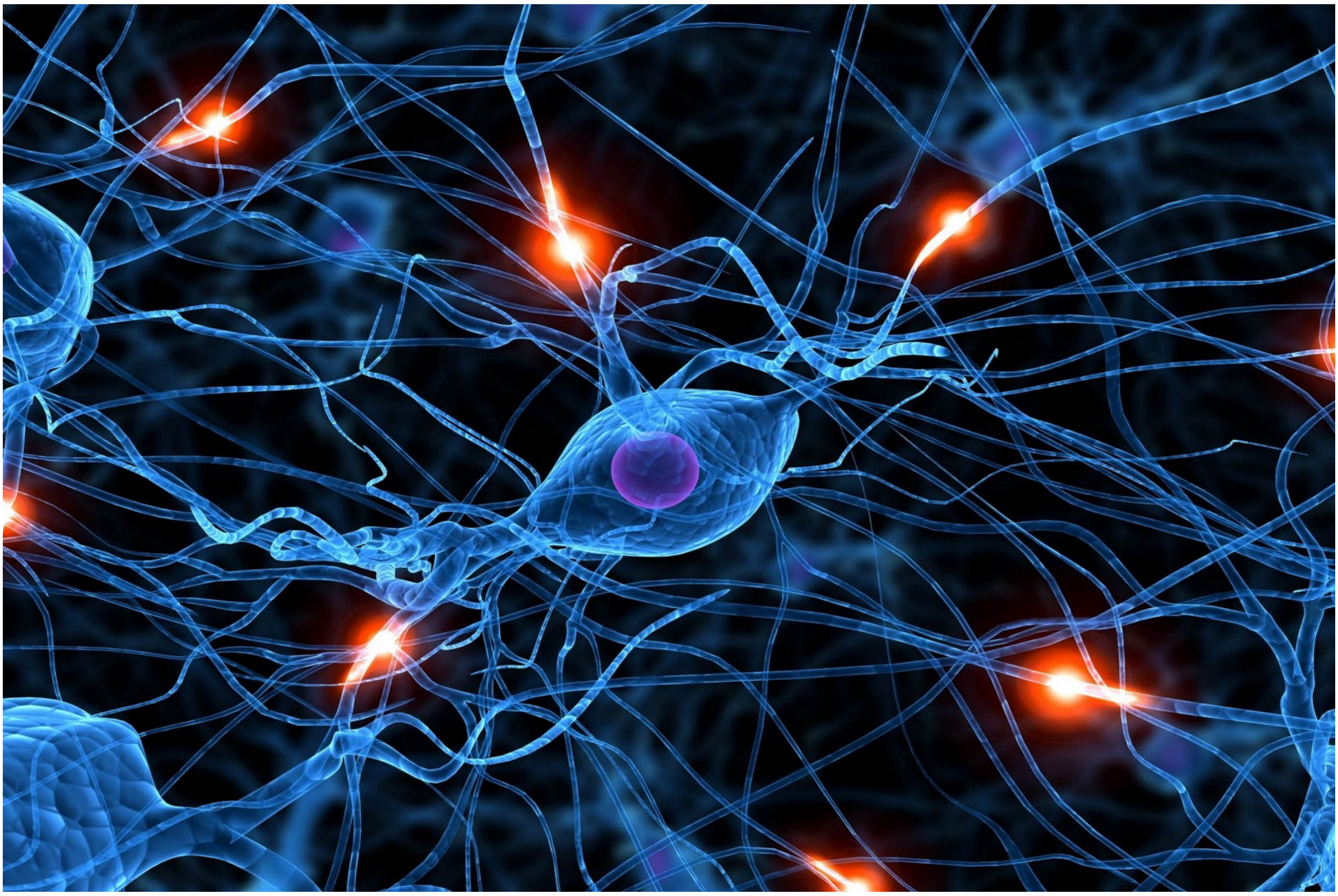


Rohan & Lenny #1: Neural Networks & The Backpropagation Algorithm, Explained

Do you know the chain rule? Then you know the neural network backpropagation algorithm!

Rohan Kapur Follow
Mar 4, 2016 · 30 min read

. . .

This is the first group (Lenny and Rohan) entry in our journey to extend our knowledge of Artificial Intelligence in the year of 2016. Learn more about our motives in this introduction post.

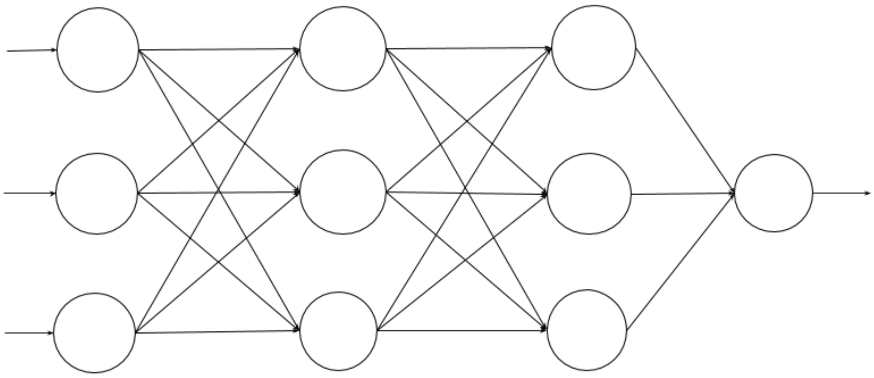
. . .

In Rohan's last post, he talked about evaluating and plugging holes in his knowledge of machine learning thus far. The backpropagation algorithm—the process of training a neural network—was a glaring one for both of us in particular. Together, we embarked on mastering backprop through some great online lectures from professors at MIT & Stanford. After attempting a few programming implementations and hand solutions, we felt equipped to write an article for AYOAI—together.

Today, we'll do our best to explain backpropagation and neural networks from the beginning. If you have an elementary understanding of differential calculus and perhaps an intuition of what machine learning is, we hope you come out of this blog post with an (acute, but existent nonetheless) understanding of neural networks and how to train them. Let us know if we succeeded!

Introduction to Neural Networks

Let's start off with a quick introduction to the concept of neural networks. Fundamentally, neural networks are nothing more than really good function approximators—you give a trained network an input vector, it performs a series of operations, and it produces an output vector. To train our network to estimate an unknown function, we give it a collection of data points—which we denote the "training set"—that the network will learn from and generalize on to make future inferences.



This is what a neural network looks like. Each circle is a **neuron**, and the arrows are connections between neurons in consecutive **layers**.

Neural networks are structured as a series of **layers**, each composed of one or more **neurons** (as depicted above). Each neuron produces an output, or **activation**, based on the outputs of the previous layer and a set of weights.

