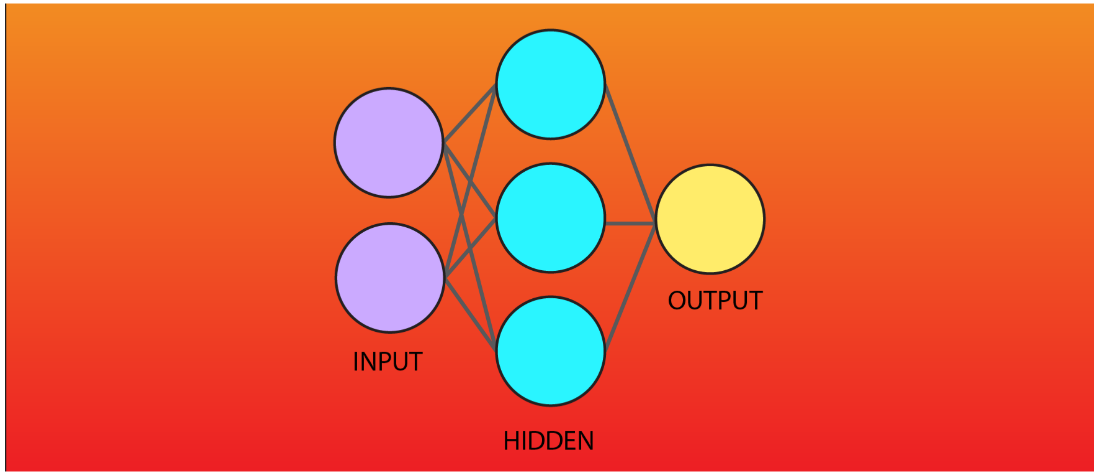
A Simple Neural Network from Scratch with PyTorch and Google Colab

Elvis Follow
Aug 14, 2018 · 6 min read



In this tutorial we will implement a simple neural network from scratch using PyTorch and Google Colab. The idea is to teach you the basics of PyTorch and how it can be used to implement a neural network from scratch. I will go over some of the basic functionalities available in PyTorch that will allow you to build your customized neural networks. I have implemented this on Google Colab already so you can take a quick peek at the result here.

This tutorial assumes you have prior knowledge of how a neural network works. Don't worry! Even if you are not so sure, you will be okay. For advanced PyTorch users this tutorial may still serve as a refresher. This tutorial is inspired by this Neural Network implementation coded purely using Numpy.

Since we are working on Google Colab, we will need to install the PyTorch library. You can do this by using the following command:

```
!pip3 install torch torchvision
```

py_install.sh hosted with ♡ by GitHub view raw

The `torch` module provides all the necessary **Tensor** operators you will need to implement your first neural network from scratch in PyTorch. That's right! In PyTorch everything is a Tensor, so this is the first thing you will need to get used to.

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

py_import.py hosted with ♡ by GitHub view raw

Data

Let's start by creating some sample data using the `torch.tensor` command. In Numpy, this could be done with `np.array`. Both functions serve the same purpose, but in PyTorch everything is a Tensor as opposed to a vector or matrix. We define types in PyTorch using the `dtype=torch.xxx` command.

In the data below, `X` represents the amount of hours studied and how much time students spent sleeping, whereas `y` represent grades. The variable `xPredicted` is a single input for which we want to predict a grade using the parameters learned by the neural network. Remember, the neural network wants to learn a mapping between `X` and `y`, so it will try to take a guess from what it has learned from the training data.

```python
X = torch.tensor(([2, 9], [1, 5], [3, 6]), dtype=torch.float) # 3 X 2 tensor
y = torch.tensor(([92], [100], [89]), dtype=torch.float) # 3 X 1 tensor
xPredicted = torch.tensor(([4, 8]), dtype=torch.float) # 1 X 2 tensor
```

py_data.py hosted with ♡ by GitHub view raw

You can check the size of the tensors we have just created with the `size` command. This is equivalent to the `shape` command used in tools such as Numpy and Tensorflow.

```python
print(X.size())
print(y.size())
```

py_size.py hosted with ♡ by GitHub view raw

The output:

> *torch.Size([3, 2])*
>
> *torch.Size([3, 1])*

Scaling

Below we are performing some scaling on the sample data. Notice that the `max` function returns both a tensor and the corresponding indices. So we use `_` to capture the indices which we won't use here because we are only interested in the max values to conduct the scaling. Perfect! Our data is now in a very nice format our neural network will appreciate later on.

```python
# scale units
X_max, _ = torch.max(X, 0)
xPredicted_max, _ = torch.max(xPredicted, 0)

X = torch.div(X, X_max)
xPredicted = torch.div(xPredicted, xPredicted_max)
y = y / 100  # max test score is 100
```

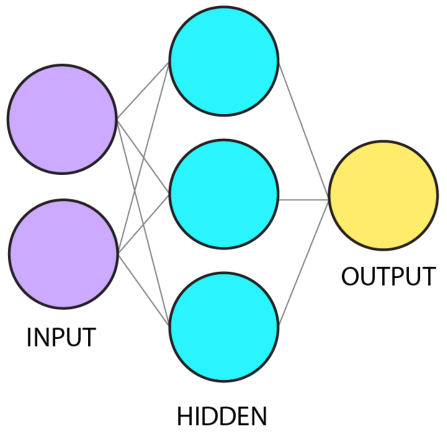
py_scaling.py hosted with ♡ by GitHub view raw

Notice that there are two functions `max` and `div` that I didn't discuss above. They do exactly what they imply: `max` finds the maximum value in a vector... I mean tensor; and `div` is basically a nice little function to divide two tensors.

Model (Computation Graph)

Once the data has been processed and it is in the proper format, all you need to do now is to define your model. Here is where things begin to change a little as compared to how you would build your neural networks using, say, something like Keras or Tensorflow. However, you will realize quickly as you go along that PyTorch doesn't differ much from other deep learning tools. At the end of the day, we are constructing a computation graph, which is used to dictate how data should flow and what type of operations are performed on this information.

For illustration purposes, we are building the following neural network or computation graph:



```python
class Neural_Network(nn.Module):
    def __init__(self, ):
        super(Neural_Network, self).__init__()
        # parameters
        # TODO: parameters can be parameterized instead of declaring them here
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3

        # weights
        self.W1 = torch.randn(self.inputSize, self.hiddenSize) # 3 X 2 tensor
        self.W2 = torch.randn(self.hiddenSize, self.outputSize) # 3 X 1 tensor

    def forward(self, X):
        self.z = torch.matmul(X, self.W1) # 3 X 3 ".dot" does not broadcast in PyTorch
        self.z2 = self.sigmoid(self.z) # activation function
        self.z3 = torch.matmul(self.z2, self.W2)
        o = self.sigmoid(self.z3) # final activation function
        return o

    def sigmoid(self, s):
        return 1 / (1 + torch.exp(-s))

    def sigmoidPrime(self, s):
        # derivative of sigmoid
        return s * (1 - s)

    def backward(self, X, y, o):
        self.o_error = y - o # error in output
        self.o_delta = self.o_error * self.sigmoidPrime(o) # derivative of sig to error
        self.z2_error = torch.matmul(self.o_delta, torch.t(self.W2))
        self.z2_delta = self.z2_error * self.sigmoidPrime(self.z2)
        self.W1 += torch.matmul(torch.t(X), self.z2_delta)
        self.W2 += torch.matmul(torch.t(self.z2), self.o_delta)

    def train(self, X, y):
        # forward + backward pass for training
        o = self.forward(X)
        self.backward(X, y, o)

    def saveWeights(self, model):
        # we will use the PyTorch internal storage functions
        torch.save(model, "NN")
        # you can reload model with all the weights and so forth with:
        # torch.load("NN")

    def predict(self):
        print ("Predicted data based on trained weights: ")
        print ("Input (scaled): \n" + str(xPredicted))
        print ("Output: \n" + str(self.forward(xPredicted)))
```

For the purpose of this tutorial, we are not going to be talking math stuff, that's for another day. I just want you to get a gist of what it takes to build a neural network from scratch using PyTorch. Let's break down the model which was declared via the class above.

