Pytorch Learning Notes

Yu Zheng
Department of Statistics, University of Florida

Based on Deep Learning with PyTorch: Zero to GANs.

The list of topics is given below:

1. Pytorch Basics
2. Gradient Descent and Linear Regression
3. Working with Images & Logistic Regression in PyTorch

Part 1: Pytorch Basics

Before we begin, we need to install the required libraries and import *torch*. The installation of PyTorch may differ based on your operating system / cloud environment. You can find detailed installation instructions here: https://pytorch.org .

```
# Uncomment and run the appropriate command for your operating system, if required

# Linux / Binder
# !pip install numpy torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==0.7.0 -f
https://download.pytorch.org/whl/torch_stable.html

# Windows
# !pip install numpy torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==0.7.0 -f
https://download.pytorch.org/whl/torch_stable.html

# MacOS
# !pip install numpy torch torchvision torchaudio
```

```
import torch
```

1.1 Tensor

At its core, PyTorch is a library for processing tensors. A tensor is a number, vector, matrix, or any n-dimensional array. We can create tensors as following:

```
# Number
t1 = torch.tensor(4.)
# Vector
t2 = torch.tensor([1., 2, 3, 4])
# Matrix
t3 = torch.tensor([[5., 6],
                   [7, 8],
                   [9, 10]])
# 3-dimensional array
t4 = torch.tensor([
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.]]])
```

Note that it's not possible to create tensors with an improper shape.

```
t5 = torch.tensor([[5., 6, 11],
                   [7, 8],
                   [9, 10]])
```

A `ValueError` is thrown because the lengths of the rows `[5., 6, 11]` and `[7, 8]` don't match.

Also, the data type for elements in a tensor should be the same. If not, it will be forced to be the same. We can check the `dtype` attribute of the tensor to know its data type.

```
t1.dtype
```

torch.float32

Tensors can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the `.shape` property of a tensor.

```
t1.shape
t2.shape
t3.shape
t4.shape
```

```
torch.Size([])
torch.Size([4])
torch.Size([3, 2])
torch.Size([2, 2, 3])
```

The way to determine the order of the lengths along different dimensions of a tensor is count the brackets from the outmost to inside.

1.2 Tensor operations and gradients

```
# Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
x, w, b
```

(tensor(3.), tensor(4., requires_grad=True), tensor(5., requires_grad=True))

```
# Arithmetic operations
y = w * x + b
y
```

tensor(17., grad_fn=)

What makes PyTorch unique is that we can automatically compute the derivative of `y` w.r.t. the tensors that have `requires_grad` set to `True` i.e. w and b. This feature of PyTorch is called *autograd* (automatic gradients).

To compute the derivatives, we can invoke the `.backward` method on our result `y`.

```
# Compute derivatives
y.backward()
```

The derivatives of `y` with respect to the input tensors are stored in the `.grad` property of the respective tensors.

```
# Display gradients
print('dy/dx:', x.grad)
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

dy/dx: None
dy/dw: tensor(3.)
dy/db: tensor(1.)

1.3 Tensor functions

Apart from arithmetic operations, the `torch` module also contains many functions for creating and manipulating tensors. Below are some examples. More tensor operations can be found here.

```
# Create a tensor with a fixed value for every element
t6 = torch.full((3, 2), 42)
t6
```

tensor([[42, 42],
[42, 42],
[42, 42]])

```
# Concatenate two tensors with compatible shapes
t7 = torch.cat((t3, t6))
t7
```

tensor([[5., 6.],
[7., 8.],
[9., 10.],
[42., 42.],
[42., 42.],
[42., 42.]])

```
# Compute the sin of each element
t8 = torch.sin(t7)
t8
```

tensor([[-0.9589, -0.2794],
[0.6570, 0.9894],
[0.4121, -0.5440],
[-0.9165, -0.9165],
[-0.9165, -0.9165],
[-0.9165, -0.9165]])

```
# Change the shape of a tensor
t9 = t8.reshape(3, 2, 2)
t9
```

tensor([[[-0.9589, -0.2794],
[0.6570, 0.9894]],
[[0.4121, -0.5440],
[-0.9165, -0.9165]],
[[-0.9165, -0.9165],
[-0.9165, -0.9165]]])

1.4 Interoperability with Numpy

Numpy is a popular open-source library used for mathematical and scientific computing in Python. It enables efficient operations on large multi-dimensional arrays and has a vast ecosystem of supporting libraries, including:

- Pandas for file I/O and data analysis
- Matplotlib for plotting and visualization
- OpenCV for image and video processing

If you're interested in learning more about Numpy and other data science libraries in Python, check out this tutorial series: https://jovian.ai/aakashns/python-numerical-computing-with-numpy .

Instead of reinventing the wheel, PyTorch interoperates well with Numpy to leverage its existing ecosystem of tools and libraries.

```
import numpy as np

# Create an array in Numpy
x = np.array([[1, 2], [3, 4.]])
x

# Convert the numpy array to a torch tensor
y = torch.from_numpy(x)
y
```

array([[1., 2.],
[3., 4.]])

tensor([[1., 2.],
[3., 4.]], dtype=torch.float64)

```
x.dtype, y.dtype
```

(dtype('float64'), torch.float64)

```
# Convert a torch tensor to a numpy array
z = y.numpy()
z
```

array([[1., 2.],
[3., 4.]])

The interoperability between PyTorch and Numpy is essential because most datasets you'll work with will likely be read and preprocessed as Numpy arrays.

Why we need a library like PyTorch at all since Numpy already provides data structures and utilities for working with multi-dimensional numeric data?There are two main reasons:

1. **Autograd**: The ability to automatically compute gradients for tensor operations is essential for training deep learning models.
2. **GPU support**: While working with massive datasets and large models, PyTorch tensor operations can be performed efficiently using a Graphics Processing Unit (GPU). Computations that might typically take hours can be completed within minutes using GPUs.

1.5 Questions for review

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is PyTorch?
2. What is a Jupyter notebook?
3. What is Google Colab?
4. How do you install PyTorch?
5. How do you import the `torch` module?
6. What is a vector? Give an example.
7. What is a matrix? Give an example.
8. What is a tensor?
9. How do you create a PyTorch tensor? Illustrate with examples.
10. What is the difference between a tensor and a vector or a matrix?
11. Is every tensor a matrix?
12. Is every matrix a tensor?
13. What does the `dtype` property of a tensor represent?
14. Is it possible to create a tensor with elements of different data types?
15. How do you inspect the number of dimensions of a tensor and the length along each dimension?
16. Is it possible to create a tensor with the values `[[1, 2, 3], [4, 5]]`? Why or why not?
17. How do you perform arithmetic operations on tensors? Illustrate with examples?
18. What happens if you specify `requires_grad=True` while creating a tensor? Illustrate with an example.
19. What is autograd in PyTorch? How is it useful?
20. What happens when you invoke the `backward` method of a tensor?
21. How do you check the derivates of a result tensor w.r.t. the tensors used to compute its value?
22. Give some examples of functions available in the `torch` module for creating tensors.
23. Give some examples of functions available in the `torch` module for performing mathematical operations on tensors.
24. Where can you find the list of tensor operations available in PyTorch?
25. What is Numpy?
26. How do you create a Numpy array?
27. How do you create a PyTorch tensor using a Numpy array?
28. How do you create a Numpy array using a PyTorch tensor?

29. Why is interoperability between PyTorch and Numpy important?
30. What is the purpose of a library like PyTorch if Numpy already provides data structures and utilities to with multi-dimensional numeric data?

Part 2: Gradient Descent and Linear Regression

2.1 Introduction to linear regression

We'll discuss one of the foundational algorithms in machine learning: *Linear regression*. We'll create a model that predicts crop yields for apples and oranges (*target variables*) by looking at the average temperature, rainfall, and humidity (*input variables or features*) in a region. Here's the training data:

Region	Temp. (F)	Rainfall (mm)	Humidity (%)	Apples (ton)	Oranges (ton)
Kanto	73	67	43	56	70
Johto	91	88	64	81	101
Hoenn	87	134	58	119	133
Sinnoh	102	43	37	22	37
Unova	69	96	70	103	119

In a linear regression model, each target variable is estimated to be a weighted sum of the input variables, offset by some constant, known as a bias :

