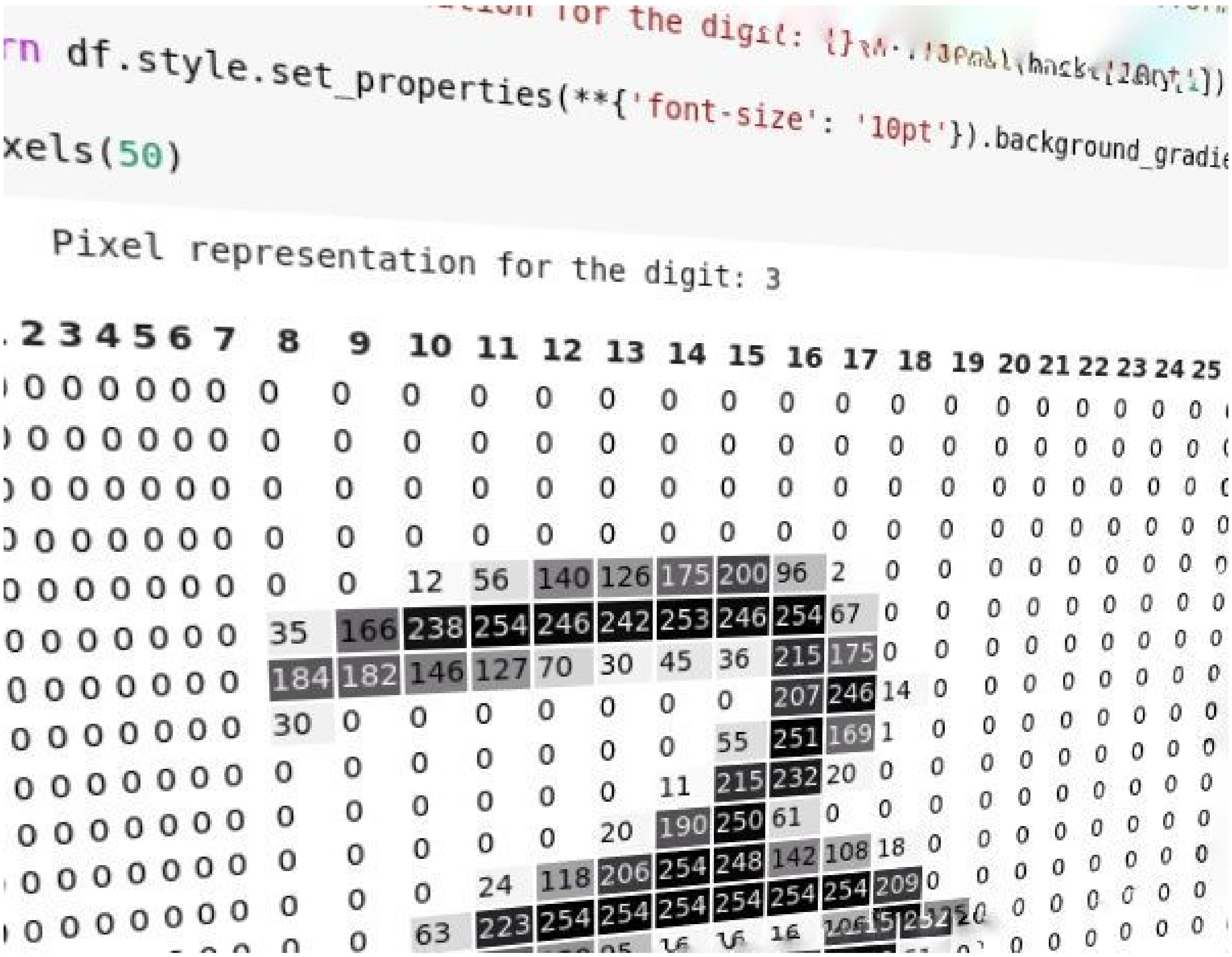
SEPTEMBER 15, 2020 / #NEURAL NETWORKS

How to Build a Neural Network from Scratch with PyTorch

Bipin Krishnan P



```
rn df.style.set_properties(**{'font-size': '10pt'}).background_gradie
xels(50)

   Pixel representation for the digit: 3
```

The one thing that excites me the most in deep learning is tinkering with code to build something from scratch. It's not an easy task, though, and teaching someone else how to do so is even more difficult.