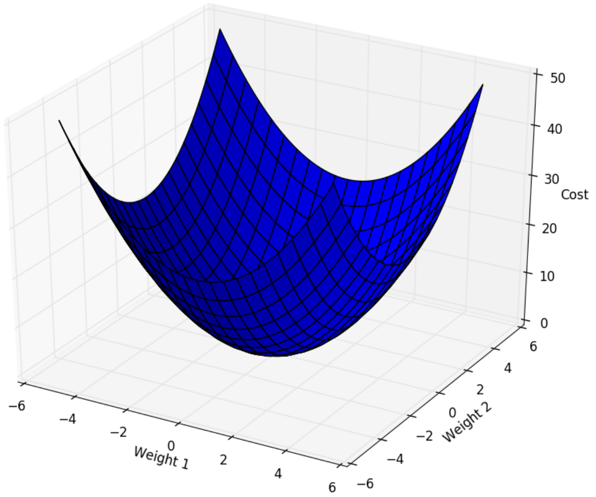
$$a_n = act(W_1 x_1 + \ldots + W_n x_n)$$

This is how each neuron computes it's own activation. It computes a weighted sum of the outputs of the previous layer (which, for the inputs, we'll call **x**), and applies an activation function (more on that later), before forwarding it on to the next layer. Each neuron in a layer has its own set of weights—so while each neuron in a layer is looking at the same inputs, their outputs will all be different.

When using a neural network to approximate a function, the data is forwarded through the network layer-by-layer until it reaches the final layer. The final layer's activations are the predictions that the network actually makes.

All this probably seems kind of magical, but it actually works. The key is finding the right set of **weights** for all of the connections to make the right decisions (this happens in a process known as **training**)—and that's what most of this post is going to be about.

When we're training the network, it's often convenient to have some metric of how good or bad we're doing; we call this metric the cost function. Generally speaking, the cost function looks at the function the network has inferred and uses it to estimate values for the data points in our training set. The discrepancies between the outputs in the estimations and the training set data points are the principle values for our cost function. When training our network, the goal will be to get the value of this cost function as low as possible (we'll see how to do that in just a bit, but for now, just focus on the intuition of what a cost function is and what it's good for). Generally speaking, the cost function *should* be more or less convex, like so:



In reality, it's impossible for any network or cost function to be truly convex. However, as we'll soon see, local minima may not be a big deal, as long as there is still a general trend for us to follow to get to the bottom. Also, notice that the cost function is parameterized by our network's weights—we control our loss function by changing the weights.

One last thing to keep in mind about the loss function is that it doesn't just have to capture how correctly your network estimates—it can specify any objective that needs to be optimized. For example, you generally want to penalize larger weights, as they could lead to overfitting. If this is the case, simply adding a **regularization term** to your cost function that expresses how big your weights will mean that, in the process of training your network, it will look for a solution that has the best estimates possible while preventing overfitting.

Now, let's take a look at how we can actually minimize the cost function during the training process to find a set of weights that work the best for our objective.

Minimizing the Cost Function

Now that we've developed a metric for "scoring" our network (which we'll denote as **J(W)**), we need to find the weights that will make that score as low as possible. If you think back to your pre-calculus days, your first instinct might be to set the derivative of the cost function to zero and solve, which would give us the locations of every minimum/maximum in the function. Unfortunately, there are a few problems with this approach:

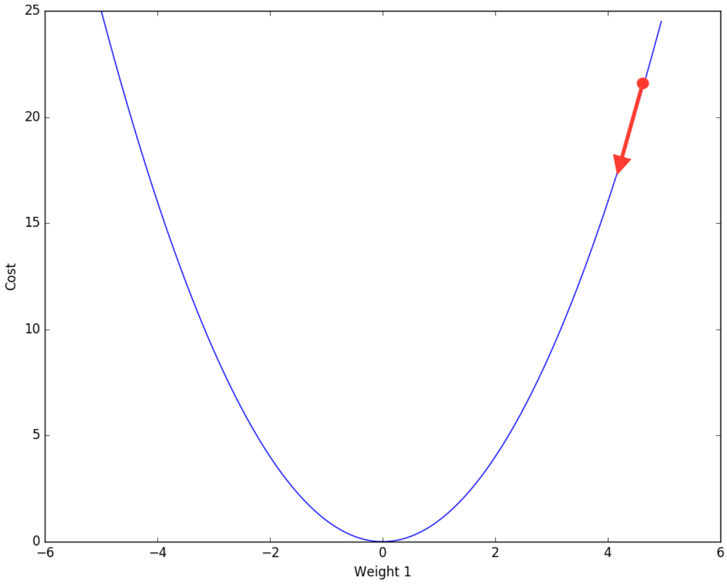
1. We don't have a simple equation for our cost function, so computing an expression for the derivative and solving it isn't trivial.
2. The function is many-dimensional (each weight gets its own dimension)—we need to find the points where all of those derivatives are zero. Also not so trivial.
3. There are lots of minimums and maximums throughout the function, and sorting out which one is the one you should be using can be computationally expensive.

Especially as the size of networks begins to scale up, solving for the weights directly becomes increasingly infeasible. Instead, we look at a different class of algorithms, called **iterative optimization**

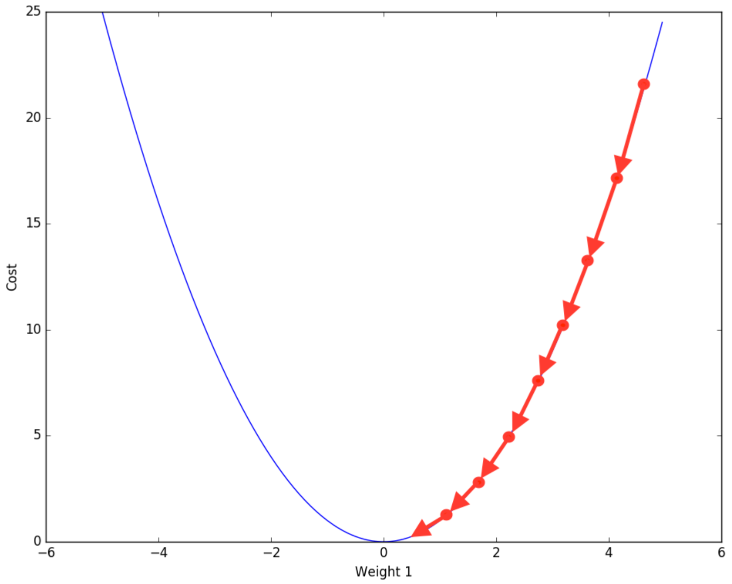
algorithms, that progressively work their way towards the optimal solution.

The most basic of these algorithms is **gradient descent**. Recall that our cost function will be *essentially* convex, and we want to get as close as possible to the global minimum. Instead of solving for it analytically, gradient descent follows the derivatives to essentially "roll" down the slope until it finds its way to the center.

Let's take the example of a single-weight neural network, whose cost function is depicted below.



We start off by initializing our weight randomly, which puts us at the red dot on the diagram above. Taking the derivative, we see the slope at this point is a pretty big positive number. We want to move closer to the center—so naturally, we should take a pretty big step in the opposite direction of the slope.



