demo01-basics

February 17, 2025

1 PyTorch Basics

I'll assume that everyone is familiar with python. Training neural nets in bare python is somewhat painful, but fortunately there are several well-established libraries which can help. I like pytorch, which is built upon an earlier library called torch. There are many others, including TensorFlow and Jax.

[3]:
```
# We start by importing the libraries we'll use today
import numpy as np
import torch
import torchvision
```

[4]:
```
a = np.random.rand(2,3)
b = torch.from_numpy(a)

print(a)
print(b)
```

```
[[0.37643267 0.27951592 0.52061261]
 [0.74981113 0.78557376 0.41705033]]
tensor([[0.3764, 0.2795, 0.5206],
        [0.7498, 0.7856, 0.4171]], dtype=torch.float64)
```

[5]:
```
print(b + 10.0)
print()
print(torch.sin(b))
print()
print(b.sum())
print()
print(b.mean())
print()
print(b.shape)
```

```
tensor([[10.3764, 10.2795, 10.5206],
        [10.7498, 10.7856, 10.4171]], dtype=torch.float64)

tensor([[0.3676, 0.2759, 0.4974],
        [0.6815, 0.7072, 0.4051]], dtype=torch.float64)
```

```
tensor(3.1290, dtype=torch.float64)
```

```
tensor(0.5215, dtype=torch.float64)
```

```
torch.Size([2, 3])
```

Torch believes everything is a *tensor*.

The main intuition is that tensors allow for intuitive and efficient matrix multiplication across different indexing dimensions. Soon, we will see that training neural nets basically consits of *forward* and *backward* passes, both of which are essentially matrix multiplies.

The other thing about torch variables is that they (natively) can be differentiated. Again, we'll see why this is important when we learn about backpropagation.

Suppose we want dy/da in the following expression: - $y = a + b$

[6]:
```
a = torch.rand(1,1, requires_grad=True)
b = torch.rand(1,1)
y = a + b
print("a:", a)
print("b:", b)
print("y:", y)
```

```
a: tensor([[0.7354]], requires_grad=True)
b: tensor([[0.5111]])
y: tensor([[1.2465]], grad_fn=<AddBackward0>)
```

Here, y is a function of the input a so we can use PyTorch to compute dy/da

[7]:
```
y.backward()
print("dy/da:", a.grad)
```

```
dy/da: tensor([[1.]])
```

Let's try this again with a more complex function: - $y = a^2 \cdot b$

[8]:
```
a = torch.rand(1,1, requires_grad=True)
b = torch.rand(1,1)
y = (a**2)*b
print("a:", a)
print("b:", b)
print("y:", y)
y.backward()
print("dy/da:", a.grad)
print("dy/da:", 2 * a * b)
```

```
a: tensor([[0.3146]], requires_grad=True)
b: tensor([[0.9741]])
y: tensor([[0.0964]], grad_fn=<MulBackward0>)
dy/da: tensor([[0.6130]])
dy/da: tensor([[0.6130]], grad_fn=<MulBackward0>)
```

Torch has calculated dy/da using backpropagation which is in agreement with our answer calculated using standard differentiation rules.

Here is an example with matrices and vectors:

```
[9]: A = torch.rand(2,2)
     b = torch.rand(2,1)
     x = torch.rand(2,1, requires_grad = True)

     y = torch.matmul(A, x) + b
     z = y.sum()
```

Here, z is a function of the input x. Let us now compute the derivative of z with respect to x using backpropagation.

```
[10]: z.backward()
      print(x)
      print(x.grad)
```

```
tensor([[0.4033],
        [0.6091]], requires_grad=True)
tensor([[1.2150],
        [0.0599]])
```

1.1 Training simple models

Let's jump in with our first, simple model. We will train a logistic classifier (equivalent to using a single-layer neural network) on a popular image dataset called *Fashion-MNIST*. Torchvision also has several other image datasets which we can directly load as variables.

```
[11]: trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/
       ↪',train=True,download=True,transform=torchvision.transforms.ToTensor())
      testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/
       ↪',train=False,download=True,transform=torchvision.transforms.ToTensor())
```