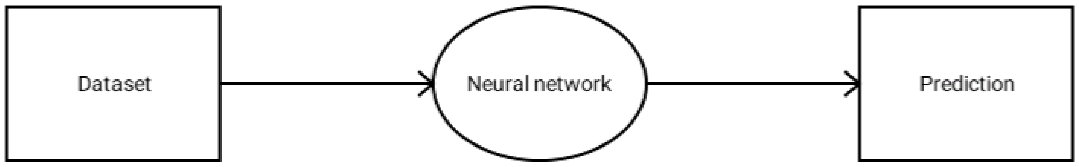
I've been working my way through the Fast.ai course and this blog is greatly inspired by my experience.

Without any further delay let's start our wonderful journey of demystifying neural networks.

How does a neural network work?

Let's start by understanding the high level workings of neural networks.

A neural network takes in a data set and outputs a prediction. It's as simple as that.



How a neural network works

Let's say that one of your friends (who is not a great football fan) points at an old picture of a famous footballer – say Lionel Messi – and asks you about him.

You will be able to identify the footballer in a second. The reason is that you have seen his pictures a thousand times before. So you can identify him even if the picture is old or was taken in dim light.

But what happens if I show you a picture of a famous baseball player (and you have never seen a single baseball game before)? You will not be able to recognize that player. In that case, even if the picture is clear and bright, you won't know who it is.

This is the same principle used for neural networks. If our goal is to build a neural network to recognize cats and dogs, we just show the neural network a bunch of pictures of dogs and cats.

More specifically, we show the neural network pictures of dogs and then tell it that these are dogs. And then show it pictures of cats, and identify those as cats.

Once we train our neural network with images of cats and dogs, it can easily classify whether an image contains a cat or a dog. In short, it can recognize a cat from a dog.

But if you show our neural network a picture of a horse or an eagle, it will never identify it as horse or eagle. This is because it has never

If you wish to improve the capability of the neural network, then all you have to do is show it pictures of all the animals that you want the neural network to classify. As of now, all it knows is cats and dogs and nothing else.

The data set we use for our training heavily depends on the problem on our hands. If you wish to classify whether a tweet has a positive or negative sentiment, then probably, you will want a data set containing a lot of tweets with their corresponding label as either positive or negative.

Now that you have a high-level overview of data sets and how a neural network learns from that data, let's dive deeper into how neural networks work.

