A4_Exercise_2

November 24, 2023

[ ]:
```
import time
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import MNIST
```

[1]:
```
import sys
import platform

print(f"Python version: {platform.python_version()}")
print(f"Numpy version: {np.__version__}")
print(f"PyTorch version: {torch.__version__}")
print(f"System: {platform.system()} {platform.release()}")
print(f"Machine: {platform.machine()}")
print(f"Processor: {platform.processor()}")
```

```
Python version: 3.10.12
Numpy version: 1.23.5
PyTorch version: 2.1.0+cu118
System: Linux 5.15.120+
Machine: x86_64
Processor: x86_64
```

0.0.1 Importing data

[ ]:
```
from torchvision.transforms.functional import pad

def resize_batch(imgs):
    resized_imgs = pad(imgs, (2, 2, 2, 2))
    return resized_imgs

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

# define a normalization transform
```

```
normalize = transforms.Normalize(mean=[0.5], std=[0.5])

# load data and select the classes of interest
transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    normalize
])

# load data
labels = ['train', 'test']
dataset = {'train': MNIST(root=root, train=True, download=True,␣
 ↪transform=transform),
           'test': MNIST(root=root, train=False, download=True,␣
 ↪transform=transform)}

# resize images to (32, 32)
for label in labels:
    dataset[label].data = resize_batch(dataset[label].data)

print(dataset['train'].data.shape)
print(type(dataset['train'].data))

# create dicts for storing sampled data
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])
        Y[label].append(dataset[label].targets[subset_idx][:600])

    # 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 Helper functions for troubleshooting

[ ]:
```python
import matplotlib.pyplot as plt

# function to display images
def show_images(images, labels, start_idx=0, num_samples=10):
    fig, axes = plt.subplots(1, num_samples, figsize=(12, 3))
    for i in range(num_samples):
        # convert the tensor to np array
        image_np = images[start_idx + i].numpy()

        axes[i].imshow(image_np, cmap='gray')
        axes[i].set_title(f'Label: {labels[start_idx + i].item()}')
        axes[i].axis('off')
    plt.show()

# display a few samples from X['train']
for i in range(10):
    show_images(X['train'], Y['train'], 600*i)
```

0.0.3 Question 1: VGG11 implementation

[38]:
```python
# VGG11 model
class VGG11(nn.Module):
    def __init__(self):
        super().__init__()
        # 1. Conv(001, 064, 3, 1, 1) - BatchNorm(064) - ReLU - MaxPool(2, 2)
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1)
        self.BatchNorm1 = nn.BatchNorm2d(64)
        self.act1 = nn.ReLU()
        self.maxPool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 2. Conv(064, 128, 3, 1, 1) - BatchNorm(128) - ReLU - MaxPool(2, 2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.BatchNorm2 = nn.BatchNorm2d(128)
        self.act2 = nn.ReLU()
        self.maxPool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 3. Conv(128, 256, 3, 1, 1) - BatchNorm(256) - ReLU
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.BatchNorm3 = nn.BatchNorm2d(256)
        self.act3 = nn.ReLU()

        # 4. Conv(256, 256, 3, 1, 1) - BatchNorm(256) - ReLU - MaxPool(2, 2)
        self.conv4 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.BatchNorm4 = nn.BatchNorm2d(256)
        self.act4 = nn.ReLU()
```

```
        self.maxPool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 5. Conv(256, 512, 3, 1, 1) - BatchNorm(512) - ReLU
        self.conv5 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm5 = nn.BatchNorm2d(512)
        self.act5 = nn.ReLU()

        # 6. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
        self.conv6 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm6 = nn.BatchNorm2d(512)
        self.act6 = nn.ReLU()
        self.maxPool6 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 7. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU
        self.conv7 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm7 = nn.BatchNorm2d(512)
        self.act7 = nn.ReLU()

        # 8. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
        self.conv8 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm8 = nn.BatchNorm2d(512)
        self.act8 = nn.ReLU()
        self.maxPool8 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 9. Linear(0512, 4096) - ReLU - Dropout(0.5)
        self.hidden9 = nn.Linear(512, 4096)
        self.act9 = nn.ReLU()
        self.drop9 = nn.Dropout(0.5)

        # 10. Linear(4096, 4096) - ReLU - Dropout(0.5)
        self.hidden10 = nn.Linear(4096, 4096)
        self.act10 = nn.ReLU()
        self.drop10 = nn.Dropout(0.5)

