Cost & Backpropagation

Week 2
warning: this is gonna suck a bit

Pickup from last presentation:

- We defined a neural network like a function
  - Takes a vector in, spits a vector out
  - Composed of smaller functions inside

- A neural network is a collection of neurons
  - Each neuron (excluding the input layer) has weights, a bias, and activation function
  - A weight is a value relating this neuron to another neuron in the previous layer
  - A bias is a constant value used in calculating a neuron’s unactivated value
  - The activation function takes the unactivated value and returns a new one, “activating” it.

Cost

How do we make sense of this vector of output values? What do they mean?

To give these values value, we assign each output neuron a label. A label could be yes, no, a color, a choice; the label is a potential outcome of our network. When training a network to identify hand-drawn numbers, there are 10 output neurons, each correlating to a possible digit.

However, there is a problem: the model is indifferent to our label; it doesn’t understand yes or no, and it is still just computing formulas and dispensing numbers. Do we have an analytical way to direct the model?

Cost

Introducing the cost function! This is an analytical approach to telling a machine that it is wrong, and provides a way to measure *how* wrong the model is.

There are quite a few, but let's start with this relatively simple one. Its called the mean squared error function, or MSE.

If a vector of n predictions is generated from a sample of n data points on all variables, and Y is the vector of observed values of the variable being predicted, with $\hat{Y}$ being the predicted values (e.g. as from a least-squares fit), then the within-sample MSE of the predictor is computed as

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n} \left(Y_i - \hat{Y}_i\right)^2$$

In other words, the MSE is the *mean* $\left(\frac{1}{n}\sum_{i=1}^{n}\right)$ of the *squares of the errors* $\left(Y_i - \hat{Y}_i\right)^2$. This is an easily computable quantity for a particular sample (and hence is sample-dependent).

Cost

Using the MSE, we can calculate the cost of our input. Using out prior example, suppose we give our model the rgb values for some shade of cyan:

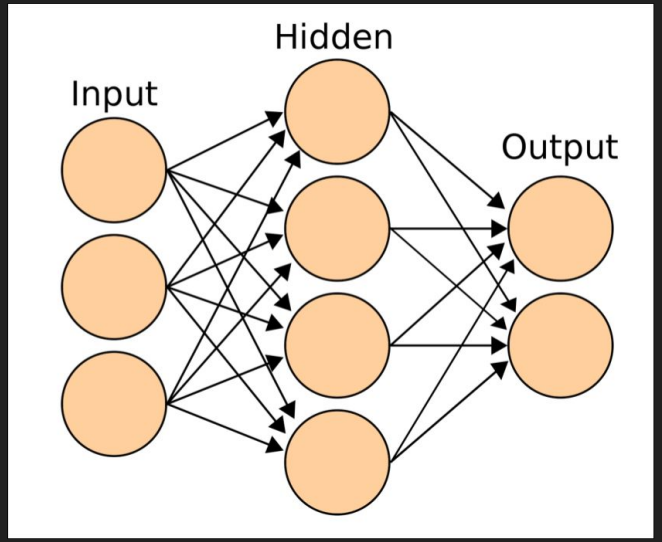
$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

I = <35, 210, 189>

And the model returns this activated output vector

O = <0.09709, 0.99983> where “yes” (O_1) = 0.097

Comparing the two magnitudes, the “no” neuron is more activated, therefore that’s the decision