```
yield_apple  = w11 * temp + w12 * rainfall + w13 * humidity + b1
yield_orange = w21 * temp + w22 * rainfall + w23 * humidity + b2
```

2.2 Training data

We can represent the training data using two matrices: `inputs` and `targets`, each with one row per observation, and one column per variable.

```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70]], dtype='float32')
```

```
# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119]], dtype='float32')

# Convert inputs and targets to tensors
inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)
print(inputs)
print(targets)
```

tensor([[73., 67., 43.],
[91., 88., 64.],
[87., 134., 58.],
[102., 43., 37.],
[69., 96., 70.]])
tensor([[56., 70.],
[81., 101.],
[119., 133.],
[22., 37.],
[103., 119.]])

2.3 Linear regression model from scratch

```
# Weights and biases
w = torch.randn(2, 3, requires_grad=True)
b = torch.randn(2, requires_grad=True)
print(w)
print(b)
```

tensor([[-0.2910, -0.3450, 0.0305],
[-0.6528, 0.7386, -0.5153]], requires_grad=True)
tensor([-0.9161, -0.7779], requires_grad=True)

Our *model* is simply a function that performs a matrix multiplication of the `inputs` and the weights `w` (transposed) and adds the bias `b` (replicated for each observation).

$$X \quad \times \quad W^T \quad + \quad b$$

$$\begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \\ b_1 & b_2 \\ \vdots & \vdots \\ b_1 & b_2 \end{bmatrix}$$

We can define the model as follows:

```
def model(x):
    return x @ w.t() + b
```

`@` represents matrix multiplication in PyTorch, and the `.t` method returns the transpose of a tensor.

The matrix obtained by passing the input data into the model is a set of predictions for the target variables.

```
# Generate predictions
preds = model(inputs)
print(preds)
```

tensor([[-43.9569, -21.1025],
[-55.7975, -28.1628],
[-70.6863, 11.5154],
[-44.2982, -54.6685],
[-51.9732, -10.9839]], grad_fn=)

```
# MSE loss
def mse(t1, t2):
    diff = t1 - t2
    return torch.sum(diff * diff) / diff.numel()
```

`torch.sum` returns the sum of all the elements in a tensor. The `.numel` method of a tensor returns the number of elements in a tensor. Let's compute the mean squared error for the current predictions of our model.

```
# Compute loss
loss = mse(preds, targets)
print(loss)
```

tensor(15813.8125, grad_fn=)

```
# Compute gradients
loss.backward()

# Gradients for weights
print(w)
print(w.grad)
```

tensor([[-0.2910, -0.3450, 0.0305],
[-0.6528, 0.7386, -0.5153]], requires_grad=True)
tensor([[-10740.7393, -12376.3008, -7471.2300],
[-9458.5078, -10033.8672, -6344.1094]])

```
with torch.no_grad():
    w -= w.grad * 1e-5
    b -= b.grad * 1e-5
```

We use `torch.no_grad` to indicate to PyTorch that we shouldn't track, calculate, or modify gradients while updating the weights and biases.

```
# Let's verify that the loss is actually lower
loss = mse(preds, targets)
print(loss)
```

tensor(15813.8125, grad_fn=)

Before we proceed, we reset the gradients to zero by invoking the `.zero_()` method. We need to do this because PyTorch accumulates gradients. Otherwise, the next time we invoke `.backward` on the loss, the new gradient values are added to the existing gradients, which may lead to unexpected results.

```
w.grad.zero_()
b.grad.zero_()
print(w.grad)
print(b.grad)
```

tensor([[0., 0., 0.],
[0., 0., 0.]])
tensor([0., 0.])

As seen above, we reduce the loss and improve our model using the gradient descent optimization algorithm. Thus, we can *train* the model using the following steps:

1. Generate predictions
2. Calculate the loss
3. Compute gradients w.r.t the weights and biases
4. Adjust the weights by subtracting a small quantity proportional to the gradient
5. Reset the gradients to zero

```
# Train for 100 epochs
for i in range(100):
    preds = model(inputs)
    loss = mse(preds, targets)
    loss.backward()
    with torch.no_grad():
        w -= w.grad * 1e-5
        b -= b.grad * 1e-5
        w.grad.zero_()
        b.grad.zero_()
```

2.4 Linear regression using PyTorch built-ins

We've implemented linear regression & gradient descent model using some basic tensor operations. However, since this is a common pattern in deep learning, PyTorch provides several built-in functions and classes to make it easy to create and train models with just a few lines of code.

2.4.1 Dataset

Let's begin by importing the `torch.nn` package from PyTorch, which contains utility classes for building neural networks.

```
import torch.nn as nn
```

```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70],
                   [74, 66, 43],
                   [91, 87, 65],
                   [88, 134, 59],
                   [101, 44, 37],
                   [68, 96, 71],
                   [73, 66, 44],
                   [92, 87, 64],
                   [87, 135, 57],
                   [103, 43, 36],
                   [68, 97, 70]],
                  dtype='float32')

# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119],
```

```
                   [57, 69],
                   [80, 102],
                   [118, 132],
                   [21, 38],
                   [104, 118],
                   [57, 69],
                   [82, 100],
                   [118, 134],
                   [20, 38],
                   [102, 120]],
                  dtype='float32')

inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)
```

We are using 15 training examples to illustrate how to work with large datasets in small batches.

2.4.2 TensorDataset and DataLoader

We'll create a `TensorDataset`, which allows access to rows from `inputs` and `targets` as tuples, and provides standard APIs for working with many different types of datasets in PyTorch.

```
from torch.utils.data import TensorDataset
```

```
# Define dataset
train_ds = TensorDataset(inputs, targets)
train_ds[0:3]
```

(tensor([[73., 67., 43.],
[91., 88., 64.],
[87., 134., 58.]]),
tensor([[56., 70.],
[81., 101.],
[119., 133.]]))

The `TensorDataset` allows us to access a small section of the training data using the array indexing notation (`[0:3]` in the above code). It returns a tuple with two elements. The first element contains the input variables for the selected rows, and the second contains the targets.

We'll also create a `DataLoader`, which can split the data into batches of a predefined size while training. It also provides other utilities like shuffling and random sampling of the data.

