Image Classification using Logistic Regression in PyTorch

Part 3 of "PyTorch: Zero to GANs"

This post is the third in a series of tutorials on building deep learning models with PyTorch, an open source neural networks library. Check out the full series:

1. [PyTorch Basics: Tensors & Gradients](#)
2. [Linear Regression & Gradient Descent](#)
3. [Image Classfication using Logistic Regression](#)
4. [Training Deep Neural Networks on a GPU](#)
5. [Image Classification using Convolutional Neural Networks](#)
6. [Data Augmentation, Regularization and ResNets](#)
7. [Generating Images using Generative Adverserial Networks](#)

In this tutorial, we'll use our existing knowledge of PyTorch and linear regression to solve a very different kind of problem: *image classification*. We'll use the famous *[MNIST Handwritten Digits Database](#)* as our training dataset. It consists of 28px by 28px grayscale images of handwritten digits (0 to 9), along with labels for each image indicating which digit it represents. Here are some sample images from the dataset:

System setup

This tutorial takes a code-first approach towards learning PyTorch, and you should try to follow along by running and experimenting with the code yourself. The easiest way to start executing this notebook is to click the **"Run"** button at the top of this page, and select **"Run on Kaggle"**. This will run the notebook on Kaggle, a free online service for running Jupyter notebooks (you might need to create an account).

Running on your computer locally

(Skip this if you're running on Kaggle) To run this notebook locally, clone this notebook, install the required dependencies using [conda](#), and start Jupyter by running the following commands on the terminal / Conda prompt:

```
pip install jovian --upgrade                    # Install the jovian library
jovian clone aakashns/03-logistic-regression    # Download notebook & dependencies
cd 03-logistic-regression                       # Enter the created directory
conda create -n 03-logistic-regression python=3.8 # Create an environment
conda activate 03-logistic-regression           # Activate virtual env
jupyter notebook                                # Start Jupyter
```

You can find the `notebook_id` by cliking the *Clone* button at the top of this page on Jovian. For a more detailed explanation of the above steps, check out the *System setup* section in the [first notebook](#).

Exploring the Data

Exploring the Data

We begin by importing `torch` and `torchvision`. `torchvision` contains some utilities for working with image data. It also contains helper classes to automatically download and import popular datasets like MNIST.

In [1]:

```
# Uncomment and run the commands below if imports fail
# !conda install numpy pytorch torchvision cpuonly -c pytorch -y
# !pip install matplotlib --upgrade --quiet
```

In [2]:

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

In [3]:

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

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to data/MNIST/raw/train-images-idx3-ubyte.gz

Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to data/MNIST/raw/train-labels-idx1-ubyte.gz

Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to data/MNIST/raw/t10k-images-idx3-ubyte.gz

Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to data/MNIST/raw/t10k-labels-idx1-ubyte.gz

Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw
Processing...
Done!
```

```
/opt/conda/conda-bld/pytorch_1587428190859/work/torch/csrc/utils/tensor_numpy.cpp:141: UserWarning: The given NumPy array is not writeable, and PyTorch does not support non-writeable tensors. This means you can write to the underlying (supposedly non-writeable) NumPy array using the tensor. You may want to copy the array to protect its data or make it writeable before converting it to a tensor. This type of warning will be suppressed for the rest of this program.
```

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.

In [4]:

```
len(dataset)
```

Out[4]:

```
60000
```

The dataset has 60,000 images which can be used to train the model. There is also an additonal test set of 10,000 images which can be created by passing `train=False` to the `MNIST` class.

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

Out[5]:

```
10000
```

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

In [6]:

```
dataset[0]
```

Out[6]:

```
(<PIL.Image.Image image mode=L size=28x28 at 0x7F21C8CD54D0>, 5)
```

It's a pair, consisting of a 28x28 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.

In [7]:

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

Along with importing `matplotlib`, a special statement `%matplotlib inline` is added to indicate 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 IPython 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.

In [8]:

```
image, label = dataset[0]
plt.imshow(image, cmap='gray')
print('Label:', label)
```
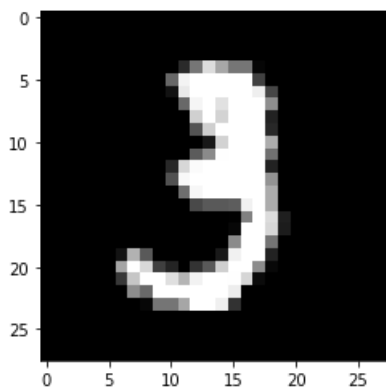
```
Label: 5
```
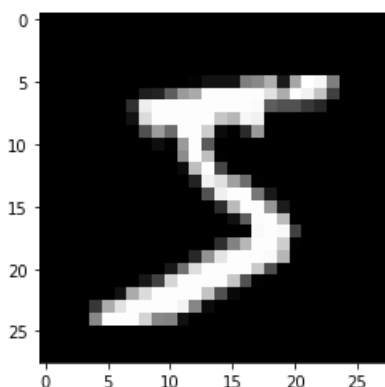

In [9]:

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

Label: 3

It's evident that these images are quite small in size, and recognizing the digits can sometimes be hard even for the human eye. 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.

In [10]:

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

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

In [11]:

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

In [12]:

```python
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 is used to keep track of the color channels. Since images in the MNIST dataset are grayscale, there's just one channel. Other datasets have images with color, in which case there are 3 channels: red, green and blue (RGB). Let's look at some sample values inside the tensor:

In [13]:

```python
print(img_tensor[:,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`.

In [14]:

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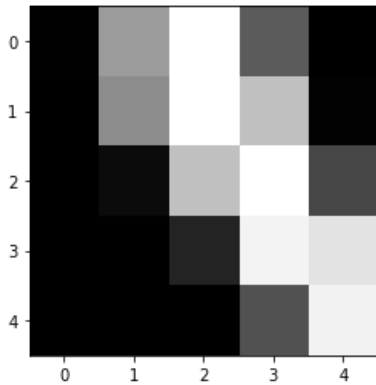

Note that we need to pass just the 28x28 matrix to `plt.imshow`, without a channel dimension. We also pass a color map (`cmap=gray`) to indicate that we want to see a grayscale image.

Training and Validation Datasets

While building real world machine learning models, it is quite common to split the dataset into 3 parts:

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

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 the results of their models against the same set 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.

In [15]:

```python
from torch.utils.data import random_split

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

Out[15]:

```
(50000, 10000)
```

It's important to choose a random sample for creating a validation set, because training data is often ordered by the target labels i.e. images of 0s, followed by images of 1s, followed by images of 2s and so on. If we were to pick a 20% validation set simply by selecting the last 20% of the images, the validation set would only consist of images of 8s and 9s, whereas the training set would contain no images of 8s and 9s. This would make it impossible to train a good model using the training set, which also performs well on the validation set (and on real world data).

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

In [16]:

```python
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 dataloader, so that the batches generated in each epoch are different, and this randomization helps generalize & speed up the training process. On the other hand, since the validation dataloader is used only for evaluating the model, there is no need to shuffle the images.

Before we move forward, let's save our progress by uploading our notebook to [Jovian.ml](#)

In [17]:

```
!pip install jovian --upgrade --quiet
```

WARNING: You are using pip version 20.1; however, version 20.1.1 is available.

You should consider upgrading via the '/opt/conda/bin/python3.7 -m pip install --upgrade pip' command.

In [18]:

```
import jovian
```

In [19]:

```
jovian.commit(project='03-logistic-regression', environment=None)
```

```
[jovian] Attempting to save notebook..
[jovian] Please enter your API key ( from https://jovian.ml/ ):
API KEY: ········
[jovian] Updating notebook "aakashns/03-logistic-regression" on https://jovian.ml/
[jovian] Uploading notebook..
[jovian] Committed successfully! https://jovian.ml/aakashns/03-logistic-regression
```

Out[19]:

```
'https://jovian.ml/aakashns/03-logistic-regression'
```

`jovian.commit` uploads the notebook to your [Jovian.ml](#) account and creates a sharable link for the notebook. You can use this link to share your work and let anyone reproduce it easily with the jovian clone command. Jovian also includes a powerful commenting interface, so you (and others) can discuss & comment on specific parts of your notebook.

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 i.e. there are weights and bias matrices, and the output is obtained using simple matrix operations (`pred = x @ w.t() + b`).
- Just as we did with linear regression, we can use `nn.Linear` to create the model instead of defining and initializing the matrices manually.
- Since `nn.Linear` expects the each training example to be a vector, each `1x28x28` image tensor needs to be flattened out into a vector of size 784 (`28*28`), before being passed into the model.
- The output for each image is vector of size 10, with each element of the vector signifying the probability a particular target label (i.e. 0 to 9). The predicted label for an image is simply the one with the highest probability.

In [20]:

```
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.