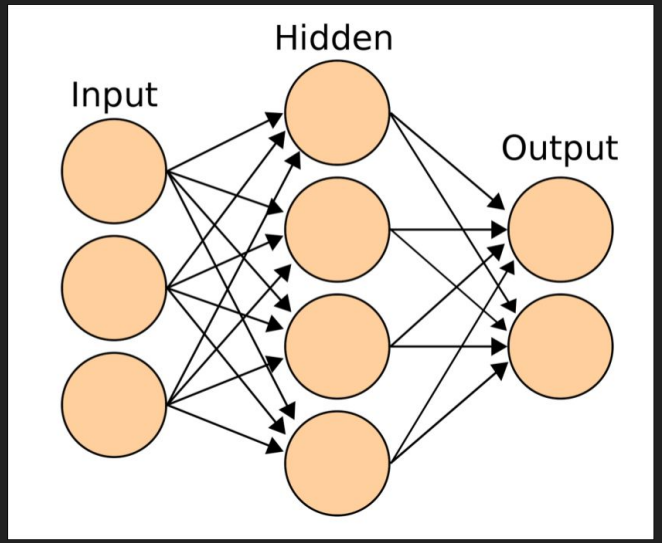


Cost

I = <35, 210, 189> O = <0.09709, 0.99983>

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

Now let's apply the MSE. Because the red value is the smallest number, probability-wise we *expect* our model to return a 0 for the yes neuron and a 1 for no neuron.



$\hat{Y}$ = <0, 1> . Y = O = <0.09709, 0.99983> n = 2

MSE = 1/n ($(Y_0 - \hat{Y}_0)^2 + (Y_1 - \hat{Y}_1)^2$)

MSE = ½ ($(0.09709 - 0)^2 + (0.99983 - 1)^2$)

MSE = ½ (0.0094264681 + 0) = 0.005

The loss is 0.005, a negligible amount. This is good!

Cost

To recap cost, it compares the expected value of a neuron with its observed value. We average the cost of all output neurons to observe the cost for a single input.

The cost function gives us an analytical model for observing how a model scores on some task. You can see which inputs gives it trouble, and where it succeeds.

In machine learning, this cost function is called the loss function, and it has a special purpose in backpropagation. Because the cost function takes as an input the vector of output neurons, it could be written as a massive sum of the products and sums of weights, input data, and biases (*foreshadowing*). And because we want to have minimal loss, backpropagation works to minimize this loss function.

Backpropagation

Continuing from that last slide, backpropagation is the process through we tweak inputs of the cost function to lower the output.

But before we get there, let’s appreciate what’s going on a bit, and understand what it means to learn.

How would a script of code “learn” ?

Think of some ways that learning could be implemented. What are some important things you would need in your program for it to work? Consider what your model must be capable of beforehand to “learn”

- A model that makes a decision
- A way to check the model’s answer
- A way for the model to correct itself

What are some ideas you have? How would you apply them to our model?

Backpropagation Overview

Here’s an overview of the backprop slides:

- Learn how a model is constructed of various inputs
- How to relate cost to certain variables
- The gradient of the Cost Function

Backpropagation

Now let’s get into it. Just a forewarning, there’s a lot of partial derivatives.

A quick recap of our model: We put numbers into the input layer, and modify these values with our weights, biases, and activation functions. To change the values we output, we consider 3 options, ranked by their viability:

- Change the weights
- Change the biases
- Change the activation functions

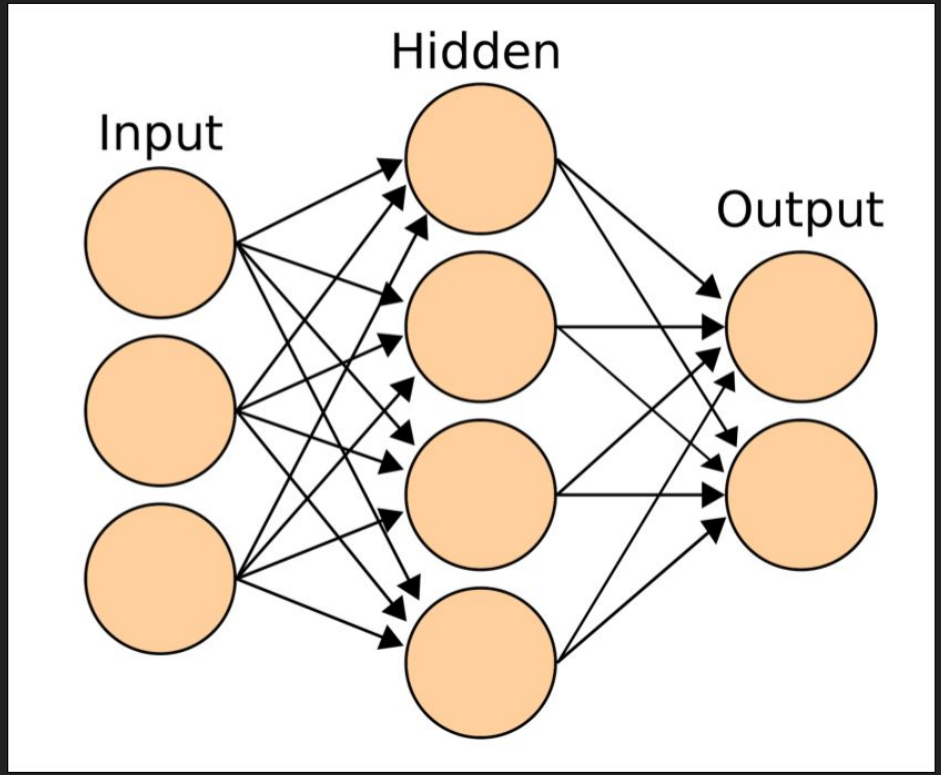
We’ll focus on the first two today, and describe them in mathematical terms.

Backpropagation

Take this neural network. Specifically, look at the 1st output neuron. Try mathematically defining the output layer vector in terms of its weights, biases, activation functions, and assign the output of this to C, the output of the cost function.

Use the sigmoid function $\sigma()$ for your activation function, and for the cost function write c().

Define neurons with dot products, then replace the dot product notation with the scalar products.



Backpropagation

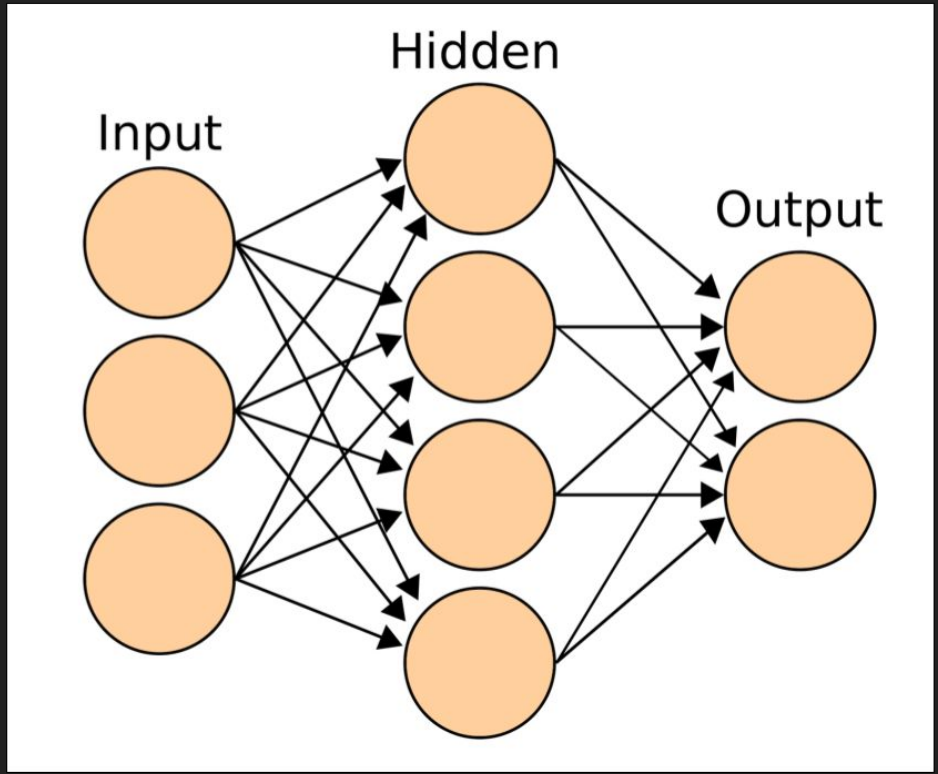
Let i, h, o, and w be vectors, *b* be a bias term, $\sigma()$ be the activation function, and <> be notation to denote a vector.

