hw_1

May 11, 2025

[25]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

# PyTorch TensorBoard support
# from torch.utils.tensorboard import SummaryWriter
# import torchvision
# import torchvision.transforms as transforms

from datetime import datetime

import torchvision
import torchvision.transforms as transforms

from torchvision.datasets import FashionMNIST
import matplotlib.pyplot as plt
%matplotlib inline

from torch.utils.data import random_split
from torch.utils.data import DataLoader
import torch.nn.functional as F

from PIL import Image
#import torchvision.transforms as T
```

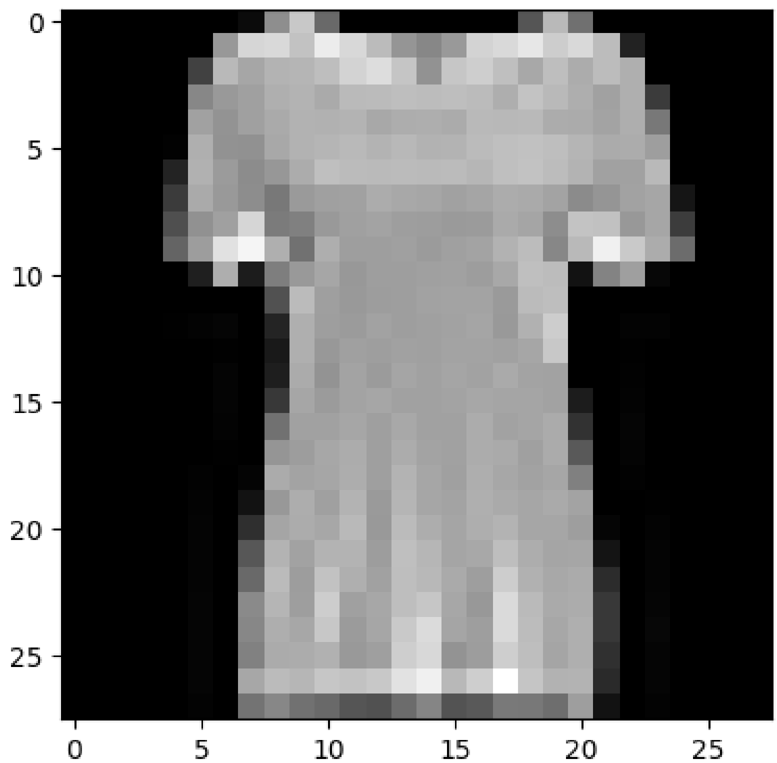
[ ]:
```
# load the dataset
fmnist_dataset = FashionMNIST(root = 'data/', download=True, train = True,␣
 ↪transform = transforms.ToTensor())
print(fmnist_dataset)
```