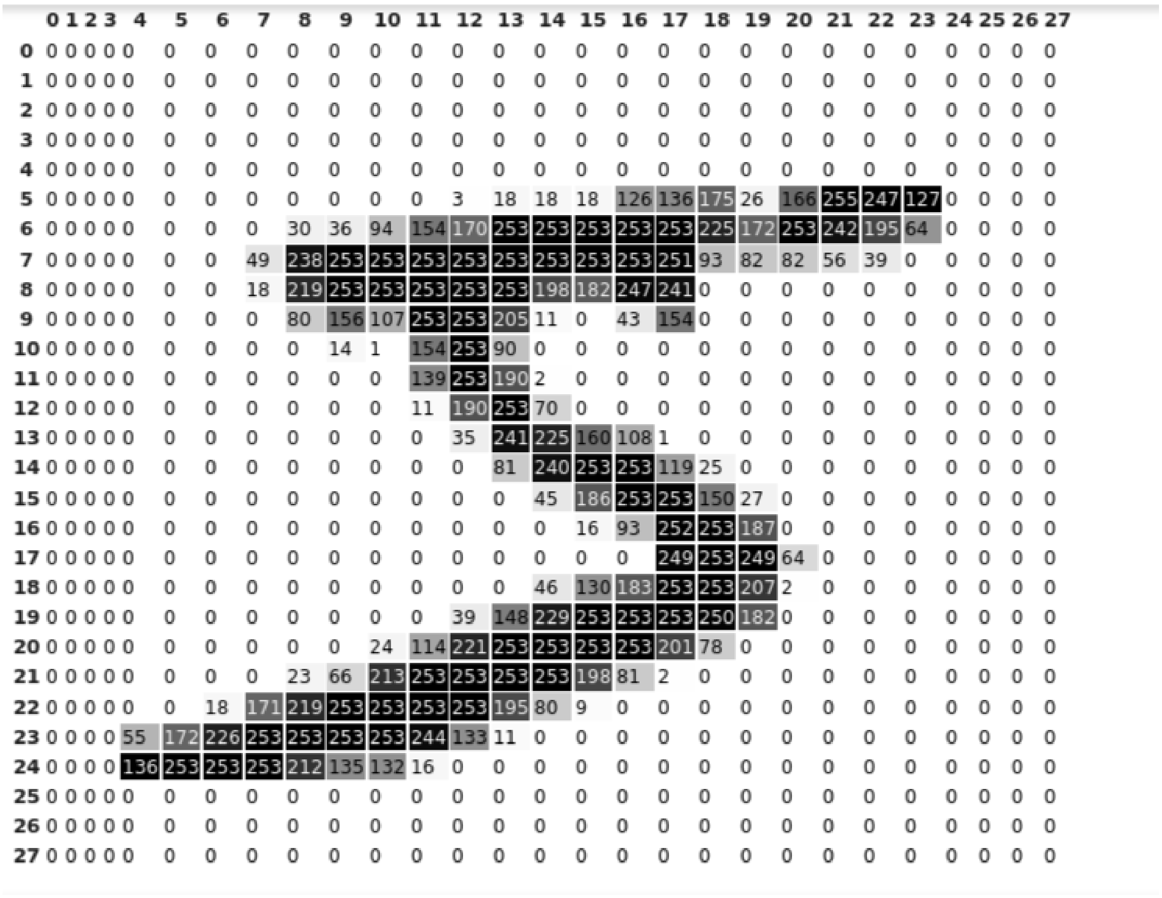
Understanding neural networks

We will be building a neural network to classify the digits three and seven from an image.

But before we build our neural network, we need to go deeper to understand how they work.

Every image that we pass to our neural network is just a bunch of numbers. That is, each of our images has a size of 28×28 which means

Here is a pixel representation of the digit five:



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255	247	127	0	0	0	0
6	0	0	0	0	0	0	0	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242	195	64	0	0	0	0
7	0	0	0	0	0	0	0	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56	39	0	0	0	0	0
8	0	0	0	0	0	0	0	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	14	1	154	253	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	139	253	190	2	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	11	190	253	70	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	35	241	225	160	108	1	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	93	252	253	187	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	249	253	249	64	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	18	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	55	172	226	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	136	253	253	253	212	135	132	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Pixel values along with shades

As you can see above, we have 28 rows and 28 columns (the index starts from 0 and ends at 27) just like a matrix. Neural networks only see these 28×28 matrices.

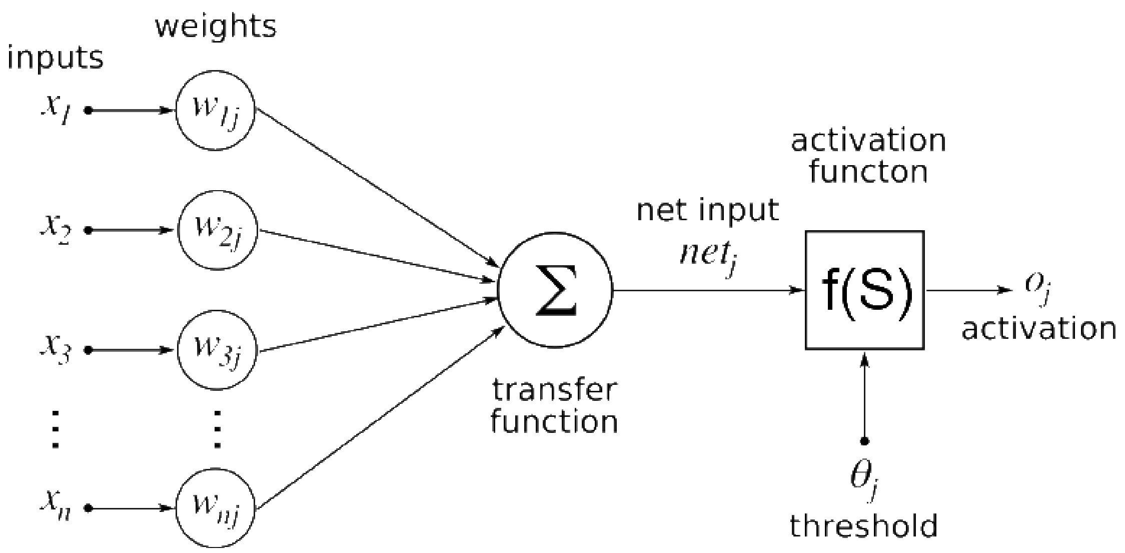
pixel values close to 255 are darker whereas the values closer to 0 are lighter in shade.