$h_0 = \sigma(h_{0,w} \cdot i + h_{0,b})$
$h_1 = \sigma(h_{1,w} \cdot i + h_{1,b})$

$h_2 = \sigma(h_{2,w} \cdot i + h_{2,b})$

$h_3 = \sigma(h_{3,w} \cdot i + h_{3,b})$

$h = < h_0, h_1, h_2, h_3 >$



Backpropagation

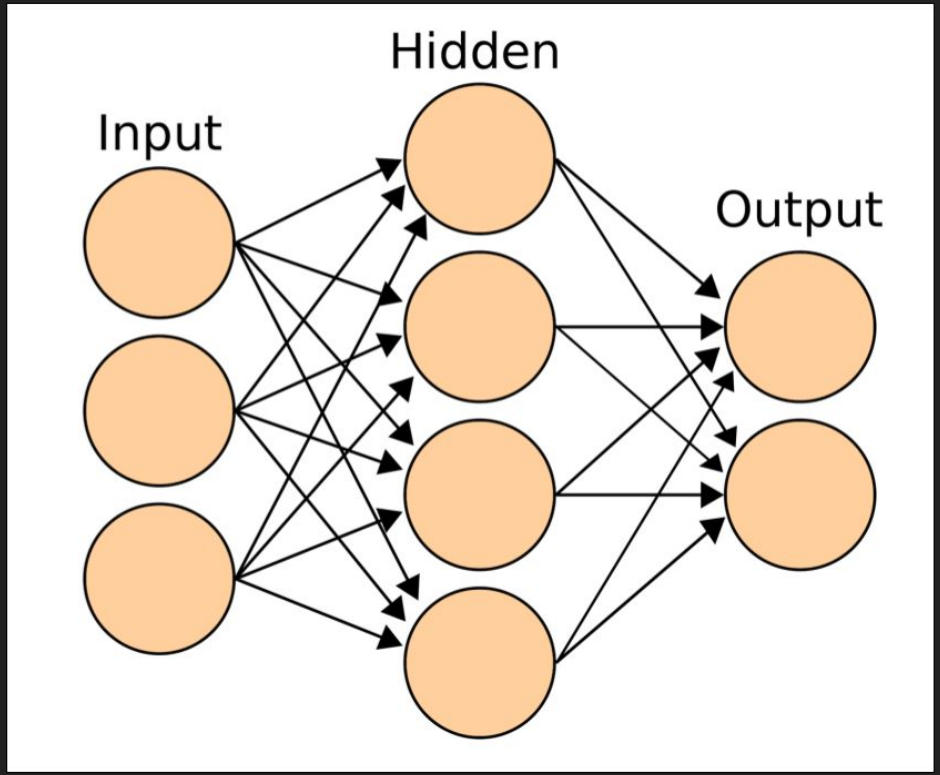
Let i, h, o, be vectors, *b* be a bias term, f() be the activation function, and <> notation to denote a vector.

$O_0 = \sigma(h \cdot O_{0,w} + O_{0,b})$

$O_1 = \sigma(h \cdot O_{1,w} + O_{1,b})$

$O = < O_0, O_1 >$

$C = c(O)$



Backpropagation

Let's start substituting our vectors for their components

$C = c(O) \rightarrow C = c(< O_0, O_1 >) \rightarrow C = c(< \sigma(h \cdot O_{0,w} + O_{0,b}) , \sigma(h \cdot O_{1,w} + O_{1,b}) >) \rightarrow$

$C = c(< \sigma(< h_0, h_1, h_2, h_3 > \cdot < O_{0,w0}, O_{0,w1}, O_{0,w2}, O_{0,w3} > + O_{0,b}), \sigma(< h_0, h_1, h_2, h_3 > \cdot < O_{1,w0}, O_{1,w1}, O_{1,w2}, O_{1,w3} > + O_{1,b}) >)$

This is getting messy very fast. Let's take a step back and process what it means.