```
Dataset FashionMNIST
    Number of datapoints: 60000
    Root location: data/
    Split: Train
    StandardTransform
Transform: ToTensor()
```

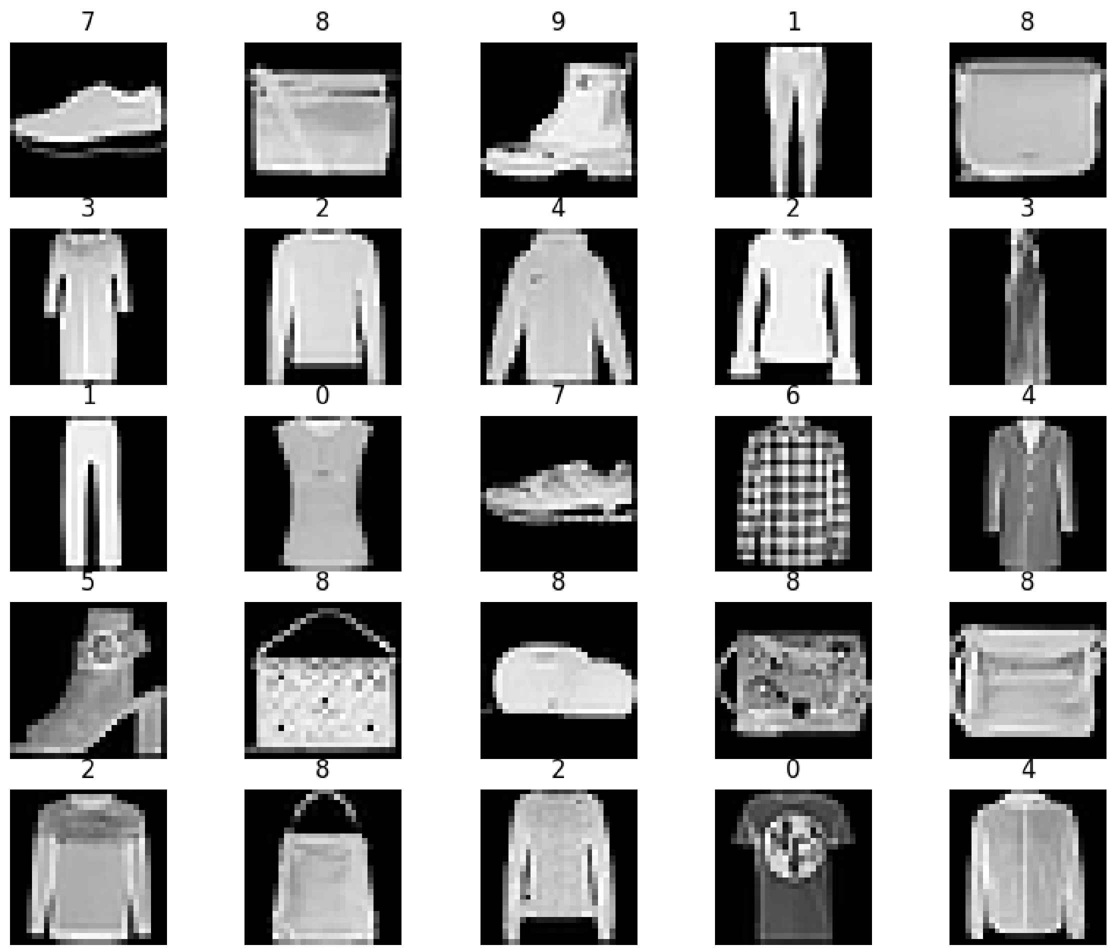
[ ]:
```python
# fmnist_dataset has 'images as tensors' so that they can't be displayed␣
↪directly
sampleTensor, label = fmnist_dataset[10]
print(sampleTensor.shape, label)
tpil = transforms.ToPILImage() # using the __call__ to
image = tpil(sampleTensor)
image.show()

# The image is now convert to a 28 X 28 tensor.
# The first dimension is used to keep track of the color channels.
# Since images in the MNIST dataset are grayscale, there's just one channel.
# The values range from 0 to 1, with 0 representing black, 1 white and the␣
↪values between different shades of grey.
print(sampleTensor[:,10:15,10:15])
print(torch.max(sampleTensor), torch.min(sampleTensor))
plt.imshow(sampleTensor[0,:,:],cmap = 'gray')
```

```
torch.Size([1, 28, 28]) 0
tensor([[[0.6510, 0.5961, 0.6196, 0.6196, 0.6275],
         [0.6235, 0.6000, 0.6157, 0.6196, 0.6353],
         [0.6196, 0.6078, 0.6353, 0.6196, 0.6275],
         [0.5961, 0.6275, 0.6196, 0.6314, 0.6275],
         [0.5765, 0.6431, 0.6078, 0.6471, 0.6314]]])
tensor(1.) tensor(0.)
```

[ ]:
```
<matplotlib.image.AxesImage at 0x7f2eb5120140>
```



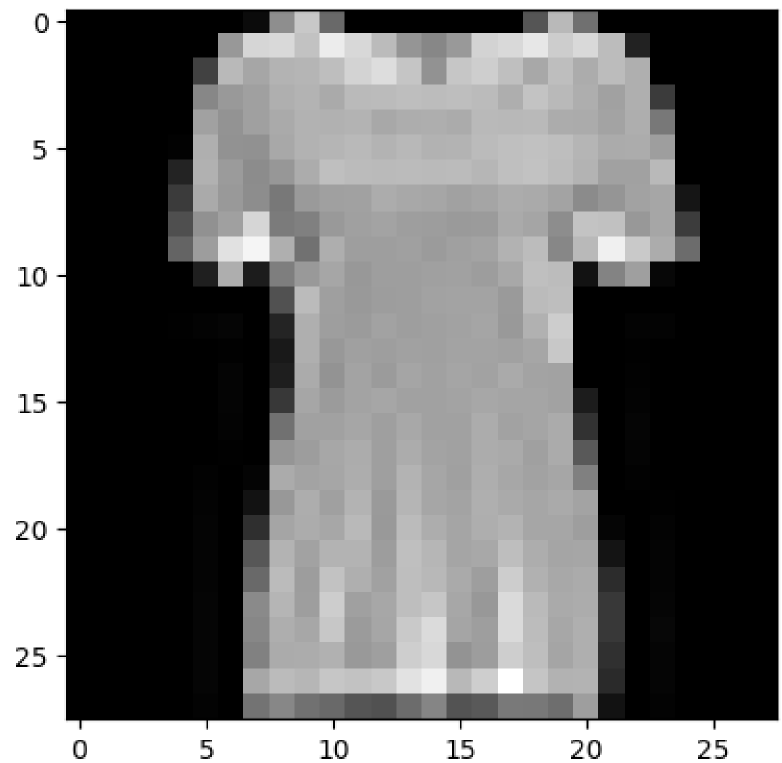
```
[ ]: # Print multiple images at once
figure = plt.figure(figsize=(10, 8))
cols, rows = 5, 5
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(fmnist_dataset), size=(1,)).item()
    img, label = fmnist_dataset[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(label)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```