        # 11. Linear(4096, 10)
        self.output = nn.Linear(4096, 10)

    def forward(self, x):
        # adjust input shape: reshape using view
        x = x.view(x.size(0), 1, 32, 32)

        # convolutional layers
        x = self.maxPool1(self.act1(self.BatchNorm1(self.conv1(x))))
        x = self.maxPool2(self.act2(self.BatchNorm2(self.conv2(x))))
        x = self.act3(self.BatchNorm3(self.conv3(x)))
        x = self.maxPool4(self.act4(self.BatchNorm4(self.conv4(x))))
        x = self.act5(self.BatchNorm5(self.conv5(x)))
```

```
        x = self.maxPool6(self.act6(self.BatchNorm6(self.conv6(x))))
        x = self.act7(self.BatchNorm7(self.conv7(x)))
        x = self.maxPool8(self.act8(self.BatchNorm8(self.conv8(x))))

        # flatten output of convolutional layers
        x = x.view(x.size(0), -1)

        # fully connected layers
        x = self.drop9(self.act9(self.hidden9(x)))
        x = self.drop10(self.act10(self.hidden10(x)))
        x = self.output(x)

        return x
```

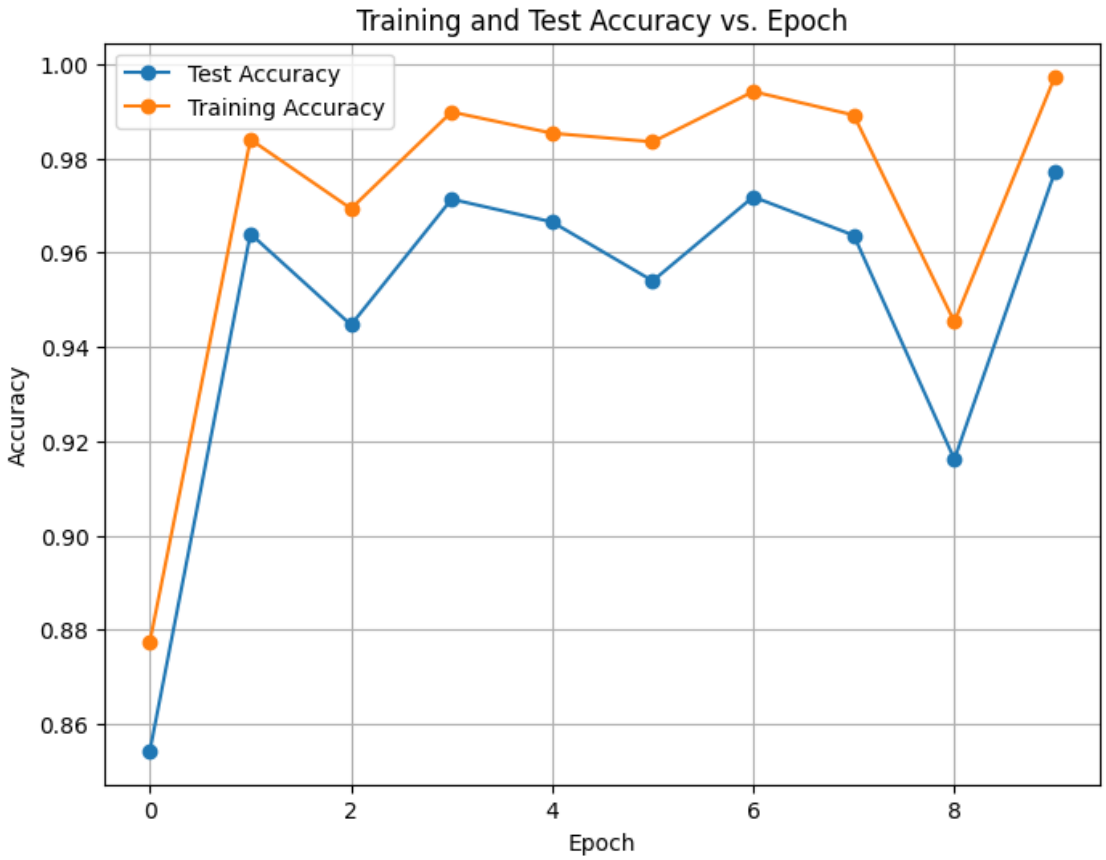
[39]:
```
# instantiate the model, loss function, and optimizer
model = VGG11()
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