From here, each h_i scalar represents the activation function of the dot product between the input vector i and the vector of weights for h_i, or $i \cdot h_{i,w}$ plus the bias.

$$h_i = \sigma(i_0 h_{i,0} + i_1 h_{i,1} + i_2 h_{i,2} + h_{i,b})$$

Backpropagation

Now lets substitute our h_i with the expression we derived last slide

$$C = c(< \sigma(< h_0, h_1, h_2, h_3 > \cdot < O_{0,w0}, O_{0,w1}, O_{0,w2}, O_{0,w3} > + O_{0,b}),\ \sigma(< h_0, h_1, h_2, h_3 > \cdot < O_{1,w0}, O_{1,w1}, O_{1,w2}, O_{1,w3} > + O_{1,b}) >) \rightarrow$$

$$C = c(< \sigma(\ (\ \sigma(\ i_0h_{0,w0} + i_1h_{0,w1} + i_2h_{0,w2} + h_{0,\,b}\ ) * O_{0,w0}\ ) + (\ \sigma(\ i_0h_{1,w0} + i_1h_{1,w1} + i_2h_{1,w2} + h_{1,\,b}\ ) * O_{0,w1}) + (\ \sigma(\ i_0h_{2,w0} + i_1h_{2,w1} + i_2h_{2,w2} + h_{2,\,b}\ ) * O_{0,w2}\ ) + (\ \sigma(\ i_0h_{3,w0} + i_1h_{3,w1} + i_2h_{3,w2} + h_{3,\,b}\ ) * O_{0,w3}\ ) + O_{0,b}) , \sigma(\ (\sigma(\ i_0h_{0,0} + i_1h_{0,1} + i_2h_{0,2} + h_{0,\,b}\ ) * O_{1,w0}) + (\ \sigma(\ i_0h_{1,0} + i_1h_{1,1} + i_2h_{1,2} + h_{1,\,b}\ ) * O_{1,w1}\ ) + (\ \sigma(\ i_0h_{2,0} + i_1h_{2,1} + i_2h_{2,2} + h_{2,\,b}\ ) * O_{1,w2}\ ) + (\ \sigma(\ i_0h_{3,0} + i_1h_{3,1} + i_2h_{3,2} + h_{3,\,b}\ ) * O_{1,w3}\ ) + O_{1,b}) >)$$

Wow, that's a lot. But, that's it. That's how our neural network works. Our output layer vector still has 2 elements, represented as the evaluated dot products between weights and values of previous layers going all the way back to the input layer.

Backpropagation

Previously, we discussed that causing learning means changing values. Well, let's look at what exactly we can change.

$$C = c(< \sigma((\sigma(i_0h_{0,w0} + i_1h_{0,w1} + i_2h_{0,w2} + h_{0,b}) * O_{0,w0}) + (\sigma(i_0h_{1,w0} + i_1h_{1,w1} + i_2h_{1,w2} + h_{1,b}) * O_{0,w1}) + (\sigma(i_0h_{2,w0} + i_1h_{2,w1} + i_2h_{2,w2} + h_{2,b}) * O_{0,w2}) + (\sigma(i_0h_{3,w0} + i_1h_{3,w1} + i_2h_{3,w2} + h_{3,b}) * O_{0,w3}) + O_{0,b}) , \sigma((\sigma(i_0h_{0,0} + i_1h_{0,1} + i_2h_{0,2} + h_{0,b}) * O_{1,w0}) + (\sigma(i_0h_{1,0} + i_1h_{1,1} + i_2h_{1,2} + h_{1,b}) * O_{1,w1}) + (\sigma(i_0h_{2,0} + i_1h_{2,1} + i_2h_{2,2} + h_{2,b}) * O_{1,w2}) + (\sigma(i_0h_{3,0} + i_1h_{3,1} + i_2h_{3,2} + h_{3,b}) * O_{1,w3}) + O_{1,b}) >)$$

This representation of the model is made of variables that we tweak during backpropagation. Excluding input data, this is what we modify to cause learning.