If we repeat the process enough, we soon find ourselves nearly at the bottom of our curve and much closer to the optimal weight configuration for our network.

More formally, gradient descent looks something like this:

$$W := W - \alpha \frac{\partial J}{\partial W}$$

This is the gradient descent **update rule**. It tells us how to update the weights of our network to get us closer to the minimum we're looking for.

Let's dissect. Every time we want to update our weights, we subtract the derivative of the cost function w.r.t. the weight itself, scaled by a **learning rate** , and—that's it! You'll see that as it gets closer and closer to the center, the derivative term gets smaller and smaller, converging to zero as it approaches the solution. The same process applies with networks that have tens, hundreds, thousands, or more parameters—compute the gradient of the cost function w.r.t. each of the weights, and update each of your weights accordingly.

I do want to say a few more words on the learning rate, because it's one of the more important **hyperparameters** ("settings" for your neural network) that you have control over. If the learning rate is too high, it could jump too far in the other direction, and you never get to the minimum you're searching for. Set it too low, and your network will take ages to find the right weights, or it will get stuck in a local minimum. There's no "magic number" to use when it comes to a learning rate, and it's usually best to try several and pick the one that works the best for your individual network and dataset. In practice, many choose to anneal the learning rate over time—it starts out high, because it's furthest from the solution, and decays as it gets closer.

But as it turns out, gradient descent is kind of slow. Really slow, actually. Earlier I used the analogy of the weights "rolling" down the gradient to get to the bottom, but that doesn't actually make any sense —it should pick up speed as it gets to the bottom, not slow down! Another iterative optimization algorithm, known as **momentum**, does just that. As the weights begin to "roll" down the slope, they pick up speed. When they get closer to the solution, the momentum that they picked up carries them closer to the optima while gradient descent would simply stop. As a result, training with momentum updates is both faster and can provide better results.

Here's what the update rule looks like for momentum:

$$V \coloneqq \mu V - \alpha \frac{\partial J}{\partial W}$$

$$W \coloneqq W + V$$

You might see the momentum update rule written differently depending on where you look, but the basic principle remains the same throughout.

As we train, we accumulate a "velocity" value **V**. At each training step, we update **V** with the gradient at the current position (once again scaled by the learning rate). Also notice that, with each time step, we decay velocity V by a factor **mu** (usually somewhere around .9), so that over time we lose momentum instead of bouncing around by the minimum forever. We then update our weight in the direction of the velocity, and repeat the process again. Over the first few training iterations, **V** will grow as our weights "pick up speed" and take successively bigger leaps. As we approach the minimum, our velocity stops accumulating as quickly, and eventually begins to decay, until we've essentially reached the minimum. An important thing to note is that we accumulate a velocity independently for each weight—just because one weight is changing particularly clearly doesn't mean any of the other weights need to be.

There are lots of other iterative optimization algorithms that are commonly used with neural networks, but I won't go into all of them here (if you're curious, some of the more popular ones include Adagrad and Adam). The basic principle remains the same throughout—gradually update the weights to get them closer to the minimum. But regardless of which optimization algorithm you use, we still need to be able to compute the gradient of the cost function w.r.t. each weight. But our cost function isn't a simple parabola anymore—it's a complicated, many-dimensional function with countless local optima that we need to watch out for. That's where backpropagation comes in.

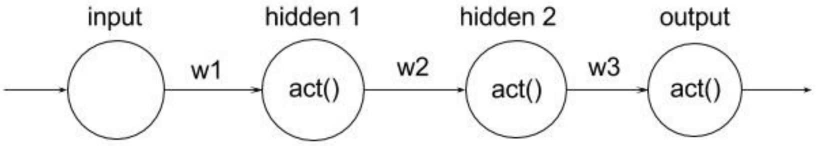
Before Backpropagation

The backpropagation algorithm was a major milestone in machine learning because, before it was discovered, optimization methods were extremely unsatisfactory. One popular method was to perturb (adjust) the weights in a random, uninformed direction (ie. increase or decrease) and see if the performance of the ANN increased. If it did not, one would attempt to either a) go in the other direction b) reduce the perturbation size or c) a combination of both. Another attempt was to use Genetic Algorithms (which became popular in AI at the same time) to evolve a high-performance neural network. In both cases, without (analytically) being informed on the correct direction, results and efficiency were suboptimal. This is where the backpropagation algorithm comes into play.

The Backpropagation Algorithm

Recall that, for any given supervised machine learning problem, we (aim to) select weights that provide the optimal estimation of a function that models our training data. In other words, we want to find a set of weights **W** that minimizes on the output of **J(W)**. We discussed the gradient descent algorithm—one where we update each weight by some negative, scalar reduction of the error derivative with respect to that weight. If we do choose to use gradient descent (or almost any other convex optimization algorithm), we need to find said derivatives in numerical form.

For other machine learning algorithms like logistic regression or linear regression, computing the derivatives is an elementary application of differentiation. This is because the outputs of these models are just the inputs multiplied by some chosen weights, and at most fed through a single activation function (the sigmoid function in logistic regression). The same, however, cannot be said for neural networks. To demonstrate this, here is a diagram of a double-layered neural network:



As you can see, each neuron is a function of the previous one connected to it. In other words, if one were to change the value of **w1**, both "hidden 1" *and* "hidden 2" (and ultimately the output) neurons would change. Because of this notion of functional dependencies, we can mathematically formulate the output as an extensive composite function:

$$output = act(w3 * hidden2)$$
$$hidden2 = act(w2 * hidden1)$$
$$hidden1 = act(w1 * input)$$

And thus:

$$output = act(w3 * act(w2 * act(w1 * input)))$$

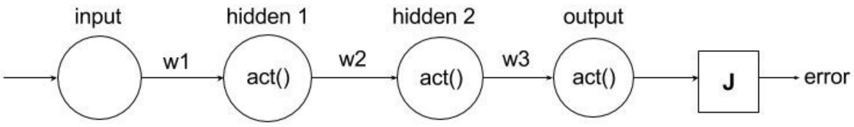
Here, the output is a composite function of the weights, inputs, and activation function(s). It is important to realize that the hidden units/nodes are simply intermediary computations that, in actuality, can be reduced down to computations of the input layer.

If we were to then take the derivative of said function with respect to some arbitrary weight (for example **w1**), we would iteratively apply the chain rule (which I'm sure you all remember from your calculus classes). The result would look similar to the following:

$$\frac{\partial}{\partial w1} output = \frac{\partial}{\partial hidden2} output * \frac{\partial}{\partial hidden1} hidden2 * \frac{\partial}{\partial w1} hidden1$$

If you fail to get an intuition of this, try researching about the chain rule.

