Report

Caleb Beers

February 17, 2025

Contents

1 Review of The Little Learner

The Little Learner is a book on deep learning. This is a summary of The Little Learner.

1.1 Scheme, Lisp, and Functional Programming: Why And How

Machine learning is basically a lot of linear algebra. Functional programming lets you reason about your programs in a manner closer to reasoning about math, by moving state outside of a boundary layer.

Moreover, function composition, in particular, is a natural way to think about neural networks. Keep in mind that neural networks are basically giant composed functions. Even pythonistas will, if mathematically literate, explain backpropagation in terms of the chain rule.

The premise of th ebook is simple. You begin with a linear equation, your old friend $y = mx + b$ from high school math. Then you compose it with another function, and another, and another. You keep doing this until you've got a giant fuckoff function that constitutes a bona fide neural net. Best of all, by currying this giant composed function, you can see how it all fits together. You can see the whole thing in terms of functions returning functions, and every layer of abstraction unfolds in naked clarity. It's what is beautiful about LISP; if you want any abstraction to be blisteringly clear, write it up as a giant composed LISP function.

I **do** wish they had set it up using a threading macro, like in Clojure, but eh. You can't have everything.

1.2 From lines to neurons

Like I said before, you can get from a simple linear equation (or function, rather) all the way up to a neural network by repeatedly composing functions. You can do this by repeatedly composing one function over another. Sometimes, we stop and generalize, meaning we rejig the function so it can take an arbitrary class of arguments instead of the special case we originally wrote it for. By composing and generalizing, we can slowly grow our linear function into a neural net.

I'm going to depart from the way this is usually taught. Instead of starting off with whole vectors of weights and inputs, I'm going to focus on case where there is only one weight and only one input.

1.2.1 Linear Functions

Begin at the beginning. Let's say that $f(x) = mx + bx$. That's your typical linear function. You've seen it a million times. We're gonna lop off the b for now, change m to w, and just make it $y = wx$

Now, let's curry that bad boy. We have $f(x\,w) = wx$. And then we curry that function and evaluate it with just x. What do we get? Well, for some concrete x (let's say it's 3) we get $f(3\,w) = 3w$. Let $f_3(w) = f(3\,w)$.

This is one of those mathematical niceties that feels almost pointless. Why bother currying? Well, the answer is, we begin with a set of x's, and a set of y's. We know that those x's are supposed to map to those y's. The x's in question are our inputs, and the y's are our training set. The x's are what we have, and the y's are what we're **supposed** to have. The question becomes, what function will give us the proper y's for those x's?

That's why we curried the function. To keep things simple, let's say we begin with two xs, and a two ys that we know those xs are supposed to map to. Say the x's are 3 and 5 and the respective ys are 9 and 15. So we know that $f(3\,w) = 10$ and $f(5\,w) = 16$. We want some function such that $f(x,\, w,) = wx = 10$. Logically, we already know what x is (since we know inputs) and we already know what $f(x,\, w,)$ is **supposed** to be. But we're missing w and b. By currying f, we've defined a higher-order function. It returns a function that can, in theory, return the correct result for y (which is $f(x)$) provided we give it the correct parameters for w.

We do some experimentation. We plug in different values. We eventually find that we can set 'w = 3' and get the right outputs for both x's. However, this requires us to either do some guesswork, or solve a system of equations.

The nice thing about currying is that we don't have to do that anymore. Once the function has been curried, we can *programatically* find the right weight and bias. We could do this with random search, for example. Or, we could find some way to get closer and closer to the right weight and bias each time. To do that, we would need two things: first, a function tells us how far we are from the correct result, and second, a function that tells us which way to adjust the weights to get closer to that result. The first thing is called a **loss function**. The second is called a **gradient**.

1.2.2 Loss Function

The output of the linear equation is called the **predicted value**. The y-value that we *know* to be correct is called the **actual value**. The correct value for w (that is, the correct **weight**) is the one that produces the actual value. The current value for w (that is, the current weight) is the one that produces the predicted value. The **loss function** takes the predicted value and the actual value as arguments. It returns the distance between them, i.e. it tells us how far we are from getting our function right.

"How far" can mean a lot of things, so there are lots of different loss functions. The one we focus on right now is **l2-loss**. **l2-loss** works by subtracting the predicted y value from the actual y value and then squaring the result. Now, we know that $f_3(3)$ should be 9. That's its **actual value**. But let's say we guess that $w = 4$. In that case, $f_3(3) = 12$, so the *predictedvalue* is 12. If we let a be the actual value and p be the predicted value, then:

$$a = 9p = 12loss(9\,12) = (b - a)^2 = 9$$

Giving us a loss of 9. This tells us roughly how far we are from the **target function**, that is, the function that produces the correct y's.

1.2.3 Gradient and Gradient Descent

The gradient is the change in the loss function divided by the change in the weights. This is our old friend from calculus, the derivative. Let L be the loss function, and let w be the weight.

$$g = \frac{dL}{dw}$$

In this case, the gradient always points in the direction of steepest ascent. That is, the derivative will always tell us, given a change in w, which direction we would change w in order to *raise* the loss as quickly as possible.

But we don't want to raise the loss. We want to lower the loss. Accordingly, we revise our weight by subtracting the gradient from it. But wait - that would be way too fast! We would run the real risk of overshooting the minimum we're aiming at if we subtract the whole loss. So we multiply the gradient by a special term called the **learning rate**, which is a tiny number between 0 and 1. By doing that, we can make sure that each step moves slowly toward the minimum without overshooting.

