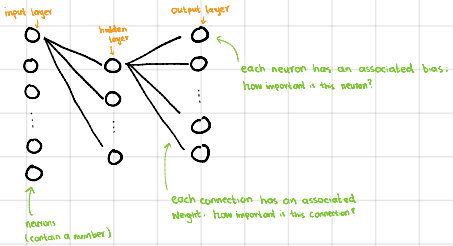
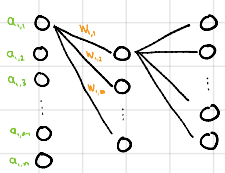
Introduction

A network of neurons; analogous to the brain.

Neurons and Layers



Forward Propagation



The activation of each neuron is given by:

1. $z' = \sum_{i=1}^{n} w_i a_i + b$

   (sum of weighted activations of previous layer; individual bias)

   $= \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ w_{2,1} & w_{2,2} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \dots & w_{m,n} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ $\left(= \begin{bmatrix} w_{1,1}a_1 + w_{1,2}a_1 + \dots + w_{1,n}a_1 \\ w_{2,1}a_2 + w_{2,2}a_1 + \dots + w_{2,n}a_2 \\ \vdots \\ w_{m,1}a_m + w_{2,m}a_m + \dots + w_{n,m}a_n \end{bmatrix} \right)$

   (outputs; input weights)

2. $z = \text{act}(z')$

   ↳ perform some activation function to turn z into an activation. E.g. ReLU $= \begin{cases} z & \text{if } z > 0 \\ 0 \end{cases}$

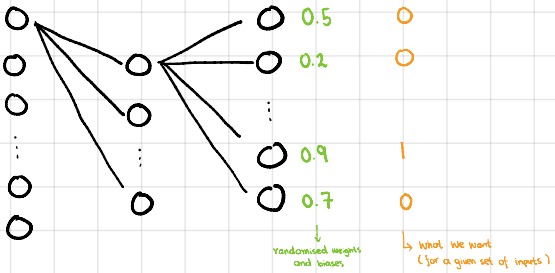
3. $a = \sigma(z)$

   ↳ squish down z into $[0,1]$. E.g. $\sigma(z) = \frac{1}{1+e^{-z}}$

Note how these activation functions and squishification functions make the network a function, not just a linear computation.

Training the Network: Cost Function

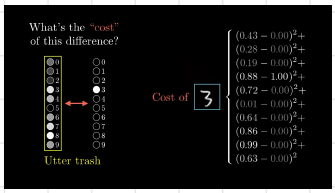
We train a neural network until the weights and biases are calibrated such that given input activations, the right output neurons will fire. We start by randomising:

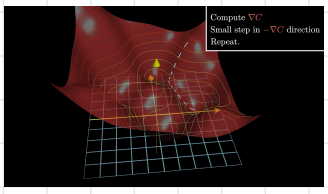


A cost function (takes in all input parameters, too) describes how off the current weights and biases are by how off the final output is. E.g. $(\text{output} - \text{expected})^2$.

Backpropagation

Backpropagation is an algorithm for computing that negative gradient. Recall that $-\nabla C$ is an n-dimensional vector that tells us how to adjust all the weights and biases to decrease the cost most efficiently. [...]





Gradient Descent

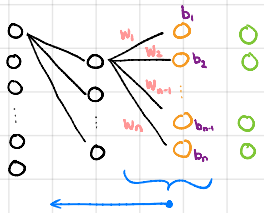
It becomes intuitive that the process of a neural network learning is really just minimising this cost function (by adjusting weights and biases).

How does one do this? Consider the graph of this multivariable function. If we start at any random point, then constantly step in a downward direction, we will eventually reach a local minimum.

From calculus III, this 'downward direction' is given by the negative of the gradient of the cost function (∇C). Further, the magnitude of this gradient vector tells us how much to proportionally step in each direction: as we get closer to the local minimum, this value becomes smaller and smaller so as to not overshoot. We can further control each step through a scalar factor — the learning rate.

We compute the value of the gradient vector (at each step) through a process known as backpropagation.

Backpropagation cont.



First recall that the cost is directly related to the output layer's activation. This, itself, is influenced by each neuron's individual bias and each connection weight. This applies iteratively for each layer.