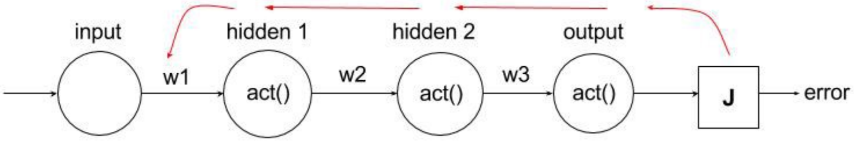
Now, let's attach a black box to the tail of our neural network. This black box will compute and return the error—using the cost function—from our output:



All we've done is add another functional dependency; our error is now a function of the output and hence a function of the input, weights, and activation function. If we were to compute the derivative of the error with any arbitrary weight (again, we'll choose **w1**), the result would be:

$$\frac{\partial error}{\partial w1} = \frac{\partial error}{\partial output} * \frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1} * \frac{\partial hidden1}{\partial w1}$$

Each of these derivatives can be simplified once we choose an activation and error function, such that the entire result would represent a numerical value. At that point, any abstraction has been removed, and the error derivative can be used in gradient descent (as discussed earlier) to iteratively improve upon the weight. We compute the error derivatives w.r.t. every other weight in the network and apply gradient descent in the same way. *This* is backpropagation—simply the computation of derivatives that are fed to a convex optimization algorithm. We call it "backpropagation" because it almost seems as if we are traversing from the output error to the weights, taking iterative steps using chain the rule until we "reach" our weight.



It's like feed-forward… but the opposite! We take the derivative of J w.r.t. the output, the output w.r.t. the hidden units, then the final hidden unit w.r.t. the weight.

When I first truly understood the backprop algorithm (just a couple of weeks ago), I was taken aback by how simple it was. Sure, the actual arithmetic/computations can be difficult, but this process is handled by our computers. In reality, backpropagation is just a rather tedious (but again, for a generalized implementation, computers will handle this) application of the chain rule. Since neural networks are convoluted multilayer machine learning model structures (at least relative to other ones), each weight "contributes" to the overall error in a more complex manner, and hence the actual derivatives require a lot of effort to produce. However, once we get past the calculus, backpropagation of neural nets is equivalent to typical gradient descent for logistic/linear regression.

Thus far, I've walked through a very abstract form of backprop for a simple neural network. However, it is unlikely that you will ever use a single-layered ANN in applications. So, now, let's make our black boxes —the activation and error functions—more concrete such that we can perform backprop on a multilayer neural net.

Recall that our error function **J(W)** will compute the "error" of our neural network based on the output predictions it produces vs. the correct *a priori* outputs we know in our training set. More formally, if we denote our predicted output estimations as vector **p**, and our actual output as vector **a**, then we can use:

$$error = J = \frac{1}{2}(\vec{p} - \vec{a})^2$$

In this case, **J** need not be a function of **W** because **p** already is. We can use vector notation here because the inputs/outputs to our neural nets our vectors/some form of tensor.

This is just one example of a possible cost function (the log-likelihood is also a popular one), and we use it because of its mathematical convenience (this is a notion one will frequently encounter in machine learning): the squared expression exaggerates poor solutions and ensures each discrepancy is positive. It will soon become clear why we multiply the expression by half.

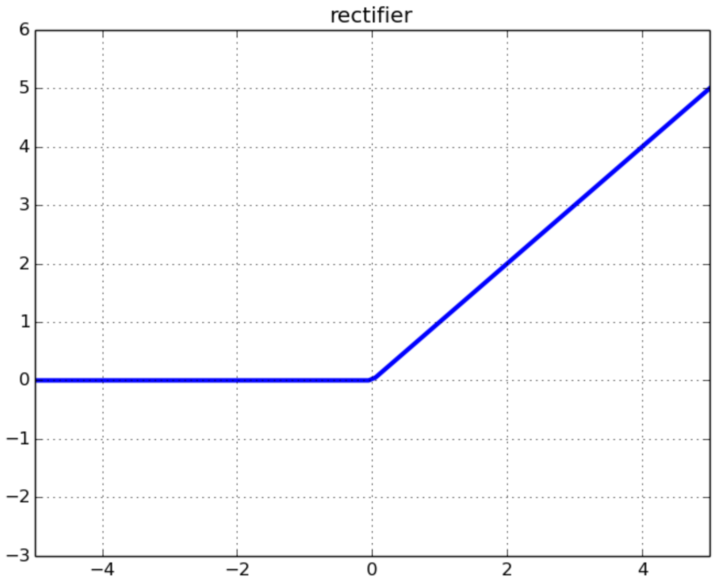
The derivative of the error w.r.t. the output was the first term in the error w.r.t. weight derivative expression we formulated earlier. Let's now compute it!

$$\frac{d}{d\vec{p}}error = \frac{d}{d\vec{p}}J = 2 * \frac{1}{2}(\vec{p} - \vec{a})^{2-1} * 1 = (\vec{p} - \vec{a})$$

With a simple application of the power and chain rule, our derivative is complete. The half gets cancelled due to the power rule.

Our result is simply our predictions take away our actual outputs.

Now, let's move on to the activation function. The activation function used depends on the context of the neural network. If we aren't in a classification context, ReLU (Rectified Linear Unit, which is zero if input is negative, and the identity function when the input is positive) is commonly used today.

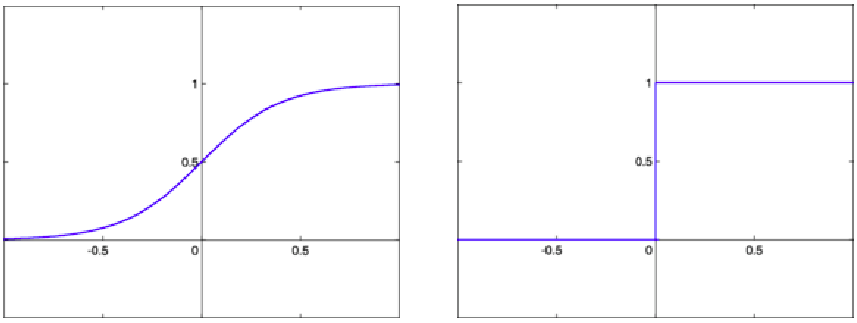


The "Rectified Linear Unit" acivation function — http://i.stack.imgur.com/8CGlM.png

If we're in a classification context (that is, predicting on a discrete state with a probability ie. if an email is spam), we can use the sigmoid or tanh (hyperbolic tangent) function such that we can "squeeze" any value into the range 0 to 1. These are used instead of a typical step function because their "smoothness" properties allows for the derivatives to be non-zero. The derivative of the step function before **and** after the origin is zero. This will pose issues when we try to update our weights (nothing much will happen!).

Now, let's say we're in a classification context and we choose to use the sigmoid function, which is of the following equation:

$$Sigmoid = S(\alpha) = \frac{1}{1+e^{-\alpha}}$$



Smooth, continuous sigmoid function on the left. Step, piece-wise function on the right. https://en.wikibooks.org/wiki/File:HardLimitFunction.png & https://en.wikibooks.org/wiki/File:SigmoidFunction.png.