Backpropagation

So how does modifying a value in that equation cause change in the output layer?

Let's pick the weight $O_{1,w2}$ and call it w. Lets define how it influences our model with functions.

This weight's value is used to calculate the unactivated value z.

$z = \Sigma w_i a_i + b$

We can represent the change in w has on z as the partial derivative of z with respect to w :

$\partial z / \partial w = 0 + 0 + h_2 + 0 + 0$

This should be a little obvious. $O_{1,w2}$ acts as a scalar, magnifying the value of h_2 while having no other effect on the definition of $O_{1,z}$

Backpropagation

Does this change in z effect any other variable in the network?

Let's observe how a change in z has a change on a, the activated value, which is used to calculate the cost, C, of neuron O_1's output

$$a = f(z),\ y = a,\ \Rightarrow\ C_a = (y - \hat{y})^2$$

Now we have a relationship between our weight and the cost of the model. Furthermore, we can represent how a change in weight has a change on cost:

$$\partial C_a / \partial w = (\partial z / \partial w)\ (\partial a / \partial z)\ (\partial C_a / \partial a)$$

Backpropagation

Now we have a relationship between our weight and the cost of the model. Furthermore, we can represent how a change in weight has a change on cost:

$$\partial C_a / \partial w = (\partial z / \partial w)\ (\partial a / \partial z)\ (\partial C_a / \partial a)$$

Lets now calculate each of these partial derivatives to create a formula for changing our weight. Do the math first, and check the next slide for answers:

Find $\partial z / \partial w_2$ given $z = a^1_0 w_1 + a^1_1 w_1 + a^1_2 w_2 + a^1_3 w_3 + b$

Find $\partial a / \partial z$ given $a = e^x / (1 + e^x)$ (this is the sigmoid function)

Find $\partial C_a / \partial a$ given $y = a$, $C_a = (y - \hat{y})^2$

Backpropagation

Now we have a relationship between our weight and the cost of the model. Furthermore, we can represent how a change in weight has a change on cost:

$$\partial C_a / \partial w = (\partial z / \partial w)\ (\partial a / \partial z)\ (\partial C_a / \partial a)$$

Lets now calculate each of these partial derivatives to create a formula for changing our weight.

$\partial z / \partial w_2$: $0 + 0 + a^1_2 + 0 + 0$

$\partial a / \partial z$: $\sigma(z)(1-\sigma(z))$ it's a weird derivative bc e^x rules are funny

$\partial C_a / \partial a$: $2(y - \hat{y})$

Backpropagation

$\partial C_a / \partial w_2 = (\partial z / \partial w)\,(\partial a / \partial z)\,(\partial C_a / \partial a)$

$\partial z / \partial w_2$: $0 + 0 + a^1_2 + 0 + 0$

$\partial a / \partial z$: $\sigma(z)(1-\sigma(z))$

$\partial C_a / \partial a$: $2(y - \hat{y})$

$$\partial C_a / \partial w_2 = (a^1_2)(\sigma(z)(1-\sigma(z))(2(y - \hat{y}))$$

This is what the partial derivative of cost is with respect to our chosen weight

Backpropagation

A general equation relating any weight from any layer with an associated cost. Remember, a change to a weight that isn't from the output layer will "ripple" through all layers until it reaches the end. This is represented by the recursive error term, δ

Let the superscript L denote a layer and the subscripts i,j,k denote the ith/jth/kth neuron in layer L/L-1/L+1. Let $w^L_{i,j}$ be the weight between neuron i of L and neuron j of L-1. Let b^L_i be the bias of neuron i in layer L. Let z^L_i and a^L_i be the unactivated and activated values of the ith neuron in layer L. Let f() be some activation function, and let δ^L_i represent the error of neuron i in layer L. Assume this term doesn't represent the error calculated by the cost function until L increments to the final layer.

