Building with Neurons

Introduction

When I first started learning about neural networks, I found it very difficult to match the math to the code. Most papers and explanations merged the math that's *required* for neural networks with the math that's an *optimization*. It made it very difficult to piece out exactly what was needed to make it work *at all* vs what was needed to make it work *faster*.

This book will build up the math and code for neural networks side by side, starting with the smallest neural network possible.

By the end of this first chapter, you will:

1. Understand the math that makes neural networks work
2. Have written code that 1:1 matches that math
3. Derived the calculus that makes backpropagation work
4. Implemented both feedforward and backpropagation
5. Built and trained the smallest neural network possible

That's right! By the end of this chapter you'll have derived and understood the calculus that makes neural networks work, and you'll have written a neural network library based on that exact same math.

If you haven't already, sign up at https://www.milestonemade.com/building-with-neurons/ to be notified when new chapters are available.

Chapter 1

What is a neural network

A neural network is a graph of single neurons, and each neuron is just small math formula. The output of some neurons is fed as input to other neurons. Mathematically, this means that a neural network is a composition of math formulas.

$$N = n_2(n_1(\text{input})) = \text{prediction}$$

Surprisingly, the coefficients in each neuron's formula are initialized to completely random numbers. Unsurprisingly, this means the initial output of a neural network is completely random as well! In order for a neural network to make accurate predictions, it needs to be trained - its neurons' coefficients need to be updated.

Input/output pairs of training data are used to train the network by comparing its prediction for a given input to the correct value. An error value is calculated from the difference, and this error value is used to gently update the neuron formula's coefficients through a process called backpropagation.

This chapter explains the bare minimum necessary to translate the math of neural networks 1:1 into code. Our goal today is *clarity*. We're not trying to build an F1 race car; we're not even trying to build a Model T. We're trying to build a tiny two-stroke lawnmower engine.

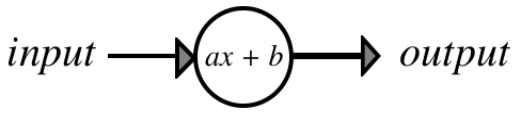
By the end of this chapter, you'll have built the smallest possible neural network, derived the math for backpropagation, and implemented all of it into code. Most importantly, you'll understand why it all works!

The smallest neural network

I think we've set ourselves a very reasonable goal: we want to define the smallest possible neural network. There's no smaller network than just 1 neuron! And since individual neurons are *essentially* just math formulas, what's the simplest formula that has at least 1 input and 1 output? A line!

$$y = ax + b$$

Perfect! Let's use that for our single neuron! Our simple network will take in a single input and provide a single output:



Let's rewrite this formula using the jargon of neural networks. Instead of calling a and b coefficients, we'll use w_1 and w_2 for "weight." And instead of y, the output of a neuron is typically called its "activation," so we'll use that term as well.

$$\begin{aligned} y &= activation \\ a &= w_1 \\ b &= w_2 \\ x &= input \\ activation &= w_1 \times input + w_2 \end{aligned}$$

Before our neuron can predict anything, we'll need to initialize our formula with some weights. If you're anything like me, this might make you a bit uncomfortable: we're just going to pick small random numbers to assign to w_1 and w_2 - yikes! Trust me this'll work out. Neural network software automatically initializes weights randomly, but we'll do it manually so we can see exactly what's happening. Let's pick 0.1 and 0.2.

$$w_1 = 0.1$$
$$w_2 = 0.2$$
$$activation = w_1 \times input + w_2$$

So let's get coding! Well, pseudo-coding. The pseudocode in this book will be easy enough to read for you to translate into your language of choice.

```
class Neuron{
    property input
    property activation
    property weights = [0.1, 0.2]

    function feedForward(){
        activation = weight[0] * input + weight[1]
    }
}
```

Note: our code indexes the weights starting with 0, but most books and academic papers use 1-indexing, so we'll start our formulas with 1 as well. It's a small detail, but something to keep in mind as we match the math to code.

Even though we've initialized our neuron with a completely random formula for a line, we're expecting and hoping that this neuron will be able to learn and predict something that's not random.

This also means that the neuron's formula will change over time. Since our neuron models the formula $ax + b$, training must change either a, x, or b. But x is provided as our input - the only values available for us to change inside the neuron are a and b! So it's *only* these weights that are allowed to change during training.

Now that we can train our neuron, we need some data to train with!

Training Data

So we've built our neuron - but what exactly are we going to predict? Well, a neuron that uses the formula for a line should be pretty good at predicting linear data, so let's train our network to predict the corresponding Fahrenheit temperature for an input Celsius temperature.

$$F = 1.8 \times C + 32$$

In code, that'd look like:

```
function convertCtoF(testInput){
    return 1.8 * testInput + 32
}
```

Perfect, now we can easily generate a piece of test data by picking a random number and running it through our new function:

```
testX = random() % 20 - 10 // pick between -10 and 10
testY = convertCtoF(testX)
```

This will make it very easy to generate test data without needing to pre-define a huge table of data.

Feedforward

At this point, we have the math and code for our single-neuron neural network and our test data. It's time to ask our network for its first prediction.

Our neuron:

$$
\begin{aligned}
w_1 &= 0.1 \\
w_2 &= 0.2 \\
activation &= w_1 \times input + w_2
\end{aligned}
$$

Sending our network an input and calculating a prediction is called *feedforward*. This step is particularly simple for our case since our network is composed of only a single neuron. For our feedforward step, we calculate the activation of our neuron for a given input. Let's find out what our network would predict for the °F value of 37°C.

$$
3.9℉ = 0.1 \times 37℃ + 0.2
$$

In code, that'd look like:

```
n = new Neuron()
n.input = 37

prediction = n.feedForward()
print prediction // 3.9
```

Ok, 3.9°F is definitely not the correct Fahrenheit value for 17°C, but just how wrong is it? Is it a lot wrong? or just a little?

To know that, we need to formalize how we calculate our error for a given prediction, and then how we can use that error measurement to help our network learn.