In [21]:

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

torch.Size([10, 784])

Out[21]:

```
Parameter containing:
tensor([[ 0.0259, -0.0354,  0.0162,  ..., -0.0020,  0.0025, -0.0085],
        [ 0.0046,  0.0331, -0.0156,  ...,  0.0020,  0.0018, -0.0183],
        [ 0.0211, -0.0052,  0.0136,  ...,  0.0283, -0.0236, -0.0110],
        ...,
        [ 0.0264,  0.0146, -0.0046,  ...,  0.0136, -0.0019,  0.0240],
        [ 0.0123, -0.0213, -0.0099,  ...,  0.0147,  0.0317, -0.0297],
        [ 0.0138, -0.0138,  0.0058,  ...,  0.0105, -0.0080,  0.0085]],
       requires_grad=True)
```

In [22]:

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

torch.Size([10])

Out[22]:

```
Parameter containing:
tensor([-0.0357,  0.0206, -0.0147, -0.0297,  0.0244,  0.0197, -0.0233, -0.0197,
         0.0335, -0.0075], 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 100 images from our dataset, and pass them into our model.

In [23]:

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

```
tensor([0, 7, 9, 5, 0, 9, 7, 3, 6, 6, 9, 7, 0, 3, 1, 3, 4, 1, 9, 3, 7, 7, 8, 6,
        4, 0, 0, 8, 4, 2, 7, 8, 6, 3, 1, 3, 1, 5, 2, 6, 2, 2, 4, 5, 8, 2, 4, 0,
        0, 1, 6, 4, 0, 8, 5, 1, 2, 1, 9, 0, 7, 0, 3, 2, 8, 0, 0, 3, 9, 9, 6, 5,
        7, 4, 7, 8, 4, 0, 4, 2, 9, 9, 9, 8, 6, 6, 5, 2, 9, 4, 4, 2, 5, 1, 7, 2,
        8, 0, 3, 3, 6, 6, 8, 4, 2, 0, 5, 8, 0, 1, 9, 5, 4, 5, 6, 8, 6, 4, 7, 3,
        5, 3, 9, 1, 8, 3, 5, 6])
torch.Size([128, 1, 28, 28])
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-23-72eddc737460> in <module>
      2         print(labels)
      3         print(images.shape)
----> 4         outputs = model(images)
      5         break

/opt/conda/lib/python3.7/site-packages/torch/nn/modules/module.py in __call__(self, *input, **kwargs)
    548                 result = self._slow_forward(*input, **kwargs)
    549             else:
--> 550                 result = self.forward(*input, **kwargs)
    551             for hook in self._forward_hooks.values():
    552                 hook_result = hook(self, input, result)

/opt/conda/lib/python3.7/site-packages/torch/nn/modules/linear.py in forward(self, input)
     85
     86     def forward(self, input):
---> 87         return F.linear(input, self.weight, self.bias)
```

```
  88
  89        def extra_repr(self):
```

/opt/conda/lib/python3.7/site-packages/torch/nn/functional.py in linear(input, weight, bias)
```
   1610              ret = torch.addmm(bias, input, weight.t())
   1611          else:
-> 1612              output = input.matmul(weight.t())
   1613              if bias is not None:
   1614                  output += bias
```

RuntimeError: size mismatch, m1: [3584 x 28], m2: [784 x 10] at /opt/conda/conda-bld/pytorch_1587428190
859/work/aten/src/TH/generic/THTensorMath.cpp:41

This 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 out. We'll use the `.reshape` method of a tensor, which will allow us to efficiently 'view' each image as a flat vector, without really chaging 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.

In [24]:

```
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 out the input tensor, and then 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, where 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 which returns a list containing the weights and bias, and can be used by a PyTorch optimizer.