```
from torch.utils.data import DataLoader
```

```
# Define data loader
batch_size = 5
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

We can use the data loader in a `for` loop.

```
for xb, yb in train_dl:
    print(xb)
    print(yb)
    break
```

tensor([[102., 43., 37.],
[92., 87., 64.],
[87., 134., 58.],
[69., 96., 70.],
[101., 44., 37.]])
tensor([[22., 37.],
[82., 100.],
[119., 133.],
[103., 119.],
[21., 38.]])

In each iteration, the data loader returns one batch of data with the given batch size. If `shuffle` is set to `True`, it shuffles the training data before creating batches. Shuffling helps randomize the input to the optimization algorithm, leading to a faster reduction in the loss.

2.4.3 nn.Linear

Instead of initializing the weights & biases manually, we can define the model using the `nn.Linear` class from PyTorch, which does it automatically.

```
# Define model
model = nn.Linear(3, 2)
print(model.weight)
print(model.bias)
```

Parameter containing:
tensor([[0.1304, -0.1898, 0.2187],
*   [0.2360, 0.4139, -0.4540]], requires_grad=True)*
Parameter containing:
tensor([0.3457, 0.3883], requires_grad=True)

PyTorch models also have a helpful `.parameters` method, which returns a list containing all the weights and bias matrices present in the model. For our linear regression model, we have one weight matrix and one bias matrix.

```
# Parameters
list(model.parameters())
```

[Parameter containing:
tensor([[0.1304, -0.1898, 0.2187],
*   [0.2360, 0.4139, -0.4540]], requires_grad=True),*
Parameter containing:
tensor([0.3457, 0.3883], requires_grad=True)]

We can use the model to generate predictions in the same way as before.

```
# Generate predictions
preds = model(inputs)
```

2.4.4 Loss function

The `nn.functional` package contains many useful loss functions and several other utilities.

```
# Import nn.functional
import torch.nn.functional as F
```

```
# Define loss function
loss_fn = F.mse_loss

loss = loss_fn(model(inputs), targets)
print(loss)
```

tensor(5427.9517, grad_fn=)

2.4.5 Optimizer

Instead of manually manipulating the model's weights & biases using gradients, we can use the optimizer `optim.SGD`. SGD is short for "stochastic gradient descent".

```
# Define optimizer
opt = torch.optim.SGD(model.parameters(), lr=1e-5)
```

Note that `model.parameters()` is passed as an argument to `optim.SGD` so that the optimizer knows which matrices should be modified during the update step. Also, we can specify a learning rate that controls the amount by which the parameters are modified.

2.4.6 Train the model

We are now ready to train the model. We'll follow the same process to implement gradient descent:

1. Generate predictions
2. Calculate the loss
3. Compute gradients w.r.t the weights and biases
4. Adjust the weights by subtracting a small quantity proportional to the gradient
5. Reset the gradients to zero

The only change is that we'll work batches of data instead of processing the entire training data in every iteration. Let's define a utility function `fit` that trains the model for a given number of epochs.

```
# Utility function to train the model
def fit(num_epochs, model, loss_fn, opt, train_dl):

    # Repeat for given number of epochs
```

```
    for epoch in range(num_epochs):

        # Train with batches of data
        for xb,yb in train_dl:

            # 1. Generate predictions
            pred = model(xb)

            # 2. Calculate loss
            loss = loss_fn(pred, yb)

            # 3. Compute gradients
            loss.backward()

            # 4. Update parameters using gradients
            opt.step()

            # 5. Reset the gradients to zero
            opt.zero_grad()

        # Print the progress
        if (epoch+1) % 10 == 0:
            print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
```

Some things to note above:

- We use the data loader defined earlier to get batches of data for every iteration.
- Instead of updating parameters (weights and biases) manually, we use `opt.step` to perform the update and `opt.zero_grad` to reset the gradients to zero.
- We've also added a log statement that prints the loss from the last batch of data for every 10th epoch to track training progress. `loss.item` returns the actual value stored in the loss tensor.

Let's train the model for 100 epochs.

```
fit(100, model, loss_fn, opt, train_dl)
```

Epoch [10/100], Loss: 818.6476
Epoch [20/100], Loss: 335.3347
Epoch [30/100], Loss: 190.3544
Epoch [40/100], Loss: 131.6701
Epoch [50/100], Loss: 77.0783
Epoch [60/100], Loss: 151.5671
Epoch [70/100], Loss: 151.0817
Epoch [80/100], Loss: 67.6262
Epoch [90/100], Loss: 53.6205
Epoch [100/100], Loss: 33.4517

Now we can use the model to make predictions of crop yields for new regions by passing a batch containing a single row of input.

```
model(torch.tensor([[75, 63, 44.]]))
```

tensor([[55.3323, 67.8895]], grad_fn=)

The predicted yield of apples is 54.3 tons per hectare, and that of oranges is 68.3 tons per hectare.

2.5 Questions for review

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is a linear regression model? Give an example of a problem formulated as a linear regression model.
2. What are input and target variables in a dataset? Give an example.
3. What are weights and biases in a linear regression model?
4. How do you represent tabular data using PyTorch tensors?
5. Why do we create separate matrices for inputs and targets while training a linear regression model?
6. How do you determine the shape of the weights matrix & bias vector given some training data?
7. How do you create randomly initialized weights & biases with a given shape?
8. How is a linear regression model implemented using matrix operations? Explain with an example.

9. How do you generate predictions using a linear regression model?
10. Why are the predictions of a randomly initialized model different from the actual targets?
11. What is a loss function? What does the term "loss" signify?
12. What is mean squared error?
13. Write a function to calculate mean squared using model predictions and actual targets.
14. What happens when you invoke the `.backward` function on the result of the mean squared error loss function?
15. Why is the derivative of the loss w.r.t. the weights matrix itself a matrix? What do its elements represent?
16. How is the derivate of the loss w.r.t. a weight element useful for reducing the loss? Explain with an example.
17. Suppose the derivative of the loss w.r.t. a weight element is positive. Should you increase or decrease the element's value slightly to get a lower loss?
18. Suppose the derivative of the loss w.r.t. a weight element is negative. Should you increase or decrease the element's value slightly to get a lower loss?
19. How do you update the weights and biases of a model using their respective gradients to reduce the loss slightly?
20. What is the gradient descent optimization algorithm? Why is it called "gradient descent"?
21. Why do you subtract a "small quantity" proportional to the gradient from the weights & biases, not the actual gradient itself?
22. What is learning rate? Why is it important?
23. What is `torch.no_grad`?
24. Why do you reset gradients to zero after updating weights and biases?
25. What are the steps involved in training a linear regression model using gradient descent?
26. What is an epoch?
27. What is the benefit of training a model for multiple epochs?
28. How do you make predictions using a trained model?
29. What should you do if your model's loss doesn't decrease while training? Hint: learning rate.
30. What is `torch.nn`?
31. What is the purpose of the `TensorDataset` class in PyTorch? Give an example.
32. What is a data loader in PyTorch? Give an example.
33. How do you use a data loader to retrieve batches of data?
34. What are the benefits of shuffling the training data before creating batches?
35. What is the benefit of training in small batches instead of training with the entire dataset?
36. What is the purpose of the `nn.Linear` class in PyTorch? Give an example.
37. How do you see the weights and biases of a `nn.Linear` model?