Calculating Error

In order for us to train our neural network, the last thing we need to define is our error formula. We need to know how wrong our network is after each activation, and since it's been randomly initialized, I suspect it will be *very* wrong. We saw above that our network predicts that 37°C is 3.9°F, but what's the real temperature?

$$
\begin{aligned}
F &= 1.8 \times C + 32 \\
62.6 &= 1.8 \times 17 + 32
\end{aligned}
$$

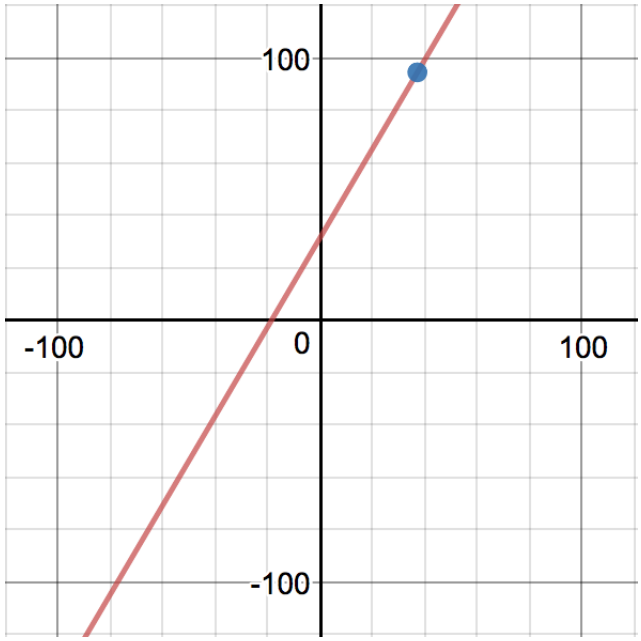
Yep! Definitely wrong! Just how wrong was our network? Let's define our rate of error as:

$$error = goal - activation$$
$$60.7 = 62.6 - 1.9$$

Great! That gives us a measured error of 60.7. And since our network is so simple, we can even graph our measured error rate for any input.

$$error = goal - activation$$
$$error = (1.8 \times C + 32) - (0.1 \times C + 0.2)$$
$$error = 1.7 \times C + 31.8$$

Let's graph that and see what we're working with:



Ok, that's odd, the error dips below 0 on the left hand side. When we predict °C for an input of -40°F, we get a *negative* error:

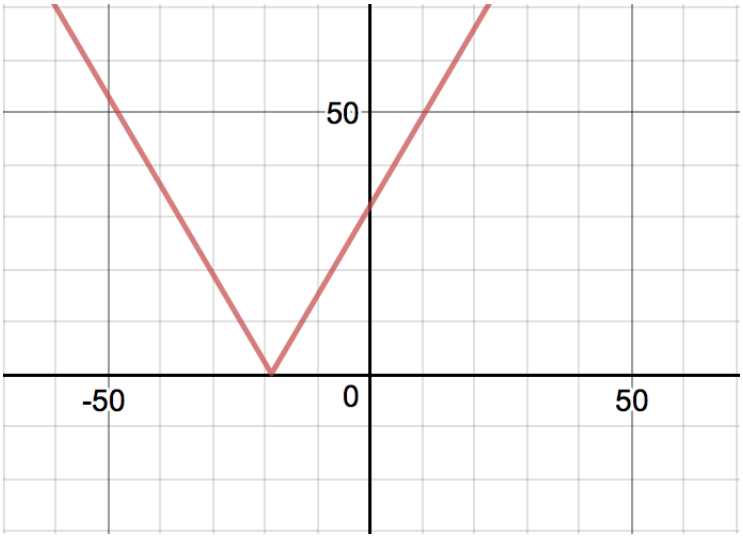
$$-36.2 = 1.7 \times -40 + 31.8$$

What does it even mean to have a negative error? Intuitively, a positive error makes sense: the larger the error the more wrong our prediction is. In the same way, we expect that the more negative our error is, the more wrong we are.

$$error_{simple} = goal - activation$$
$$error = \left|error_{simple}\right|$$

Perfect, now our error measure will always be positive, which makes more sense. The further away our prediction is from reality, the larger the error. If we had asked to predict with -18.7°C, then our neuron would have correctly predicted -1.66°F with 0 error.

In the graph below we see the error line hit 0 at -18.7°C.



Let's update our code:

```
class Neuron{
    property input
    property activation
    property weights = [0.1, 0.2]

    function feedForward(){
        activation = weight[0] * input + weight[1]
    }

    function simpleErrorFor(goal){
        return goal - activation
    }

    function errorFor(goal){
        return ABS(simpleErrorFor(goal))
    }
}
```

Next up: training! When we train, we'll be trying to reduce the network's error as close to zero as possible. We saw above that we can get 0 error for one specific input, training will help us drive toward 0 error for all inputs. We want that error graph to be as flat on the X axis as possible.

Training

Ok, we're getting very very close to the magic now! Let's get all of our formulas in one place. We'll use w_1 and w_2 to represent the two *weights* in our neuron.

$$
\begin{aligned}
input &= 17℃ \\
goal &= 62.6℉
\end{aligned}
$$

$$
\begin{aligned}
activation &= w_1 \times input + w_2 \\
&= 0.1 \times 17 + 0.2 \\
&= 1.9
\end{aligned}
$$

$$
\begin{aligned}
error &= \left| goal - activation \right| \\
&= \left| 62.6 - 1.9 \right| \\
&= 60.7
\end{aligned}
$$

There appears to be a surprising number of moving parts to this very simple neural network! We have:

1. An input number (17)
2. A correct goal output number (62.6)
3. An incorrect output number (1.9)
4. An error measurement (60.7)
5. And a bunch of formulas

Remember, there's only one piece of this puzzle that we can change during training - the weights. Let's take a second look at activation and error formulas, defining them in terms of w_1 and w_2. With the constants filled in and using a for activation and e for error, we get:

$$
\begin{aligned}
a(w_1, w_2) &= w_1 \times 17 + w_2 \\
e(a) &= \left| 62.6 - a \right|
\end{aligned}
$$

Activation is a function with weights as input, and error is a function with activation as input. And now we see we can substitute the activation formula into the error formula!

$$
e(w_1, w_2) = \left| 62.6 - (w_1 \times 17 + w_2) \right|
$$

Holding each of the weights constant, we can define our error in terms of each weight:

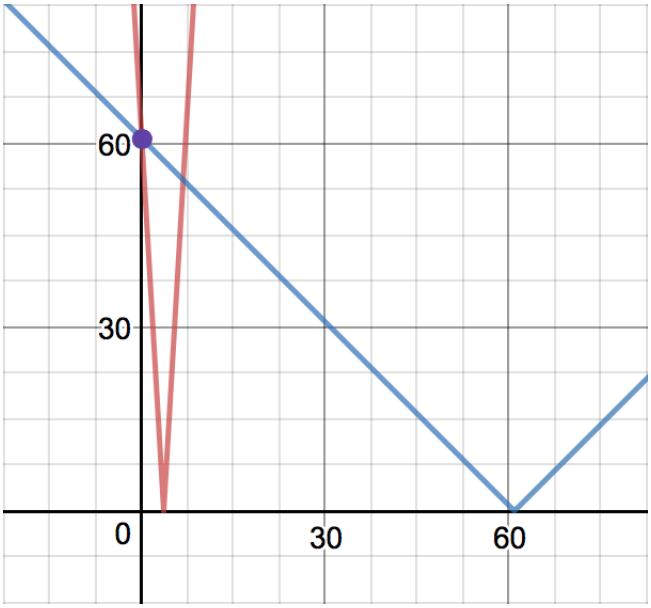
$$
\begin{aligned}
e_{w1} &= \left| 62.6 - (w_1 \times 17 + 0.2) \right| \\
e_{w2} &= \left| 62.6 - (0.1 \times 17 + w_2) \right|
\end{aligned}
$$

Let's simplify these a bit:

$$
\begin{aligned}
e_{w1} &= \left| 62.4 - w_1 \times 17 \right| \\
e_{w2} &= \left| 60.9 - w_2 \right|
\end{aligned}
$$

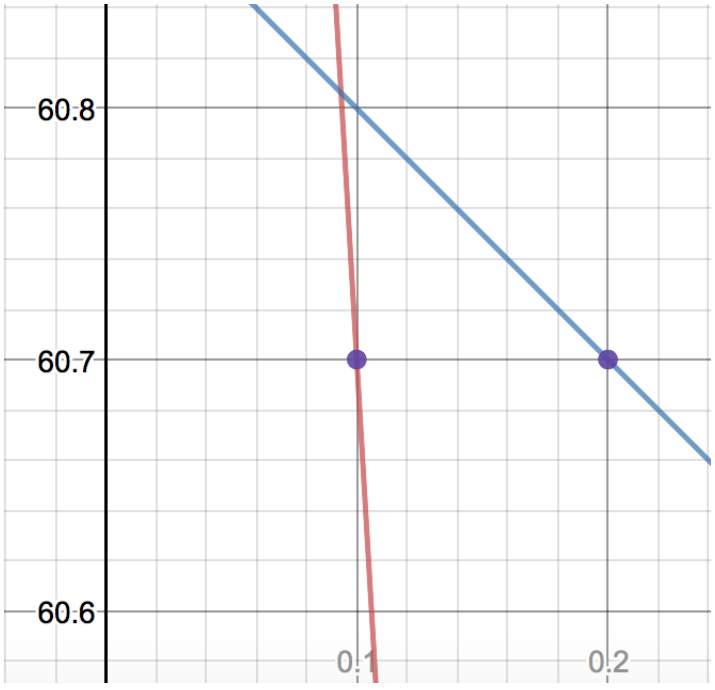
This means that we can now see how *each weight* affects error! This is different than the error graph in the previous section. There we graphed how error changed as we changed the input. Here, we're holding the input constant to see how changing each weight affects the error.

What can we learn when we graph these? I've plotted e_{w1} in red and e_{w2} in blue below, with the purple dot showing total error $e(.1, .2)$:



It might look like our current error is at the exact intersection of the two lines - but that's an artifact of the graph's zoom. If we zoom in, it'll be clear what's happening: the e_{w1} error is at $w_1 = .1$, and the e_{w2} error is at $w_2 = .2$.

$$
\begin{aligned}
w_1 &= .1 \\
w_2 &= .2 \\
e_{w1} &= \left|62.4 - w_1 \times 17\right| = 60.7 \\
e_{w2} &= \quad \left|60.9 - w_2\right| \quad\ = 60.7
\end{aligned}
$$



Having exact equations like this means that we can solve for $e = 0$ directly! When we do that, we see that setting either $w_1 \approx 3.7$ or $w_2 = 60.9$ will reduce our error to zero.

Changing one weight

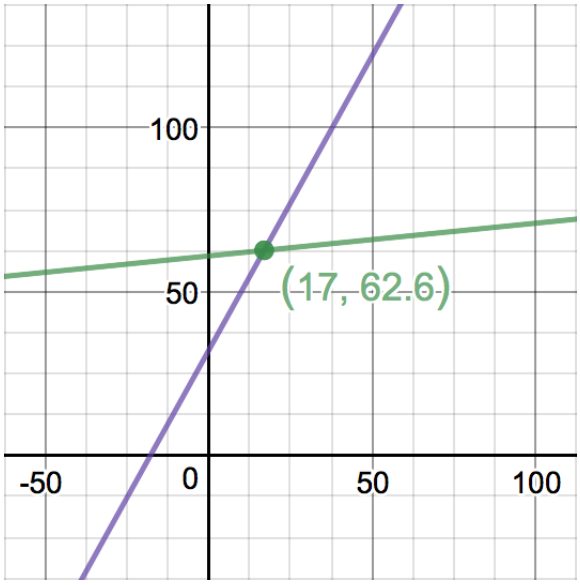
Let's try it out and test updating a weight to one of these roots. Using the new value for w_2, our neuron would look like:

$$\begin{aligned} w_1 &= 0.1 \\ w_2 &= 60.9 \\ activation &= w_1 \times input + w_2 \end{aligned}$$

Which would make our activation for that sample input:

$$\begin{aligned} activation &= 0.1 \times 17 + 60.9 \\ &= 1.7 + 60.9 \\ &= 62.6 \quad (!) \end{aligned}$$

Look at that! We've "corrected" our answer by updating just one of our weights and accurately predicted 17°C is 62.6°F! Unfortunately, our neuron would still be wrong for every *other* input - it's only corrected for that single input. Let's look at the graph of what our neuron predicts (green) for °C inputs vs the true formula (purple).