In PyTorch we don't use the term matrix. Instead, we use the term tensor. Every number in PyTorch is represented as a tensor. So, from now on, we will use the term tensor instead of matrix.

Visualizing a neural network

A neural network can have any number of neurons and layers.

This is how a neural network looks:



Artificial neural network

down for you:

Take the case of predicting whether a patient will survive or not based on a data set containing the name of the patient, temperature, blood pressure, heart condition, monthly salary, and age.

In our data set, only the temperature, blood pressure, heart condition, and age have significant importance for predicting whether the patient will survive or not. So we will assign a higher weight value to these values in order to show higher importance.

But features like the name of the patient and monthly salary have little or no influence on the patient's survival rate. So we assign smaller weight values to these features to show less importance.

In the above figure, x1, x2, x3...xn are the features in our data set which may be pixel values in the case of image data or features like blood pressure or heart condition as in the above example.

The feature values are multiplied by the corresponding weight values referred to as w1j, w2j, w3j...wnj. The multiplied values are summed together and passed to the next layer.

The optimum weight values are learned during the training of the neural network. The weight values are updated continuously in such a way as to maximize the number of correct predictions.

The activation function is nothing but the sigmoid function in our case.

You will understand the importance of the sigmoid layer once we start building our neural network model.

There are a lot of other activation functions that are even simpler to learn than sigmoid.

This is the equation for a sigmoid function:

$$y(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid function

The circular-shaped nodes in the diagram are called neurons. At each layer of the neural network, the weights are multiplied with the input data.

the number of neurons in that layer.

Understanding our data set

The first thing we need in order to train our neural network is the data set.

Since the goal of our neural network is to classify whether an image contains the number three or seven, we need to train our neural network with images of threes and sevens. So, let's build our data set.

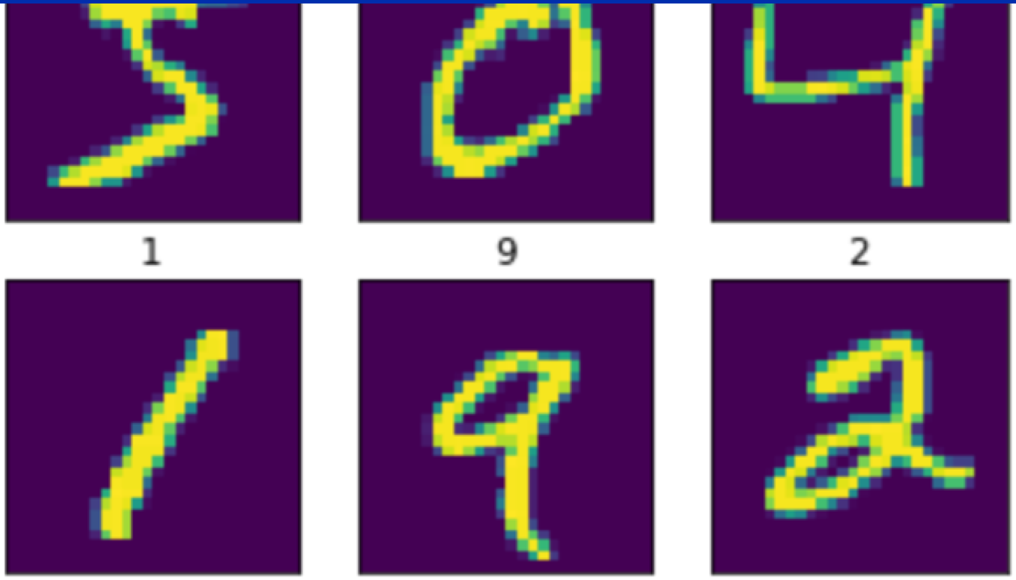
Luckily, we don't have to create the data set from scratch. Our data set is already present in PyTorch. All we have to do is just download it and do some basic operations on it.

We need to download a data set called **MNIST** (Modified National Institute of Standards and Technology) from the torchvision library of PyTorch.

Now let's dig deeper into our data set.

What is the MNIST data set?

The MNIST data set contains handwritten digits from zero to nine with their corresponding labels as shown below:



MNIST data set

So, what we do is simply feed the neural network the images of the digits and their corresponding labels which tell the neural network that this is a three or seven.

How to prepare our data set

The downloaded MNIST data set has images and their corresponding labels.