[29]:
```
# The image is now convert to a 28 X 28 tensor.
# The first dimension is used to keep track of the color channels.
# Since images in the MNIST dataset are grayscale, there's just one channel.
# The values range from 0 to 1, with 0 representing black, 1 white and the␣
 ↪values between different shades of grey.
print(sampleTensor[:,10:15,10:15])
print(torch.max(sampleTensor), torch.min(sampleTensor))
plt.imshow(sampleTensor[0,:,:],cmap = 'gray')
```

```
tensor([[[0.6510, 0.5961, 0.6196, 0.6196, 0.6275],
         [0.6235, 0.6000, 0.6157, 0.6196, 0.6353],
         [0.6196, 0.6078, 0.6353, 0.6196, 0.6275],
         [0.5961, 0.6275, 0.6196, 0.6314, 0.6275],
         [0.5765, 0.6431, 0.6078, 0.6471, 0.6314]]])
tensor(1.) tensor(0.)
```

[29]: <matplotlib.image.AxesImage at 0x7f2eb58337a0>



0.1 Training and validation data

[ ]:
```
train_data, validation_data = random_split(fmnist_dataset, [50000, 10000])
## Print the length of train and validation datasets
print("length of Train Datasets: ", len(train_data))
print("length of Validation Datasets: ", len(validation_data))

batch_size = 128
train_loader = DataLoader(train_data, batch_size, shuffle = True)
val_loader = DataLoader(validation_data, batch_size, shuffle = False)
## MNIST data from pytorch already provides held-out test set!
```

```
length of Train Datasets:  50000
length of Validation Datasets:  10000
```

0.2 Multi-class Logistic Regression (a building block of DNN)

[31]:
```
## Basic set up for a logistic regression model (won't be used in practice or
 ↪for training)
input_size = 28 * 28
num_classes = 10
```

```
# we gradually build on this inherited class from pytorch
model = nn.Linear(input_size, num_classes)
```

We define the class with multiple methods so that we can train, evaluate, and do many other routine tasks with the model.

Particularly, we are looking at multi-class logistic regression (a generalization of one-class logistic regression) using the softmax function (more about this in a few cells down)

[32]:
```
# Slowly build the model, first with basic
class MnistModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, xb):
        # view xb with two dimensions, 28 * 28(i.e 784)
        # One argument to .reshape can be set to -1(in this case the first␣
 ↪dimension),
        # to let PyTorch figure it out automatically based on the shape of the␣
 ↪original tensor.
        xb = xb.reshape(-1, 784)
        print(xb)
        out = self.linear(xb)
        print(out)
        return(out)

model = MnistModel()
print(model.linear.weight.shape, model.linear.bias.shape)
list(model.parameters())
```

```
torch.Size([10, 784]) torch.Size([10])
```