As per usual, we'll compute the derivative using differentiation rules as:

$$
\begin{aligned}
&\frac{d}{d\alpha}S = \frac{d}{d\alpha}(1+e^{-\alpha})^{-1} = -1(1+e^{-\alpha})^{-1-1} * -1 * e^{-\alpha} = e^{-\alpha}(1+e^{-\alpha})^{-2} \\
&= \frac{e^{-\alpha}}{(1+e^{-\alpha})^{-2}} = \frac{1+e^{-\alpha}-1}{(1+e^{-\alpha})^{-2}} = \frac{1}{1+e^{-\alpha}}\left[\frac{1+e^{-\alpha}}{1+e^{-\alpha}} - \frac{1}{1+e^{-\alpha}}\right] \\
&= S(1-S)
\end{aligned}
$$

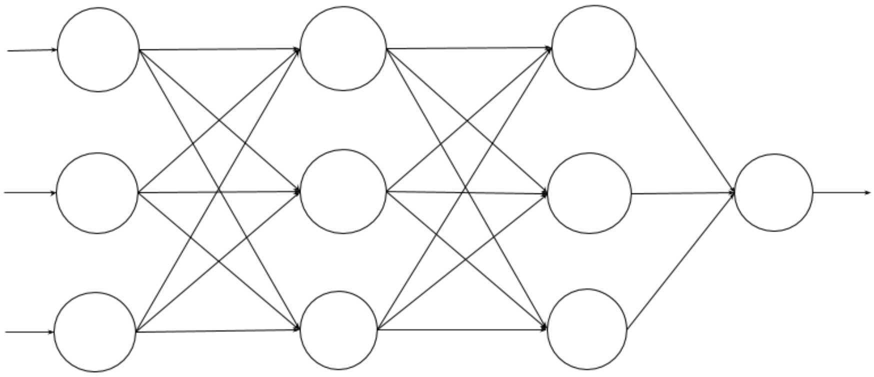
Looks confusing? Forgot differentiation? Don't worry! Just take my word for it. It's not necessary to have a complete mathematical comprehension of this derivation.

EDIT: On the 2nd line, the denominator should be raised to +2, not -2. Thanks to a reader for pointing this out.

Sidenote: ReLU activation functions are also commonly used in classification contexts. There are downsides to using the sigmoid function—particularly the "vanishing gradient" problem—which you can read more about here.

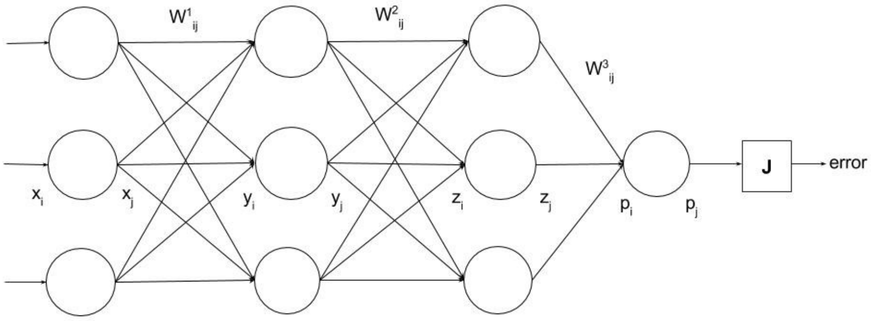
The sigmoid function is mathematically convenient (there it is again!) because we can represent its derivative in terms of the output of the function. Isn't that cool?

We are now in a good place to perform backpropagation on a multilayer neural network. Let me introduce you to the net we are going to work with:



This net is still not as complex as one you may use in your programming, but its architecture allows us to nevertheless get a good grasp on backprop. In this net, we have 3 input neurons and one output neuron. There are four layers in total: one input, one output, and two hidden layers. There are 3 neurons in each hidden layer, too (which, by the way, need not be the case). The network is fully connected; there are no missing connections. Each neuron/node (save the inputs, which are usually pre-processed anyways) is an **activity**; it is the weighted sum of the previous neurons' activities applied to the sigmoid activation function.

To perform backprop by hand, we need to introduce the different variables/states at *each* point (layer-wise) in the neural network:



It is important to note that every variable you see here is a **generalization** on the entire layer at that point. For example, when I say **x_i**, I am referring to the input to any input neuron (arbitrary value of **i**). I chose to place it in the middle of the layer for visibility purposes, but that does ***not*** mean that **x_i** refers to the middle neuron. I'll demonstrate and discuss the implications of this later on.

x refers to the input layer, **y** refers to hidden layer 1, **z** refers to hidden layer 2, and **p** refers to the prediction/output layer (which fits in nicely with the notation used in our cost function). If a variable has the subscript **i**, it means that the variable is the *input* to the relevant neuron at that layer. If a variable has the subscript **j**, it means that the variable is the *output* of the relevant neuron at that layer. For example, **x_i** refers to any input value we enter into the network. **x_j** is actually equal to **x_i**, but this is only because we choose not to use an activation function—or rather, we use the identity activation function—in the input layer's activities. We only include these two separate variables to retain consistency. **y_i** is the input to any neuron in the first hidden layer; it is the weighted sum of all previous neurons (each neuron in the input layer multiplied by the corresponding connecting weights). **y_j** is the output of any neuron at the hidden layer, so it is equal to **activation_function(y_i) = sigmoid(y_i) = sigmoid(weighted_sum_of_x_j)**. We can apply the same logic for **z** and **p**. Ultimately, **p_j** is the sigmoid output of **p_i** and hence is the output of the entire neural network that we pass to the error/cost function.

The weights are organized into three separate variables: **W1**, **W2**, and **W3**. Each **W** is a matrix (if you are not comfortable with Linear Algebra, think of a 2D array) of all the weights at the given layer. For example, **W1** are the weights that connect the input layer to the hidden layer 1. **Wlayer_ij** refers to any arbitrary, single weight at a given layer. To get an intuition of **ij** (which is really **i, j**), **Wlayer_i** are all the weights that connect arbitrary neuron **i** at a given layer to the next layer. **Wlayer_ij** (adding the **j** component) is the weight that connects arbitrary neuron **i** at a given layer to an arbitrary neuron **j** at the next layer. Essentially, **Wlayer** is a vector of **Wlayer_i**s, which is a vector of real-valued **Wlayer_ij**s.

NOTE: Please note that the **i**'s and **j**'s in the weights and other variables are ***completely different***. These indices do not correspond in any way. In fact, for **x/y/z/p**, **i** and **j** do not represent tensor indices at all, they simply represent the input and output of a neuron. **Wlayer_ij** represents an arbitrary weight at an index in a weight matrix, and **x_j/y_j/z_j/p_j** represent an arbitrary input/output point of a neuron unit.

That last part about weights was tedious! It's crucial to understand how we're separating the neural network here, especially the notion of generalizing on an entire layer, before moving forward.

