def forward(self, x):
        x = self.act1(self.hidden1(x))
        x = self.act2(self.batchNorm2(self.hidden2(x)))
        x = self.act3(self.batchNorm3(self.hidden3(x)))
        x = self.output(x)
        return x

# instantiate the model, loss function, and optimizer
model = MLP()
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

n_epochs = 20
batch_size = 64

# grabbing training data
x = X['train'].float()
y = Y['train']

accuracies = {'train': [], 'test': []}
losses = {'train': [], 'test': []}

# gradient descent to train
for epoch in range(n_epochs):
    model.train()

    start = time.time()

    # generate a random permutation of indices for shuffling
    perm_indices = torch.randperm(len(x))

    for i in range(0, len(x), batch_size):
        # use shuffled indices to create shuffled mini-batches
        batch_indices = perm_indices[i:i + batch_size]
        x_batch = x[batch_indices]
```

```
        y_batch = y[batch_indices]

        y_pred = model(x_batch)
        loss = loss_fn(y_pred, y_batch)

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

    model.eval()
    for label in labels:
        with torch.no_grad():
            y_pred = model(X[label].float())
            _, predicted = torch.max(y_pred, dim=1) # returns index of class␣
↪with highest probability
            accuracies[label].append((predicted == Y[label]).float().mean())
            losses[label].append(loss_fn(y_pred, Y[label]).item())

    end = time.time()
    print(f'Epoch [{epoch + 1}/{n_epochs}], Time elapsed: {(end - start):.4f}s')
```

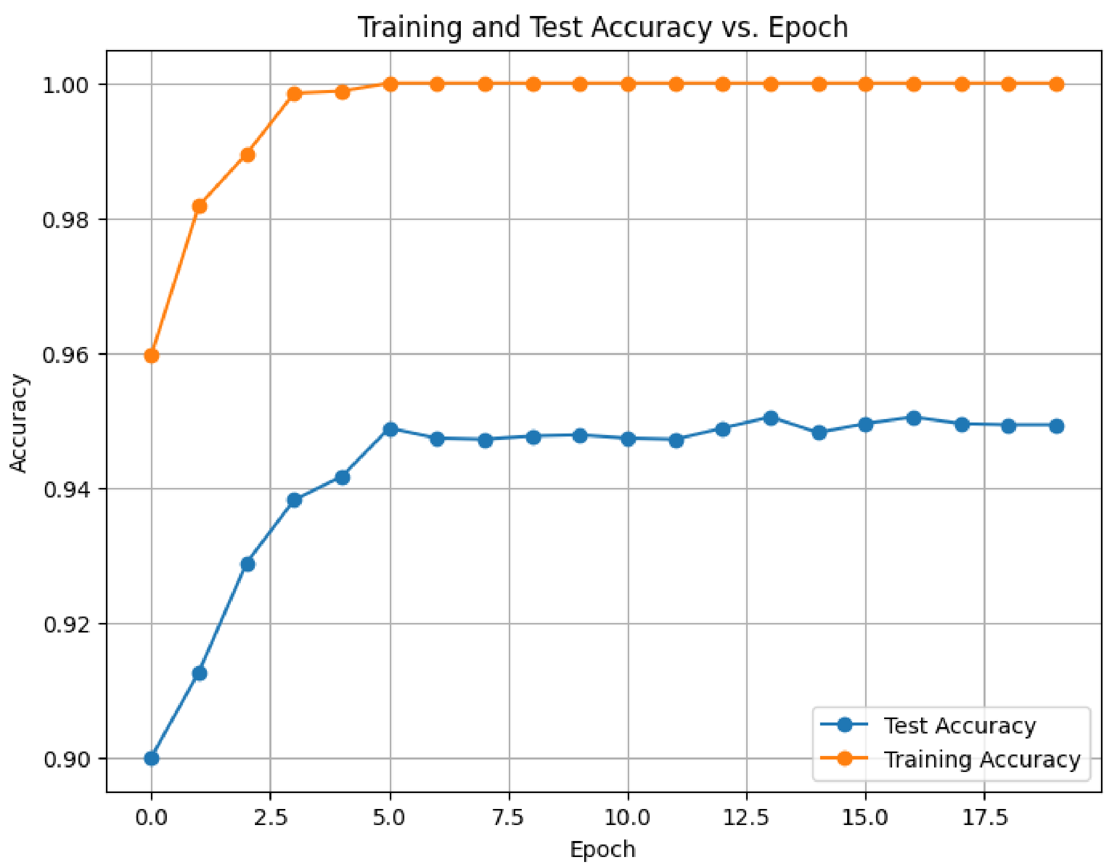
```
Epoch [1/20], Time elapsed: 1.2946s
Epoch [2/20], Time elapsed: 1.2316s
Epoch [3/20], Time elapsed: 1.2387s
Epoch [4/20], Time elapsed: 1.2561s
Epoch [5/20], Time elapsed: 1.2644s
Epoch [6/20], Time elapsed: 1.5446s
Epoch [7/20], Time elapsed: 1.8839s
Epoch [8/20], Time elapsed: 1.6772s
Epoch [9/20], Time elapsed: 1.2212s
Epoch [10/20], Time elapsed: 1.2368s
Epoch [11/20], Time elapsed: 1.2484s
Epoch [12/20], Time elapsed: 1.2824s
Epoch [13/20], Time elapsed: 1.2923s
Epoch [14/20], Time elapsed: 1.2840s
Epoch [15/20], Time elapsed: 1.2636s
Epoch [16/20], Time elapsed: 1.5018s
Epoch [17/20], Time elapsed: 1.9247s
Epoch [18/20], Time elapsed: 1.7779s
Epoch [19/20], Time elapsed: 1.2578s
Epoch [20/20], Time elapsed: 1.2766s
```

[15]:

```
# accuracy vs number of epochs
import matplotlib.pyplot as plt
epochs = [i for i in range(0, n_epochs)]

plt.figure(figsize=(8, 6))
```

```
plt.plot(epochs, accuracies['test'], marker='o', linestyle='-', label='Test␣
 ↪Accuracy')
plt.plot(epochs, accuracies['train'], marker='o', linestyle='-', label='Training␣
 ↪Accuracy')
plt.title('Training and Test Accuracy vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```



When comparing the accuracy vs epoch graphs for the two MLP models, we observe that the network with 4 layers performs better. The 4-layer MLP achieves a higher initial accuracy on the training data and converges in the same number of epochs, which is not a notable difference. However, the 4-layer MLP evidently outperforms the 3-layer MLP evaluating on the testing data, where the former converges in 6 epochs versus 10. This is because the additional layer increases the model's capacity to learn complex patterns through the increase in number of parameters and the addition of a non-linear activation function. As a result, it is able to generalize better and achieves better performance.