38. What is the purpose of the `torch.nn.functional` module?
39. How do you compute mean squared error loss using a PyTorch built-in function?
40. What is an optimizer in PyTorch?
41. What is `torch.optim.SGD`? What does SGD stand for?
42. What are the inputs to a PyTorch optimizer?
43. Give an example of creating an optimizer for training a linear regression model.
44. Write a function to train a `nn.Linear` model in batches using gradient descent.
45. How do you use a linear regression model to make predictions on previously unseen data?

Part 3: Working with Images & Logistic Regression in PyTorch

3.1 Data (MNIST)

```
# Imports
import torch
import torchvision
from torchvision.datasets import MNIST

# Download training dataset
dataset = MNIST(root='data/', download=True)
```

When this statement is executed for the first time, it downloads the data to the `data/` directory next to the notebook and creates a PyTorch `Dataset`. On subsequent executions, the download is skipped as the data is already downloaded. Let's check the size of the dataset.

```
len(dataset)
```

60000

The dataset has 60,000 images that we'll use to train the model. There is also an additional test set of 10,000 images used for evaluating models and reporting metrics in papers and reports. We can create the test dataset using the `MNIST` class by passing `train=False` to the constructor.

```
test_dataset = MNIST(root='data/', train=False)
len(test_dataset)
```

10000

Let's look at a sample element from the training dataset.

```
dataset[0]
```

(<PIL.Image.Image image mode=L size=28x28 at 0x7F625B9FD710>, 5)

It's a pair, consisting of a 28x28px image and a label. The image is an object of the class `PIL.Image.Image`, which is a part of the Python imaging library Pillow. We can view the image within Jupyter using `matplotlib`, the de-facto plotting and graphing library for data science in Python.

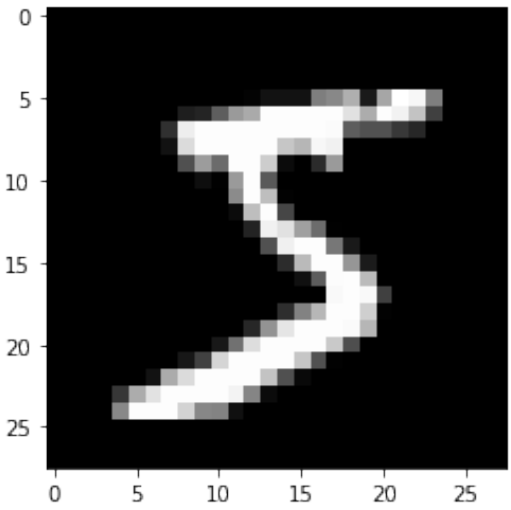
```
import matplotlib.pyplot as plt
%matplotlib inline
```

The statement `%matplotlib inline` indicates to Jupyter that we want to plot the graphs within the notebook. Without this line, Jupyter will show the image in a popup. Statements starting with % are called magic commands and are used to configure the behavior of Jupyter itself. You can find a full list of magic commands here: https://ipython.readthedocs.io/en/stable/interactive/magics.html .

Let's look at a couple of images from the dataset.

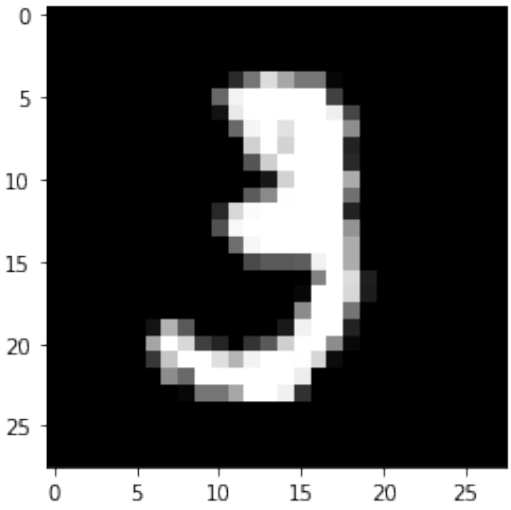
```
image, label = dataset[0]
plt.imshow(image, cmap='gray')
print('Label:', label)
```

Label: 5




```
image, label = dataset[10]
plt.imshow(image, cmap='gray')
print('Label:', label)
```

Label: 3



While it's useful to look at these images, there's just one problem here: PyTorch doesn't know how to work with images. We need to convert the images into tensors. We can do this by specifying a transform while creating our dataset.

```
import torchvision.transforms as transforms
```

PyTorch datasets allow us to specify one or more transformation functions that are applied to the images as they are loaded. The `torchvision.transforms` module contains many such predefined functions. We'll use the `ToTensor` transform to convert images into PyTorch tensors.

```
# MNIST dataset (images and labels)
dataset = MNIST(root='data/',
                train=True,
                transform=transforms.ToTensor())
```

```
img_tensor, label = dataset[0]
print(img_tensor.shape, label)
```

torch.Size([1, 28, 28]) 5

The image is now converted to a 1x28x28 tensor. The first dimension tracks color channels. The second and third dimensions represent pixels along the height and width of the image, respectively. Since images in the MNIST dataset are grayscale, there's just one channel. Other datasets have images with color, in which case there are three channels: red, green, and blue (RGB).

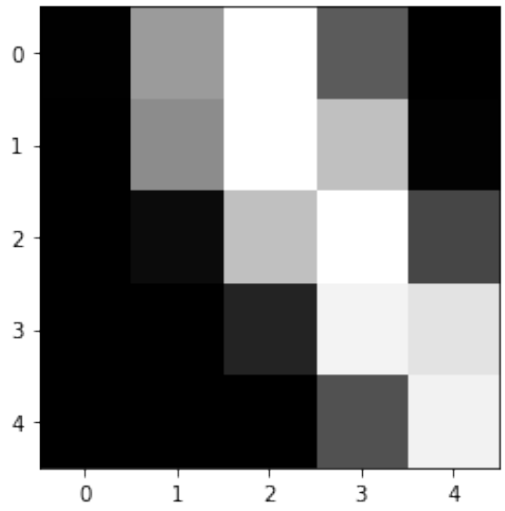
Let's look at some sample values inside the tensor.

```
print(img_tensor[0,10:15,10:15])
print(torch.max(img_tensor), torch.min(img_tensor))
```

tensor([[0.0039, 0.6039, 0.9922, 0.3529, 0.0000],
[0.0000, 0.5451, 0.9922, 0.7451, 0.0078],
[0.0000, 0.0431, 0.7451, 0.9922, 0.2745],
[0.0000, 0.0000, 0.1373, 0.9451, 0.8824],
[0.0000, 0.0000, 0.0000, 0.3176, 0.9412]])
tensor(1.) tensor(0.)

The values range from 0 to 1, with `0` representing black, `1` white, and the values in between different shades of grey. We can also plot the tensor as an image using `plt.imshow`.

```
# Plot the image by passing in the 28x28 matrix
plt.imshow(img_tensor[0,10:15,10:15], cmap='gray');
```



Note that we need to pass just the 28x28 matrix to `plt.imshow`, without a channel dimension. We also pass a color map (`cmap=gray`) to indicate that we want to see a grayscale image.