n_epochs = 10
batch_size = 64

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

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

for epoch in range(n_epochs):
    model.train() # training mode

    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 [1/10], Time elapsed: 281.3921s
Epoch [2/10], Time elapsed: 276.9482s
Epoch [3/10], Time elapsed: 281.0960s
Epoch [4/10], Time elapsed: 281.6837s
Epoch [5/10], Time elapsed: 276.9935s
Epoch [6/10], Time elapsed: 281.8110s
Epoch [7/10], Time elapsed: 280.3179s
Epoch [8/10], Time elapsed: 277.7942s
Epoch [9/10], Time elapsed: 280.7869s
Epoch [10/10], Time elapsed: 282.0842s
```

0.0.4 Question 2: Inspecting the training process

[40]:
```
# accuracy vs number of epochs
import matplotlib.pyplot as plt
epochs = [i for i in range(0, 10)]

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()
```



[41]:
```
# 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()
```

0.0.5 Question 3: CNN with Adam

[ ]:
```
# VGG11 model
class VGG11(nn.Module):
    def __init__(self):
        super().__init__()
        # 1. Conv(001, 064, 3, 1, 1) - BatchNorm(064) - ReLU - MaxPool(2, 2)
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1)
        self.BatchNorm1 = nn.BatchNorm2d(64)
        self.act1 = nn.ReLU()
        self.maxPool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 2. Conv(064, 128, 3, 1, 1) - BatchNorm(128) - ReLU - MaxPool(2, 2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.BatchNorm2 = nn.BatchNorm2d(128)
        self.act2 = nn.ReLU()
        self.maxPool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 3. Conv(128, 256, 3, 1, 1) - BatchNorm(256) - ReLU
```

```
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.BatchNorm3 = nn.BatchNorm2d(256)
        self.act3 = nn.ReLU()

        # 4. Conv(256, 256, 3, 1, 1) - BatchNorm(256) - ReLU - MaxPool(2, 2)
        self.conv4 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.BatchNorm4 = nn.BatchNorm2d(256)
        self.act4 = nn.ReLU()
        self.maxPool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 5. Conv(256, 512, 3, 1, 1) - BatchNorm(512) - ReLU
        self.conv5 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm5 = nn.BatchNorm2d(512)
        self.act5 = nn.ReLU()

        # 6. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
        self.conv6 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm6 = nn.BatchNorm2d(512)
        self.act6 = nn.ReLU()
        self.maxPool6 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 7. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU
        self.conv7 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm7 = nn.BatchNorm2d(512)
        self.act7 = nn.ReLU()

        # 8. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
        self.conv8 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm8 = nn.BatchNorm2d(512)
        self.act8 = nn.ReLU()
        self.maxPool8 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 9. Linear(0512, 4096) - ReLU - Dropout(0.5)
        self.hidden9 = nn.Linear(512, 4096)
        self.act9 = nn.ReLU()
        self.drop9 = nn.Dropout(0.5)

        # 10. Linear(4096, 4096) - ReLU - Dropout(0.5)
        self.hidden10 = nn.Linear(4096, 4096)
        self.act10 = nn.ReLU()
        self.drop10 = nn.Dropout(0.5)

        # 11. Linear(4096, 10)
        self.output = nn.Linear(4096, 10)

    def forward(self, x):
        # adjust input shape: reshape using view
```

```
        x = x.view(x.size(0), 1, 32, 32)

        # convolutional layers
        x = self.maxPool1(self.act1(self.BatchNorm1(self.conv1(x))))
        x = self.maxPool2(self.act2(self.BatchNorm2(self.conv2(x))))
        x = self.act3(self.BatchNorm3(self.conv3(x)))
        x = self.maxPool4(self.act4(self.BatchNorm4(self.conv4(x))))
        x = self.act5(self.BatchNorm5(self.conv5(x)))
        x = self.maxPool6(self.act6(self.BatchNorm6(self.conv6(x))))
        x = self.act7(self.BatchNorm7(self.conv7(x)))
        x = self.maxPool8(self.act8(self.BatchNorm8(self.conv8(x))))