So. There are three (3) avenues that influence an output neuron's activation (and hence the cost):

1. Change the bias.
2. Change the weights.
3. Change the activations from the previous layer.

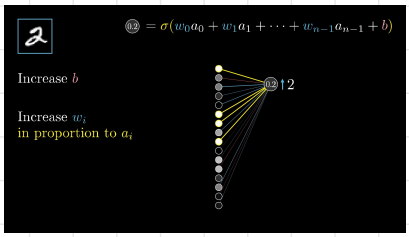
Changing Bias

The effect of a change to the bias on the weighted sum is constant and predictable. Increase if output < desired activation and vice versa.

Changing the Weights

How should we change the weights? First note that changing the weights associated with large activation values will have a stronger effect than changing the weights associated with small activation values.

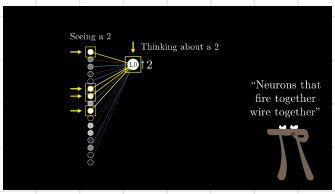
For $-\nabla C$, we don't just care about what should be moved up or down, but also by how much. (what gives you most bang for your buck).



This is somewhat similar to biological neural networks under Hebbian theory. "neurons that fire together wire together". Here, the biggest changes to weights — the biggest strengthening of connections — happens between neurons that are the most active and the ones which wish to become more active.

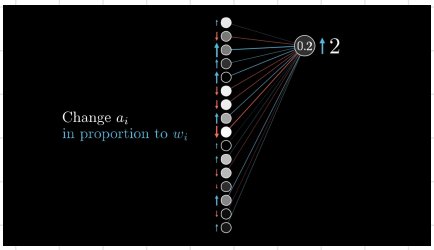
(strong activations ⇒ more strong activations).

Pictorially:

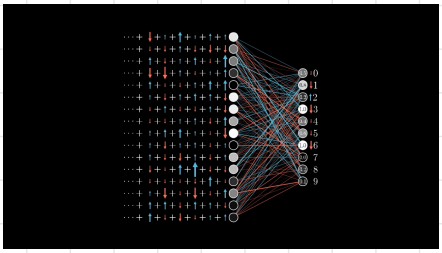


Changing the Activations

We cannot change the activations in a previous layer directly. So. We change the weights of the preceding layer (and biases of the previous layer). This will likely cause competing requests. Again, these changes are proportional to the weights.



To balance competing requests between neurons in the same layer, we take a sum.



Here is where the idea of propagating backwards comes in. By adding all these desired effects, we can get a list of the nudges we want to happen to this 2nd last layer. From there, we recursively apply this process to the relevant weights and biases determining those values, repeating as you move backward through the network.

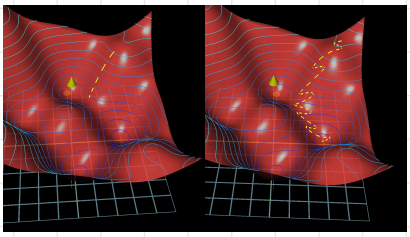
Repeating for All Training Examples

We take an average of all versions/suggestions of weights and biases.

Stochastic Gradient Descent

It is too exhaustive to train every sample. So we train in 'mini-batches'.

Then we treat the average weight/bias suggestion of each mini-batch as one training sample, and find the average between these averages.



It looks like larger strides (rather than steps) down the graph.

Backpropagation Calculus

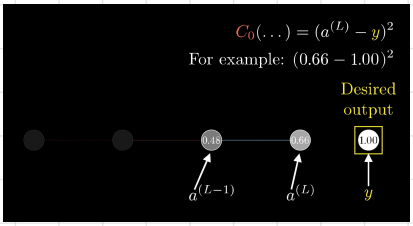
The cost function is a function whose variables are the weights and biases of the neural network. As such, its gradient can be given as

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial b_1} \\ \vdots \\ \frac{\partial C}{\partial w_n} \\ \frac{\partial C}{\partial b_n} \end{bmatrix}.$$

The goal with this is to find the relationship between a change to C and the necessary change to a given weight and bias to incur that change.

Simple Example

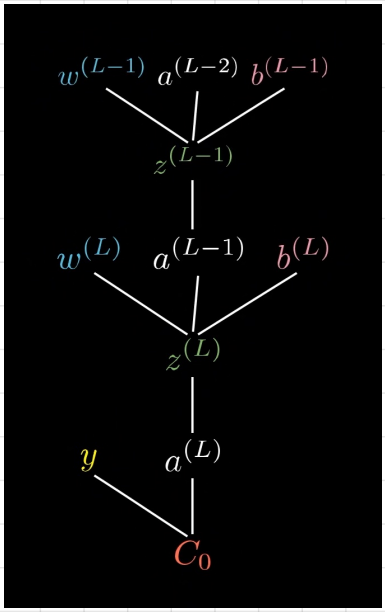
Let's begin by considering a simple network.