In [25]:

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

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

Out[25]:

```
[Parameter containing:
 tensor([[-0.0005, -0.0333,  0.0218,  ..., -0.0252, -0.0036, -0.0218],
         [-0.0006, -0.0019,  0.0209,  ..., -0.0135,  0.0084, -0.0339],
         [ 0.0156, -0.0026,  0.0324,  ...,  0.0137, -0.0011,  0.0164],
         ...,
         [-0.0179, -0.0130,  0.0106,  ..., -0.0334, -0.0087, -0.0024],
         [ 0.0260, -0.0239, -0.0128,  ...,  0.0348, -0.0138, -0.0109],
         [-0.0113,  0.0029,  0.0326,  ..., -0.0353,  0.0054, -0.0277]],
        requires_grad=True),
 Parameter containing:
 tensor([ 0.0023,  0.0008, -0.0276,  0.0191,  0.0057, -0.0251, -0.0249, -0.0162,
         -0.0140,  0.0050], requires_grad=True)]
```

Our new custom model can be used in the exact same way as before. Let's see if it works.

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

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

```
outputs.shape :  torch.Size([128, 10])
Sample outputs :
 tensor([[ 0.3503,  0.0264,  0.1077,  0.1449,  0.2338, -0.2030, -0.3445, -0.2275,
         -0.0768, -0.2349],
        [ 0.0698,  0.1846,  0.1106,  0.3781, -0.1750,  0.2208, -0.0559,  0.0833,
         -0.2360, -0.1506]])
```

For each of the 100 input images, we get 10 outputs, one for each class. As discussed earlier, we'd like these outputs to represent probabilities, but for that the elements of each output row must lie between 0 to 1 and add up to 1, which is clearly not the case here.

To convert the output rows into probabilities, we use the softmax function, which has the following formula:
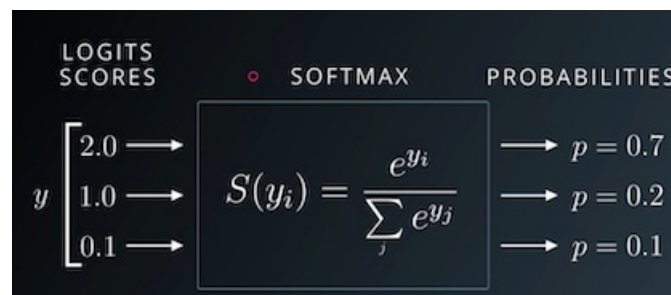
First we replace each element `yi` in an output row by `e^yi` , which makes all the elements positive, and then we divide each element by the sum of all elements to ensure that they add up to 1.

While it's easy to implement the softmax function (you should try it!), 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).

```python
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 softmax must be applied.

```python
# 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([[0.1417, 0.1025, 0.1112, 0.1154, 0.1261, 0.0815, 0.0707, 0.0795, 0.0925,
         0.0789],
        [0.1010, 0.1133, 0.1052, 0.1374, 0.0790, 0.1174, 0.0890, 0.1023, 0.0744,
         0.0810]])
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. This is done using `torch.max` , which returns the largest element and the index of the largest element along a particular dimension of a tensor.

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

```
tensor([0, 3, 0, 3, 4, 3, 3, 3, 5, 0, 3, 3, 2, 3, 3, 8, 4, 2, 1, 3, 1, 4, 7, 3,
        3, 0, 3, 3, 3, 6, 6, 3, 7, 1, 3, 3, 3, 2, 6, 2, 1, 2, 3, 6, 1, 3, 1, 3,
        3, 0, 3, 0, 3, 3, 4, 3, 3, 3, 2, 5, 3, 6, 3, 0, 3, 3, 8, 2, 4, 3, 4, 6,
        2, 0, 3, 3, 0, 3, 2, 3, 3, 3, 3, 3, 3, 3, 1, 0, 3, 2, 2, 3, 2, 3, 3,
        3, 3, 4, 7, 5, 2, 3, 0, 2, 3, 3, 5, 3, 3, 3, 6, 3, 1, 2, 3, 2, 3, 2, 4,
        3, 3, 2, 1, 3, 1, 3, 4])
tensor([0.1417, 0.1374, 0.1305, 0.1195, 0.1160, 0.1401, 0.1343, 0.1452, 0.1207,
        0.1188, 0.1356, 0.1468, 0.1506, 0.1554, 0.1358, 0.1290, 0.1236, 0.1470,
        0.1250, 0.1504, 0.1245, 0.1248, 0.1399, 0.1271, 0.1249, 0.1274, 0.1270,
        0.1461, 0.1252, 0.1213, 0.1256, 0.1381, 0.1301, 0.1328, 0.1363, 0.1545,
        0.1343, 0.1443, 0.1383, 0.1246, 0.1256, 0.1277, 0.1439, 0.1319, 0.1283,
        0.1291, 0.1247, 0.1307, 0.1482, 0.1269, 0.1407, 0.1187, 0.1479, 0.1387,
        0.1134, 0.1463, 0.1330, 0.1424, 0.1489, 0.1217, 0.1432, 0.1283, 0.1357,
        0.1218, 0.1448, 0.1361, 0.1172, 0.1275, 0.1293, 0.1593, 0.1334, 0.1431,
        0.1209, 0.1282, 0.1415, 0.1514, 0.1225, 0.1406, 0.1235, 0.1329, 0.1417,
        0.1520, 0.1262, 0.1510, 0.1184, 0.1318, 0.1268, 0.1170, 0.1234, 0.1387,
        0.1348, 0.1507, 0.1579, 0.1242, 0.1249, 0.1384, 0.1242, 0.1361, 0.1157,
        0.1249, 0.1386, 0.1410, 0.1231, 0.1407, 0.1304, 0.1411, 0.1324, 0.1294,
        0.1451, 0.1393, 0.1533, 0.1624, 0.1204, 0.1202, 0.1297, 0.1656, 0.1412,
        0.1347, 0.1481, 0.1243, 0.1323, 0.1203, 0.1321, 0.1160, 0.1386, 0.1244,
        0.1347, 0.1349], grad_fn=<MaxBackward0>)
```