Let's check that everything has been downloaded.

```
[12]: print(len(trainingdata))
      print(len(testdata))
```

```
60000
10000
```

Let's investigate to see what's inside the dataset.

```
[13]: image, label = trainingdata[0]
      print(image.shape, label)
```

```
torch.Size([1, 28, 28]) 9
```

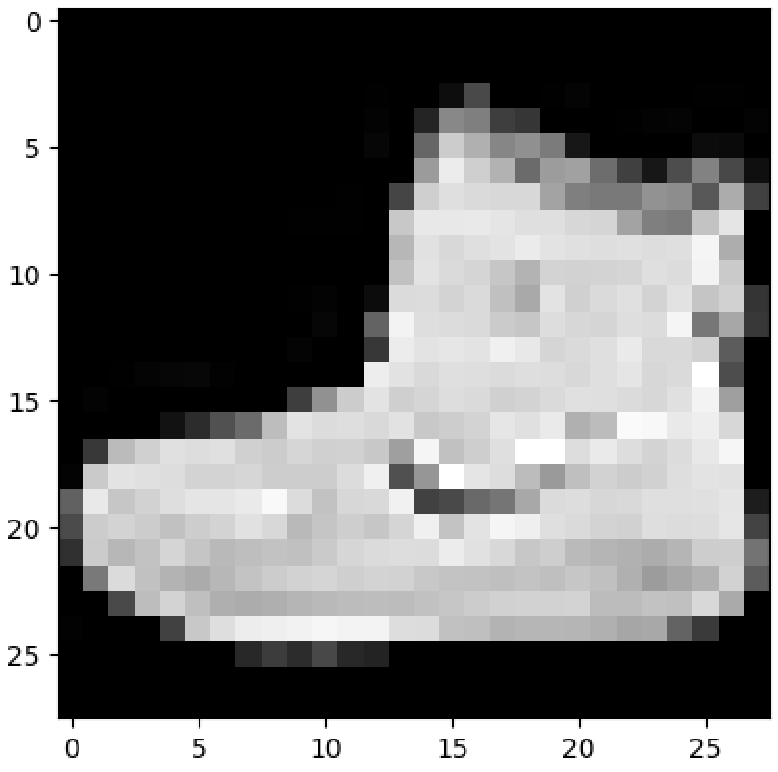
We cannot directly plot the image object given that its first dimension has a size of 1. So we will use the `squeeze` function to get rid of the first dimension.

[14]:
```
print(image.squeeze().shape)
```

```
torch.Size([28, 28])
```

[15]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(image.squeeze(), cmap=plt.cm.gray)
```

[15]:
```
<matplotlib.image.AxesImage at 0x146aee360>
```



Looks like a shoe? Fashion-MNIST is a bunch of different black and white images of clothing with a corresponding label identifying the category the clothing belongs to. It looks like label 9 corresponds to shoes.

In order to nicely wrap the process of iterating through the dataset, we'll use a dataloader.

[16]:
```
trainDataLoader = torch.utils.data.
  ↪DataLoader(trainingdata,batch_size=64,shuffle=True)
testDataLoader = torch.utils.data.
  ↪DataLoader(testdata,batch_size=64,shuffle=False)
```

Let's also check the length of the train and test dataloader

[17]:
```
print(len(trainDataLoader))
print(len(testDataLoader))
```

```
938
157
```

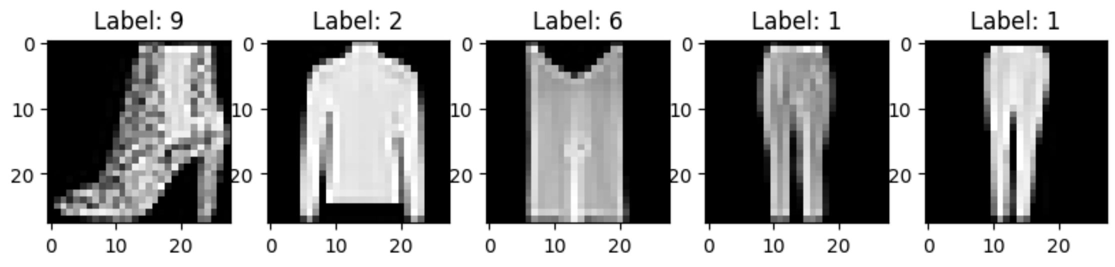
The length here depends upon the batch size defined above. Multiplying the length of our dataloader by the batch size should give us back the number of samples in each set.

[18]:
```
print(len(trainDataLoader) * 64) # batch_size from above
print(len(testDataLoader) * 64)
```