[32]:
```
[Parameter containing:
 tensor([[ 0.0302, -0.0244, -0.0221,  …, -0.0354, -0.0011, -0.0191],
         [ 0.0059, -0.0298, -0.0335,  …, -0.0161, -0.0181,  0.0215],
         [ 0.0003,  0.0097,  0.0043,  …,  0.0126,  0.0231, -0.0112],
         …,
         [ 0.0083,  0.0251,  0.0005,  …, -0.0079,  0.0301, -0.0188],
         [ 0.0186,  0.0183, -0.0347,  …, -0.0200,  0.0312,  0.0024],
         [ 0.0249,  0.0170, -0.0062,  …,  0.0070, -0.0275, -0.0029]],
        requires_grad=True),
 Parameter containing:
 tensor([-0.0216, -0.0106,  0.0346,  0.0050, -0.0052,  0.0162,  0.0206,  0.0106,
          0.0218, -0.0220], requires_grad=True)]
```

[33]:
```
# Alway check the dimensions and sample data/image
for images, labels in train_loader:
```

```
    outputs = model(images)
    break

print('Outputs shape: ', outputs.shape) # torch.Size([128, 10])
print('Sample outputs: \n', outputs[:2].data) # example outputs
```

```
tensor([[0.0000, 0.0000, 0.0000,  …, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  …, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  …, 0.0000, 0.0000, 0.0000],
        …,
        [0.0000, 0.0000, 0.0000,  …, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  …, 0.0039, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  …, 0.0000, 0.0000, 0.0000]])
tensor([[-0.2234,  0.2386, -0.0340,  …,  0.2979,  0.0118, -0.0658],
        [ 0.3455, -0.1725,  0.2791,  …,  0.3818, -0.0107, -0.2146],
        [ 0.4592,  0.1558,  0.0924,  …,  0.4541,  0.1828,  0.0899],
        …,
        [-0.1451,  0.3672, -0.1604,  …,  0.4152,  0.1399,  0.0320],
        [ 0.5077,  0.1205,  0.1281,  …,  0.2762, -0.1545, -0.0600],
        [ 0.2511,  0.1209,  0.2638,  …,  0.5946,  0.0303, -0.0083]],
       grad_fn=<AddmmBackward0>)
Outputs shape:  torch.Size([128, 10])
Sample outputs:
 tensor([[-0.2234,  0.2386, -0.0340, -0.1299,  0.1609,  0.2886,  0.0154,
0.2979,
          0.0118, -0.0658],
        [ 0.3455, -0.1725,  0.2791, -0.0059, -0.0450,  0.2795,  0.2659,  0.3818,
         -0.0107, -0.2146]])
```

0.3 Softmax function

[34]:
```
## Apply softmax for each output row
probs = F.softmax(outputs, dim = 1)

## chaecking at sample probabilities
print("Sample probabilities:\n", probs[:2].data)

# print(preds)
# print("\n")
# print(max_probs)
```

```
Sample probabilities:
 tensor([[0.0745, 0.1183, 0.0900, 0.0818, 0.1094, 0.1243, 0.0946, 0.1255,
0.0943,
         0.0872],
        [0.1238, 0.0737, 0.1158, 0.0871, 0.0838, 0.1159, 0.1143, 0.1283, 0.0867,
         0.0707]])
```

0.4 Evaluation Metric and Loss Function

[35]:
```
# accuracy calculation
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim = 1)
    return(torch.tensor(torch.sum(preds == labels).item()/ len(preds)))

print("Accuracy: ", accuracy(outputs, labels))
print("\n")
loss_fn = F.cross_entropy
print("Loss Function: ",loss_fn)
print("\n")
## Loss for the current batch
loss = loss_fn(outputs, labels)
print(loss)
```

```
Accuracy:  tensor(0.1641)


Loss Function:  <function cross_entropy at 0x7f2f26975d00>


tensor(2.3216, grad_fn=<NllLossBackward0>)
```