3.2 Training and validation datasets

While building real-world machine learning models, it is quite common to split the dataset into three parts:

1. **Training set** - used to train the model, i.e., compute the loss and adjust the model's weights using gradient descent.
2. **Validation set** - used to evaluate the model during training, adjust hyperparameters (learning rate, etc.), and pick the best version of the model.
3. **Test set** - used to compare different models or approaches and report the model's final accuracy.

In the MNIST dataset, there are 60,000 training images and 10,000 test images. The test set is standardized so that different researchers can report their models' results against the same collection of images.

Since there's no predefined validation set, we must manually split the 60,000 images into training and validation datasets. Let's set aside 10,000 randomly chosen images for validation. We can do this using the `random_spilt` method from PyTorch.

```
from torch.utils.data import random_split

train_ds, val_ds = random_split(dataset, [50000, 10000])
len(train_ds), len(val_ds)
```

(50000, 10000)

It's essential to choose a random sample for creating a validation set. Training data is often sorted by the target labels, i.e., images of 0s, followed by 1s, followed by 2s, etc. If we create a validation set using the last 20% of images, it would only consist of 8s and 9s. In contrast, the training set would contain no 8s or 9s. Such a training-validation would make it impossible to train a useful model.

We can now create data loaders to help us load the data in batches. We'll use a batch size of 128.

```
from torch.utils.data import DataLoader

batch_size = 128

train_loader = DataLoader(train_ds, batch_size, shuffle=True)
val_loader = DataLoader(val_ds, batch_size)
```

We set `shuffle=True` for the training data loader to ensure that the batches generated in each epoch are different. This randomization helps generalize & speed up the training process. On the other hand, since the validation data loader is used only for evaluating the model, there is no need to shuffle the images.

3.3 Model

Now that we have prepared our data loaders, we can define our model.

- A **logistic regression** model is almost identical to a linear regression model. It contains weights and bias matrices, and the output is obtained using simple matrix operations (`pred = x @ w.t() + b`).
- As we did with linear regression, we can use `nn.Linear` to create the model instead of manually creating and initializing the matrices.
- Since `nn.Linear` expects each training example to be a vector, each `1x28x28` image tensor is *flattened* into a vector of size 784 `(28*28)` before being passed into the model.
- The output for each image is a vector of size 10, with each element signifying the probability

of a particular target label (i.e., 0 to 9). The predicted label for an image is simply the one with the highest probability.

```
import torch.nn as nn

input_size = 28*28
num_classes = 10

# Logistic regression model
model = nn.Linear(input_size, num_classes)
```

Of course, this model is a lot larger than our previous model in terms of the number of parameters. Let's take a look at the weights and biases.

```
print(model.weight.shape)
model.weight
```

torch.Size([10, 784])
Parameter containing:
tensor([[0.0009, -0.0116, -0.0353, ..., 0.0250, 0.0174, -0.0170],
[0.0273, -0.0075, -0.0141, ..., -0.0279, 0.0321, 0.0207],
[0.0115, 0.0028, 0.0332, ..., 0.0286, -0.0246, -0.0237],
...,
[-0.0151, 0.0339, 0.0293, ..., 0.0080, -0.0065, 0.0281],
[-0.0011, 0.0064, 0.0177, ..., -0.0050, 0.0324, -0.0150],
[0.0147, -0.0001, -0.0042, ..., -0.0102, 0.0343, -0.0263]],
requires_grad=True)

```
print(model.bias.shape)
model.bias
```

torch.Size([10])
Parameter containing:
tensor([0.0080, 0.0105, -0.0150, -0.0245, 0.0057, -0.0085, 0.0240, 0.0297,
0.0087, 0.0296], requires_grad=True)

Although there are a total of 7850 parameters here, conceptually, nothing has changed so far. Let's try and generate some outputs using our model. We'll take the first batch of images from our dataset and pass them into our model.

```
for images, labels in train_loader:
    outputs = model(images)
    break
```

The code above leads to an error because our input data does not have the right shape. Our images are of the shape 1x28x28, but we need them to be vectors of size 784, i.e., we need to flatten them. We'll use the `.reshape` method of a tensor, which will allow us to efficiently 'view' each image as a flat vector without really creating a copy of the underlying data. To include this additional functionality within our model, we need to define a custom model by extending the `nn.Module` class from PyTorch.

```
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

model = MnistModel()
```

Inside the `__init__` constructor method, we instantiate the weights and biases using `nn.Linear`. And inside the `forward` method, which is invoked when we pass a batch of inputs to the model, we flatten the input tensor and pass it into `self.linear`.

`xb.reshape(-1, 28*28)` indicates to PyTorch that we want a *view* of the `xb` tensor with two dimensions. The length along the 2nd dimension is 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.

Note that the model no longer has `.weight` and `.bias` attributes (as they are now inside the `.linear` attribute), but it does have a `.parameters` method that returns a list containing the weights and bias.

```
model.linear
```

Linear(in_features=784, out_features=10, bias=True)

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

torch.Size([10, 784]) torch.Size([10])
[Parameter containing:
tensor([[-0.0090, -0.0159, -0.0224, ..., -0.0215, 0.0283, -0.0320],
[0.0287, -0.0217, -0.0159, ..., -0.0089, -0.0199, -0.0269],
[0.0217, 0.0316, -0.0253, ..., 0.0336, -0.0165, 0.0027],
...,
[0.0079, -0.0068, -0.0282, ..., -0.0229, 0.0235, 0.0244],
[0.0199, -0.0111, -0.0084, ..., -0.0271, -0.0252, 0.0264],
[-0.0146, 0.0340, -0.0004, ..., 0.0189, 0.0017, 0.0197]],
requires_grad=True), Parameter containing:
tensor([-0.0108, 0.0181, -0.0022, 0.0184, -0.0075, -0.0040, -0.0157, 0.0221,
-0.0005, 0.0039], requires_grad=True)]

We can use our new custom model in the same way as before.

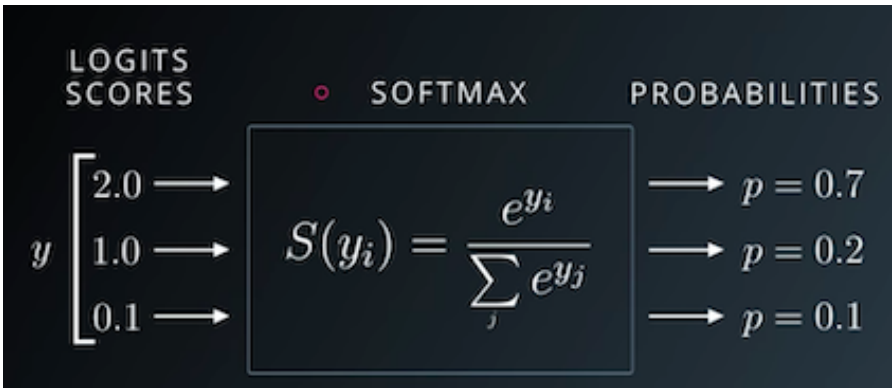
```
for images, labels in train_loader:
    print(images.shape)
    outputs = model(images)
    break

print('outputs.shape : ', outputs.shape)
print('Sample outputs :\n', outputs[:2].data)
```

torch.Size([128, 1, 28, 28])
outputs.shape : torch.Size([128, 10])
Sample outputs :
tensor([[0.0967, 0.2726, -0.0690, -0.0207, -0.3350, 0.2693, 0.2956, -0.1389,

0.0620, -0.0604],
[-0.0071, 0.1541, -0.0696, -0.0508, -0.2197, 0.1108, 0.0263, -0.0490,
0.0100, -0.1085]])

To convert the output rows into probabilities, we use the softmax function, which has the following formula:



While it's easy to implement the softmax function, we'll use the implementation that's provided within PyTorch because it works well with multidimensional tensors (a list of output rows in our case).

```
import torch.nn.functional as F
```

The softmax function is included in the `torch.nn.functional` package and requires us to specify a dimension along which the function should be applied.

