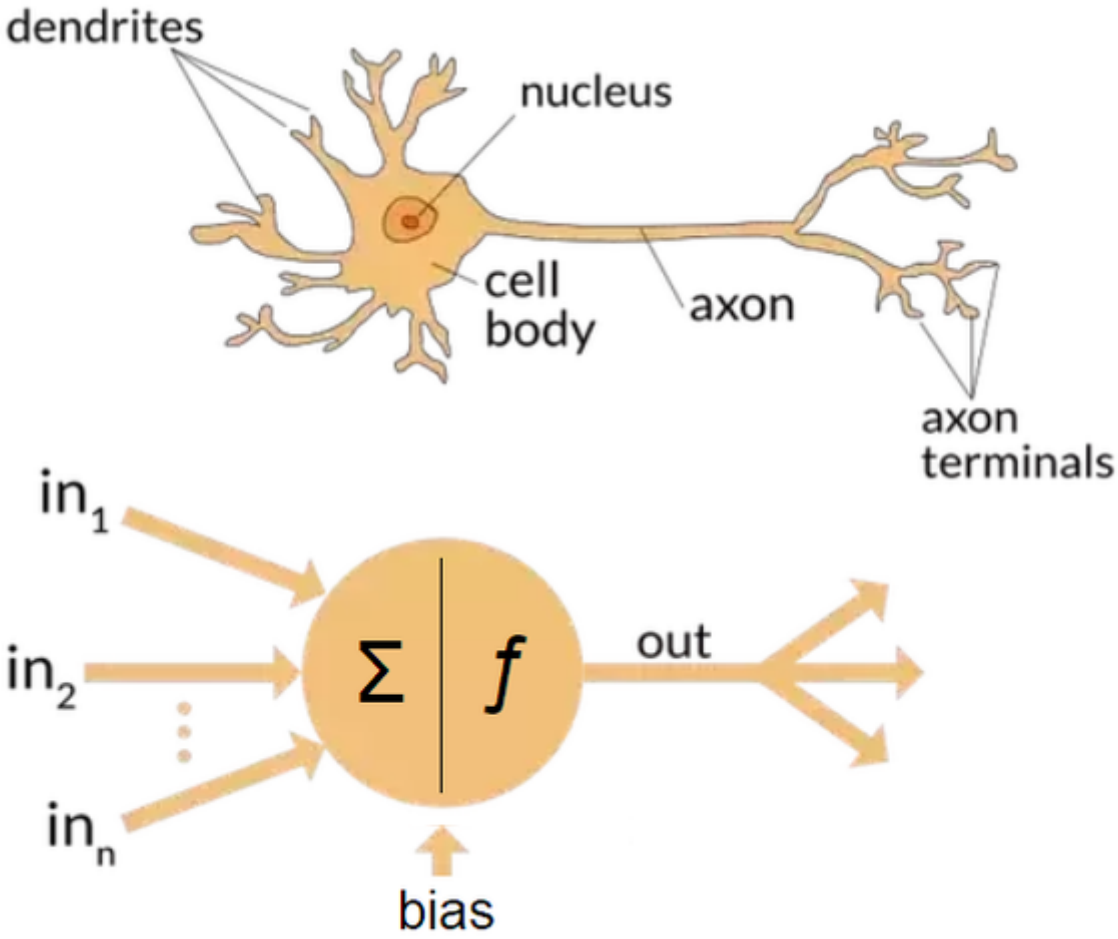
Training Neural Network using PyTorch

Tasnuva Zaman
Aug 6, 2019 · 6 min read ★

> “A little learning is a dangerous thing; drink deep or taste not Pierian Spring” (Alexander Pope)



Human brain vs Neural network (image source here)

So in the previous article we’ve build a very simple and “naive”neural network which doesn’t know the function mapping the inputs to the outputs. To make it more intelligent, we will train the network by showing it the example of ‘real data’ and then adjusting the network parameters(weight and bias). In short, We increase the accuracy by iterating over a training data set while tweaking the parameters(the weights and biases) of our model.