0.5 Cross-Entropy

[36]:
```
# We put all of the above:
class MnistModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, xb):
        xb = xb.reshape(-1, 784)
        out = self.linear(xb)
        return(out)

    # We add extra methods
    def training_step(self, batch):
        # when training, we compute the cross entropy, which help us update␣
↪weights
        images, labels = batch
        out = self(images) ## Generate predictions
        loss = F.cross_entropy(out, labels) ## Calculate the loss
        return(loss)

    def validation_step(self, batch):
        images, labels = batch
```

```
        out = self(images) ## Generate predictions
        loss = F.cross_entropy(out, labels) ## Calculate the loss
        # in validation, we want to also look at the accuracy
        # idealy, we would like to save the model when the accuracy is the␣
↪highest.
        acc = accuracy(out, labels) ## calculate metrics/accuracy
        return({'val_loss':loss, 'val_acc': acc})

    def validation_epoch_end(self, outputs):
        # at the end of epoch (after running through all the batches)
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()
        return({'val_loss': epoch_loss.item(), 'val_acc' : epoch_acc.item()})

    def epoch_end(self, epoch,result):
        # log epoch, loss, metrics
        print("Epoch [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch,␣
↪result['val_loss'], result['val_acc']))

# we instantiate the model
model = MnistModel()

# a simple helper function to evaluate
def evaluate(model, data_loader):
    # for batch in data_loader, run validation_step
    outputs = [model.validation_step(batch) for batch in data_loader]
    return(model.validation_epoch_end(outputs))

# actually training
def fit(epochs, lr, model, train_loader, val_loader, opt_func = torch.optim.
↪SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        ## Training Phase
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward() ## backpropagation starts at the loss and goes␣
↪through all layers to model inputs
            optimizer.step() ## the optimizer iterate over all parameters␣
↪(tensors); use their stored grad to update their values
            optimizer.zero_grad() ## reset gradients

        ## Validation phase
        result = evaluate(model, val_loader)
```

```
        model.epoch_end(epoch, result)
        history.append(result)
    return(history)
```

[37]:
```
# test the functions, with a randomly initialized model (weights are random, e.
 ↪g., untrained)
result0 = evaluate(model, val_loader)
result0
```

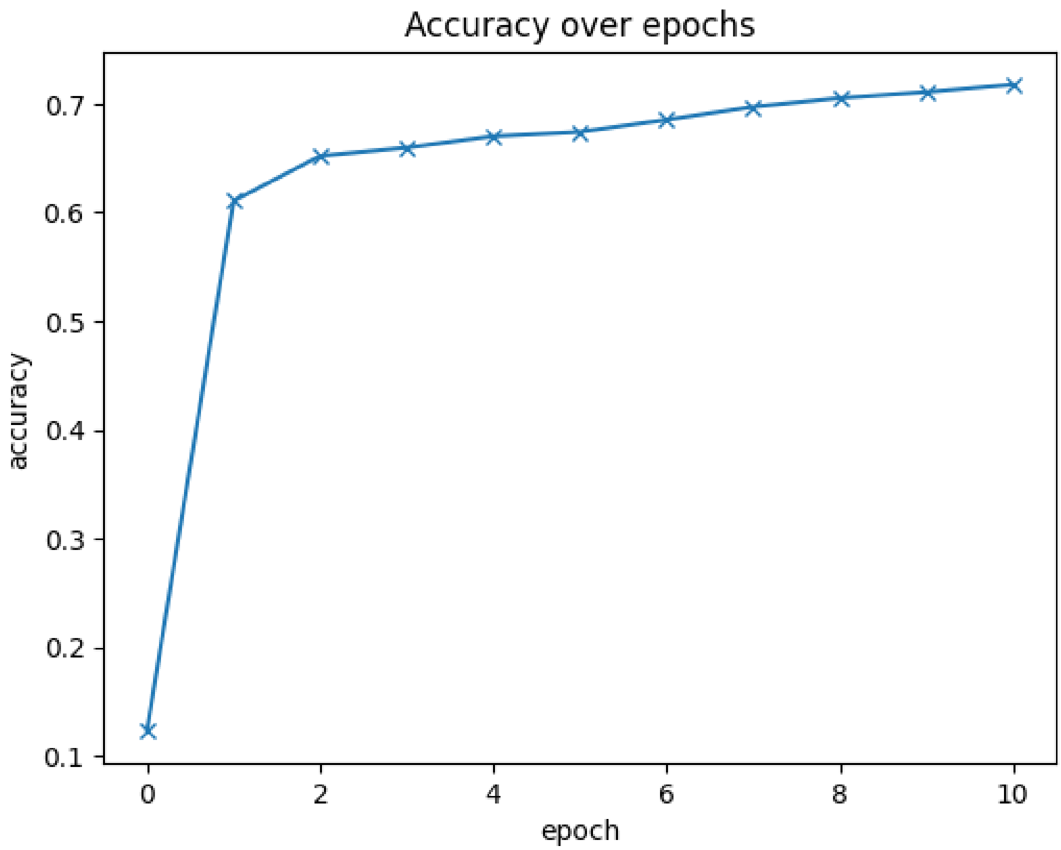
[37]:
```
{'val_loss': 2.325157642364502, 'val_acc': 0.12321993708610535}
```