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

Clearly, the predicted and the actual labels are completely different. Obviously, 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.

Evaluation Metric and Loss Function

Just as with linear regression, we need a way to evaluate how well our model is performing. A natural way to do this would be to find the percentage of labels that were predicted correctly i.e. the **accuracy** of the predictions.

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

The `==` performs an element-wise comparison of two tensors with the same shape, and returns a tensor of the same shape, containing 0s for unequal elements, and 1s for equal elements. Passing the result to `torch.sum` returns the number of labels that were predicted correctly. Finally, we divide by the total number of images to get the accuracy.

Note that we don't need to apply softmax to the outputs, since it doesn't change the relative order of the results. This is because `e^x` is an increasing function i.e. if `y1 > y2`, then `e^y1 > e^y2` and the same holds true after averaging out the values to get the softmax.

Let's calculate the accuracy of the current model, on the first batch of data. Obviously, we expect it to be pretty bad.

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

```
tensor(0.0781)
```

While the accuracy is a great way for us (humans) to evaluate the model, it can't be used 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.

Due to these reasons, accuracy is a great **evaluation metric** for classification, but not a good loss function. A commonly used loss function for classification problems is the **cross entropy**, which has the following formula:

$$\hat{\mathbf{y}} \quad \begin{bmatrix} 0.1 \\ 0.5 \\ 0.4 \end{bmatrix} \quad D(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{j} y_j \ln \hat{y}_j \quad \mathbf{y} \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

While it looks complicated, it's actually quite simple:

- For each output row, pick the predicted probability for the correct label. E.g. if the predicted probabilities for an image are `[0.1, 0.3, 0.2, ...]` and the correct label is `1`, we pick the corresponding element `0.3` and ignore the rest.
- Then, take the [logarithm](#) of the picked probability. If the probability is high i.e. close to 1, then its logarithm is a very small negative value, close to 0. And if the probability is low (close to 0), then the logarithm is a very large negative value. We also multiply the result by -1, which results is a large postive value of the loss for poor predictions.
- Finally, take the average of the cross entropy across all the output rows to get the overall loss for a batch of data.

Unlike accuracy, cross-entropy is a continuous and differentiable function that also provides good feedback for incremental improvements in the model (a slightly higher probability for the correct label leads to a lower loss). This makes it a good 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 outputs of the model without converting them into probabilities.

```
loss_fn = F.cross_entropy
```

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

```
tensor(2.3522, grad_fn=<NllLossBackward>)
```

Since the cross entropy is the negative logarithm of the predicted probability of the correct label averaged over all training samples, 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. *Lower the loss, better the model.*

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. Let's impelment the problem-specific parts within our `MnistModel` class:

In [35]:

```python
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()
```

Now we'll define an `evaluate` function, which will perform the validation phase, and a `fit` function which will peform the entire training process.