```
60032
10048
```

Now let's use it to look at a few images.

[19]:
```
images, labels = next(iter(trainDataLoader))

plt.figure(figsize=(10,4))
for index in np.arange(0,5):
  plt.subplot(1,5,index+1)
  plt.title(f'Label: {labels[index].item()}')
  plt.imshow(images[index].squeeze(),cmap=plt.cm.gray)
```



Now let's set up our model.

[20]:
```
class LinearReg(torch.nn.Module):
  def __init__(self):
    super(LinearReg, self).__init__()
    self.linear = torch.nn.Linear(28*28, 10)

  def forward(self, x):
    x = x.view(-1, 28*28) # change so 784 vector instead of 28x28 matrix
    return self.linear(x)

model = LinearReg().to("mps")
loss = torch.nn.CrossEntropyLoss() # Step 2: loss
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # Step 3: training␣
  ↪method
```

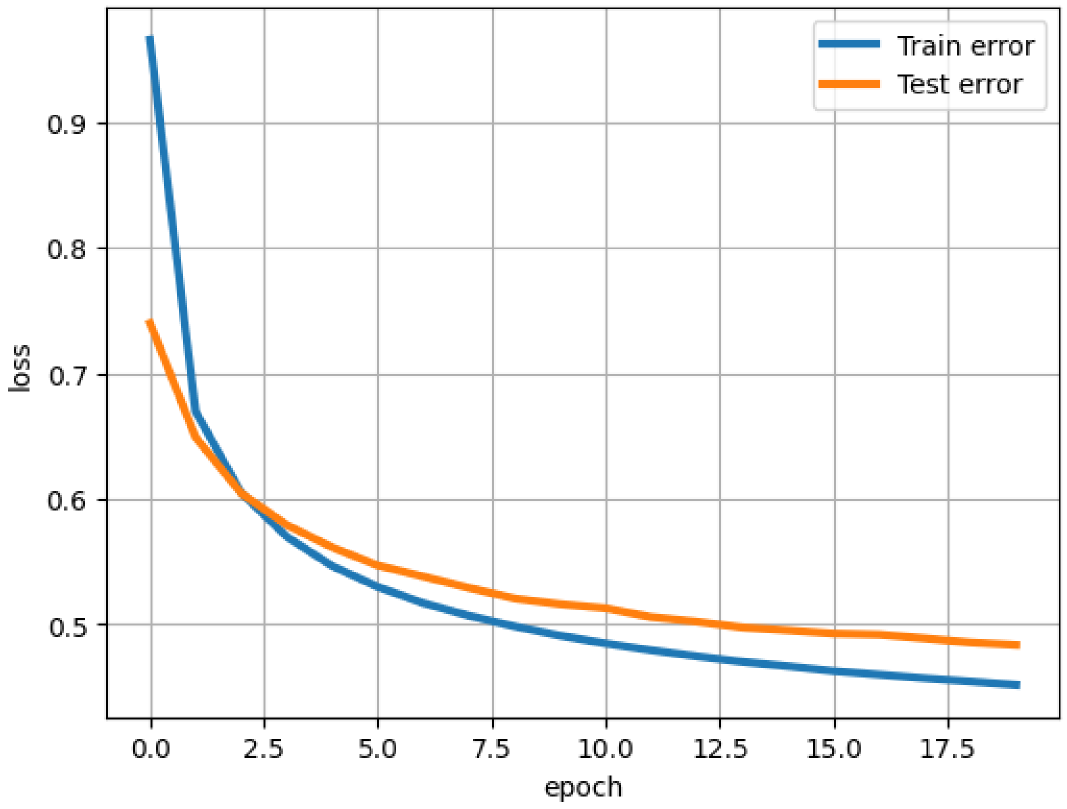
Now let's train our model!

[21]:
```
train_loss_history = []
test_loss_history = []