We just write the code to index out only the images with a label of three or seven. Thus, we get a data set of threes and sevens.

First, let's import all the necessary libraries.

```
import torch
```

Import the libraries

We import the PyTorch library for building our neural network and the torchvision library for downloading the MNIST data set, as discussed before. The Matplotlib library is used for displaying images from our data set.

Now, let's prepare our data set.

```
mnist = datasets.MNIST('./data', download=True)

threes = mnist.data[(mnist.targets == 3)]/255.0
sevens = mnist.data[(mnist.targets == 7)]/255.0

len(threes), len(sevens)
```

Preparing the data set

As we learned above, everything in PyTorch is represented as tensors. So our data set is also in the form of tensors.

We download the data set in the first line. We index out only the images whose target value is equal to 3 or 7 and normalize them by dividing with 255 and store them separately.

We can check whether our indexing was done properly by running the

```python
def show_image(img):
  plt.imshow(img)
  plt.xticks([])
  plt.yticks([])
  plt.show()

show_image(threes[3])
show_image(sevens[8])
```

Check the indexed images

Using the Matplotlib library, we create a function to display the images.

Let's do a quick sanity check by printing the shape of our tensors.

```python
print(threes.shape, sevens.shape)
```

Check the size of the tensors

If everything went right, you will get the size of threes and sevens as ([6131, 28, 28]) and ([6265, 28, 28]) respectively. This means that we have 6131 28×28 sized images for threes and 6265 28×28 sized images for sevens.

network.

```
combined_data = torch.cat([threes, sevens])
combined_data.shape
```

Concatenate the tensors

We will concatenate the two tensors using PyTorch and check the shape of the combined data set.

Now we will flatten the images in the data set.

```
flat_imgs = combined_data.view((-1, 28*28))
flat_imgs.shape
```

Flatten the images

We will flatten the images in such a way that each of the 28×28 sized images becomes a single row with 784 columns (28×28=784). Thus the shape gets converted to ([12396, 784]).

We need to create labels corresponding to the images in the combined data set.

```
target = torch.tensor([1]*len(threes)+[2]*len(sevens))
```

We assign the label 1 for images containing a three, and the label 0 for images containing a seven.

How to train your Neural Network

To train your neural network, follow these steps.

Step 1: Building the model

Below you can see the simplest equation that shows how neural networks work:

$$y = Wx + b$$

Here, the term 'y' refers to our prediction, that is, three or seven. 'W' refers to our weight values, 'x' refers to our input image, and 'b' is the bias (which, along with weights, help in making predictions).

In short, we multiply each pixel value with the weight values and add them to the bias value.

The weights and bias value decide the importance of each pixel value while making predictions.

We are classifying three and seven, so we have only two classes to predict.

and 1.

This allows us to create a threshold of 0.5. That is, if the predicted value is less than 0.5 then it is a seven. Otherwise it is a three.

We use a sigmoid function to get a value between 0 and 1.

We will create a function for sigmoid using the same equation shown earlier. Then we pass in the values from the neural network into the sigmoid.

We will create a single layer neural network.

We cannot create a lot of loops to multiply each weight value with each pixel in the image, as it is very expensive. So we can use a magic trick to do the whole multiplication in one go by using matrix multiplication.

```
def sigmoid(x): return 1/(1+torch.exp(-x))

def simple_nn(data, weights, bias): return sigmoid((data@weights) + bias)
```

Defining the neural network

value is different from that of the ground truth.

For example, if the predicted value is 0.3 but the ground truth is 1, then our loss is very high. So our model will try to reduce this loss by updating the weights and bias so that our predictions become close to the ground truth.

We will be using mean squared error to check the loss value. Mean squared error finds the mean of the square of the difference between the predicted value and the ground truth.

```
def error(pred, target): return ((pred-target)**2).mean()
```

Defining the loss

Step 3: Initialize the weight values

We just randomly initialize the weights and bias. Later, we will see how these values are updated to get the best predictions.

```
w = torch.randn((flat_imgs.shape[1], 1), requires_grad=True)
b = torch.randn((1, 1), requires_grad=True)
```

Initialize the parameters

(Number of neurons in the previous layer, number of neurons in the next layer)

We use a method called gradient descent to update our weights and bias to make the maximum number of correct predictions.

Our goal is to optimize or decrease our loss, so the best method is to calculate gradients.

