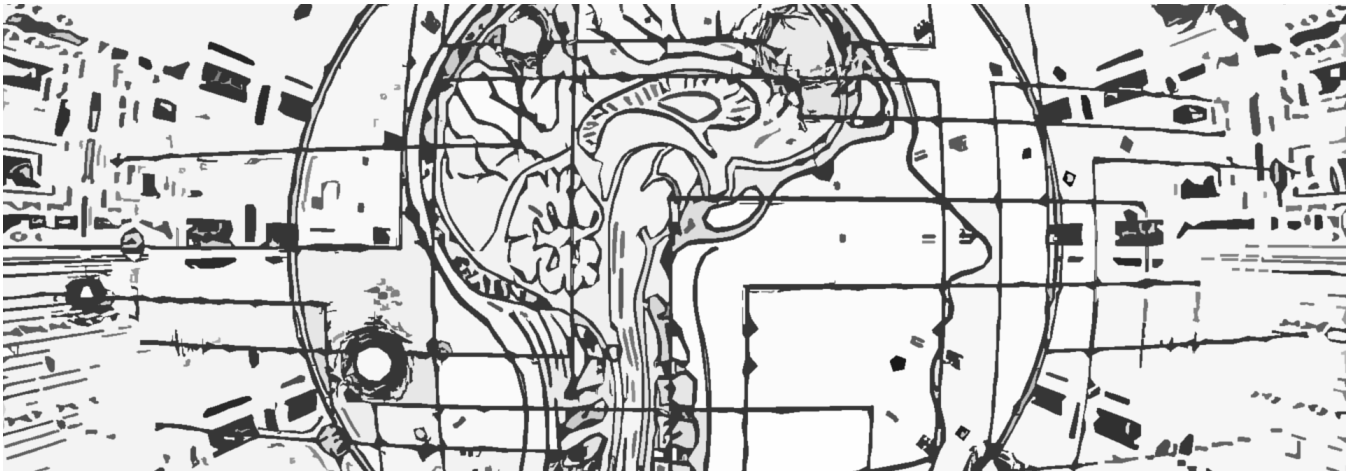
Building Neural Network Using PyTorch

Tasnuva Zaman
Jul 16, 2019 · 5 min read ★

“The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.”
— **Edsger W. Dijkstra**



source: here

In this tutorial we will implement a simple neural network from scratch using PyTorch. I am sharing what I have learnt from my recent facebook-udacity scholarship challenge program. This tutorial assumes you have prior knowledge of **how a neural network works**.

Though there are many libraries out there that can be used for deep learning I like the PyTorch most. As a python programmer, one of the reasons behind my liking is pythonic behavior of PyTorch. It mostly uses the style and power of python which is easy to understand and use.

At its core, PyTorch provides two main features:

- An n-dimensional Tensor, similar to numpy but can run on GPUs
- Automatic differentiation for building and training neural networks

What is Neural Network?

Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. The networks are built from individual parts approximating neurons, typically called units or simply "**neurons**." Each unit has some number of weighted inputs. These weighted inputs are summed together (a linear combination) then passed through an activation function to get the unit's output.

Types of Nodes in a Neural Network:

1. Input Units — Provides information from the outside world to the network and are together referred to as the "Input Layer". These nodes do not perform any computation, they just pass on the information to the hidden nodes.

2. Hidden Units — These nodes do not have any direct connection with the outside world. They perform computations and transfer information from Input nodes to Output nodes. A collection of hidden nodes forms a "Hidden Layer". While a feed-forward network will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers.

3. Output Units — The Output nodes are collectively referred to as the "Output Layer" and are responsible for computations and transferring information from the network to the outside world.

Each layer comprises one or more nodes.

Building Neural Network

PyTorch provides a module `nn` that makes building networks much simpler. We'll see how to build a neural network with `784 inputs`, `256 hidden units`, `10 output units` and a `softmax output`.

```
from torch import nn

class Network(nn.Module):
    def __init__(self):
        super().__init__()

        # Inputs to hidden layer linear transformation
        self.hidden = nn.Linear(784, 256)
        # Output layer, 10 units - one for each digit
```

```
        self.output = nn.Linear(256, 10)

        # Define sigmoid activation and softmax output
        self.sigmoid = nn.Sigmoid()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # Pass the input tensor through each of our operations
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.softmax(x)

        return x
```

> *Note: The* ***`softmax` function,*** *also known as* ***`softargmax`*** *or* ***`normalized exponential function`*** *is a function that takes as input a vector of* K *real numbers, and normalizes it into a probability distribution consisting of* K *probabilities.*