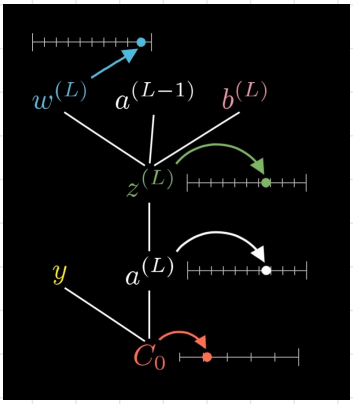
Recall that the last activation ($a^{(L)}$) is determined by a weight, a bias, and the activation of the previous neuron, all pumped through some special non-linear activation function.

$$\begin{aligned} a^{(L)} &= \sigma\left(w^{(L)} a^{(L-1)} + b^{(L)}\right) \\ &= \sigma\left(z^{(L)}\right), \end{aligned}$$

Where z denotes the weighted sum of the L^{th} layer. Let's visually keep track of this.



Then to adjust each weight and bias to incur a change in C_0, we can use the chain rule. (Rate of change with respect to hidden/embedded variables.



Backpropagation Calculus cont.

So. We have that:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial C_0}{\partial a^{(L)}} \times \frac{\partial a^{(L)}}{\partial z^{(L)}} \times \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

Practically:

- $z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$
  - $\hookrightarrow \frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$
- $a^{(L)} = \sigma(z^{(L)})$
  - $\hookrightarrow \frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$
- $C_0 = (a^{(L)} - y)^2$
  - $\hookrightarrow \frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$

We can apply this entire process similarly for the bias and find that $\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1$.

Now. Note that for these functions, the first derivative, $\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$, tells us that changing the weight has a stronger effect on $z^{(L)}$ (and therefore on the cost) when the previous neuron is more active. This is where that biological idea that "neurons that fire together, wire together" is mirrored in our math. (Strong activations $\Rightarrow$ more strong activations).

Further, the third derivative is directly dependent on the loss $a^{(L)} - y$. This rejects the intuitive idea that a more inaccurate set of weights and biases (i.e. greater cost) would incur greater adjustments to the weights and biases.

For All Training Data

So far, this has only been for one training sample. The full cost function for the network is the average all the individual costs for each training example:

$$C = \frac{1}{n} \sum_{k=0}^{n-1} C_k$$

If we want the derivative of C with respect to the weight, we need to take the average of all the individual derivatives.

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}$$

But this is just one entry in ∇C.

The Derivatives for All Weights And Biases

The full gradient is given as

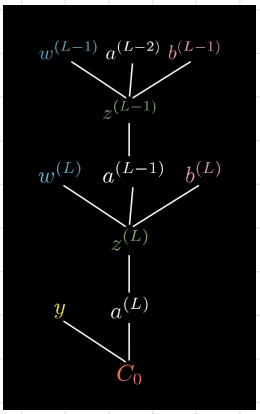
$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial b_1} \\ \vdots \\ \frac{\partial C}{\partial w_n} \\ \frac{\partial C}{\partial b_n} \end{bmatrix}.$$

Now, we need to account for the rest of the weights and biases in the network, the 'backward' part. We first think literally:

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \times \frac{\partial a^{(L)}}{\partial z^{(L)}} \times \frac{\partial C_0}{\partial a^{(L)}}.$$

We can compute $\frac{\partial z^{(L)}}{\partial a^{(L-1)}}$ to be $= w^{(L)}$, but this just means changing the activation in L-1 will incur a change in $z^{(L)}$ proportional to $w^{(L)}$. This is somewhat meaningless because we don't care about changing $a^{(L-1)}$ directly.

So. We treat $a^{(L-1)}$ as $a^{(L)}$ and iterate the previously seen chain rule.



Here,

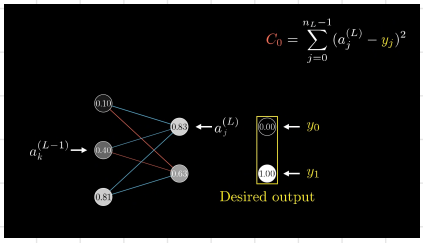
$$\frac{\partial C_0}{\partial w^{(L-1)}} = \frac{\partial C_0}{\partial a^{(L)}} \times \frac{\partial a^{(L)}}{\partial z^{(L)}} \times \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \times \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \times \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}}.$$

Note, the further back we go, the less we affect the cost.