for epoch in range(20):
  train_loss = 0.0
  test_loss = 0.0

  model.train()
  for i, data in enumerate(trainDataLoader):
    images, labels = data
    images = images.to("mps")
    labels = labels.to("mps")
    optimizer.zero_grad() # zero out any gradient values from the previous␣
  ↪iteration
    predicted_output = model(images) # forward propagation
    fit = loss(predicted_output, labels)  # calculate our measure of goodness
    fit.backward() # backpropagation
    optimizer.step() # update the weights of our trainable parameters
    train_loss += fit.item()

  model.eval()
  for i, data in enumerate(testDataLoader):
    with torch.no_grad():
      images, labels = data
      images = images.to("mps")
      labels = labels.to("mps")
      predicted_output = model(images)
      fit = loss(predicted_output, labels)
      test_loss += fit.item()
  train_loss = train_loss / len(trainDataLoader)
  test_loss = test_loss / len(testDataLoader)
  train_loss_history += [train_loss]
  test_loss_history += [test_loss]
  print(f'Epoch {epoch}, Train loss {train_loss}, Test loss {test_loss}')
```

```
Epoch 0, Train loss 0.9653658385851236, Test loss 0.7398990548340378
Epoch 1, Train loss 0.6695687851227169, Test loss 0.6489718693077184
Epoch 2, Train loss 0.60478076338768, Test loss 0.6044306525379229
Epoch 3, Train loss 0.5698058851746354, Test loss 0.5791268225308437
Epoch 4, Train loss 0.5464037097911082, Test loss 0.5613235191554781
Epoch 5, Train loss 0.5298061437571226, Test loss 0.5468807429265065
Epoch 6, Train loss 0.5169029303514627, Test loss 0.5380505723938062
Epoch 7, Train loss 0.506880147148297, Test loss 0.5290696228006083
Epoch 8, Train loss 0.49837095434985945, Test loss 0.5206071379458069
Epoch 9, Train loss 0.491007157067246, Test loss 0.5160953309505608
Epoch 10, Train loss 0.484764043043163, Test loss 0.5128690571921646
Epoch 11, Train loss 0.4794131822105664, Test loss 0.5057890430377547
Epoch 12, Train loss 0.47458388548352315, Test loss 0.5021672539270607
Epoch 13, Train loss 0.47021598972554907, Test loss 0.49749891374521193
Epoch 14, Train loss 0.4666251786100839, Test loss 0.4951605720884481
Epoch 15, Train loss 0.46287032354996405, Test loss 0.4926913665358428
Epoch 16, Train loss 0.4599651808836567, Test loss 0.4918951703484651
Epoch 17, Train loss 0.4570500953301692, Test loss 0.4888207606828896
Epoch 18, Train loss 0.454567590470253, Test loss 0.4855632717442361
Epoch 19, Train loss 0.45179840212247013, Test loss 0.4836295729230164
```

Let's plot our loss by training epoch to see how we did.

[22]:
```
plt.plot(range(20),train_loss_history,'-',linewidth=3,label='Train error')
plt.plot(range(20),test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
plt.show()
```



Why is test loss larger than training loss?

We definitely see some improvement. Let's look at the images, the predictions our model makes and the true label.

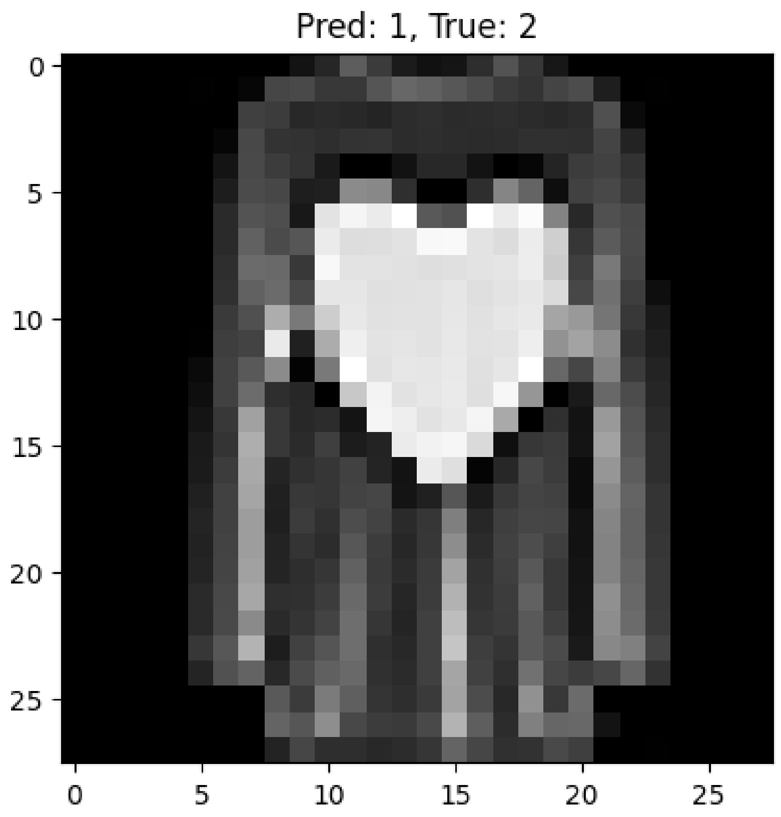
Now for the labels and predicted labels.

[23]:
```
predicted_outputs = model(images)
predicted_classes = torch.max(predicted_outputs, 1)[1]
print('Predicted:', predicted_classes)
fit = loss(predicted_output, labels)
print('True labels:', labels)
print(fit.item())
```