Class Header

First, we defined our model via a class because that is the recommended way to build the computation graph. The class header contains the name of the class `Neural Network` and the parameter `nn.Module` which basically indicates that we are defining a customized neural network.

```
class Neural_Network(nn.Module):
```

Initialization

```
def __init__(self, ):
    super(Neural_Network, self).__init__()
    # parameters
    # TODO: parameters can be parameterized instead of declaring them
here
    self.inputSize = 2
```

```
        self.outputSize = 1
        self.hiddenSize = 3

    # weights
        self.W1 = torch.randn(self.inputSize, self.hiddenSize)
        self.W2 = torch.randn(self.hiddenSize, self.outputSize)
```

The next step is to define the initializations (`def __init__(self,)`) that will be performed upon creating an instance of the customized neural network. You can declare the parameters of your model here, but typically, you would declare the structure of your network in this section — the size of the hidden layers and so forth. Since we are building the neural network from scratch, we explicitly declared the size of the weights matrices: one that stores the parameters from the input to hidden layer; and one that stores the parameter from the hidden to output layer. Both weight matrices are initialized with values randomly chosen from a normal distribution via `torch.randn(...)` . Note that we are not using bias just to keep things as simple as possible.

```
    def forward(self, X):
        self.z = torch.matmul(X, self.W1)
        self.z2 = self.sigmoid(self.z) # activation function
        self.z3 = torch.matmul(self.z2, self.W2)
        o = self.sigmoid(self.z3) # final activation function
        return o
```

The `forward` function shown above is where all the magic happens (see below). This is where the data enters and is fed into the computation graph (i.e., the neural network structure we have built). Since we are building a simple neural network with one hidden layer, our forward function looks very simple.

The `forward` function above takes the input `X` and then performs a matrix multiplication (`torch.matmul(...)`) with the first weight matrix `self.W1` . Then the result is applied an activation function, `sigmoid` . The resulting matrix of the activation is then multiplied with the second weight matrix `self.W2` . Then another activation if performed, which renders the output of the neural network or computation graph. The process I described above is simply what's known as a `forward pass` . In order for the weights to optimize when training, we need a `backward pass` as well.

The Backward Function (Backpropagation)

The `backward` function contains the backpropagation algorithm, where the goal is to essentially minimize the loss with respect to our weights. In other words, the weights need to be updated in such a way that the loss decreases while the neural network is training (well, that is what we hope for). All this magic is possible with the gradient descent algorithm which is declared in the `backward` function. Take a minute or two to inspect what is happening in the code below:

```
def backward(self, X, y, o):
    self.o_error = y - o # error in output
    self.o_delta = self.o_error * self.sigmoidPrime(o)
    self.z2_error = torch.matmul(self.o_delta, torch.t(self.W2))
    self.z2_delta = self.z2_error * self.sigmoidPrime(self.z2)
    self.W1 += torch.matmul(torch.t(X), self.z2_delta)
    self.W2 += torch.matmul(torch.t(self.z2), self.o_delta)
```

Notice that we are performing a lot of matrix multiplications along with the transpose operations via the `torch.matmul(...)` and `torch.t(...)` operations, respectively. The rest is simply gradient descent — there is nothing to it.

Training

All that is left now is to train the neural network. First we create an instance of the computation graph we have just built:

```
NN = Neural_Network()
```

Then we train the model for `1000` rounds. Notice that in PyTorch `NN(X)` automatically calls the `forward` function so there is no need to explicitly call `NN.forward(X)`.

After we have obtained the predicted output for ever round of training, we compute the loss, with the following code:

```
torch.mean((y - NN(X))**2).detach().item()
```

The next step is to start the training (foward + backward) via `NN.train(X, y)`. After we have trained the neural network, we can store the model and output the predicted value of the single instance we declared in the beginning, `xPredicted`.

Let’s train!

```python
NN = Neural_Network()
for i in range(1000):  # trains the NN 1,000 times
    print ("#" + str(i) + " Loss: " + str(torch.mean((y - NN(X))**2).detach().item()))  # mean su
    NN.train(X, y)
NN.saveWeights(NN)
NN.predict()
```

py_train.py hosted with ♡ by GitHub view raw

Below is a snippet of the loss in the final rounds of training:

> *#997 Loss: 0.0016838625306263566*
>
> *#998 Loss: 0.0016829235246405005*
>
> *#999 Loss: 0.0016819849843159318*

The loss keeps decreasing, which means that the neural network is learning something. That’s it. Congratulations! You have just learned how to create and train a neural network from scratch using PyTorch. There are so many things you can do with the shallow network we have just implemented. You can add more hidden layers or try to incorporate the bias terms for practice. I would love to see what you will build from here. Reach me out on Twitter if you have any further questions or leave your comments here. Until next time!

Machine Learning Artificial Intelligence Data Science Technology Innovation