        # flatten output of convolutional layers
        x = x.view(x.size(0), -1)

        # fully connected layers
        x = self.drop9(self.act9(self.hidden9(x)))
        x = self.drop10(self.act10(self.hidden10(x)))
        x = self.output(x)

        return x
```

[ ]:
```
# instantiate the model, loss function, and optimizer
model = VGG11()
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

n_epochs = 10
batch_size = 64

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

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

for epoch in range(n_epochs):
    model.train() # training mode

    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 [1/10], Time elapsed: 314.2779s
Epoch [2/10], Time elapsed: 300.4791s
Epoch [3/10], Time elapsed: 311.4415s
Epoch [4/10], Time elapsed: 300.8686s
Epoch [5/10], Time elapsed: 309.8756s
Epoch [6/10], Time elapsed: 298.9292s
Epoch [7/10], Time elapsed: 312.8635s
Epoch [8/10], Time elapsed: 335.9928s
Epoch [9/10], Time elapsed: 342.9294s
Epoch [10/10], Time elapsed: 324.8344s
```

[ ]:
```
# 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()
```

The performance of the model using Adam as the optimizer instead of SGD is improved. This can be attributed to Adam's adaptive learning rate, where the learning rates for each parameter are adjusted individually based on the historical gradients. This allows it to converge faster, as seen above. Adam also incorporates a momentum term that does not need to be tuned, and it contributes to faster convergence and reduced oscillations.

0.0.6 Question 4: Replace ReLU with Sigmoid

[ ]:
```
# VGG11 model
class VGG11(nn.Module):
    def __init__(self):
        super().__init__()
        # 1. Conv(001, 064, 3, 1, 1) - BatchNorm(064) - Sigmoid - MaxPool(2, 2)
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1)
        self.BatchNorm1 = nn.BatchNorm2d(64)
        self.act1 = nn.Sigmoid()
```

```
        self.maxPool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 2. Conv(064, 128, 3, 1, 1) - BatchNorm(128) - Sigmoid - MaxPool(2, 2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.BatchNorm2 = nn.BatchNorm2d(128)
        self.act2 = nn.Sigmoid()
        self.maxPool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 3. Conv(128, 256, 3, 1, 1) - BatchNorm(256) - Sigmoid
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.BatchNorm3 = nn.BatchNorm2d(256)
        self.act3 = nn.Sigmoid()

        # 4. Conv(256, 256, 3, 1, 1) - BatchNorm(256) - Sigmoid - MaxPool(2, 2)
        self.conv4 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.BatchNorm4 = nn.BatchNorm2d(256)
        self.act4 = nn.Sigmoid()
        self.maxPool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 5. Conv(256, 512, 3, 1, 1) - BatchNorm(512) - Sigmoid
        self.conv5 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm5 = nn.BatchNorm2d(512)
        self.act5 = nn.Sigmoid()

        # 6. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - Sigmoid - MaxPool(2, 2)
        self.conv6 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm6 = nn.BatchNorm2d(512)
        self.act6 = nn.Sigmoid()
        self.maxPool6 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 7. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - Sigmoid
        self.conv7 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm7 = nn.BatchNorm2d(512)
        self.act7 = nn.Sigmoid()

        # 8. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - Sigmoid - MaxPool(2, 2)
        self.conv8 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm8 = nn.BatchNorm2d(512)
        self.act8 = nn.Sigmoid()
        self.maxPool8 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 9. Linear(0512, 4096) - Sigmoid - Dropout(0.5)
        self.hidden9 = nn.Linear(512, 4096)
        self.act9 = nn.Sigmoid()
        self.drop9 = nn.Dropout(0.5)

        # 10. Linear(4096, 4096) - Sigmoid - Dropout(0.5)
```

```
        self.hidden10 = nn.Linear(4096, 4096)
        self.act10 = nn.Sigmoid()
        self.drop10 = nn.Dropout(0.5)

        # 11. Linear(4096, 10)
        self.output = nn.Linear(4096, 10)

    def forward(self, x):
        # adjust input shape: reshape using view
        x = x.view(x.size(0), 1, 32, 32)