```
Predicted: tensor([3, 1, 7, 5, 8, 2, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5],
device='mps:0')
True labels: tensor([3, 2, 7, 5, 8, 4, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5],
device='mps:0')
0.31733202934265137
```

[24]:
```
plt.imshow(images[1].squeeze().cpu(), cmap=plt.cm.gray)
plt.title(f'Pred: {predicted_classes[1].item()}, True: {labels[1].item()}')
```

[24]:
```
Text(0.5, 1.0, 'Pred: 1, True: 2')
```



1.2 Training a more complex model

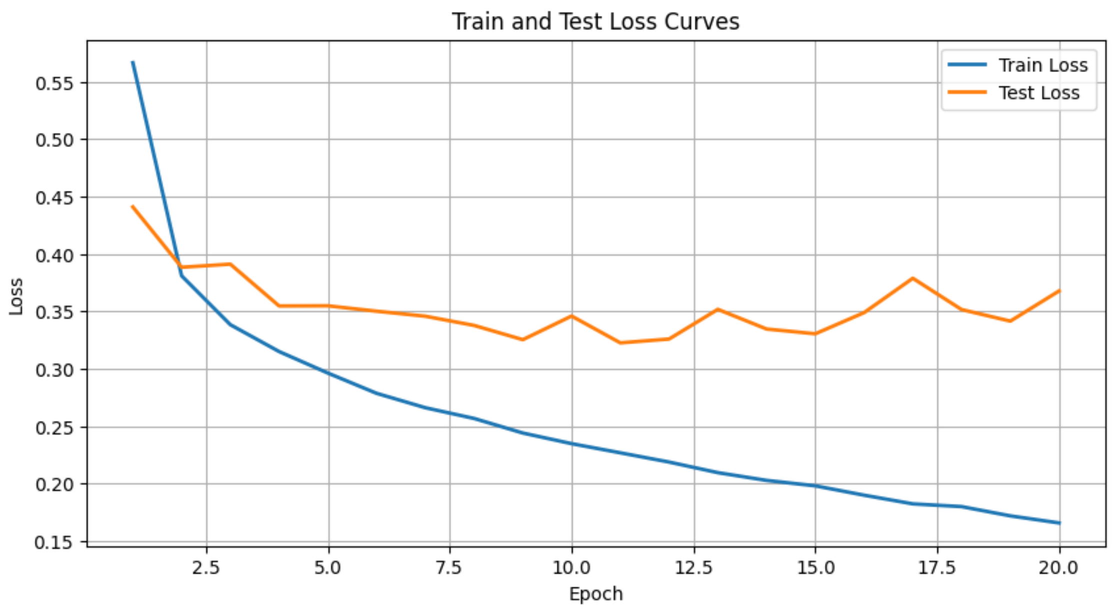
[25]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import random
# Load FashionMNIST datasets
train_data = torchvision.datasets.FashionMNIST(
    './FashionMNIST/', train=True, download=True, transform=transforms.
 ↪ToTensor()
)
test_data = torchvision.datasets.FashionMNIST(
    './FashionMNIST/', train=False, download=True, transform=transforms.
 ↪ToTensor()
)

# Create DataLoaders
batch_size = 64
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,␣
 ↪shuffle=True)
test_loader  = torch.utils.data.DataLoader(test_data, batch_size=batch_size,␣
 ↪shuffle=False)
```