To find these parameters we need to know how poorly our network is predicting the real outputs. For this we will calculate the `cost` which also called the `loss function` .

Cost

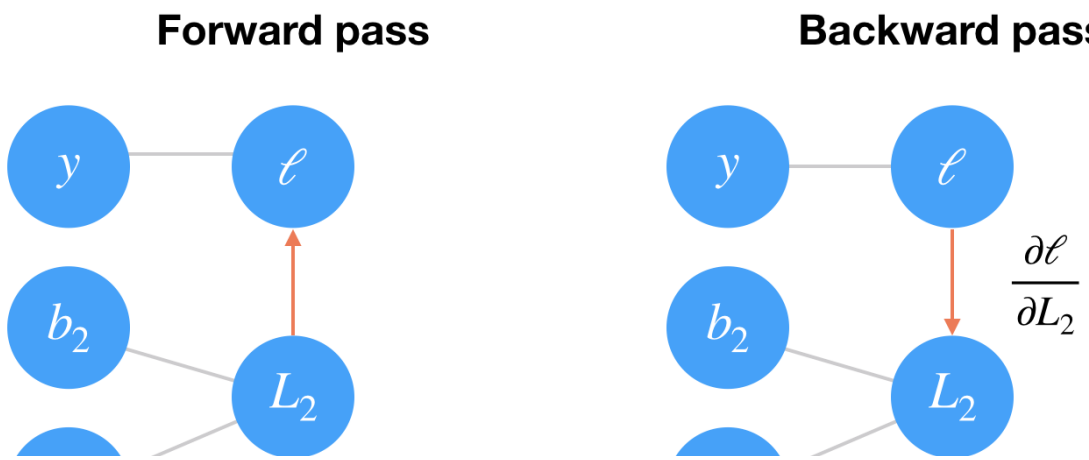
`Cost` or `loss function` is the measure of our prediction error. By minimizing the `loss` with respect to the network parameters, we can find a state where the `loss` is at a minimum and the network is able to predict the correct labels at a high accuracy. We find this `minimum loss` using a process called `gradient descent` . Check different kinds of `cost` function here

Gradient Descent

Gradient Descent requires a cost function. We need this `cost` function because we need to minimize this in order to acquire high prediction accuracy. The whole point of GD is to minimize the `cost` function. The aim of the algorithm is the process of getting to the lowest `error value` . To get the lowest `error value` in the cost function(with respect to one weight) we need to tweak the parameters of our model. So, how much do we need to tweak the parameters? We can find it using `calculus` . Using `calculus` we know that the `slope` of a function is the `derivative` of the function with respect to the value. The `gradient` is the slope of the loss function and points in the direction of fastest change.

Backpropagation

`Gradient Descent` is straightforward to implement for single layer network but for multi-layer network it is more complicated and deeper. Training multilayer networks is done through **backpropagation** which is really just an application of the chain rule from calculus. It's easiest to understand if we convert a two layer network into a graph representation.