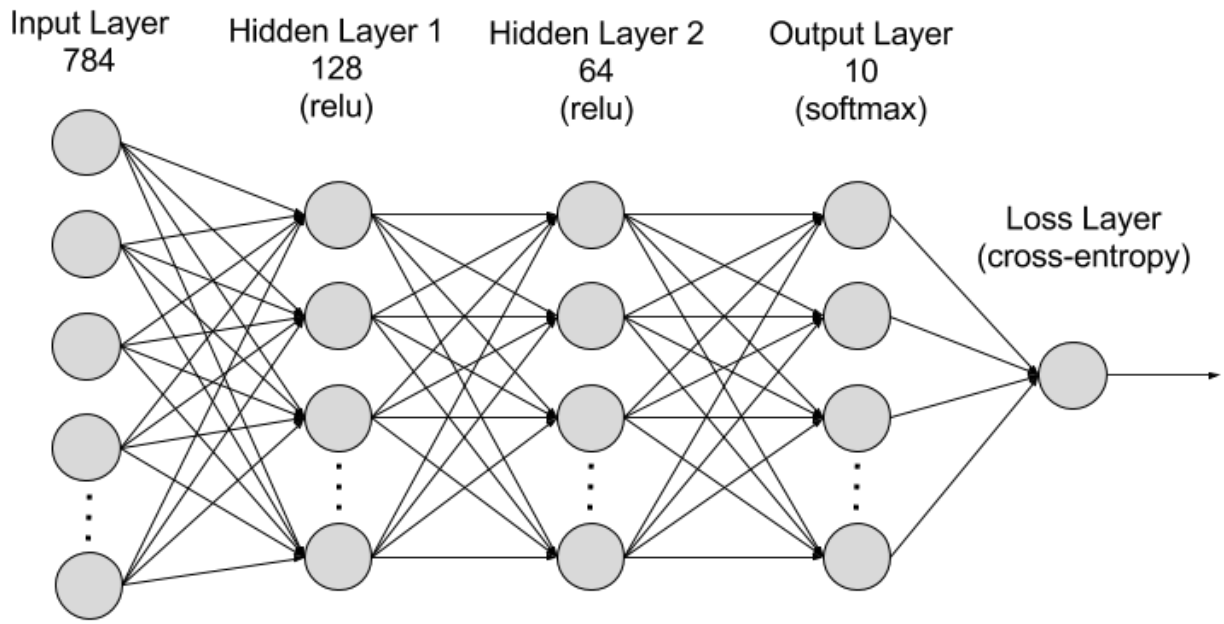


image from google

Let's go through this line by line.

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

Here we're inheriting from `nn.Module`. Combined with `super().__init__()` this creates a class that tracks the architecture and provides a lot of useful methods and attributes. It is

mandatory to inherit from `nn.Module` when you're creating a class for your network. The name of the class itself can be anything.

```
self.hidden = nn.Linear(784, 256)
```

This line creates a module for a linear transformation, $x\mathbf{W}+b$xW+b, with 784 inputs and 256 outputs and assigns it to `self.hidden`. The module automatically creates the weight and bias tensors which we'll use in the `forward` method. You can access the weight and bias tensors once the network (`net`) is created with `net.hidden.weight` and `net.hidden.bias`.

```
self.output = nn.Linear(256, 10)
```

Similarly, this creates another linear transformation with 256 inputs and 10 outputs.

```
self.sigmoid = nn.Sigmoid()
self.softmax = nn.Softmax(dim=1)
```

Here I defined operations for the sigmoid activation and softmax output.
Setting `dim=1` in `nn.Softmax(dim=1)` calculates softmax across the columns.

```
def forward(self, x):
```

PyTorch networks created with `nn.Module` must have a `forward` method defined. It takes in a tensor `x` and passes it through the operations you defined in the `__init__` method.

```
x = self.hidden(x)
x = self.sigmoid(x)
x = self.output(x)
x = self.softmax(x)
```

Here the input tensor `x` is passed through each operation and reassigned to `x`. We can see that the input tensor goes through the hidden layer, then a sigmoid function, then the output

layer, and finally the softmax function. It doesn't matter what you name the variables here, as long as the inputs and outputs of the operations match the network architecture you want to build. The order in which you define things in the `__init__` method doesn't matter, but you'll need to sequence the operations correctly in the `forward` method.

```
# Create the network and look at it's text representation
model = Network()
model
```

Building Neural Network using `nn.Sequential`

PyTorch provides a convenient way to build networks like this where a tensor is passed sequentially through operations, `nn.Sequential` (documentation). Using this to build the equivalent network:

```
# Hyperparameters for our network
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

# Build a feed-forward network
model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1], output_size),
                      nn.Softmax(dim=1))
print(model)
```

> *Here our model is the same as before:* `784 input units`, `a hidden layer with 128 units`, `ReLU activation`, `64 unit hidden layer`, *another* `ReLU`, *then the* `output layer with 10 units`, *and the* `softmax output`.

You can also pass in an `OrderedDict` to name the individual layers and operations, instead of using incremental integers. Note that dictionary keys must be unique, so *each operation must have a different name*.

```
from collections import OrderedDict
model = nn.Sequential(OrderedDict([
                      ('fc1', nn.Linear(input_size,
hidden_sizes[0])),
                      ('relu1', nn.ReLU()),
                      ('fc2', nn.Linear(hidden_sizes[0],
hidden_sizes[1])),
                      ('relu2', nn.ReLU()),
                      ('output', nn.Linear(hidden_sizes[1],
output_size)),
                      ('softmax', nn.Softmax(dim=1))]))


model
```

Now you can access layers either by integer or the name

```
print(model[0])
print(model.fc1)
```

That’s all for today. Next we will be training a neural network. You will find it here.

You are always welcome with any constructive criticism or feedback.