[26]:
```
class DenseNet(torch.nn.Module):
    def __init__(self):
        super(DenseNet, self).__init__()
        self.flatten = torch.nn.Flatten()  # Flatten 28x28 images into 784-dim␣
 ↪vectors
        self.fc1 = torch.nn.Linear(28*28, 256)
        self.relu1 = torch.nn.ReLU()
        self.fc2 = torch.nn.Linear(256, 128)
        self.relu2 = torch.nn.ReLU()
        self.fc3 = torch.nn.Linear(128, 64)
        self.relu3 = torch.nn.ReLU()
        self.fc4 = torch.nn.Linear(64, 10)  # 10 output classes

    def forward(self, x):
        x = self.flatten(x)
        x = self.relu1(self.fc1(x))
        x = self.relu2(self.fc2(x))
        x = self.relu3(self.fc3(x))
        x = self.fc4(x)
        return x

model = DenseNet().to("mps")

# Loss function and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

[27]:
```
# Training loop
num_epochs = 20
train_loss_history = []
test_loss_history = []
test_accuracy_history = []

for epoch in range(num_epochs):
    model.train()
    running_train_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to("mps"), labels.to("mps")

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

        running_train_loss += loss.item() * images.size(0)
    train_loss = running_train_loss / len(train_loader.dataset)
    train_loss_history.append(train_loss)