        # convolutional layers
        x = self.maxPool1(self.act1(self.BatchNorm1(self.conv1(x))))
        x = self.maxPool2(self.act2(self.BatchNorm2(self.conv2(x))))
        x = self.act3(self.BatchNorm3(self.conv3(x)))
        x = self.maxPool4(self.act4(self.BatchNorm4(self.conv4(x))))
        x = self.act5(self.BatchNorm5(self.conv5(x)))
        x = self.maxPool6(self.act6(self.BatchNorm6(self.conv6(x))))
        x = self.act7(self.BatchNorm7(self.conv7(x)))
        x = self.maxPool8(self.act8(self.BatchNorm8(self.conv8(x))))

        # flatten output of convolutional layers
        x = x.view(x.size(0), -1)

        # fully connected layers
        x = self.drop9(self.act9(self.hidden9(x)))
        x = self.drop10(self.act10(self.hidden10(x)))
        x = self.output(x)

        return x
```

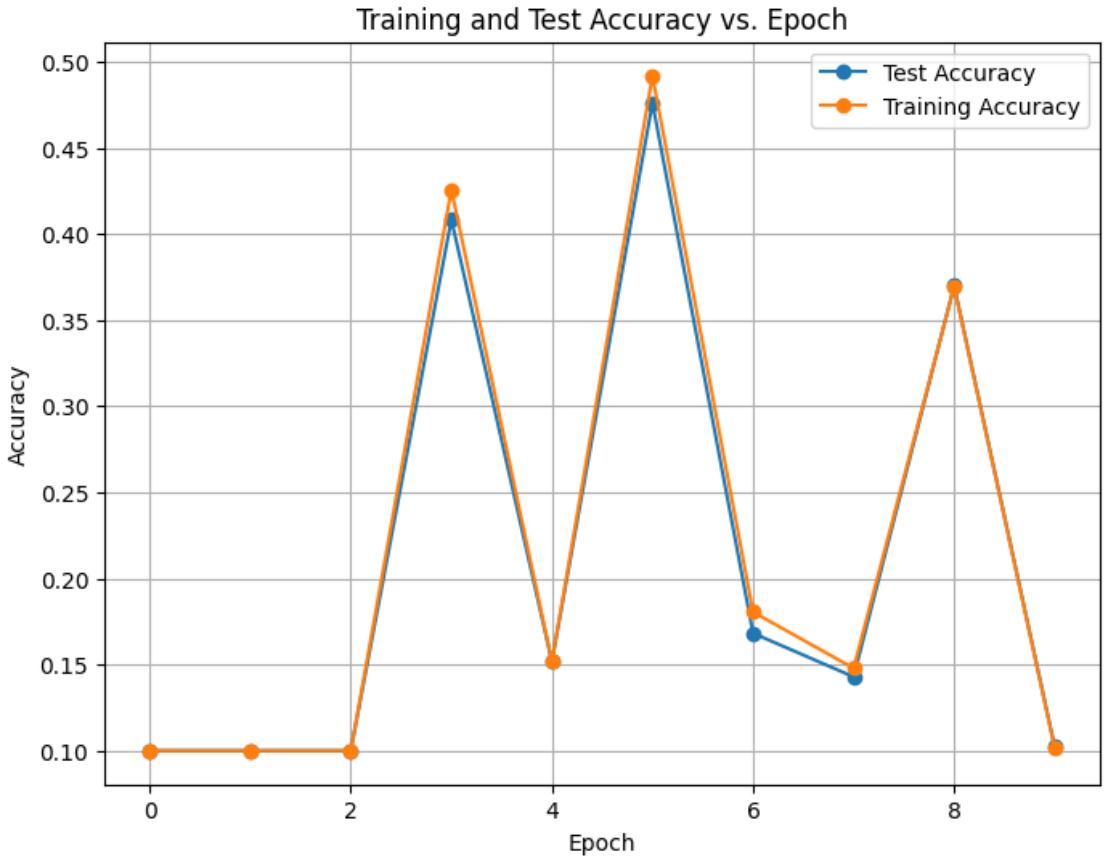
[ ]:
```
# instantiate the model, loss function, and optimizer
model = VGG11()
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