The graph makes it obvious that we'll only correctly predict that single input of 17°C. And even looking at the formulas it's clear they won't behave the same.

$$\begin{aligned} our\ neuron &= 0.1 \times input + 60.9 \\ reality &= 1.8 \times input + 32 \end{aligned}$$

What we'd really like to see after training our neuron on many test cases is:

$$\begin{aligned} w_1 &\approx 1.8 \\ w_2 &\approx 32 \\ our\ neuron &\approx 1.8 \times input + 32 \end{aligned}$$

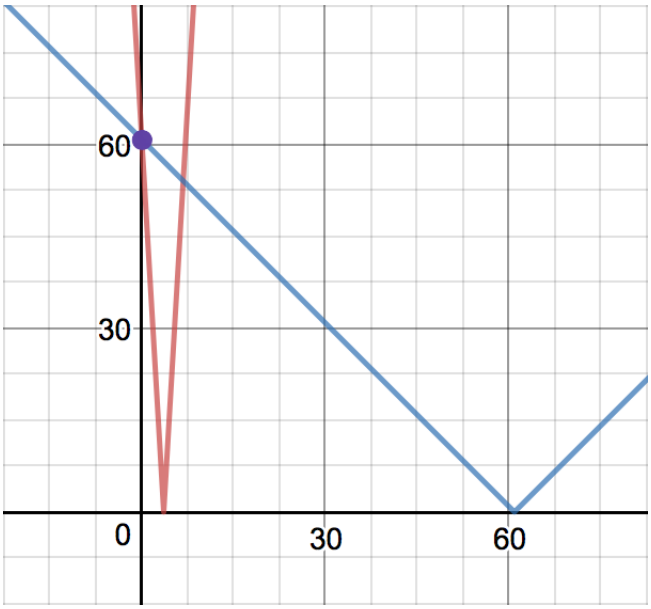
It's obviously not enough to train by just using the root values of that error graph. If we just change w_2 over and over again, we'll just be shifting that red line up and down. And if we only change w_1 over and over, then we'll just be rotating our line around the point (0, w_2). We need

our training to shift *and* rotate our red line into position to overlap the blue line.

Let's try a different approach and see what else these error formulas are telling us.

Changing both weights

Let's look at the e_{w1} and e_{w2} graphs again.



Notice that for both e_{w1} and e_{w2} formulas, both error lines have negative slopes at their input weights. What if we move the weights to the right along those lines? If we increase both values, it should decrease the error!

So let's try something different. Instead of correcting one weight *a lot*, let's try correcting both weights just a little:

$$
\begin{aligned}
bump &= 0.1 \\
w_1 &= 0.1 + bump \\
w_2 &= 0.2 + bump \\
activation &= w_1 \times input + w_2 \\
&= 0.2 \times 17 + 0.3 \\
&= 3.4 + 0.3 \\
&= 3.7
\end{aligned}
$$

We still didn't predict the correct answer of 62.6, but our new activation of 3.7 is definitely better than the original activation of 1.9! Our neuron still isn't *accurate*, but it's certainly *more* accurate!

Look again at the e_{w1} and e_{w2} graphs above. If we bump w_1 even a tiny bit, we can see that the error will change dramatically! Similarly, if we change w_2 by the same amount, the error won't change nearly as much. That's the clue we're looking for! We want to bump each weight according to how much it would affect the error! The steep slope of e_{w1} and more shallow slope of e_{w2} are our clues for how much to adjust each weight.

This is the essence of how neural networks learn. Do this lots of times for lots of input/output

pairs, bumping the weights just a bit each time, and voilà! The network has learned!

Finding the right bump

We're so close! We know *why* we should bump each weight, and we know *what direction* to bump each weight, but we still don't know exact formulas for *how much* to bump each weight. In the above example, I chose 0.1, but that's hardly scientific. Instead we should somehow be using the slope of the error function.

This is our next challenge, and this is where the magic of neural networks really comes alive! And by "magic," I mean "calculus." And by "calculus," I mean "it's not as bad as it sounds." Let's use calculus magic with these error graphs to determine how much we should bump each weight. To do that, we'll need to find the derivatives of the e_{w1} and e_{w2} functions.

A brief review: Derivatives and the Chain Rule

The slope of a line tells us how much that line changes vertically for every step horizontally. It's a measure of rate of change for that line, and that's exactly what the derivative of a formula tells us. For a formula $f(x)$, the derivative $f'(x)$ tells us how fast or slow the value of $f(x)$ is changing at that x.

For our purposes, we only need to remember a few key specifics about derivatives. First, the derivative of $|x|$ is:

$$\frac{\partial}{\partial x}|x| = \begin{cases} -1 & x < 0 \\ 1 & x > 0 \end{cases}$$

We'll also need to remember the Chain Rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

or put another way:

$$F(x) = f(g(x))$$
$$F'(x) = f'(g(x))g'(x)$$

Bringing these two together, we get:

$$\frac{\partial}{\partial x}|g(x)| = g'(x) \begin{cases} -1 & g(x) < 0 \\ 1 & g(x) > 0 \end{cases}$$

Backpropagation

We saw in the previous section that the error of our neuron is simply a function of its weights, $e(w_1, w_2)$. We also saw that to minimize the error, we should move in the opposite direction of e_{w1} and e_{w2}'s slope. When the slope of the line is negative, we should *increase* that weight. And when the slope is positive, we should *decrease* that weight. Using this process to update our neuron's weights is called *backpropagation*.

Let's try using the $e(w_1, w_2)$ function's derivative with respect to each weight to help us determine *how much* we should move down the slope.

Let's start with our error and activation formulas. It'll also be helpful for us to separate out e_{simple} from the absolute value in e:

$$\begin{aligned} a(w_1, w_2) &= w_1 \times input + w_2 \\ e_{simple}(a) &= goal - a \\ e_{abs}(e_{simple}) &= \left|e_{simple}\right| \end{aligned}$$

We now see that our total error is the composition of these three functions.

$$error = e_{abs}(e_{simple}(a(w_1, w_2)))$$

Conveniently, we just reviewed the Chain Rule to help us calculate this exact sort of derivative! So we know that:

$$\frac{\partial e}{\partial w} = \frac{\partial e_{abs}}{\partial e_{simple}} \frac{\partial e_{simple}}{\partial a} \frac{\partial a}{\partial w}$$

Let's solve for those derivatives:

$$\frac{\partial e_{abs}}{\partial e_{simple}} = \frac{\partial}{\partial e_{simple}} \left|e_{simple}\right| = \begin{cases} -1 & e_{simple} < 0 \\ 1 & e_{simple} > 0 \end{cases}$$

$$\frac{\partial e_{simple}}{\partial a} = \frac{\partial}{\partial a}(goal - a) = -1$$

$$\frac{\partial a}{\partial w_1} = \frac{\partial}{\partial w_1}(w_1 \times input + w_2) = input$$

$$\frac{\partial a}{\partial w_2} = \frac{\partial}{\partial w_2}(w_1 \times input + w_2) = 1$$

Now we can solve for e's derivative with respect to each weight! This will tell us how much each weight is responsible for the error.

$$e'_{w1}(e_{simple}) = \frac{\partial e}{\partial w_1} = \left(\begin{cases} -1 & e_{simple} < 0 \\ 1 & e_{simple} > 0 \end{cases} \right) (-1)(input) = \begin{cases} input & e_{simple} < 0 \\ -input & e_{simple} > 0 \end{cases}$$