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

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

# Add up the probabilities of an output row
print("Sum: ", torch.sum(probs[0]).item())
```

Sample probabilities:
tensor([[0.1042, 0.1242, 0.0883, 0.0927, 0.0677, 0.1238, 0.1271, 0.0823, 0.1006,
0.0890],
[0.1008, 0.1185, 0.0947, 0.0965, 0.0815, 0.1134, 0.1042, 0.0967, 0.1026,
0.0911]])

Sum: 1.0

Finally, we can determine the predicted label for each image by simply choosing the index of the element with the highest probability in each output row. We can do this using `torch.max`, which returns each row's largest element and the corresponding index.

```
max_probs, preds = torch.max(probs, dim=1)
print(preds)
print(max_probs)
```

tensor([6, 1, 6, 1, 5, 1, 0, 5, 5, 6, 5, 5, 1, 4, 6, 5, 1, 6, 0, 6, 4, 1, 5, 1,
5, 5, 5, 1, 2, 5, 3, 5, 5, 0, 1, 1, 1, 5, 1, 1, 5, 5, 5, 5, 8, 1, 5, 4,
0, 5, 5, 5, 1, 1, 1, 1, 5, 1, 1, 5, 6, 5, 1, 1, 5, 5, 5, 5, 0, 5, 5, 9,
0, 3, 5, 3, 5, 5, 5, 6, 5, 5, 0, 9, 5, 5, 0, 8, 2, 3, 1, 1, 1, 5, 5, 0,
1, 1, 5, 2, 1, 1, 5, 0, 5, 0, 1, 5, 1, 5, 1, 5, 5, 2, 1, 5, 1, 5, 0, 4,
5, 3, 5, 5, 1, 0, 0, 5])
tensor([0.1271, 0.1185, 0.1188, 0.1175, 0.1220, 0.1229, 0.1318, 0.1710, 0.1210,
0.1396, 0.1285, 0.1397, 0.1276, 0.1250, 0.1390, 0.1707, 0.1208, 0.1291,
0.1232, 0.1262, 0.1235, 0.1293, 0.1354, 0.1374, 0.1518, 0.1911, 0.1292,
0.1317, 0.1381, 0.1327, 0.1189, 0.1326, 0.1310, 0.1183, 0.1467, 0.1130,
0.1292, 0.1280, 0.1329, 0.1276, 0.1552, 0.1281, 0.1146, 0.1223, 0.1201,
0.1321, 0.1272, 0.1356, 0.1205, 0.1410, 0.1164, 0.1287, 0.1425, 0.1222,
0.1364, 0.1418, 0.1303, 0.1262, 0.1371, 0.1371, 0.1306, 0.1278, 0.1461,
0.1272, 0.1433, 0.1267, 0.1369, 0.1153, 0.1262, 0.1252, 0.1268, 0.1163,
0.1229, 0.1275, 0.1426, 0.1180, 0.1248, 0.1319, 0.1329, 0.1216, 0.1492,
0.1208, 0.1583, 0.1354, 0.1339, 0.1218, 0.1224, 0.1296, 0.1301, 0.1239,
0.1281, 0.1275, 0.1270, 0.1189, 0.1246, 0.1167, 0.1192, 0.1337, 0.1245,
0.1484, 0.1202, 0.1299, 0.1174, 0.1448, 0.1440, 0.1305, 0.1297, 0.1454,
0.1701, 0.1270, 0.1465, 0.1339, 0.1216, 0.1232, 0.1193, 0.1353, 0.1269,
0.1252, 0.1216, 0.1222, 0.1359, 0.1332, 0.1442, 0.1237, 0.1275, 0.1272,
0.1217, 0.1240], grad_fn=)

The numbers printed above are the predicted labels for the first batch of training images. Let's compare them with the actual labels.

```
labels
```

tensor([7, 4, 7, 0, 3, 4, 0, 0, 1, 6, 1, 3, 4, 1, 7, 0, 3, 7, 7, 7, 3, 4, 0, 6,
0, 8, 1, 0, 3, 9, 9, 1, 9, 4, 0, 9, 0, 9, 6, 9, 8, 4, 3, 5, 9, 0, 6, 2,
8, 7, 1, 4, 6, 9, 9, 5, 2, 5, 6, 5, 7, 3, 9, 6, 4, 1, 7, 5, 5, 1, 5, 1,
9, 0, 8, 0, 0, 1, 5, 0, 8, 7, 2, 5, 5, 1, 6, 4, 6, 1, 0, 6, 3, 4, 6, 5,
9, 9, 9, 5, 6, 0, 6, 7, 4, 9, 6, 6, 0, 7, 0, 1, 1, 3, 0, 4, 6, 1, 2, 1,
9, 8, 0, 1, 9, 9, 7, 2])

Most of the predicted labels are different from the actual labels. That's because we have started with randomly initialized weights and biases. We need to train the model, i.e., adjust the weights using gradient descent to make better predictions.

3.4 Evaluation Metric and Loss Function

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

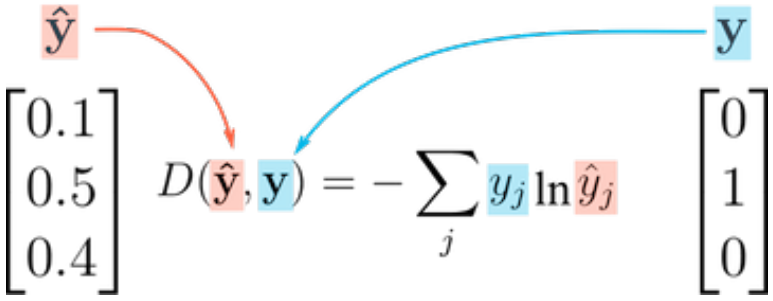
```
accuracy(outputs, labels)
```

tensor(0.0625)

Accuracy is an excellent way for us (humans) to evaluate the model. However, we can't use it as a loss function for optimizing our model using gradient descent for the following reasons:

1. It's not a differentiable function. `torch.max` and `==` are both non-continuous and non-differentiable operations, so we can't use the accuracy for computing gradients w.r.t the weights and biases.
2. It doesn't take into account the actual probabilities predicted by the model, so it can't provide sufficient feedback for incremental improvements.

For these reasons, accuracy is often used as an **evaluation metric** for classification, but not as a loss function. A commonly used loss function for classification problems is the **cross-entropy**, which has the following formula:



Unlike accuracy, cross-entropy is a continuous and differentiable function. It also provides useful feedback for incremental improvements in the model (a slightly higher probability for the correct label leads to a lower loss). These two factors make cross-entropy a better choice for the loss function.

As you might expect, PyTorch provides an efficient and tensor-friendly implementation of cross-entropy as part of the `torch.nn.functional` package. Moreover, it also performs softmax internally, so we can directly pass in the model's outputs without converting them into probabilities.