n_epochs = 10
batch_size = 64

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

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

for epoch in range(n_epochs):
```

```
    model.train() # training mode

    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 [1/10], Time elapsed: 300.4017s
Epoch [2/10], Time elapsed: 300.2452s
Epoch [3/10], Time elapsed: 296.5676s
Epoch [4/10], Time elapsed: 297.1819s
Epoch [5/10], Time elapsed: 297.3954s
Epoch [6/10], Time elapsed: 298.7076s
Epoch [7/10], Time elapsed: 297.0209s
Epoch [8/10], Time elapsed: 297.9962s
Epoch [9/10], Time elapsed: 288.9264s
Epoch [10/10], Time elapsed: 289.3546s
```

[34]:
```
# 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()
```



The use of the Sigmoid function for activation results in poorer performance compared to ReLU. Specifically, the accuracy is very low (less than 50%) and is wild oscillating. This is expected, because the Sigmoid activation function forces input values between 0 and 1, and consequently can cause gradients to become extremely small, or "vanish," through multiplication of layers. This makes it difficult for the model to learn and updated the weights effectively, especially in networks with many layers. ReLU does not have the vanishing gradient issue to the same extent and is thus particularly suitable for deep networks.

0.0.7 Question 5: Remove dropout

[35]:
```
# VGG11 model
class VGG11(nn.Module):
    def __init__(self):
        super().__init__()
        # 1. Conv(001, 064, 3, 1, 1) - BatchNorm(064) - Sigmoid - MaxPool(2, 2)
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1)
        self.BatchNorm1 = nn.BatchNorm2d(64)
        self.act1 = nn.Sigmoid()
        self.maxPool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 2. Conv(064, 128, 3, 1, 1) - BatchNorm(128) - Sigmoid - MaxPool(2, 2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.BatchNorm2 = nn.BatchNorm2d(128)
        self.act2 = nn.Sigmoid()
        self.maxPool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 3. Conv(128, 256, 3, 1, 1) - BatchNorm(256) - Sigmoid
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.BatchNorm3 = nn.BatchNorm2d(256)
        self.act3 = nn.Sigmoid()

        # 4. Conv(256, 256, 3, 1, 1) - BatchNorm(256) - Sigmoid - MaxPool(2, 2)
        self.conv4 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.BatchNorm4 = nn.BatchNorm2d(256)
        self.act4 = nn.Sigmoid()
        self.maxPool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 5. Conv(256, 512, 3, 1, 1) - BatchNorm(512) - Sigmoid
        self.conv5 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm5 = nn.BatchNorm2d(512)
        self.act5 = nn.Sigmoid()

        # 6. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - Sigmoid - MaxPool(2, 2)
        self.conv6 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm6 = nn.BatchNorm2d(512)
        self.act6 = nn.Sigmoid()
        self.maxPool6 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 7. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - Sigmoid
        self.conv7 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.BatchNorm7 = nn.BatchNorm2d(512)
        self.act7 = nn.Sigmoid()

        # 8. Conv(512, 512, 3, 1, 1) - BatchNorm(512) - Sigmoid - MaxPool(2, 2)
        self.conv8 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
```

```
        self.BatchNorm8 = nn.BatchNorm2d(512)
        self.act8 = nn.Sigmoid()
        self.maxPool8 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 9. Linear(0512, 4096) - Sigmoid
        self.hidden9 = nn.Linear(512, 4096)
        self.act9 = nn.Sigmoid()

        # 10. Linear(4096, 4096) - Sigmoid
        self.hidden10 = nn.Linear(4096, 4096)
        self.act10 = nn.Sigmoid()

        # 11. Linear(4096, 10)
        self.output = nn.Linear(4096, 10)

    def forward(self, x):
        # adjust input shape: reshape using view
        x = x.view(x.size(0), 1, 32, 32)

        # convolutional layers
        x = self.maxPool1(self.act1(self.BatchNorm1(self.conv1(x))))
        x = self.maxPool2(self.act2(self.BatchNorm2(self.conv2(x))))
        x = self.act3(self.BatchNorm3(self.conv3(x)))
        x = self.maxPool4(self.act4(self.BatchNorm4(self.conv4(x))))
        x = self.act5(self.BatchNorm5(self.conv5(x)))
        x = self.maxPool6(self.act6(self.BatchNorm6(self.conv6(x))))
        x = self.act7(self.BatchNorm7(self.conv7(x)))
        x = self.maxPool8(self.act8(self.BatchNorm8(self.conv8(x))))