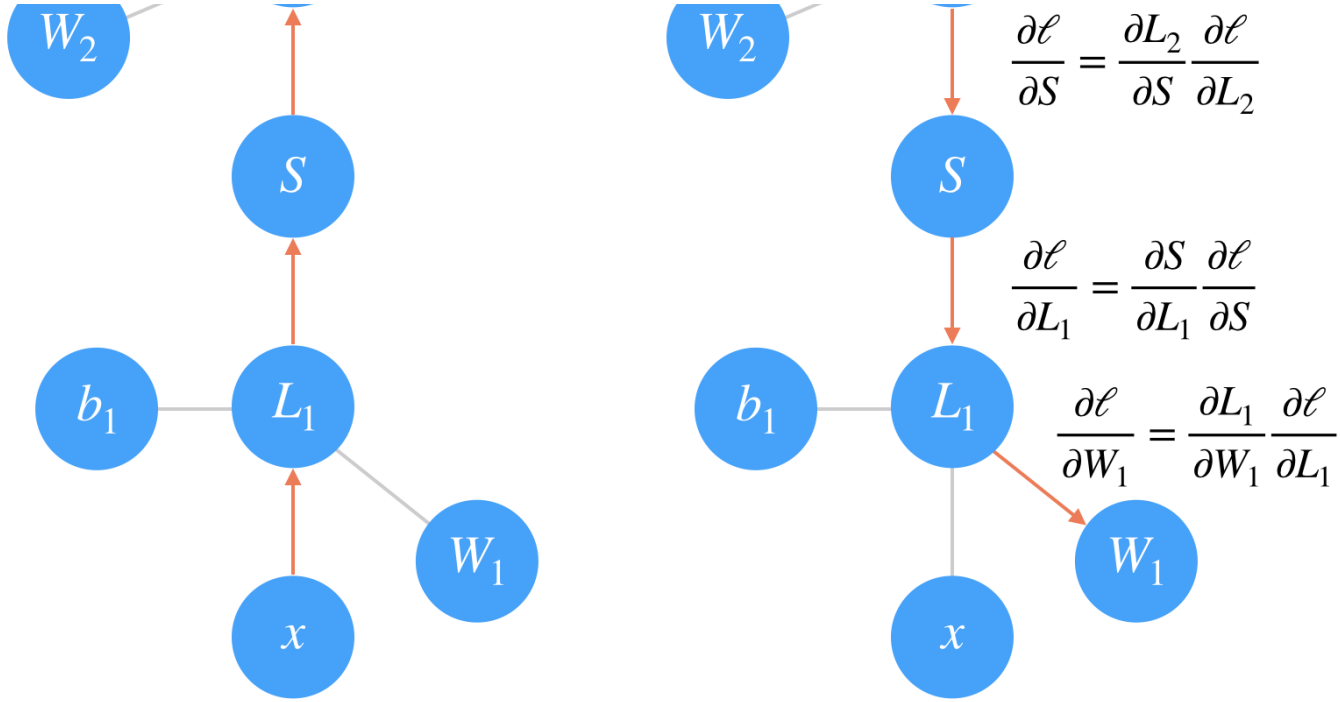


image source: udacity course material

forwards pass

In the forward pass data and operations go from bottom to top.

step 1: We pass the input x through a linear transformation $L1$ with weights $W1$ and biases $b1$

step 2: The output then goes through the sigmoid operation S and another linear transformation $L2$

step 3: Finally we calculate the loss ℓ.

We use the loss as a measure of how bad the network's predictions are. The goal then is to adjust the `weights` and `biases` to minimize the loss.

Backward pass

To train the weights with gradient descent, we propagate the gradient of the loss backwards through the network.

Each operation has some gradient between the inputs and outputs.

As we send the gradients backwards, we multiply the incoming gradient with the gradient for the operation.

Mathematically, this is really just calculating the gradient of the loss with respect to the weights using the chain rule.

$$\frac{\partial \ell}{\partial W_1} = \frac{\partial L_1}{\partial W_1}\frac{\partial S}{\partial L_1}\frac{\partial L_2}{\partial S}\frac{\partial \ell}{\partial L_2}$$

We update our weights using this gradient with some learning rate α.

$$W_1' = W_1 - \alpha \frac{\partial \ell}{\partial W_1}$$

The learning rate `α` is set such that the weight update steps are small enough that the iterative method settles in a minimum.

know more about backprop here

Losses in PyTorch

PyTorch provides losses such as the cross-entropy loss `nn.CrossEntropyLoss`. With a classification problem such as MNIST, we're using the softmax function to predict class probabilities.

To calculate the `loss` we first define the `criterion` then pass in the `output` of our network and correct labels.

> *The `nn.CrossEntropyLoss` criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.*
>
> *The input is expected to contain scores for each class.*

i.e we need to pass in the raw output of our network into the loss not the output of the softmax function. This raw output is usually called the *`logits`* or *`scores`* We use the `logits` because `softmax` gives us probabilities which will often be very close to `zero` or `one` but floating-point numbers can't accurately represent values near zero or one (read more here). It's usually best to avoid doing calculations with probabilities, typically we use log-probabilities.