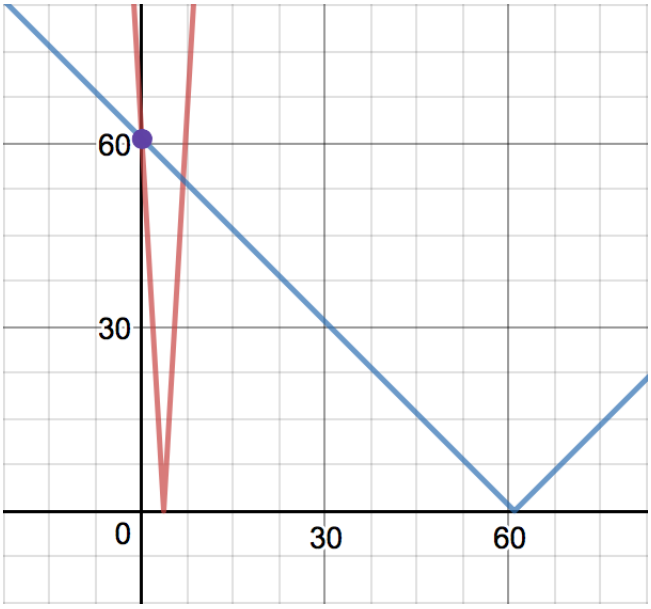
$$e'_{w2}(e_{simple}) = \frac{\partial e}{\partial w_2} = \left(\begin{cases} -1 & e_{simple} < 0 \\ 1 & e_{simple} > 0 \end{cases} \right) (-1)(1) = \begin{cases} 1 & e_{simple} < 0 \\ -1 & e_{simple} > 0 \end{cases}$$

Since we want to move the *opposite* direction of the slope of the error, we'll want to *subtract* this derivative from each weight. Every time we train a new °C/°F pair, we'll subtract e'_{w1} from w_1 and e'_{w2} from w_2.

$$w_{1\,next} = w_1 - e'_{w1}(e_{simple})$$
$$w_{2\,next} = w_2 - e'_{w2}(e_{simple})$$

Bringing It Home

With our new knowledge of the error derivative formulas with respect to each weight, let's look again at the graph for simple error and see what this means graphically. We can see clearly below that our slope for e_1 is steeper than for e_2, so our adjustment to w_1 should be larger than our adjustment to w_2.



Since e_{simple} is positive, we can now calculate $w_{1\,next}$ and $w_{2\,next}$.

$$input = 17$$
$$e_{simple} = 60.7$$
$$e'_{w1} = -input = -17$$
$$e'_{w2} = -1$$

$$w_{1\,next} = w_1 - (-input) = 17.1$$
$$w_{2\,next} = w_2 - (-1) = 1.2$$

And thankfully, we do see that e'_{w1} is steeper than e'_{w2}, so our math seems to be matching our expectations.

Coding Time

Let's translate everything we just did into code! I find that the math always makes much more sense to me once I can see it operate in code, and compute step by step.

We'll update our Neuron class below:

```
class Neuron{
    property input
    property activation
    property weights = [0.1, 0.2]

    function feedForward(){
        activation = weights[0] * input + weights[1]
        return activation
    }

    function backpropagate(goal){
        e_simple = simpleErrorFor(goal)
        delta_w0 = errorDerivativeForWeight0(e_simple)
        delta_w1 = errorDerivativeForWeight1(e_simple)

        // subtract the derivative to move _opposite_ the slope
        weights[0] -= delta_w0
        weights[1] -= delta_w1
    }

    function simpleErrorFor(goal){
        return goal - activation
    }

    function errorFor(goal){
        return ABS(simpleErrorFor(goal))
    }

    function errorDerivativeForWeight0(e_simple){
        e_deriv = (e_simple < 0) ? -1 : 1
        e_simple_deriv = -1
        a_deriv = input
        return e_deriv * e_simple_deriv * a_deriv
    }

    function errorDerivativeForWeight1(e_simple){
        e_deriv = (e_simple < 0) ? -1 : 1
        e_simple_deriv = -1
        a_deriv = 1
        return e_deriv * e_simple_deriv * a_deriv
    }
}
```

Training Time

And now, at long last, we can train our neuron!

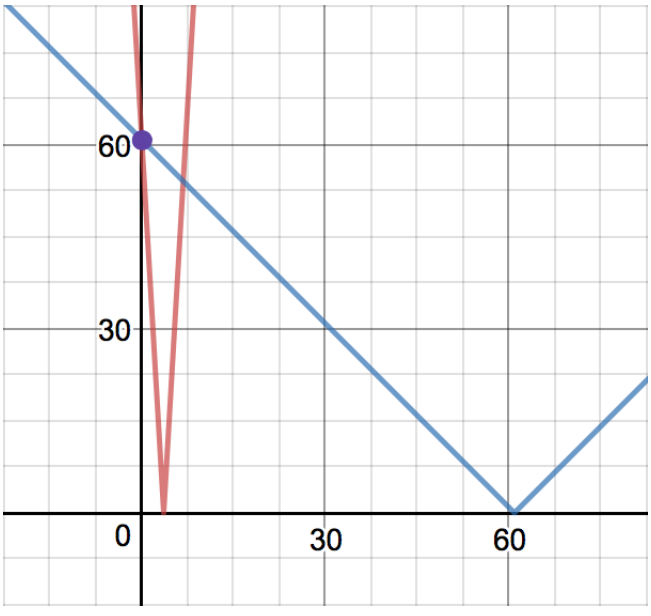
```
n = new Neuron()

for(iter = 1 ... 3600){
    celsius = random() % 20 - 10
    fahrenheit = 1.8 * celsius + 32

    n.input = celsius
    n.feedForward()
    n.backpropagate(fahrenheit)

    error = n.errorFor(fahrenheit)
    avgError = avgError * 0.99 + abs(error) * 0.01

    log("Asked for %℃ => %℉. Predicted %℉. Error %", celsius, fahrenheit, n

    if(avgError < .25){
        log("Error is less than 0.25 degrees at iteration %", iter)
        break
    }
}
```

Uh oh! It doesn't seem to be working. Instead of getting more accurate, our neuron is spiraling out of control! Its predictions are getting worse and worse.

There's one last piece for us to implement to get our neuron behaving properly!

Learning Rate

Let's pull up the graph of our error functions one more time. The red line, e_{w1}, has a very large slope, so a very small change in w_1 will produce a *huge* change in the weight.



This dramatic change in the weight will produce a large change in our predictions too. But remember our experiment where we *bumped* the neuron's weights? We were making only tiny changes to the weights to improve the prediction, but the slope of our lines is a much larger number in comparison.