In [36]:

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

```python
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()
            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 and returns a history of the training process. This is useful for debuggin & visualizing the training process. 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.

Configurations like batch size, learning rate etc. need to picked in advance while training machine learning models, and are called hyperparameters. Picking the right hyperparameters is critical for training an accurate model within a reasonable amount of time, and is an active area of research and experimentation. Feel free to try different learning rates and see how it affects the training process.

In [37]:

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

Out[37]:

```
{'val_loss': 2.305769443511963, 'val_acc': 0.09581001102924347}
```

The initial accuracy is around 10%, which is what one might expect from a randomly intialized model (since it has a 1 in 10 chance of getting a label right by guessing randomly). Also note that we are using the `.format` method with the message string to print only the first four digits after the decimal point.

We are now ready to train the model. Let's train for 5 epochs and look at the results.

In [38]:

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

```
Epoch [0], val_loss: 1.8825, val_acc: 0.6854
Epoch [1], val_loss: 1.5864, val_acc: 0.7516
Epoch [2], val_loss: 1.3771, val_acc: 0.7808
Epoch [3], val_loss: 1.2260, val_acc: 0.7983
Epoch [4], val_loss: 1.1135, val_acc: 0.8110
```

That's a great result! With just 5 epochs of training, our model has reached an accuracy of over 80% on the validation set. Let's see if we can improve that by training for a few more epochs.

In [39]:

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

```
Epoch [0], val_loss: 1.0272, val_acc: 0.8200
Epoch [1], val_loss: 0.9592, val_acc: 0.8262
Epoch [2], val_loss: 0.9042, val_acc: 0.8320
Epoch [3], val_loss: 0.8589, val_acc: 0.8352
Epoch [4], val_loss: 0.8208, val_acc: 0.8390
```

In [40]:

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

```
Epoch [0], val_loss: 0.7884, val_acc: 0.8419
Epoch [1], val_loss: 0.7604, val_acc: 0.8448
Epoch [2], val_loss: 0.7359, val_acc: 0.8471
Epoch [3], val_loss: 0.7144, val_acc: 0.8491
Epoch [4], val_loss: 0.6953, val_acc: 0.8513
```

In [41]:

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

```
Epoch [0], val_loss: 0.6781, val_acc: 0.8532
Epoch [1], val_loss: 0.6627, val_acc: 0.8552
Epoch [2], val_loss: 0.6486, val_acc: 0.8563
Epoch [3], val_loss: 0.6359, val_acc: 0.8579
Epoch [4], val_loss: 0.6242, val_acc: 0.8593
```

While the accuracy does continue to increase as we train for more epochs, the improvements get smaller with every epoch. This is easier to see using a line graph.

In [42]:

```
# Replace these values with your results
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');
```
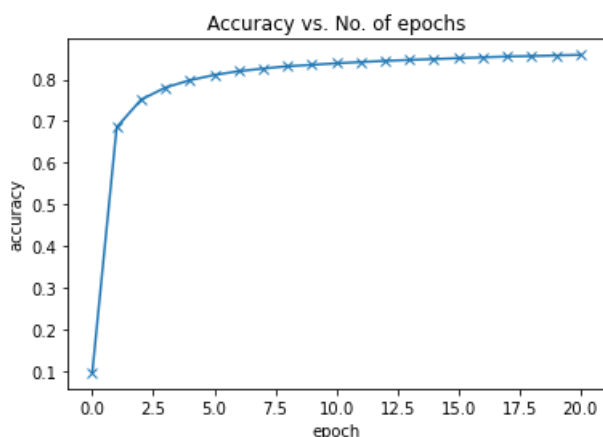

It's quite clear from the above picture that the model probably won't cross the accuracy threshold of 90% even after training for a very long time. One possible reason for this is that the learning rate might be too high. It's possible that the model's paramaters are "bouncing" around the optimal set of parameters that have 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.

This would be a good time to save our work. Along with the notebook, we can also record some metrics from our training.

In [43]:

```
jovian.log_metrics(val_acc=history[-1]['val_acc'], val_loss=history[-1]['val_loss'])
```

```
[jovian] Metrics logged.
```

```
jovian.commit(project='03-logistic-regression', environment=None)
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/03-logistic-regression" on https://jovian.ml/
[jovian] Uploading notebook..
[jovian] Attaching records (metrics, hyperparameters, dataset etc.)
[jovian] Committed successfully! https://jovian.ml/aakashns/03-logistic-regression
```

Out[44]:

```
'https://jovian.ml/aakashns/03-logistic-regression'
```

Testing with individual images

While we have been tracking the overall accuracy of a model so far, it's also a good idea to look at model's results on some sample images. Let's test out our model with some images from the predefined test dataset of 10000 images. We begin by recreating the test dataset with the `ToTensor` transform.

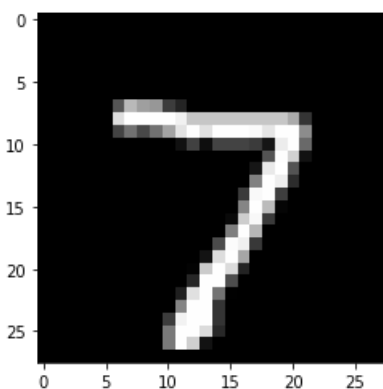In [45]:

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

Here's a sample image from the dataset.

In [46]:

```
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
```


In [47]:

```
img.unsqueeze(0).shape
```

Out[47]:

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

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

In [48]:

```python
def predict_image(img, model):
    xb = img.unsqueeze(0)
    yb = model(xb)
    _, preds  = torch.max(yb, dim=1)
    return preds[0].item()
```
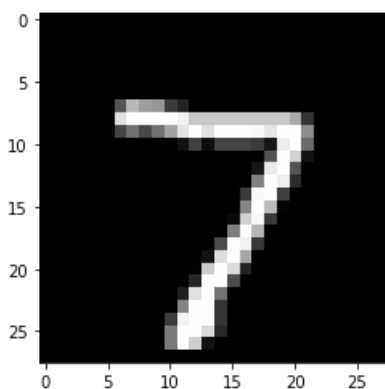
`img.unsqueeze` simply adds another dimension at the begining of the 1x28x28 tensor, making it a 1x1x28x28 tensor, which the model views as a batch containing a single image.

Let's try it out with a few images.

In [49]:

```python
img, label = test_dataset[0]
plt.imshow(img[0], cmap='gray')
print('Label:', label, ', Predicted:', predict_image(img, model))
```
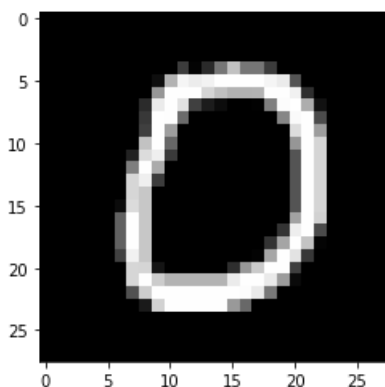
Label: 7 , Predicted: 7

In [50]:

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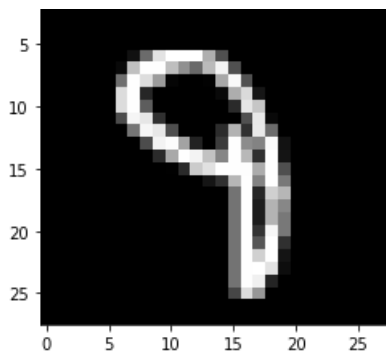
Label: 0 , Predicted: 0

In [51]:

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

Label: 9 , Predicted: 9

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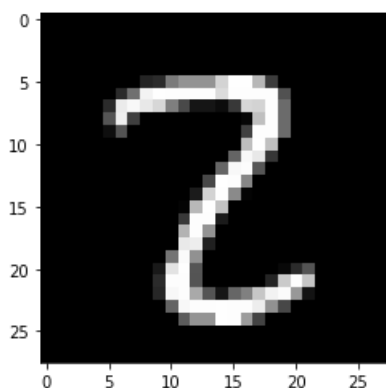```

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_loss': 0.5958371162414551, 'val_acc': 0.867382824420929}

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).

Saving and loading the model

Since we've trained our model for a long time and achieved a resonable accuracy, it would be a good idea to save the weights and bias matrices to disk, so that we can reuse the model later and avoid retraining from scratch. Here's how you can save the model.