[38]:
```
# let's train for 10 epochs
history1 = fit(10, 0.001, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 1.7219, val_acc: 0.6107
Epoch [1], val_loss: 1.4309, val_acc: 0.6520
Epoch [2], val_loss: 1.2635, val_acc: 0.6597
Epoch [3], val_loss: 1.1558, val_acc: 0.6701
Epoch [4], val_loss: 1.0801, val_acc: 0.6741
Epoch [5], val_loss: 1.0237, val_acc: 0.6853
Epoch [6], val_loss: 0.9798, val_acc: 0.6972
Epoch [7], val_loss: 0.9444, val_acc: 0.7053
Epoch [8], val_loss: 0.9155, val_acc: 0.7108
Epoch [9], val_loss: 0.8904, val_acc: 0.7178
```

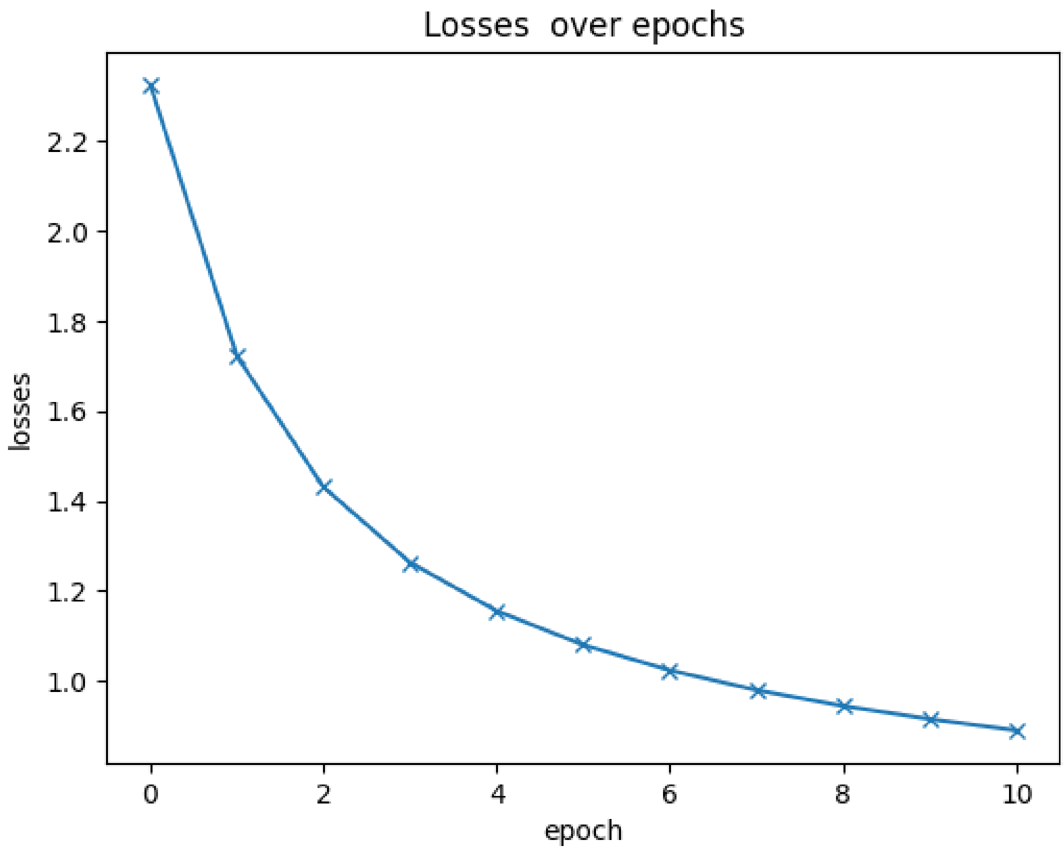
[39]:
```
# we combine the first result (no training) and the training results of 5␣
 ↪epoches
# plotting accuracy
history = [result0] + history1
accuracies = [result['val_acc'] for result in history]
plt.plot(accuracies, '-x')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Accuracy over epochs')
```