```
loss_fn = F.cross_entropy
```

```
# Loss for current batch of data
loss = loss_fn(outputs, labels)
print(loss)
```

tensor(2.3418, grad_fn=)

We know that cross-entropy is the negative logarithm of the predicted probability of the correct label averaged over all training samples. Therefore, one way to interpret the resulting number e.g. `2.23` is look at `e^-2.23` which is around `0.1` as the predicted probability of the correct label, on average. *The lower the loss, The better the model.*

3.5 Training the model

Now that we have defined the data loaders, model, loss function and optimizer, we are ready to train the model. The training process is identical to linear regression, with the addition of a "validation phase" to evaluate the model in each epoch. Here's what it looks like in pseudocode:

```
for epoch in range(num_epochs):
    # Training phase
    for batch in train_loader:
```

```
        # Generate predictions
        # Calculate loss
        # Compute gradients
        # Update weights
        # Reset gradients

    # Validation phase
    for batch in val_loader:
        # Generate predictions
        # Calculate loss
        # Calculate metrics (accuracy etc.)
    # Calculate average validation loss & metrics

    # Log epoch, loss & metrics for inspection
```

Some parts of the training loop are specific the specific problem we're solving (e.g. loss function, metrics etc.) whereas others are generic and can be applied to any deep learning problem.

We'll include the problem-independent parts within a function called `fit`, which will be used to train the model. The problem-specific parts will be implemented by adding new methods to the `nn.Module` class.

```
def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD):
    optimizer = opt_func(model.parameters(), lr)
    history = [] # for recording epoch-wise results

    for epoch in range(epochs):

        # Training Phase
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

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

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

    return history
```

The `fit` function records the validation loss and metric from each epoch. It returns a history of the training, useful for debugging & visualization.

Configurations like batch size, learning rate, etc. (called hyperparameters), need to picked in advance while training machine learning models. Choosing the right hyperparameters is critical for training a reasonably accurate model within a reasonable amount of time. It is an active area of research and experimentation in machine learning.

Let's define the `evaluate` function, used in the validation phase of `fit`.

```
def evaluate(model, val_loader):
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)
```

Finally, let's redefine the `MnistModel` class to include additional methods `training_step`, `validation_step`, `validation_epoch_end`, and `epoch_end` used by `fit` and `evaluate`.

```
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

    def training_step(self, batch):
        images, labels = batch
        out = self(images)                  # Generate predictions
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss
```

```
    def validation_step(self, batch):
        images, labels = batch
        out = self(images)                    # Generate predictions
        loss = F.cross_entropy(out, labels)   # Calculate loss
        acc = accuracy(out, labels)           # Calculate accuracy
        return {'val_loss': loss, 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()      # Combine accuracies
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch,
result['val_loss'], result['val_acc']))
```

```
model = MnistModel()
```

Before we train the model, let's see how the model performs on the validation set with the initial set of randomly initialized weights & biases.

```
result0 = evaluate(model, val_loader)
result0
```

{'val_acc': 0.10977056622505188, 'val_loss': 2.3349318504333496}

The initial accuracy is around 10%, which one might expect from a randomly initialized model (since it has a 1 in 10 chance of getting a label right by guessing randomly).

We are now ready to train the model. Let's train for five epochs and look at the results.

```
history1 = fit(5, 0.001, model, train_loader, val_loader)
```

Epoch [0], val_loss: 1.9552, val_acc: 0.6153
Epoch [1], val_loss: 1.6839, val_acc: 0.7270
Epoch [2], val_loss: 1.4819, val_acc: 0.7587
Epoch [3], val_loss: 1.3295, val_acc: 0.7791
Epoch [4], val_loss: 1.2124, val_acc: 0.7969

```
history2 = fit(5, 0.001, model, train_loader, val_loader)
```

Epoch [0], val_loss: 1.1205, val_acc: 0.8081
Epoch [1], val_loss: 1.0467, val_acc: 0.8165
Epoch [2], val_loss: 0.9862, val_acc: 0.8237
Epoch [3], val_loss: 0.9359, val_acc: 0.8281
Epoch [4], val_loss: 0.8934, val_acc: 0.8322

```
history3 = fit(5, 0.001, model, train_loader, val_loader)
```

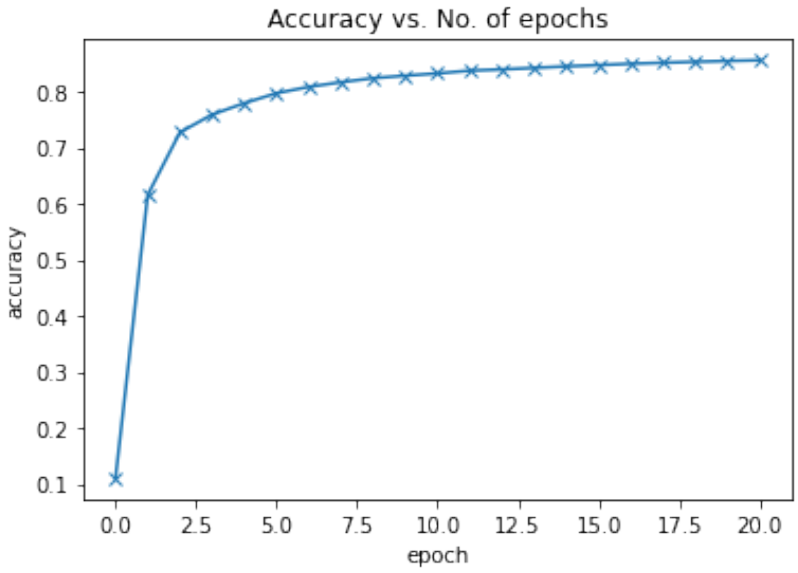
Epoch [0], val_loss: 0.8569, val_acc: 0.8371
Epoch [1], val_loss: 0.8254, val_acc: 0.8393
Epoch [2], val_loss: 0.7977, val_acc: 0.8420
Epoch [3], val_loss: 0.7733, val_acc: 0.8447
Epoch [4], val_loss: 0.7515, val_acc: 0.8470

```
history4 = fit(5, 0.001, model, train_loader, val_loader)
```

Epoch [0], val_loss: 0.7320, val_acc: 0.8494
Epoch [1], val_loss: 0.7144, val_acc: 0.8512
Epoch [2], val_loss: 0.6985, val_acc: 0.8528
Epoch [3], val_loss: 0.6839, val_acc: 0.8543
Epoch [4], val_loss: 0.6706, val_acc: 0.8557

While the accuracy does continue to increase as we train for more epochs, the improvements get smaller with every epoch. Let's visualize this using a line graph.