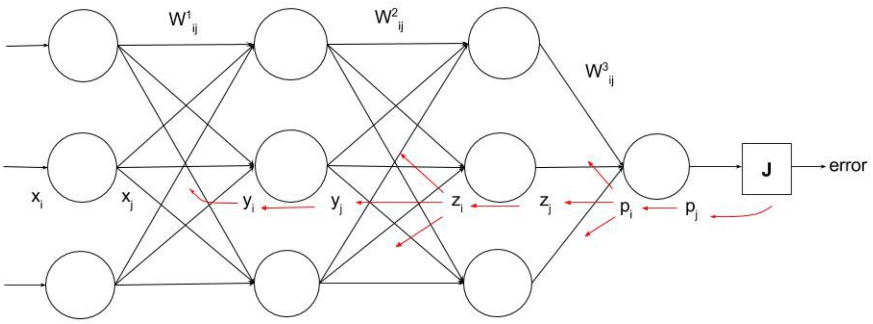
To acquire a comprehensive intuition of backpropagation, we're going to backprop this neural net as discussed before. More specifically, we're going to find the **derivative of the error w.r.t. an arbitrary weight in**

the input layer (**W1_ij**). We could find the derivative of the error w.r.t. an arbitrary weight in the first or second hidden layer, but let's go as far back as we can; the more backprop, the better!

So, mathematically, we are trying to obtain (to perform our iterative optimization algorithm with):

$$\frac{\partial error}{\partial W^1_{ij}} = \frac{\partial J}{\partial W^1_{ij}}$$

We can express this graphically/visually, using the same principles as earlier (chain rule), like so:



We backpropagate from the error all the way to the weights in the input layer. Note that the weight itself is any **arbitrary** one in that layer. In a **fully connected** neural net, we can make these generalizations considering we are consistent with our indices.

In two layers, we have three red lines pointing in three different directions, instead of just one. This is a reinforcement of (and why it is important to understand) the fact that **variable j** is just a generalization/represents any point in the layer. So, when we differentiate **p_i** with respect to the layer before that, there are **three different weights**, as I hope you can see, in **W3_ij** that contribute to the value **p_i**. There also happen to be three weights in **W3** in total, but this isn't the case for the layers before; it is only the case because layer **p** has one neuron—the output—in it. We stop backprop at the input layer and so we just point to the single weight we are looking for.

Wonderful! Now let's work out all this great stuff mathematically. Immediately, we know:

$$\frac{\partial error}{\partial W^1_{ij}} = \frac{\partial J}{\partial W^1_{ij}} = \frac{\partial J}{\partial p_j}\frac{\partial p_j}{\partial W^1_{ij}}$$

We have already established the left hand side, so now we just need to use the chain rule to simplify it further. The derivative of the error w.r.t. the weight can be written as the derivative of the error w.r.t. the output prediction multiplied by the derivative of the output prediction w.r.t. the weight. At this point, we've traversed one red line back. We know this because

$$\frac{\partial J}{\partial p_j}$$

is reducible to a numerical value. Specifically, the derivative of the error w.r.t. the output prediction is:

$$(p_j - \vec{a})$$

We know this from our manual derivation earlier.

Hence:

$$\frac{\partial J}{\partial W^1_{ij}} = (p_j - \vec{a})\frac{\partial p_j}{\partial W^1_{ij}}$$

Going one more layer backwards, we can determine that:

$$\frac{\partial p_j}{\partial W^1_{ij}} = \frac{\partial p_j}{\partial p_i}\frac{\partial p_i}{\partial W^1_{ij}}$$

In other words, the derivative of the output prediction w.r.t. the weight is the derivative of the output w.r.t. the input to the output layer (**p_i**) multiplied by the derivative of that value w.r.t. the weight. This represents our second red line. We can solve the first term like so:

$$\frac{\partial p_j}{\partial p_i} = p_j(1 - p_j)$$

This corresponds with the derivative of the sigmoid function we solved earlier, which was equal to the output multiplied by one minus the output. In this case, **p_j** *is* the output of the sigmoid function. Now, we have:

$$\frac{\partial p_j}{\partial W^1_{ij}} = p_j(1 - p_j)\frac{\partial p_i}{\partial W^1_{ij}}$$

$$\frac{\partial J}{\partial W^1_{ij}} = (p_j - \vec{a}) * p_j(1 - p_j) * \frac{\partial p_i}{\partial W^1_{ij}}$$

Let's move on to the third red line(s). This one is interesting because we begin to "spread" out. Since there are multiple different weights that contribute to the value of **p_i**, we need to take into account their individual "pull" factors into our derivative:

$$\frac{\partial p_i}{\partial W^1_{ij}} = \sum_j \frac{\partial p_i}{\partial z_j}\frac{\partial z_j}{\partial W^1_{ij}}$$

If you're a mathematician, this notation may irk you slightly; sorry if that's the case! In computer science, we tend to stray from the notion of completely legal mathematical expressions. This is yet again again another reason why it's key to understand the role of layer generalization; **z_j** here is not just referring to the middle neuron, it's referring to an arbitrary neuron. The actual *value* of **j** in the summation is not changing (it's not even an index or a value in the first place), and we don't really consider it. It's less of a mathematical expression and more of a ***statement*** that we will iterate through each generalized neuron **z_j** and use it. In other words, we iterate over the derivative terms and sum them up using **z_1**, **z_2**, and **z_3**. Before, we could write **p_j** as any single value because the output layer just contains one node; there is just one **p_j**. But we see here that this is no longer the case. We have multiple **z_j** values, and **p_i** is functionally dependent on each of these **z_j** values. So, when we traverse from **p_j** to the preceding layer, we need to consider each contribution from layer **z** to **p_j** separately and add them up to create one total contribution. There's no upper bound to the summation; we just assume that we start at zero and end at our maximum value for the number of neurons in the layer. Please again note that the same changes are not reflected in **W1_ij**, where **j** refers to an entirely different thing. Instead, we're just ***stating*** that we will use the different **z_j** neurons in layer **z**.

Since **p_i** is a summation of each weight multiplied by each **z_j** (weighted sum), if we were to take the derivative of **p_i** with any arbitrary **z_j**, the result would be the connecting weight since said weight would be the coefficient of the term (derivative of m*x w.r.t. x is just m):

$$\frac{\partial p_i}{\partial z_j} = W^3_{ij}$$

W3_ij is loosely defined here. **ij** still refers to any arbitrary weight—where **ij** are still separate from the **j** used in **p_i/z_j**—but again, as computer scientists and not mathematicians, we need not be pedantic about the legality and intricacy of expressions; we just need an intuition of what the expressions imply/mean. It's almost a succinct form of psuedo-code! So, even though this defines an arbitrary weight, we know it means the **connecting** weight. We can also see this from the diagram: when we walk from **p_j** to an arbitrary **z_j**, we walk along the connecting weight. So now, we have:

$$\frac{\partial p_i}{\partial W^1_{ij}} = W^3_{ij} * \frac{\partial z_j}{\partial W^1_{ij}}$$