Instead of using the full magnitude of the slope to update the weight, we should try scaling it to ensure its a very small bump to the weight. It turns out that the exact value of each slope is much less important than the *relative slopes* of the error lines. It's more important that $e'_{w1} > e'_{w2}$ than it is that e'_{w1} is a large number.

Let's update our Neuron code on the following page and add in a learning rate.

With our new learning rate implemented, it's time to train again! This time you should see the weights converge to match the correct formula. After 3600 iterations, the neuron will predict within just $\frac{1}{4}$°F of the correct value!

In the next chapter, we'll look at how different error formulas affect our math and our neuron's training speed.

```
class Neuron{
    property input
    property activation
    property weights = [0.1, 0.2]
    property learningRate = 0.01

    function feedForward(){
        activation = weights[0] * input + weights[1]
        return activation
    }

    function backpropagate(goal){
        e_simple = simpleErrorFor(goal)
        delta_w0 = errorDerivativeForWeight0(e_simple)
        delta_w1 = errorDerivativeForWeight1(e_simple)

        // subtract the derivative to move _opposite_ the slope.
        // use learningRate to ensure _small_ bump
        weights[0] -= learningRate * delta_w0
        weights[1] -= learningRate * delta_w1
    }

    function simpleErrorFor(goal){
        return goal - activation
    }

    function errorFor(goal){
        return ABS(simpleErrorFor(goal))
    }

    function errorDerivativeForWeight0(e_simple){
        e_deriv = (e_simple < 0) ? -1 : 1
        e_simple_deriv = -1
        a_deriv = input
        return e_deriv * e_simple_deriv * a_deriv
    }

    function errorDerivativeForWeight1(e_simple){
        e_deriv = (e_simple < 0) ? -1 : 1
        e_simple_deriv = -1
        a_deriv = 1
        return e_deriv * e_simple_deriv * a_deriv
    }
}
```

Chapter 2

Limitations of Absolute Error

In the last chapter, we saw how gently bumping the weights of our neuron would train the neuron toward the correct answer. To find how much to bump each weight, we calculated the total error as the difference between our desired output and our actual output, and we then calculated the derivative of our neuron's formula to assign some of that error to each weight. The formal name for this measure of error is the mean absolute error (MAE). Error functions are also called cost functions or loss functions.

Let's take a look at the weights' error formulas again.

$$e'_{w1}(e_{simple}) = \begin{cases} input & e_{simple} < 0 \\ -input & e_{simple} > 0 \end{cases}$$

$$e'_{w2}(e_{simple}) = \begin{cases} 1 & e_{simple} < 0 \\ -1 & e_{simple} > 0 \end{cases}$$

Notice that the amount we change each weight depends on two things:

1. The sign of e_{simple}, and
2. The size of $input$

Notably, the size of a weight's adjustment does *not* depend on the size of our error! The adjustment to the weights depends on the slope of the error, not its magnitude. This means that very large errors are having the same sized effect on our neuron as very small errors.

In this chapter we'll be learning about the Mean Squared Error, which adjusts each weight proportionally based on how much each weight contributed to the neuron's error.

As an example, imagine that a criminal gang is captured and charged for various crimes they have committed. The leader (e'_{w1}), who bears more responsibility, will be punished more harshly than a low-level member (e'_{w2}) who was less involved, and the size of their punishment will rise if more serious crimes were committed (e_{simple}). Similarly, for our neuron, very large mistakes deserve larger corrections, and very small mistakes can be ignored all together.

We might think of e'_{w1} and e'_{w2} as the amount of blame to assign to each weight. A larger e'_w means more blame assigned to that weight. Next, let's take a look at the Mean Squared Error, which adjusts the weights of our neuron proportionally to each weight's blame.

Mean Squared Error

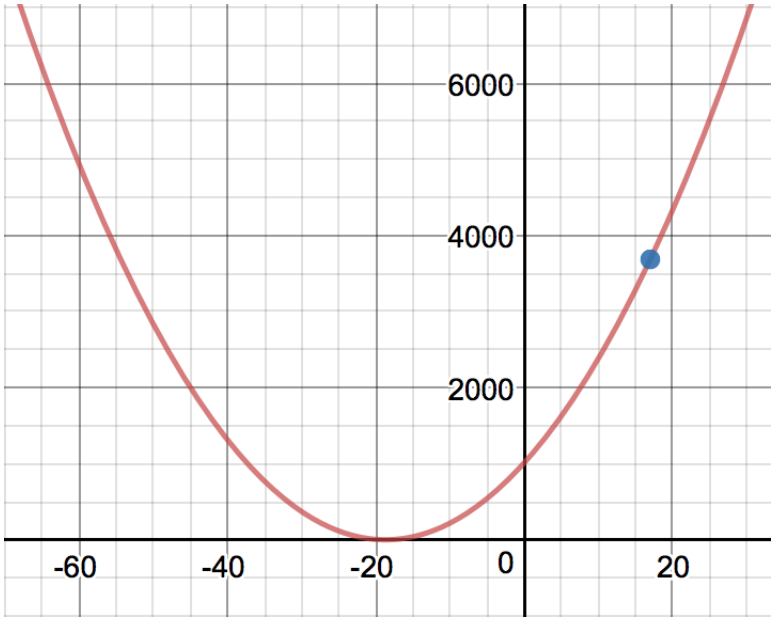
The mean squared error (MSE) is defined as the square of the goal g minus the neuron's activation a. We're just squaring the simple error that we'd calculated last chapter - that's it!

$$\begin{aligned} e_{simple} &= g - a \\ e_{mse} &= (e_{simple})^2 \end{aligned}$$

Let's get a quick idea of how this new error formula behaves. Let's calculate the mean squared error formula for the entire neuron from the last chapter.

$$\begin{aligned} e &= (g - a)^2 \\ e &= ((1.8 \times C + 32) - (0.1 \times C + 0.2))^2 \\ e &= (1.7 \times C + 31.8)^2 \\ e &= 2.89 \times C^2 + 108.12 \times C + 1011.24 \end{aligned}$$

With the mean squared error, we see that the slope is much steeper further away from zero error, but it starts to flatten out as the error approaches zero. It behaves just like it looks: If we could place a ball at the top of the slope, it'd fall quickly toward zero, but if we placed a ball very close to the bottom of the valley, then it'd roll much slower. Similarly, the closer the error is to zero, the smaller the change in the weight.