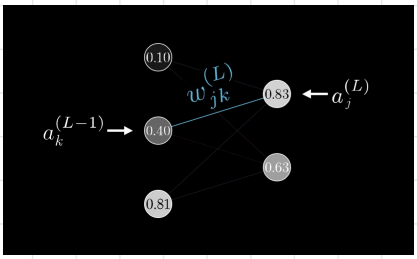
More Complicated Networks

With multiple neurons in each layer, we will need to keep track of the neurons in each layer more closely.

Let's use k to index neurons of layer $(L-1)$ and j to index the layer L. Then we have the cost as a sum:



Additionally, each weight must be made more specific:



Note: we say jk as opposed to kj to align with the weight matrix.

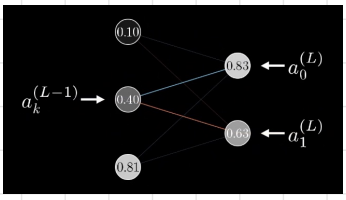
So now we have $z_j^{(L)} = \sigma(w_{j0}^{(L)} a_0^{(L-1)} + \dots + w_{jk}^{(L)} a_k^{(L-1)})$.

Then: $\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial C_0}{\partial a_j^{(L)}} \times \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \times \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}$.

Note that these new equations are essentially the same — we just became more specific about which weight (and bias), i.e. letters to index, to account for the many weights.

What <u>does</u> change, however, is the derivative of the cost with respect to one of the activations in the layer $(L-1)$, $\frac{\partial C_0}{\partial a_k^{(L-1)}}$.

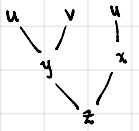
This neuron now influences the cost through multiple paths.



(i.e. through $w_{0,k}$ and $w_{1,k}$).

The total influence of $a_k^{(L-1)}$ is then the sum of the derivatives along multiple paths of influence.

One way to think of it is now, $a_k^{(L-1)}$ is a shared variable:

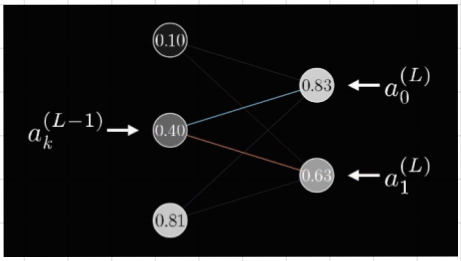


$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial u} + \frac{dx}{du}.$$

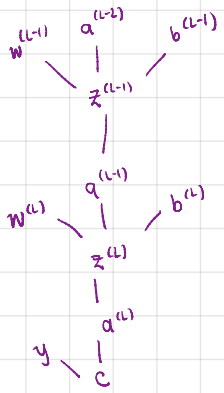
So we finally have that

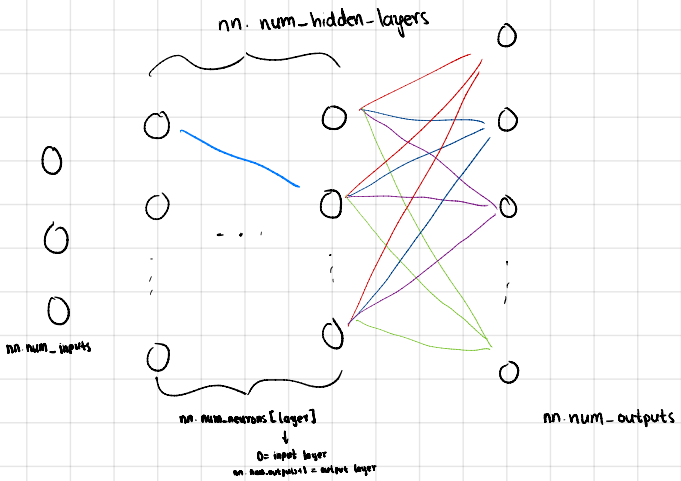
$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial C_0}{\partial a_j^{(L)}} \times \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \times \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}}.$$

↓ each path (that ends at $a_1^{(L)}, a_2^{(L)}, \dots, a_j^{(L)}$)



Practise Implementation





nn.connections[2][2][1]
layer, destination, source

$$\sigma = \frac{1}{1+e^{-x}} = (1+e^{-x})^{-1}$$

$$\sigma'(x) = -(1+e^{-x})^{-2}(-e^{-x})$$

$$= \frac{e^{-x}}{(1+e^{-x})^2}$$