We need to take the derivative of each and every weight and bias with respect to the loss function. Then we have to subtract this value from our weights and bias.

In this way, our weights and bias values are updated in such a way that our model makes a good prediction.

Updating a parameter for optimizing a function is not a new thing – you can optimize any arbitrary function using gradients.

We've set a special parameter (called requires_grad) to true to calculate the gradient of weights and bias.

Step 4: Update the weights

If our prediction does not come close to the ground truth, that means that we've made an incorrect prediction. This means that our weights are not correct. So we need to update our weights until we get good predictions.

At each iteration, the loss is calculated and the weights and biases are updated to get a better prediction on the next iteration.

Thus our model becomes better after each iteration by finding the optimal weight value suitable for our task in hand.

Each task requires a different set of weight values, so we can't expect our neural network trained for classifying animals to perform well on musical instrument classification.

This is how our model training looks like:

```
for i in range(2000):
  pred = simple_nn(flat_imgs, w, b)
  loss = error(pred, target.unsqueeze(1))
  loss.backward()

  w.data -= 0.001*w.grad.data
  b.data -= 0.001*b.grad.data

  w.grad.zero_()
  b.grad.zero_()

print("Loss: ", loss.item())
```

Training the model

We will calculate the predictions and store it in the 'pred' variable by calling the function that we've created earlier. Then we calculate the mean squared error loss.

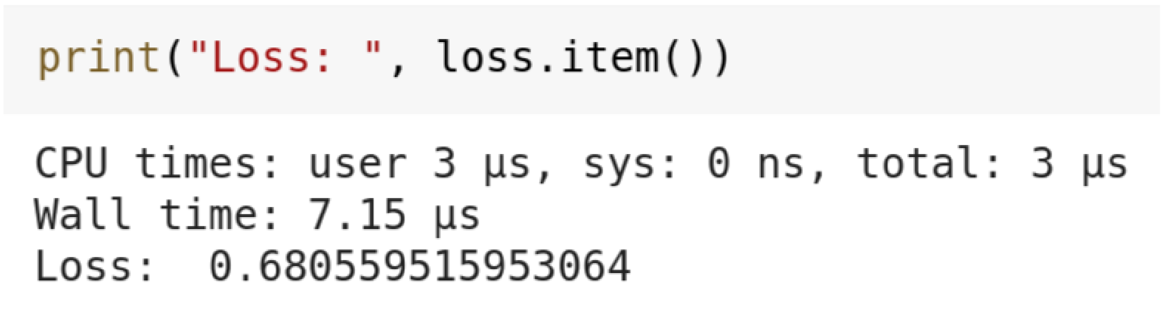
We've multiplied the gradients by 0.001, and this is called learning rate. This value decides the rate at which our model will learn, if it is too low, then the model will learn slowly, or in other words, the loss will be reduced slowly.

If the learning rate is too high, our model will not be stable, jumping between a wide range of loss values. This means it will fail to converge.

We do the above steps for 2000 times, and each time our model tries to reduce the loss by updating the weights and bias values.

We should zero out the gradients at the end of each loop or epoch so that there is no accumulation of unwanted gradients in the memory which will affect our model's learning.

Since our model is very small, it doesn't take much time to train for 2000 epochs or iterations. After 2000 epochs, our neural netwok has given a loss value of 0.6805 which is not bad from such a small model.



```
print("Loss: ", loss.item())
```

```
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 7.15 µs
Loss:  0.6805591595306
```

Final result

Conclusion

There is a huge space for improvement in the model that we've just created.

This is just a simple model, and you can experiment on it by increasing the number of layers, number of neurons in each layer, or increasing the number of epochs.

In short, machine learning is a whole lot of magic using math. Always learn the foundational concepts – they may be boring, but eventually you will understand that those boring math concepts created these cutting edge technologies like deepfakes.

You can get the complete code on GitHub or play with the code in Google colab.

Bipin Krishnan P
Mechanical engineering undergrad with a drag to machine learning stuff. Connect with me on LinkedIn: https://linkedin.com/in/bipin-krishnan

Trending Guides

What is Docker?	What is STEM?
TCP/IP Model	JavaScript Void 0
RTF File	SQL Delete Row
CSS Transition	JavaScript Replace
How to Use Instagram?	Python JSON Parser
MBR VS GPT	cmd Delete Folder
FAT32 Format	What is NFC?
Error 503 Code	Content Type JSON
Windows Hosts File	Convert HEIC to JPG