So how does this new formula affect the blame we assign each weight? How do their derivatives change? Remembering our chain rule,

$$\frac{\partial e}{\partial w} = \frac{\partial e_{mse}}{\partial e_{simple}} \frac{\partial e_{simple}}{\partial a} \frac{\partial a}{\partial w}$$

we calculate:

$$\frac{\partial e_{mse}}{\partial e_{simple}} = \frac{\partial}{\partial e_{simple}} (e_{simple})^2 = 2 \times e_{simple}$$

Conveniently, the remaining derivatives remain unchanged.

$$\frac{\partial e_{simple}}{\partial a} = \frac{\partial}{\partial a}(g - a) = -1$$

$$\frac{\partial a}{\partial w_1} = \frac{\partial}{\partial w_1}(w_1 \times input + w_2) = input$$

$$\frac{\partial a}{\partial w_2} = \frac{\partial}{\partial w_2}(w_1 \times input + w_2) = 1$$

Like last time, we solve for e's derivative with respect to each weight.

$$e'_{w1}(input, e_{simple}) = \frac{\partial e}{\partial w_1} = (2 \times e_{simple})(-1)(input) = -2 \times e_{simple} \times input$$

$$e'_{w2}(1, e_{simple}) = \frac{\partial e}{\partial w_2} = (2 \times e_{simple})(-1)(1) = -2 \times e_{simple}$$

> **Note:** You may sometimes see the MSE calculated without multiplying e_{simple} by 2. This is done by defining the error function as $\frac{1}{2}(g - a)^2$, so that the 2s in the derivative will cancel out altogether.[1] Tensorflow's implementation of MSE, however, does not simplify with $\frac{1}{2}$, so we'll be leaving the 2 in our derivative as well. (Initially, the thought of willy-nilly adding or removing a 2 made me very uncomfortable, but in fact, our network already scales everything by a learning rate anyway, so any additional linear scale doesn't matter.)

Unlike in Chapter 1, e'_w is a function of both e_{simple} and the neuron's $input$ to update the weight's value. So with MSE, the amount we update each weight is a function of more than just e_{simple}.

Looking at our neuron's formula $input \times w_1 + w_2$, we see that no matter what the value of $input$ is, only the first half of the formula is affected $(input \times w_1)$, and the second half of the formula is always the same (w_2). Similarly, when we calculate adjustments to our weights, the $input$ should only affect the adjustment the first weight w_1 and not the second weight w_2. We'll talk more about this in Chapter 3.

MSE in Code

Updating our code to implement MSE is straight forward. The only changes to our Neuron class are the formula in `errorFor()` and the `e_deriv` portion of the derivative functions:

```
    function errorFor(goal){
        return pow(simpleErrorFor(goal), 2)         // updated
    }

    function errorDerivativeForWeight0(e_simple){
        e_deriv = 2 * e_simple                      // updated
        e_simple_deriv = -1
        a_deriv = input
        return e_deriv * e_simple_deriv * a_deriv
    }

    function errorDerivativeForWeight1(e_simple){
        e_deriv = 2 * e_simple                      // updated
        e_simple_deriv = -1
        a_deriv = 1
        return e_deriv * e_simple_deriv * a_deriv
    }
```

Those three lines are all that's needed to swap out our old error measure for our new mean squared error.

Learning Speed

So is this new error measurement better than our original one? Let's find out how a new error formula affects how fast we can train our network. When we used $\left|e_{simple}\right|$ for our error calculation in Chapter 1, it took us 3600 training iterations until our neuron could predict within $\frac{1}{2}$ a degree. I've added a rolling average calculation to our training code below, so let's find out how many iterations our updated neuron will take to train.

```
n = new Neuron()

for(iter = 1 ... 3600){
    celsius = random() % 20 - 10
    fahrenheit = 1.8 * celsius + 32

    n.input = celsius
    n.feedForward()
    n.backpropagate(fahrenheit)

    avgError = avgError * 0.99 + abs(n.errorFor(y)) * 0.01

    if(avgError < .25){
        log("Error is less than 0.25 degrees at iteration %", iter)
        break
    }
}
```

Impressive! The new neuron takes only 800 iterations until it can predict within $\frac{1}{2}$ a degree, far less than the 3600 needed when we used MAE. Our neuron converges to the correct answer dramatically faster with this new error formula. And that makes sense: at the beginning of training, the slope of the error is much steeper with $(e_{simple})^2$ than with $\left|e_{simple}\right|$, so our neuron makes more dramatic changes to its weights earlier in the training process.

Flipping e_{simple}

So far we've been defining $e_{simple} = (g - a)$, but in some places you'll see error defined as $e_{simple} = (a - g)$. How does defining e_{simple} in this opposite way affect the adjustments to our weight? Let's recalculate the derivatives for MSE using this new $(a - g)$ measure for error, which we'll call $e_{flipped}$.

$$e_{flipped} = a - g = -e_{simple}$$

This flips the sign of our error, does it also flip the sign of our adjustment? Why or why not?

$$\frac{\partial e_{mse}}{\partial e_{flipped}} = \frac{\partial}{\partial e_{flipped}}(e_{flipped})^2 = 2 \times e_{flipped}$$

$$\frac{\partial e_{flipped}}{\partial a} = \frac{\partial}{\partial a}(a - g) = 1 \qquad \text{(different)}$$

$$\frac{\partial a}{\partial w_1} = \frac{\partial}{\partial w_1}(w_1 \times input + w_2) = input$$

$$\frac{\partial a}{\partial w_2} = \frac{\partial}{\partial w_2}(w_1 \times input + w_2) = 1$$

And now we see that the adjustments to our weights are flipped as well, but remember that the sign of e_{simple} has flipped as well, so our weights are changed by the same amount in the same direction.

$$\begin{aligned} e'_{w1}(input, e_{flipped}) = \frac{\partial e}{\partial w_1} &= (2 \times e_{flipped})(1)(input) \\ &= 2 \times e_{flipped} \times input \\ &= -2 \times e_{simple} \times input \qquad \text{(same!)} \end{aligned}$$

$$\begin{aligned} e'_{w2}(1, e_{flipped}) = \frac{\partial e}{\partial w_2} &= (2 \times e_{flipped})(1)(1) \\ &= 2 \times e_{flipped} \\ &= -2 \times e_{simple} \qquad \text{(same!)} \end{aligned}$$

So if you ever see error definitions using $a - g$ instead of $g - a$, you can rest easy that all of the math works out exactly the same. Our neuron's weights will be updated the same direction either way.

Activation vs Error

As I was first learning about neural networks, it was at this point that things really started to make sense from the math perspective, but I was still unsure from the code perspective.