At this point, I like to continue playing the "reduction test". The reduction test states that, if we can further simplify a derivative term, we still have more backprop to do. Since we can't yet quite put the derivative of **z_j** w.r.t. **W1_ij** into a numerical term, let's keep going (and fast-forward a bit). Using chain rule, we follow the fourth line back to determine that:

$$\frac{\partial z_j}{\partial W^1_{ij}} = \frac{\partial z_j}{\partial z_i}\frac{\partial z_i}{\partial W^1_{ij}} = z_j(1-z_j) * \frac{\partial z_i}{\partial W^1_{ij}}$$

Since **z_j** is the sigmoid of **z_i**, we use the same logic as the previous layer and apply the sigmoid derivative. The derivative of **z_i** w.r.t. **W1_ij**, demonstrated by the fifth line(s) back, requires the same idea of "spreading out" and summation of contributions:

$$\frac{\partial z_i}{\partial W^1_{ij}} = \sum_j \frac{\partial z_i}{\partial y_j}\frac{\partial y_j}{\partial W^1_{ij}} = \sum_j W^2_{ij}\frac{\partial y_j}{\partial W^1_{ij}}$$

Briefly, since **z_i** is the weighted sum of each **y_j** in **y**, we sum over the derivatives which, similar to before, simplifies to the relevant connecting weights in the preceding layer (**W2** in this case).

We're almost there, let's go further; there's still more reduction to do:

$$\frac{\partial y_j}{\partial W^1_{ij}} = \frac{\partial y_j}{\partial y_i}\frac{\partial y_i}{\partial W^1_{ij}} = y_j(1-y_j)\frac{\partial y_i}{\partial W^1_{ij}}$$

We have, of course, another sigmoid activation function to deal with. This is the sixth red line. Notice, now, that we have just *one* line remaining. In fact, our last derivative term here passes (or rather, fails) the reduction test! The last line traverses from the input at **y_i** to **x_j**, walking along **W1_ij**. Wait a second—is this not what we are attempting to backprop to? Yes, it is! Since we are, for the first time, *directly* deriving **y_i** w.r.t. the weight **W1_ij**, we can think of the coefficient of **W1_ij** as being **x_j** in our weighted sum (instead of the vice versa as used previously). Hence, the simplification follows:

$$\frac{\partial y_i}{\partial W^1_{ij}} = \sum_j x_j$$

Of course, since each **x_j** in layer **x** contributes to the weighted sum **y_i**, we sum over the effects. And that's it! We can't reduce any further from here. Now, let's tie all these individual expressions together:

$$\frac{\partial J}{\partial W^{layer}_{ij}} = (p_j - \vec{a}) * p_j(1-p_j) * \sum_j \left[W^3_{ij} * z_j(1-z_j) * \sum_j \left[W^2_{ij} * y_j(1-y_j) * \sum_j x_j \right]\right]$$

Our final expression for the derivative of the error w.r.t. any weight in **W1**

EDIT: The denominator on the left hand side should say **dW^{1}ij** instead of "layer".

With no more partial derivative terms left, our work is complete! This gives us the derivative of the error w.r.t. any arbitrary weight in the input layer/**W1**. That was a lot of work—maybe now we can sympathize with the poor computers!

Something you should notice is that values such as **p_j, a, z_j, y_j, x_j** etc. are the values of the network at the different points. But where do they come from? Actually, we would need to perform a feed-forward of the neural network first and then capture these variables.

Neural Network Training Overview

Our task is to now perform Gradient Descent to train the neural net:

$$repeat\ \ until\ \ \frac{\partial J}{\partial W^{layer}{}_{ij}} \rightarrow 0:$$

$$\{ \quad W^{layer}{}_{ij} := W^{layer}{}_{ij} - \alpha \frac{\partial J}{\partial W^{layer}{}_{ij}}$$

We perform gradient descent on each weight in each layer. Notice that the resulting gradient should change each time because the weight itself changes, (and as a result, the performance and output of the entire net should change) even if it's a small perturbation. This means that, at each update, we need to do a feed-forward of the neural net. Not just once before, but once *each iteration*.

These are then the steps to train an entire neural network:

1. Create our connected neural network and prepare training data
2. Initialize all the weights to random values

It's important to note that one must not initialize the weights to zero, similar to what may be done in other machine learning algorithms. If weights *are* initialized to zero, after each update, the outgoing weights of each neuron will be identical, because the gradients will be identical (which can be proved). Because of this, the proceeding hidden units will remain the same value and will continue to follow each other. Ultimately, this means that our training will become extremely constrained (due to the "symmetry"), and we won't be able to build interesting functions. Also, neural networks may get stuck at local optima (places where the gradient is zero but are not the global minima), so random weight initialization allows one to hopefully have a chance of circumventing that by starting at many different random values.

3. Perform one feed-forward using the training data

4. Perform backpropagation to get the error derivatives w.r.t. each and every weight in the neural network

5. Perform gradient descent to update each weight by the negative scalar reduction (w.r.t. some learning rate alpha) of the respective error derivative. Increment the number of iterations.

6. If we have converged (in reality, though, we just stop when we have reached the number of maximum iterations) training is complete. Else, repeat starting at step 3.

If we initialize our weights randomly (and not to zero) and then perform gradient descent with derivatives computed from backpropagation, we should expect to train a neural network in no time! I hope this example brought clarity to how backprop works and the intuition behind it. If you didn't understand the intricacies of the example but understand and appreciate the concept of backprop as a whole, you're still in a great place! Next we'll go ahead and explain backprop code that works on any generalized architecture of a neural network using the ReLU activation function.

Implementing Backpropagation

Now that we've developed the math and intuition behind backpropagation, let's try to implement it. We'll divide our implementation into three distinct steps:

1. **Feed-forward**. In this step, we take the inputs and forward them through the network, layer by layer, to generate the output activations (as well as all of the activations in the hidden layers). When we are actually using our network (rather than training it), this is the only step we'll need to perform.
2. **Backpropagation**. Here, we'll take our error function and compute the weight gradients for each layer. We'll use the algorithm just described to compute the derivative of the cost function w.r.t. each of the weights in our network, which will in turn allow us to complete step 3.
3. **Weight update**. Finally, we'll use the gradients computed in step 2 to update our weights. We can use any of the update rules discussed previously during this step (gradient descent, momentum, and so on).

Let's start off by defining what the API we're implementing looks like. We'll define our network as a series of Layer instances that our data passes through—this means that instead of modeling each individual neuron, we group neurons from a single layer together. This makes it a bit easier to reason about larger networks, but also makes the actual computations faster (as we'll see shortly). Also—we're going to write the code in Python.

