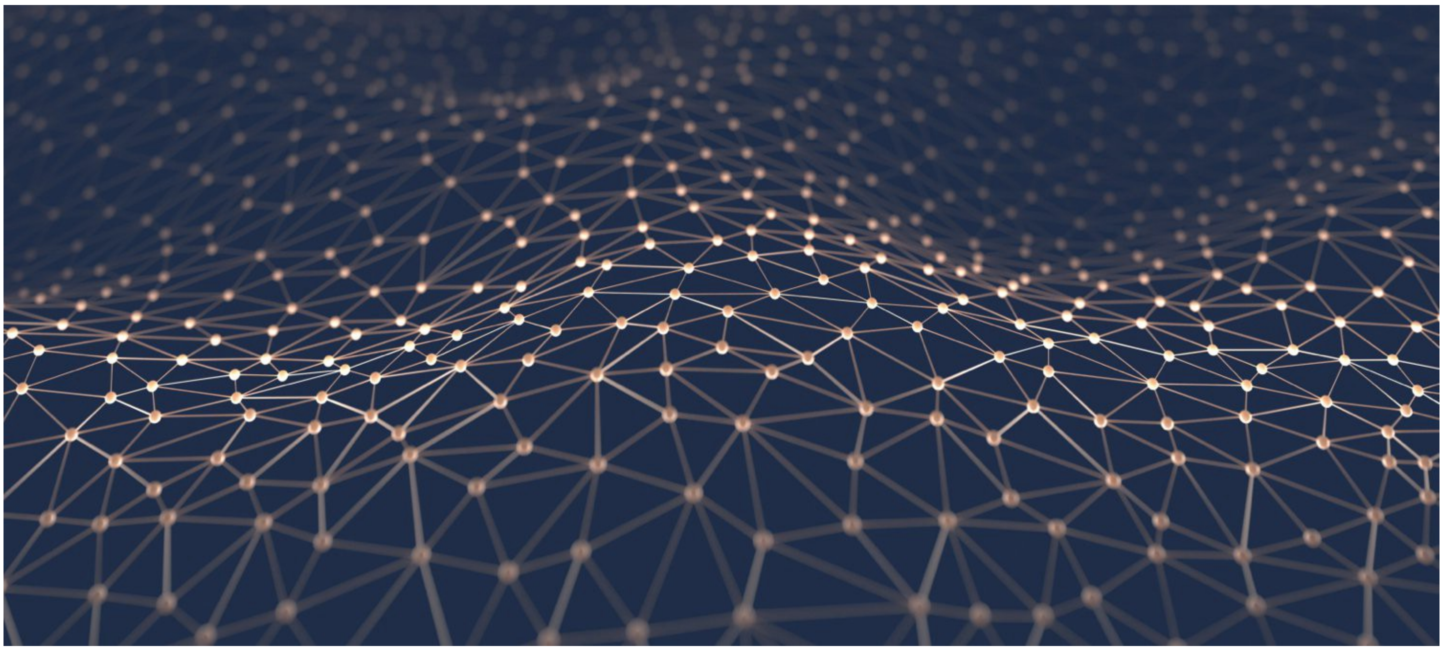


Rohan #4: The vanishing gradient problem

Oh no — an obstacle to deep learning!

Rohan Kapur Follow
Mar 16, 2016 · 11 min read

. . .

This is the fourth entry in my journey to extend my knowledge of Artificial Intelligence in the year of 2016. Learn more about my motives in this introduction post.

This post assumes sound knowledge of neural networks and the backpropagation algorithm. If that does not describe your current understanding, no worries. You can catch up here.

. . .

It's summer time, and you recently **read my Medium post** on the backpropagation algorithm. Excited to hack away at your own implementation, you create a deep, multi-layer neural network and begin running the program. But, to your avail, it's either taking forever to train or not performing accurately. Why? You thought neural networks were state of the art, didn't you!

I was, in fact, stuck on this issue for a while.

Here's what we know about iterative optimization algorithms: they slowly make their way to the local optima by perturbing weights in a direction inferred from the gradient such that the cost function's output is decreased. The gradient descent algorithm, in specific, updates the weights by the negative of the gradient multiplied by some small (between 0 and 1) scalar value.

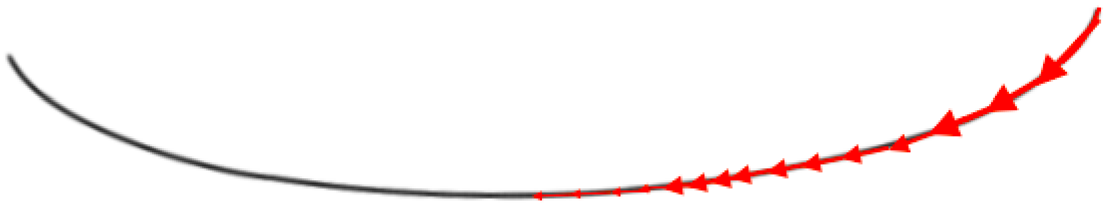
$$\textit{repeat until } \frac{\partial J}{\partial {W^{layer}}_{ij}} \rightarrow 0:$$

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

As you can see, we are to *"repeat"* until convergence. In reality, though, we actually set a hyper-parameter for the number of max iterations. If the number of iterations is too small for certain deep neural nets, we will have inaccurate results. If the number is too large, the training duration will become infeasibly long. It's an unsettling tradeoff between training time and accuracy.