Most tutorials either used a library like Keras that had already separated error from activation, or by building a bare bones python network that had the error formula baked in and unchangeable, as we've done so far.

Specifically, in our case, the error calculations in `errorDerivativeForWeight` are tied much too tightly to the neuron itself.

```
function errorDerivativeForWeight0(e_simple){
    e_deriv = 2 * e_simple // updated
    e_simple_deriv = -1
    a_deriv = input
    return e_deriv * e_simple_deriv * a_deriv
}
```

It's convenient for the neuron to calculate its own error from a goal, and then to calculate its own `e_deriv`, but it makes it much harder to hot-swap out different error methods to see how they behave.

Ideally, the error calculations would be factored out from the neuron's code, and it wasn't immediately clear which pieces of that function should stay in the neuron, if any, and which would be pulled out into an Error class.

Looking at the math again, we can separate out the forward propagation into two sections: everything up to the neuron's activation (the weights, input, and neuron formula), and everything after the neuron's activation (the simple error and final error).

$$\underbrace{w, i \Rightarrow a}_{\text{neuron}} \Rightarrow \underbrace{e_{simple} \Rightarrow e_{final}}_{\text{error}}$$

For backpropagation, we align these same responsibilities to their derivatives. Since we are splitting $\frac{\partial e}{\partial w}$ into parts with the chain rule, the error function is responsible for the first part $\frac{\partial e}{\partial a}$, and the neuron is responsible for calculating the last part $\frac{\partial a}{\partial w}$.

$$\frac{\partial e}{\partial w} = \underbrace{\frac{\partial e_{final}}{\partial e_{simple}} \frac{\partial e_{simple}}{\partial a}}_{\text{error}} \underbrace{\times \frac{\partial a}{\partial w}}_{\text{neuron}}$$

For us to be able to swap out error implementations, the derivative $\frac{\partial e}{\partial a}$ that we calculate in `e_deriv` should be handled outside of the neuron, and $\frac{\partial a}{\partial w}$ from `a_deriv` onward should stay inside the neuron. Our neuron shouldn't care at all about how the error's derivative was calculated, it only needs to know the final derivative.

In that separation above, the neuron needs to know about $\frac{\partial e_{final}}{\partial a}$, and then it can calculate its own $\frac{\partial a}{\partial w}$ and apply that error to each weight.

In code, this separation of responsibilities looks like:

```
    // Somewhere else, outside the Neuron class
    e_final_deriv = 2 * e_simple
    e_simple_deriv = -1
    e_deriv = e_final_deriv * e_simple_deriv

    // a much simpler function for the neuron
    function errorDerivativeForWeight0(e_deriv){
        a_deriv = input
        return e_deriv * a_deriv
    }
```

So our future error calculator will calculate `e_deriv`, and our neuron will still calculate `a_deriv`. That's the correct separation of powers, as the neuron controls its activation, so it should control the $\frac{\partial a}{\partial w}$, and the error calculator handles calculating error from activation, so it should handle $\frac{\partial e}{\partial a}$.

Refactoring Error Functions

Currently, our neuron's code contains 100% of the formulas that make the neural network work. Just as we've done in the math formulas above, I find it very helpful to separate out the error function from the neuron in our code too. This way, I can think of our $a = w_1 i + w_2$ neuron separately from our $(e_{simple})^2$ or $\left|e_{simple}\right|$ error functions, whereas right now all of that code is jumbled into the same class.

Let's start by pull the errors out into their own classes. Each error option should be able to calculate its absolute error and its derivative with respect to activation, so we'll only need those two functions for each error.

```
abstract class SimpleError{
    function error(goal, activation){
        return goal - activation
    }
    function derivative(goal, activation){
        return -1
    }
}

class ABSError : SimpleError{
    function error(goal, activation){
        return abs(super.error(goal, activation))
    }

    function derivative(goal, activation){
        derivSimpleError = super.derivative(goal, activation)
        if (super.error(goal, activation) > 0) {
            derivABS = 1
        } else {
            derivABS = -1
        }
        return derivABS * derivSimpleError
    }
}

class MSEError : SimpleError{
    function error(goal, activation){
        simpleErr = super.error(goal, activation)
        return simpleErr * simpleErr
    }

    function derivative(goal, activation){
        derivSimpleError = super.derivative(goal, activation)
        derivMSE = 2 * super.error(goal, activation)
        return derivMSE * derivSimpleError
    }
}
```

Looking at these error classes, it's easier to see the requirements for building even more error functions: we need to calculate the error given a goal and activation, and also be able to compute its derivative with the same inputs. As long as we can compute the total error and derivative with just a goal and activation as inputs, then we have enough to build a new error function.

Now that we have our error functions separated out into their own classes. Let's update our neuron class to use these error classes instead of calculating its own error.

```
class Neuron{
    property input
    property activation
    property weights = [0.1, 0.2]
    property learningRate = 0.01

    function feedForward(){
        activation = weights[0] * input + weights[1]
        return activation
    }

    function backpropagate(e_deriv){
        delta_w0 = errorDerivativeForWeight0(e_deriv)
        delta_w1 = errorDerivativeForWeight1(e_deriv)
        weights[0] -= learningRate * delta_w0
        weights[1] -= learningRate * delta_w1
    }

    function errorDerivativeForWeight0(e_deriv){
        a_deriv = input
        return e_deriv * a_deriv
    }

    function errorDerivativeForWeight1(e_deriv){
        a_deriv = 1
        return e_deriv * e_simple_deriv * a_deriv
    }
}
```

These new error classes and updated Neuron class let us easily swap out the error method used during training.

```
n = new Neuron()
n = new Neuron()
errCalc = MSEError.singleton
// or errCalc = ABSError.singleton

for(1 ... 3600){
    celsius = random() % 20 - 10
    fahrenheit = 1.8 * celsius + 32

    n.input = celsius
    n.feedForward()

    e_deriv = errCalc.derivative(fahrenheit, n.activation)
    n.backpropagate(e_deriv)

    log("Asked for %℃ => %℉. Predicted %℉. Error %",
        celsius, fahrenheit, n. activation, error)
}
```

That's all we need to do to change how we calculate a neuron's error. Next, we'll start looking at different kinds of neurons. So far we've only seen linear neurons $w_1 i + w_2$, but in Chapter 3 we'll look into non-linear neurons.

I Want Your Feedback!

Thanks for reading! I'd love to know what you thought of Chapter 1 and 2, and what you'd like to see in upcoming chapters. Please send me your feedback at adam@milestonemade.com.

And if you haven't already, sign up at https://www.milestonemade.com/building-with-neurons/ to be notified when new chapters are available!