The partial derivative of the cost with respect to the weight connecting neuron j with neuron i of layer L is the following:

If L is the output layer:

$$\partial C_i / \partial w^L_{i,j} = (a^{L-1}_i) * f'(z^L_i) * (2(y_i - \hat{y}_i))$$

Else:

$$\partial C_i / \partial w^L_{i,j} = (a^{L-1}_i) * \delta^L_i \text{ where } \delta^L_i = f'(z^L_i) * \Sigma_k (w^{L+1}_{k,i} * \delta^{L+1}_k)$$

Backpropagation

If L is the output layer:

$$\partial C_i / \partial w^L_{i,j} = (a^{L-1}_i) * f'(z^L_i) * (2(y_i - \hat{y}_i))$$

Else:

$$\partial C_i / \partial w^L_{i,j} = (a^{L-1}_i) * \delta^L_i \text{ where } \delta^L_i = f'(z^L_i) * \Sigma_k (w^{L+1}_{k,i} * \delta^{L+1}_k)$$

Let's unpack. The top one should make some sense, it's the products of the individual derivatives now expressed in the formal notation. The bottom one is definitely different. Why does the computation of non-output layer weights require this recursive term? Think back to writing out every component of the network. If you modified some weight, its change would ripple through the network. This summation describes those changes as the activation function times the sum of weights and activated values.

Backpropagation

Let’s backtrack to the case of an output layer weight, defined as such:

$$\partial C_i / \partial w^L_{i,j} = (a^{L-1}_i) * f'(z^L_i) * (2(y_i - \hat{y}_i))$$

Now that we have a way to calculate this partial derivative, let’s think about what it represents. This value is going to be a scalar, and it describes the rate of change of the cost function with respect to this weight variable that was chosen.

The derivative’s sign tells us the direction of change, and the magnitude describes how fast this change is occurring. Using this information, we are going to adjust the weight value as to minimize the cost.

Backpropagation

Let's backtrack to the case of an output layer weight, defined as such:

$$\partial C_i / \partial w^L_{i,j} = (a^{L-1}_i) * f'(z^L_i) * (2(y_i - \hat{y}_i))$$

If the derivative is positive, that means this weight is *increasing* the cost function. To combat this, we will *decrease* the weight variable so it has less of an effect on the cost.

Conversely, if the derivative is negative, that means this weight is *decreasing* the cost function. This is good. We will increase the value of this weight to maximize its cost-negating effects.

We determine how much to change the weight by referencing the magnitude of the derivative. If the magnitude is *big*, then a *small* change will have a *big* impact. The opposite is true. If the magnitude is *small,* then a *big* change will have a *small* impact.

Backpropagation

A little challenge: We analyzed how adjustments to an output-layer weight affect the cost of the network. But what about the bias?

Compute the $\partial C / \partial b$, the partial derivative of cost with respect to bias.

Assume the bias is in an output-layer neuron.

First write it as the product of derivatives (see how we did weights) then substitute each derivative in.

Finally, describe what this partial derivative means. Is it any different from the partial derivative used to relate weight to cost? Why or why not?

Cost & Backpropagation, Recap

In today's slides we covered the following concepts:

- We can evaluate the effectiveness of a model with a cost function
- By using the cost function, we can build a backpropagation algorithm
- This algorithm adjusts weights and biases inside our network to minimize cost
- The algorithm analyzes partial derivatives of weights and biases
- Through the sign and magnitude of these derivatives adjustments are made

This is the hardest part of machine learning, so congratulations on getting through this! There will probably be another slideshow on backpropagation, specifically about building an algorithm to do backpropagation from a coding perspective.