```
torch.save(model.state_dict(), 'mnist-logistic.pth')
```

The `.state_dict` method returns an `OrderedDict` containing all the weights and bias matrices mapped to the right attributes of the model.

In [55]:

```
model.state_dict()
```

Out[55]:

```
OrderedDict([('linear.weight',
              tensor([[ 0.0251,  0.0241, -0.0165,  ..., -0.0104, -0.0195,  0.0201],
                      [-0.0234,  0.0099,  0.0338,  ..., -0.0203, -0.0133,  0.0212],
                      [ 0.0063, -0.0131,  0.0202,  ..., -0.0164,  0.0221,  0.0011],
                      ...,
                      [-0.0079,  0.0126, -0.0063,  ..., -0.0049, -0.0159,  0.0094],
                      [-0.0264, -0.0351,  0.0151,  ...,  0.0070, -0.0234, -0.0352],
                      [ 0.0140, -0.0059,  0.0071,  ...,  0.0348,  0.0120,  0.0213]])),
             ('linear.bias',
              tensor([-0.0809,  0.1206, -0.0430, -0.0242, -0.0004,  0.0362,  0.0046,  0.0181,
                      -0.0809, -0.0435]))])
```

To load the model weights, we can instante a new object of the class `MnistModel` , and use the `.load_state_dict` method.

In [56]:

```
model2 = MnistModel()
model2.load_state_dict(torch.load('mnist-logistic.pth'))
model2.state_dict()
```

Out[56]:

```
OrderedDict([('linear.weight',
              tensor([[ 0.0251,  0.0241, -0.0165,  ..., -0.0104, -0.0195,  0.0201],
                      [-0.0234,  0.0099,  0.0338,  ..., -0.0203, -0.0133,  0.0212],
                      [ 0.0063, -0.0131,  0.0202,  ..., -0.0164,  0.0221,  0.0011],
                      ...,
                      [-0.0079,  0.0126, -0.0063,  ..., -0.0049, -0.0159,  0.0094],
                      [-0.0264, -0.0351,  0.0151,  ...,  0.0070, -0.0234, -0.0352],
                      [ 0.0140, -0.0059,  0.0071,  ...,  0.0348,  0.0120,  0.0213]])),
             ('linear.bias',
              tensor([-0.0809,  0.1206, -0.0430, -0.0242, -0.0004,  0.0362,  0.0046,  0.0181,
                      -0.0809, -0.0435]))])
```

Just as a sanity check, let's verify that this model has the same loss and accuracy on the test set as before.

In [57]:

```
test_loader = DataLoader(test_dataset, batch_size=256)
result = evaluate(model2, test_loader)
result
```

Out[57]:

```
{'val_loss': 0.5958371162414551, 'val_acc': 0.867382824420929}
```

Commit and upload the notebook

As a final step, we can save and commit our work using the jovian library. Along with the notebook, we can also attach the weights of our trained model, so that we can use it later.

In [ ]:

```
jovian.commit(project='03-logistic-regression', environment=None, outputs=['mnist-logistic.pth'])
```

```
[jovian] Attempting to save notebook..
```

Summary and Further Reading

We've created a fairly sophisticated training and evaluation pipeline in this tutorial. Here's a list of the topics we've covered:

- Working with images in PyTorch (using the MNIST dataset)
- Splitting a dataset into training, validation and test sets
- Creating PyTorch models with custom logic by extending the `nn.Module` class
- Interpreting model ouputs as probabilities using softmax, and picking predicted labels
- Picking a good evaluation metric (accuracy) and loss function (cross entropy) for classification problems
- Setting up a training loop that also evaluates the model using the validation set
- Testing the model manually on randomly picked examples
- Saving and loading model checkpoints to avoid retraining from scratch

There's a lot of scope to experiment here, and I encourage you to use the interactive nature of Jupyter to play around with the various parameters. Here are a few ideas:

- Try making the validation set smaller or larger, and see how it affects the model.
- Try changing the learning rate and see if you can achieve the same accuracy in fewer epochs.
- Try changing the batch size. What happens if you use too high a batch size, or too low?
- Modify the `fit` function to also track the overall loss and accuracy on the training set, and see how it compares with the validation loss/accuracy. Can you explain why it's lower/higher?
- Train with a small subset of the data, and see if you can reach a similar level of accuracy.
- Try building a model for a different dataset, such as the CIFAR10 or CIFAR100 datasets.

Here are some references for further reading:

- For a more mathematical treatment, see the popular Machine Learning course on Coursera. Most of the images used in this tutorial series have been taken from this course.
- The training loop defined in this notebook was inspired from FastAI development notebooks which contain a wealth of other useful stuff if you can read and understand the code.
- For a deep dive into softmax and cross entropy, see this blog post on DeepNotes.

With this we complete our discussion of logistic regression, and we're ready to move on to the next topic: *feedforward neural networks*!