Autograd

Torch provides a module called `autograd` to calculate the ***gradients*** of tensors automatically. It is kind of engine which calculates derivatives. It records a graph of all the operations performed on a **gradient enabled tensor** and creates an **acyclic** graph called the **dynamic computational graph**. The *leaves* of this graph are input tensors and the *roots* are output tensors.

`Autograd` works by keeping track of operations performed on tensors, then going *backwards* through those operations, calculating gradients along the way.

To make sure PyTorch keeps track of operations on a tensor and calculates the gradients we need to set `requires_grad = True` . we can turn off gradients for a block of code with the `torch.no_grad()` .

Training The Network

Lastly we'll in need of an `optimizer` that we'll use to update the weights with the gradients. We get these from PyTorch's `optim` package. For example we can use stochastic gradient descent with `optim.SGD` .

Process of training a neural network:

- Make a forward pass through the network
- Use the network output to calculate the loss
- Perform a backward pass through the network with `loss.backward()` to calculate the gradients
- Take a step with the optimizer to update the weights

We'll prepare our data for training:

```
import torch
from torch import nn
import torch.nn.functional as F
from torchvision import datasets, transforms

# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,),
(0.5,)),
                               ])
```

```
# Download and load the training data
trainset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True,
train=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
shuffle=True)
```

Training with real data:

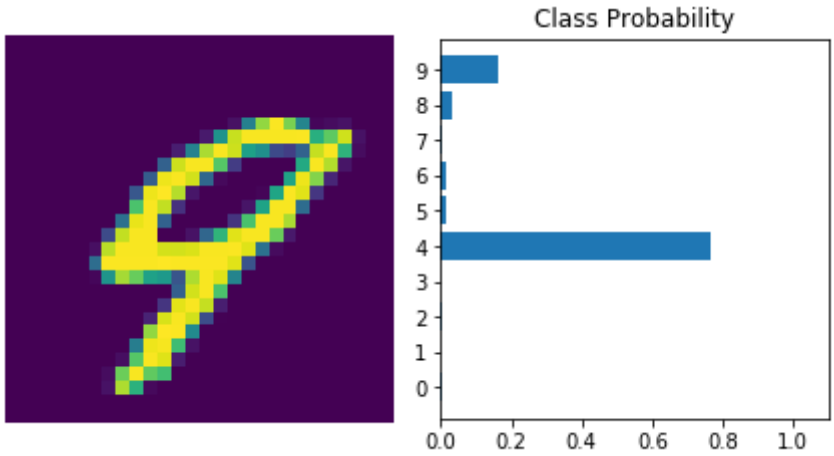
Some nomenclature, one pass through the entire dataset is called an *epoch*. So here we're going to loop through `trainloader` to get our training batches. For each batch, we'll doing a training pass where we calculate the loss, do a backwards pass, and update the weights.

```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10),
                      nn.LogSoftmax(dim=1))

# Define the loss
criterion = nn.NLLLoss()

# Optimizers require the parameters to optimize and a learning
rate
optimizer = optim.SGD(model.parameters(), lr=0.003)

epochs = 5
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        # Flatten MNIST images into a 784 long vector
        images = images.view(images.shape[0], -1)

        # Training pass
        optimizer.zero_grad()

        output = model(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    else:
        print(f"Training loss: {running_loss/len(trainloader)}")
```

> *Note: `optimizer.zero_grad()` : When we do multiple backwards passes with the same parameters, the gradients are accumulated. This means that we need to zero the gradients on each training pass or we'll retain gradients from previous training batches.*

See! the training loss is dropping with each epoch.

With the network trained, we can check out it's predictions.



Prediction result after training

Now our network is brilliant! It can accurately predict the digits in our images. Isn't it cool?

Disclaimer: This article is based on my learning in facebook-udacity scholarship challenge program.