[39]:
```
Text(0.5, 1.0, 'Accuracy over epochs')
```




```
[40]: # plotting losses
      history = [result0] + history1
      losses = [result['val_loss'] for result in history]
      plt.plot(losses, '-x')
      plt.xlabel('epoch')
      plt.ylabel('losses')
      plt.title('Losses  over epochs')
```

```
[40]: Text(0.5, 1.0, 'Losses  over epochs')
```

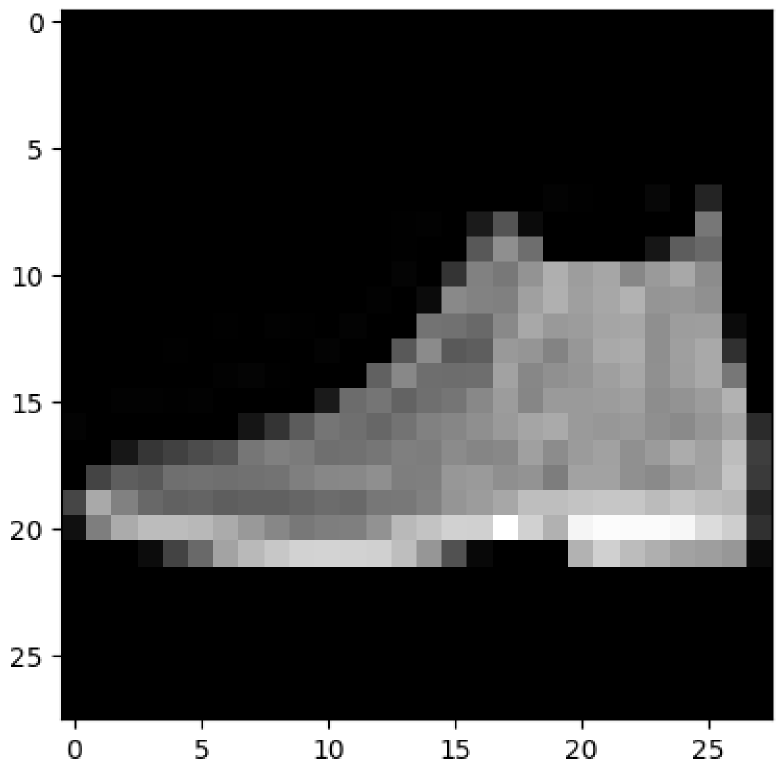


0.6 Final check using the (held-out) test dataset.

We will first load the test dataset (from MNIST) and individually check the prediction made by the model. And then, we will put through all images in the test dataset to obtain the final accuracy

[41]:
```
# Testing with individual images
## Define the test dataset
test_dataset = FashionMNIST(root = 'data/', train = False, transform =␣
↪transforms.ToTensor())
print("Length of Test Datasets: ", len(test_dataset))
img, label = test_dataset[0]
plt.imshow(img[0], cmap = 'gray')
print("Shape: ", img.shape)
print('Label: ', label)
```

```
Length of Test Datasets:  10000
Shape:  torch.Size([1, 28, 28])
Label:  9
```



[42]:
```
def predict_image(img, model):
    xb = img.unsqueeze(0)
    yb = model(xb)
    _, preds = torch.max(yb, dim = 1)
    return(preds[0].item())
```

[43]:
```
img, label = test_dataset[0]
print('Label:', label, ', Predicted :', predict_image(img, model))
```

```
Label: 9 , Predicted : 9
```

[44]:
```
# the final check on the test dataset (not used in any training)
test_loader = DataLoader(test_dataset, batch_size = 256, shuffle = False)
result = evaluate(model, test_loader)
result
```

[44]:
```
{'val_loss': 0.8996326327323914, 'val_acc': 0.708691418170929}
```

1 Convolutional Neural Network (CNN)

[45]:
```python
# We construct a fundamental CNN class.
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        # fully connected layer, output 10 classes
        self.out = nn.Linear(32 * 7 * 7, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output, x    # return x for visualization

cnn = CNN()
print(cnn)
```

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
```

```
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
```

[46]:
```
loss_func = nn.CrossEntropyLoss()
loss_func