```
history = [result0] + history1 + history2 + history3 + history4
accuracies = [result['val_acc'] for result in history]
plt.plot(accuracies, '-x')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Accuracy vs. No. of epochs');
```



It's quite clear from the above picture that the model probably won't cross the accuracy threshold of 90% even after training for a very long time. One possible reason for this is that the learning rate might be too high. The model's parameters may be "bouncing" around the optimal set of parameters for the lowest loss. You can try reducing the learning rate and training for a few more epochs to see if it helps.

The more likely reason that **the model just isn't powerful enough**. If you remember our initial hypothesis, we have assumed that the output (in this case the class probabilities) is a **linear function** of the input (pixel intensities), obtained by perfoming a matrix multiplication with the weights matrix and adding the bias. This is a fairly weak assumption, as there may not actually exist a linear relationship between the pixel intensities in an image and the digit it represents. While it works reasonably well for a simple dataset like MNIST (getting us to 85% accuracy), we need more sophisticated models that can capture non-linear relationships between image pixels and labels for complex tasks like recognizing everyday objects, animals etc.

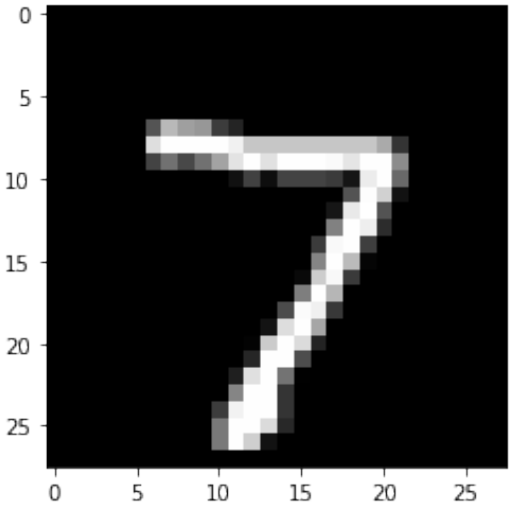
3.6 Testing with individual images

```
# Define test dataset
test_dataset = MNIST(root='data/',
                     train=False,
                     transform=transforms.ToTensor())
```

Here's a sample image from the dataset.

```
img, label = test_dataset[0]
plt.imshow(img[0], cmap='gray')
print('Shape:', img.shape)
print('Label:', label)
```

Shape: torch.Size([1, 28, 28])
Label: 7



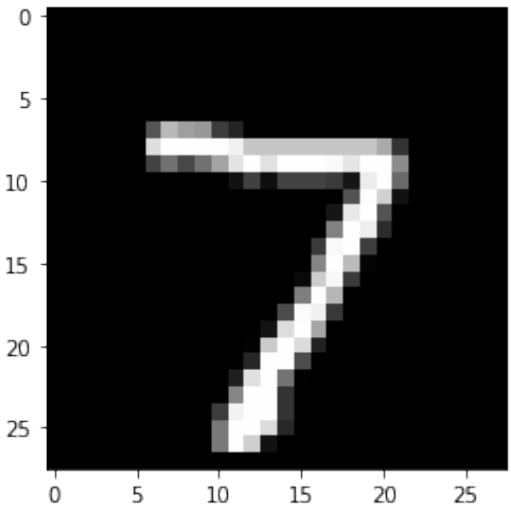
Let's define a helper function `predict_image`, which returns the predicted label for a single image tensor.

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

`img.unsqueeze` simply adds another dimension at the begining of the 1x28x28 tensor, making it a 1x1x28x28 tensor, which the model views as a batch containing a single image.

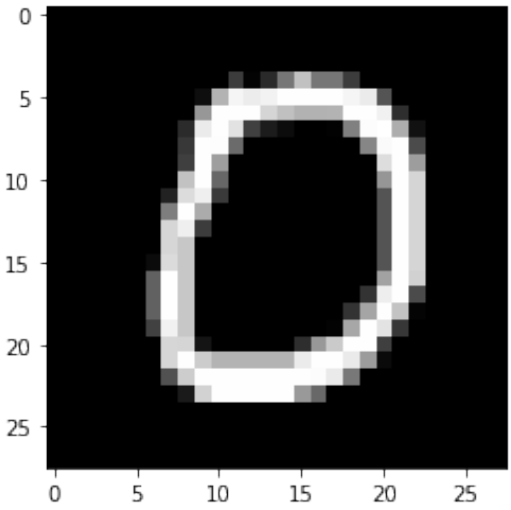
```
img, label = test_dataset[0]
plt.imshow(img[0], cmap='gray')
print('Label:', label, ', Predicted:', predict_image(img, model))
```

Label: 7 , Predicted: 7



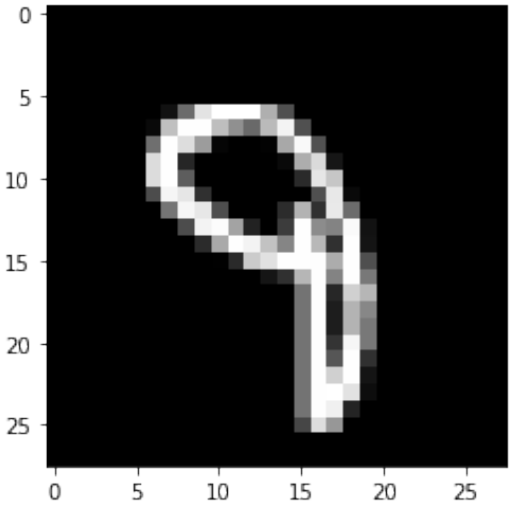

```
img, label = test_dataset[10]
plt.imshow(img[0], cmap='gray')
print('Label:', label, ', Predicted:', predict_image(img, model))
```

Label: 0 , Predicted: 0



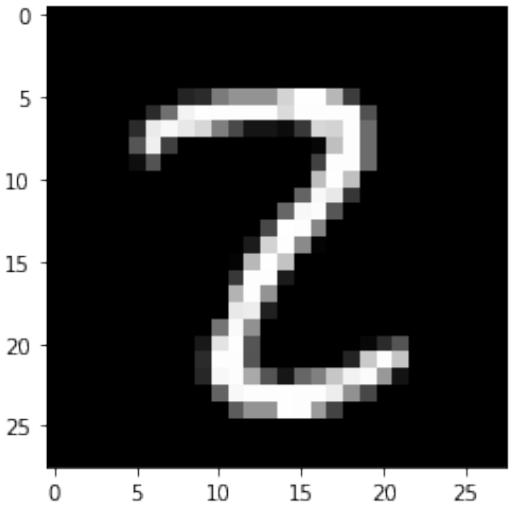
```
img, label = test_dataset[193]
plt.imshow(img[0], cmap='gray')
print('Label:', label, ', Predicted:', predict_image(img, model))
```

Label: 9 , Predicted: 4




```
img, label = test_dataset[1839]
plt.imshow(img[0], cmap='gray')
print('Label:', label, ', Predicted:', predict_image(img, model))
```

Label: 2 , Predicted: 8



Identifying where our model performs poorly can help us improve the model, by collecting more training data, increasing/decreasing the complexity of the model, and changing the hypeparameters.

As a final step, let's also look at the overall loss and accuracy of the model on the test set.

```
test_loader = DataLoader(test_dataset, batch_size=256)
result = evaluate(model, test_loader)
result
```

{'val_acc': 0.86083984375, 'val_loss': 0.6424765586853027}

We expect this to be similar to the accuracy/loss on the validation set. If not, we might need a better validation set that has similar data and distribution as the test set (which often comes from real world data).