    # Evaluate on test set
    model.eval()
    running_test_loss = 0.0
    correct = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to("mps"), labels.to("mps")
            outputs = model(images)
            loss = criterion(outputs, labels)
            running_test_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
    test_loss = running_test_loss / len(test_loader.dataset)
    test_loss_history.append(test_loss)
    accuracy = 100 * correct / len(test_loader.dataset)
    test_accuracy_history.append(accuracy)
    print(f"Epoch {epoch+1}/{num_epochs}: Train Loss = {train_loss:.4f}, Test␣
↪Loss = {test_loss:.4f}, Test Accuracy = {accuracy:.2f}%")
```

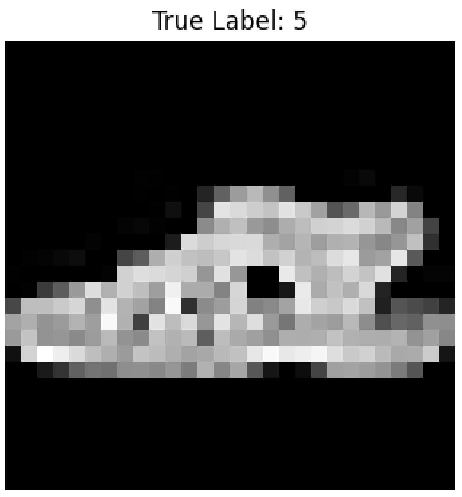
```
Epoch 1/20: Train Loss = 0.5664, Test Loss = 0.4408, Test Accuracy = 83.72%
Epoch 2/20: Train Loss = 0.3808, Test Loss = 0.3882, Test Accuracy = 85.85%
Epoch 3/20: Train Loss = 0.3382, Test Loss = 0.3909, Test Accuracy = 85.80%
Epoch 4/20: Train Loss = 0.3148, Test Loss = 0.3545, Test Accuracy = 87.23%
Epoch 5/20: Train Loss = 0.2961, Test Loss = 0.3547, Test Accuracy = 87.32%
Epoch 6/20: Train Loss = 0.2784, Test Loss = 0.3500, Test Accuracy = 87.47%
Epoch 7/20: Train Loss = 0.2659, Test Loss = 0.3456, Test Accuracy = 88.00%
Epoch 8/20: Train Loss = 0.2566, Test Loss = 0.3376, Test Accuracy = 88.42%
Epoch 9/20: Train Loss = 0.2439, Test Loss = 0.3252, Test Accuracy = 88.75%
Epoch 10/20: Train Loss = 0.2346, Test Loss = 0.3458, Test Accuracy = 88.43%
Epoch 11/20: Train Loss = 0.2266, Test Loss = 0.3224, Test Accuracy = 88.80%
Epoch 12/20: Train Loss = 0.2186, Test Loss = 0.3257, Test Accuracy = 89.07%
Epoch 13/20: Train Loss = 0.2094, Test Loss = 0.3516, Test Accuracy = 88.03%
Epoch 14/20: Train Loss = 0.2026, Test Loss = 0.3345, Test Accuracy = 88.61%
Epoch 15/20: Train Loss = 0.1979, Test Loss = 0.3303, Test Accuracy = 88.64%
Epoch 16/20: Train Loss = 0.1898, Test Loss = 0.3487, Test Accuracy = 88.58%
Epoch 17/20: Train Loss = 0.1822, Test Loss = 0.3787, Test Accuracy = 88.16%
Epoch 18/20: Train Loss = 0.1798, Test Loss = 0.3514, Test Accuracy = 89.00%
Epoch 19/20: Train Loss = 0.1717, Test Loss = 0.3413, Test Accuracy = 89.24%
Epoch 20/20: Train Loss = 0.1656, Test Loss = 0.3676, Test Accuracy = 88.88%
```

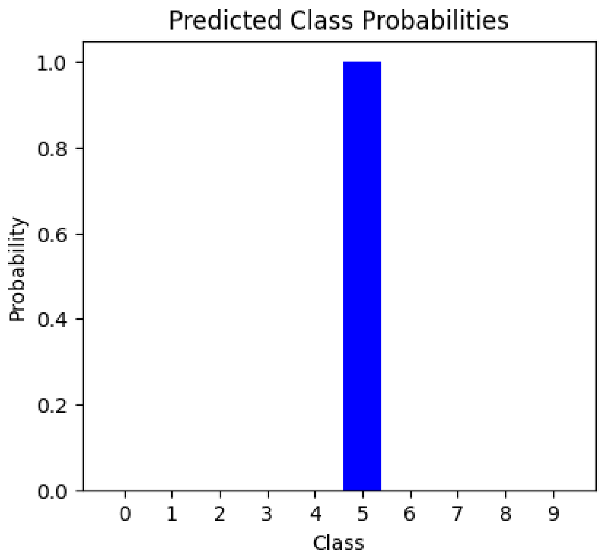
[28]:
```
# Plot loss curves
plt.figure(figsize=(10,5))
plt.plot(range(1, num_epochs+1), train_loss_history, label='Train Loss',␣
 ↪linewidth=2)
plt.plot(range(1, num_epochs+1), test_loss_history, label='Test Loss',␣
 ↪linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Train and Test Loss Curves")
plt.legend()
plt.grid(True)
plt.show()