        # flatten output of convolutional layers
        x = x.view(x.size(0), -1)

        # fully connected layers
        x = self.act9(self.hidden9(x))
        x = self.act10(self.hidden10(x))
        x = self.output(x)

        return x
```

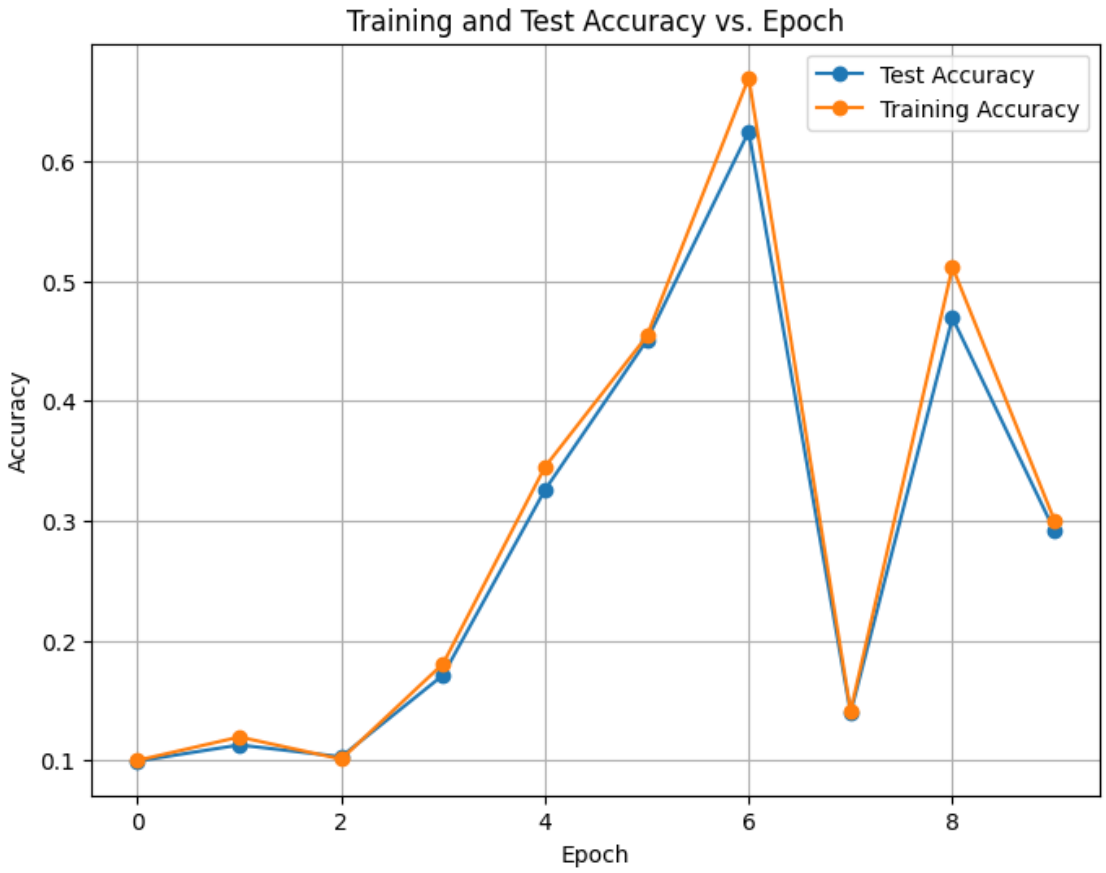
[36]:
```
# instantiate the model, loss function, and optimizer
model = VGG11()
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

n_epochs = 10
batch_size = 64
```

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

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

for epoch in range(n_epochs):
    model.train() # training mode

    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 [1/10], Time elapsed: 298.3262s
Epoch [2/10], Time elapsed: 294.2948s
Epoch [3/10], Time elapsed: 296.4877s
Epoch [4/10], Time elapsed: 289.8472s
Epoch [5/10], Time elapsed: 295.5214s
Epoch [6/10], Time elapsed: 296.8478s
Epoch [7/10], Time elapsed: 297.7347s
Epoch [8/10], Time elapsed: 296.8194s
```

```
Epoch [9/10], Time elapsed: 294.8299s
Epoch [10/10], Time elapsed: 299.3141s
```

[37]:
```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()
```



The removal of Dropout appears to improve the model by reducing oscillations, although I cannot

confirm if this is a pattern that is seen beyond 10 epochs. Dropout is typically used to prevent overfitting by randomly dropping out a fraction of neurons during training. While it normally enhances generalization by preventing the network from relying on certain neurons, it can also introduce instability during training. This might explain the seemingly improved behaviour with the removal of Dropout.