In gradient descent, we continually revise our function. It goes like this:

1. Try some weight
2. Calculate loss for that weight
3. Multiply loss by learning rate and subtract from weight

4. Try new weight

5. If close enough, end. Otherwise, return to step 1.

If we keep doing this, we slowly converge toward the proper value for w. Yay!

1.2.4 Enter The Vectors

Now, notice how we had only one weight so far? Well, let's expand that idea. Instead of one x, let's say we have a whole set of 'em. They look like this: $[x_1\, x_2\, x_3\, x_4\, x_5\, x_6\, x_7\, x_8\, x_9\, x_{10}]$. And let's say that, furthermore, we've got a whole set of y's. They look like this $[y_1\, y_2\, y_3\, y_4\, y_5\, y_6\, y_7\, y_8\, y_9\, y_{10}]$

Now, this does *not* mean that we have ten functions. Instead, we have one function that looks like this:

$$w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7 + w_8x_8 + w_9x_9 + w_{10}x_{10}$$

So instead of a single x-value (x here is an *argument* where w is a *parameter*), f takes a vector as an argument. The vector has i x values. f also takes a vector of weights, which also has i values.

If you've ever taken multivariate calculus, you know that it's possible to take a derivative with respect to one variable. In this case, there are i gradients, with each gradient g_i being the derivative of the function with respect to x_i.

1.3 WTF Is A Tensor?

1.3.1 This is what an effing tensor is

Sometimes, when we're learning a difficult new concept, it's easier to start with a little fib. We lie about what the concept is to make it seem simpler and less scary. But then we *correct* the lie, and say, "Actually, it works like *this*". In the first grade, they told you that you can't subtract a bigger number from a smaller one. Then you got older and they said, "Actually, we lied. You *can* subtract a bigger number from a smaller number, 'cause there's this thing called a negative number."

Here's the lie: you've probably seen vectors and scalars. You've probably seen matrices. And, if you're a programmer, you know all about multidimensional arrays. You know that it's just a list of lists of lists of lists. Well, that's all a tensor is. A vector has one dimension, a matrix has two. A scalar has zero dimensions, like a point. Well, a scalar *is* a kind of tensor, what we call a tensor of “rank” zero. A vector is a tensor of rank one. A matrix is a tensor of rank two. If it has more dimensions than that, then it’s a tensor of higher rank. A tensor, therefore, is just a generalization of our familiar notion of vector.

Okay, so that was a lie. What part of it was wrong? Well, strictly speaking, the only part that was wrong was where we said that that’s *all* a tensor is. A tensor is a list of list of lists of lists of.... but it’s also much more than that. A tensor has a few other special features.

Imagine a two-dimensional array. If that array is to represent a tensor, it has to have these qualities:

1. Each “axis” or “dimension” of the tensor must have lists of equal length. [[0 0 0] [0 0 0]] would be valid. But you CAN’T have [[0 0 0] [0]]. That’s no good, because every list along the “”width" axis must be the same length.

2. The tensor must obey certain transformation rules. There are ways of “rotating” tensors around a given axis, and tensors must obey certain rules for how to be rotated. This is similar to what you learned about matrices in school: there are certain operations defined on them.

And that’s about it. There are lots of operations defined on tensors, but once you get the gist for how they work, they’re really not hard to understand. For example, two tensors can be “zipped” into a larger. So if ‘˜t1 = [2 3 5 7]˜‘ and ‘*t*2 = [11131719]‘, then you can zip ’em into a tensor that looks like this:

```
[[2 11]
[3 13]
[5 17]
[7 19]]
```

We can also think of this in terms of zipping a single tensor *along an axis*. For example, let’s say that we didn’t start with two tensors. Instead, we have one big tensor that looks like this:

```
[[ 2  3  5  7 ]
[  11 13 17 19]]
```

We can think of zipping this one tensor along its “length” axis, corresponding to the vertical dimension here. This is very similar to transposing a matrix.

Fun fact: I thought that the idea of "zipping" a tensor was a nicety that only programmers used. Moreover, when I conceived of zipping a *single* tensor "along an axis", I thought I was being awfully clever, since that wasn't in the programming books I was reading. I even observed that zipping a single tensor decreases its rank by one.

Lo and behold, the mathematicians thought of both things! The kind of zipping where you start with two tensors is called *concatenation* in mathematics, and my extension of that to zipping a single tensor along an axis is called *tensor contraction*. And, of course, it's called contraction because it shrinks the tensor's rank by one!

1.3.2 Shape

There's one other very important thing you need to know about tensors. They have a **shape**. The shape of a tensor is the length of each of its dimensions. Notice how the conventions around shape very closely align with the nature of tensors themselves. A tensor must be uniform along each axis; the shape specification seems to assume this. So, for example, this tensor:

```
[[ [ 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 0 ]]]
```

This tensor has shape (4 2 3).

1.4 Approximation In Deep Learning

1.4.1 Successive Approximation

1.4.2 Gradients

1.5 Hyperparameters

1.5.1 WTF is a hyper parameter and why do we have them?

1.5.2 How do we implement them in the context of Scheme and functional programming more generally?

1.6 Gradient Descent

1.6.1 The Skeleton Of Gradient Descent

1.6.2 Problems With Gradient Descent (and solutions thereto)

See Chapters 8 and 9 + Interlude IV for some of this

1.7 Neurons To Networks

1.7.1 Neurons As Functions

1.7.2 Layers As Functions

1.7.3 Shapelists

1.7.4 Blocks

1.8 Encoding