# Report final test accuracy
print(f"Final Test Accuracy: {test_accuracy_history[-1]:.2f}%")
```

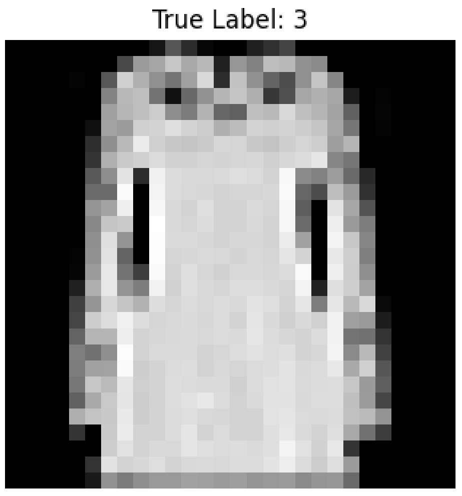


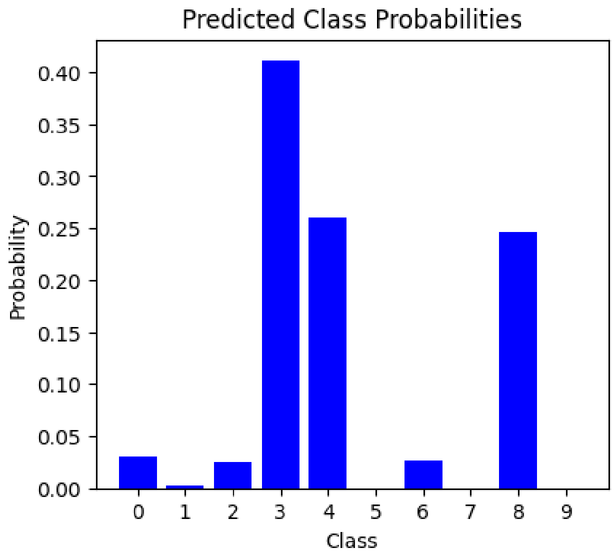
```
Final Test Accuracy: 88.88%
```

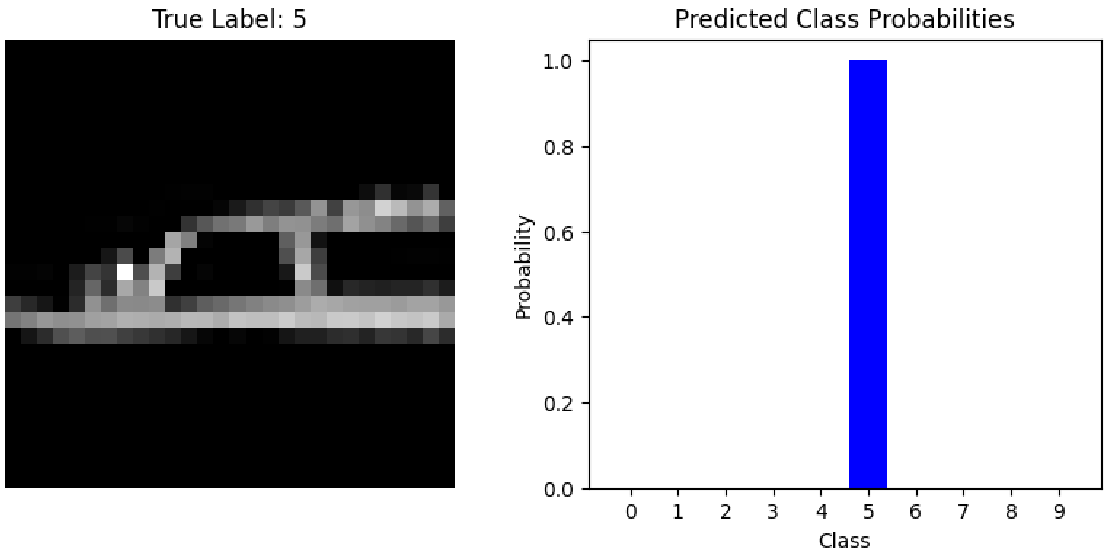
[29]:
```
import random
# Visualize predictions for 3 random test images
model.eval()
for _ in range(3):
    idx = random.randint(0, len(test_data) - 1)
    image, true_label = test_data[idx]

    # Prepare image for the model: add batch dimension and send to device
    input_img = image.unsqueeze(0).to("mps")
    output = model(input_img)

    # Convert logits to probabilities
    probabilities = torch.softmax(output, dim=1).cpu().detach().numpy()[0]

    # Plot the image and a bar chart of predicted class probabilities
    plt.figure(figsize=(10,4))

    # Image subplot
    plt.subplot(1,2,1)
    plt.imshow(image.squeeze(), cmap='gray')
    plt.title(f"True Label: {true_label}")
    plt.axis("off")

    # Bar chart subplot
    plt.subplot(1,2,2)
    plt.bar(np.arange(10), probabilities, color='blue')
    plt.xlabel("Class")
    plt.ylabel("Probability")
    plt.title("Predicted Class Probabilities")
    plt.xticks(np.arange(10))

    plt.show()
```










True Label: 5
Predicted Class Probabilities
Probability
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6
7
8
9
Class