# unlike earlier example using optim.SGD, we use optim.Adam as the optimizer
# lr(Learning Rate): Rate at which our model updates the weights in the cells␣
 ↪each time back-propagation is done.
optimizer = optim.Adam(cnn.parameters(), lr = 0.01)
optimizer
```

[46]:
```
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    decoupled_weight_decay: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.01
    maximize: False
    weight_decay: 0
)
```

[ ]:
```
# train_data, validation_data = random_split(fmnist_dataset, [50000, 10000])
# ## Print the length of train and validation datasets
# print("length of Train Datasets: ", len(train_data))
# print("length of Validation Datasets: ", len(validation_data))

# batch_size = 128
# train_loader = DataLoader(train_data, batch_size, shuffle = True)
# val_loader = DataLoader(validation_data, batch_size, shuffle = False)
from torch.autograd import Variable

def train(num_epochs, cnn, loaders):
    cnn.train()
    optimizer = optim.Adam(cnn.parameters(), lr = 0.01)
    loss_func = nn.CrossEntropyLoss()
    # Train the model
    total_step = len(loaders)

    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(loaders):
```

```
            # gives batch data, normalize x when iterate train_loader
            b_x = Variable(images)   # batch x
            b_y = Variable(labels)   # batch y
            output = cnn(b_x)[0]
            loss = loss_func(output, b_y)

            # clear gradients for this training step
            optimizer.zero_grad()

            # backpropagation, compute gradients
            loss.backward()
            # apply gradients
            optimizer.step()

            if (i+1) % 100 == 0:
                print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch␣
↪+ 1, num_epochs, i + 1, total_step, loss.item()))
                pass
        pass
    pass
```

[48]:
```
# instiate the CNN model
cnn = CNN()
# for testing purpose, we calculate the accuracy of the initial
cnn.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in train_loader:
        test_output, last_layer = cnn(images)
        pred_y = torch.max(test_output, 1)[1].data.squeeze()
        accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
print('Accuracy of the model on the 10000 test images: %.2f' % accuracy)
```

```
Accuracy of the model on the 10000 test images: 0.19
```

[49]:
```
train(num_epochs=5, cnn=cnn, loaders=train_loader)
```

```
Epoch [1/5], Step [100/391], Loss: 0.5404
Epoch [1/5], Step [200/391], Loss: 0.2914
Epoch [1/5], Step [300/391], Loss: 0.2988
Epoch [2/5], Step [100/391], Loss: 0.3219
Epoch [2/5], Step [200/391], Loss: 0.2439
Epoch [2/5], Step [300/391], Loss: 0.2370
Epoch [3/5], Step [100/391], Loss: 0.3342
Epoch [3/5], Step [200/391], Loss: 0.2703
Epoch [3/5], Step [300/391], Loss: 0.2695
```

```
Epoch [4/5], Step [100/391], Loss: 0.2814
Epoch [4/5], Step [200/391], Loss: 0.2218
Epoch [4/5], Step [300/391], Loss: 0.2696
Epoch [5/5], Step [100/391], Loss: 0.2278
Epoch [5/5], Step [200/391], Loss: 0.3272
Epoch [5/5], Step [300/391], Loss: 0.3563
```

2 Evaluate the model on test data

[50]:
```python
# Test the model, after the training
cnn.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in train_loader:
        test_output, last_layer = cnn(images)
        pred_y = torch.max(test_output, 1)[1].data.squeeze()
        accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
print('Test Accuracy of the model on the 10000 test images: %.2f' % accuracy)
```

```
Test Accuracy of the model on the 10000 test images: 0.89
```

[51]:
```python
# Test the model, after the training
cnn.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        test_output, last_layer = cnn(images)
        pred_y = torch.max(test_output, 1)[1].data.squeeze()
        accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
print('Test Accuracy of the model on the 10000 test images: %.2f' % accuracy)
```

```
Test Accuracy of the model on the 10000 test images: 0.94
```

Run inference on individual images

[52]:
```python
sample = next(iter(test_loader))
imgs, lbls = sample

actual_number = lbls[:10].numpy()
actual_number

test_output, last_layer = cnn(imgs[:10])
pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
print(f'Prediction number: {pred_y}')
```

```python
print(f'Actual number: {actual_number}')
```

```
Prediction number: [9 2 1 1 0 1 4 6 5 7]
Actual number: [9 2 1 1 6 1 4 6 5 7]
```