Each layer will have the following API:

```
class ReLULayer(object):
    def __init__(self, size_in, size_out):
        # Initialize this layer with random weights and any
other
        # parameters we may need.
        pass

    def forward(self, in_act):
        # Compute the outgoing activations from this layer,
given
        # the activations from the previous layer.
        pass

    def backward(self, out_grad):
        # out_grad is the derivative of the cost function
w.r.t. the
        # inputs to all of the neurons for the following
layer. We
        # need to compute the gradient of our own weights,
and
        # return another the gradient of the inputs to this
layer to
        # continue the backpropagation.
        pass

    def update(self, alpha):
        # Perform the actual weight update step.
        pass
```

(This isn't great API design—ideally, we would decouple the backprop and weight update from the rest of the object, so the specific algorithm we use for updating weights isn't tied to the layer itself. But that's not the point, so we'll stick with this design for the purposes of explaining how backpropagation works in a real-life scenario. Also: we'll be using numpy throughout the implementation. It's an awesome tool for mathematical operations in Python (especially tensor based ones), but we don't have the time to get into how it works—if you want a good introduction, here ya' go.)

We can start by implementing the weight initialization. As it turns out, how you initialize your weights is actually kind of a big deal for both network performance and convergence rates. Here's how we'll initialize our weights:

```
self.W = np.random.randn(self.size_in, self.size_out) *
np.sqrt(2.0/self.size_in)
```

This initializes a weight matrix of the appropriate dimensions with random values sampled from a normal distribution. We then scale it rad(2/self.size_in), giving us a variance of 2/self.size_in (derivation here).

And that's all we need for layer initialization! Let's move on to implementing our first objective—feed-forward. This is actually pretty simple—a dot product of our input activations with the weight matrix, followed by our activation function, will give us the activations we need. The dot product part should make intuitive sense; if it doesn't, you should sit down and try to work through it on a piece of paper. This is where the performance gains of grouping neurons into layers comes from: instead of keeping an individual weight vector for each neuron, and performing a series of vector dot products, we can just do a single matrix operation (which, thanks to the wonders of modern processors, is significantly faster). In fact, we can compute all of the activations from a layer in just two lines:

```
# Compute weighted sum for each neuron
self.out_act = np.dot(self.in_act, self.W)

# Activation function (any sum < 0 is capped at 0)
self.out_act[self.out_act < 0] = 0

return self.out_act
```

Simple enough. Let's move on to backpropagation.

This one's a bit more involved. First, we compute the derivative of the output w.r.t. the weights, then the derivative of the cost w.r.t. the output, followed by chain rule to get the derivative of the cost w.r.t. the weights.

Let's start with the first part—the derivative of the output w.r.t. the weights. That should be simple enough; because you're multiplying the weight by the corresponding input activation, the derivative will just be the corresponding input activation.

```
output_wrt_weights = np.ones(self.W.shape) * self.in_act[:,
None]
```

Except, because we’re using the ReLU activation function, the weights have no effect if the corresponding output is < 0 (because it gets capped anyway). This should take care of that hiccup:

```
output_wrt_weights[:, self.out_act < 0] = 0
```

(More formally, you’re multiplying by the derivative of the activation function, which is 0 when the activation is < 0 and 1 elsewhere.)

Let’s take a brief detour to talk about the **out_grad** parameter that our **backward** method gets. Let’s say we have a network with two layers: the first has **m** neurons, and the second has **n**. Each of the **m** neurons produces an activation, and each of the **n** neurons looks at each of the **m** activations. The **out_grad** parameter is an **m** x **n** matrix of how each **m** affects each of the **n** neurons it feeds into.

Now, we need the derivative of the cost w.r.t. each of the outputs— which is essentially the **out_grad** parameter we’re given! We just need to sum up each row of the matrix we’re given, as per the backpropagation formula.

```
cost_wrt_output = np.sum(np.atleast_2d(grad), axis=1)
```

Finally, we end up with something like this:

```
self.dW = cost_wrt_weights
```

Now, we need to compute the derivative of our inputs to pass along to the next layer. We can perform a similar chain rule—derivative of the output w.r.t. the inputs times the derivative of the cost w.r.t. the outputs.

```
output_wrt_inputs = self.W
output_wrt_inputs[:, self.out_act < 0] = 0

cost_wrt_inputs = cost_wrt_output * output_wrt_inputs
```

```
return cost_wrt_inputs
```

And that’s it for the backpropagation step.

The final step is the weight update. Assuming we’re sticking with gradient descent for this example, this can be a simple one-liner:

```
self.W = self.W — self.dW * alpha
```

To actually train our network, we take one of our training samples and call **forward** on each layer consecutively, passing the output of the previous layer as the input of the following layer. We compute dJ, passing that as the **out_grad** parameter to the last layer’s **backward** method. We call **backward** on each of the layers in reverse order, this time passing the output of the further layer as **out_grad** to the previous layer. Finally, we call **update** on each of our layers and repeat.

There’s one last detail that we should include, which is the concept of a **bias** (akin to that of a constant term in any given equation). Notice that, with our current implementation, the activation of a neuron is determined solely based on the activations of the previous layer. There’s no bias term that can shift the activation up or down independent of the inputs. A bias term isn’t strictly necessary—in fact, if you train your network as-is, it would probably still work fine. But if you do need a bias term, the code stays almost the same—the only difference is that you need to add a column of 1s to the incoming activations, and update your weight matrix accordingly, so one of your weights gets treated as a bias term. The only other difference is that, when returning **cost_wrt_inputs**, you can cut out the first row—nobody cares about the gradients associated with the bias term because the previous layer has no say in the activation of the bias neuron.

Implementing backpropagation can be kind of tricky, so it’s often a good idea to check your implementation. You can do so by computing the gradient numerically (by literally perturbing the weight and calculating the difference in your cost function) and comparing it to your backpropagation-computed gradient. This **gradient check** doesn’t need to be run once you’ve verified your implementation, but it could save a lot of time tracking down potential problems with your network.

Nowadays, you often don't even need to implement a neural network on your own, as libraries such as Caffe, Torch, or TensorFlow will have implementations ready to go. That being said, it's often a good idea to try implementing it on your own to get a better grasp of how everything works under the hood.

Learning More about Neural Networks

Intrigued? Looking to learn more about neural networks? Here are some great online classes to get you started:

Stanford's CS231n. Although it's technically about convolutional neural networks, the class provides an excellent introduction to and survey of neural networks in general. Class videos, notes, and assignments are all posted here, and if you have the patience for it I would strongly recommend walking through the assignments so you can really get to know what you're learning.

MIT 6.034. This class, taught by Prof. Patrick Henry Winston, explores many different algorithms and disciplines in Artificial Intelligence. There's a great lecture on backprop that I actually used as a stepping stone to getting setup writing this article. I also learned genetic algorithms from Prof. Winston—he's a great teacher!

. . .

We hope that, if you visited this article without knowing how the backpropagation algorithm works, you are reading this with an (at least rudimentary) mathematical or conceptual intuition of it. Writing and conveying such a complex algorithm to a supposed beginner has proven to be an extremely difficult task for us, but it's helped us truly understand what we've been learning about. With greater knowledge in a fundamental area of machine learning, we are now excited to take a look at new, interesting algorithms and disciplines in the field. We are looking forward to continue documenting these endeavors **together**.