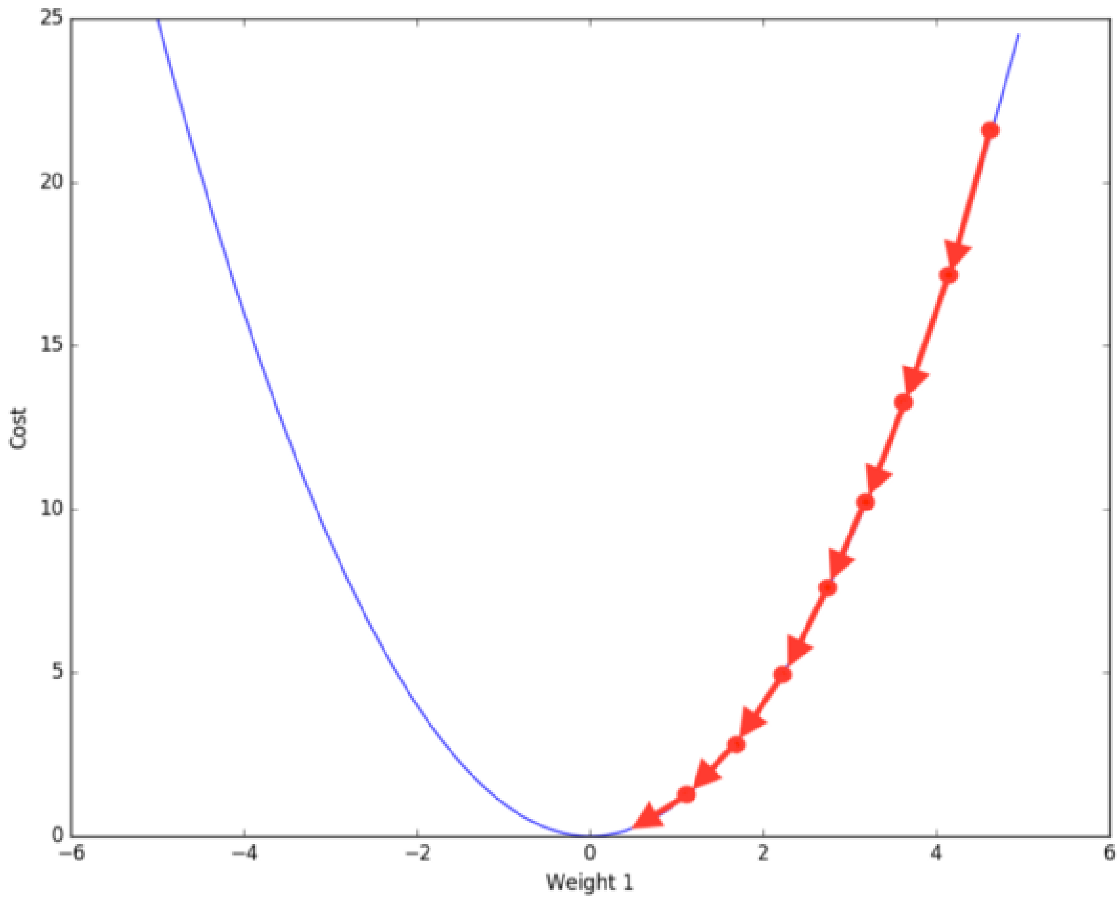
So, why is this the case? Well, put simply, if the gradient at each step is too small, then greater repetitions will be needed until convergence, because the weight is not changing enough at each iteration. Or, the weights will not move as close to the minimum (versus greater gradients) in the set number of iterations. And with *really really* small gradients, this becomes a problem. It becomes infeasible to train neural networks, and they start predicting poorly.

You would get an elongated cost function, like the following:



Visual representation of the many more updates needed with flat gradients and an elongated cost function

Compare that shape to the following more optimal one, for example:



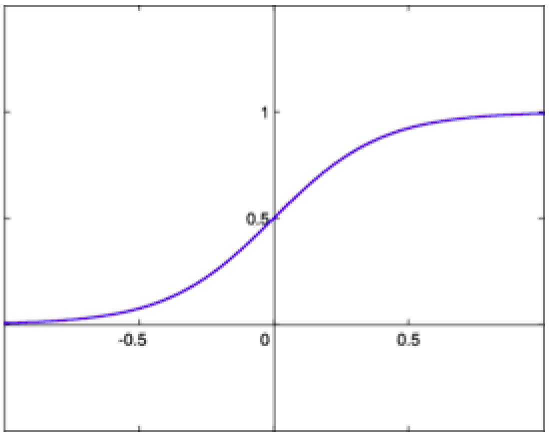
Since the latter has larger gradients, gradient descent can converge much quicker.

This isn't the only issue. When gradients become too small, it also becomes difficult to represent the numerical value in computers. We call this underflow — the opposite of overflow.

Okay. Small gradients = bad news, got it. The question, then, is: does this problem exist? In many cases it indeed does, and we call it the **vanishing gradient problem**.

Recall the sigmoid function, one that was almost always used as an activation function for ANNs in a classification context:

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



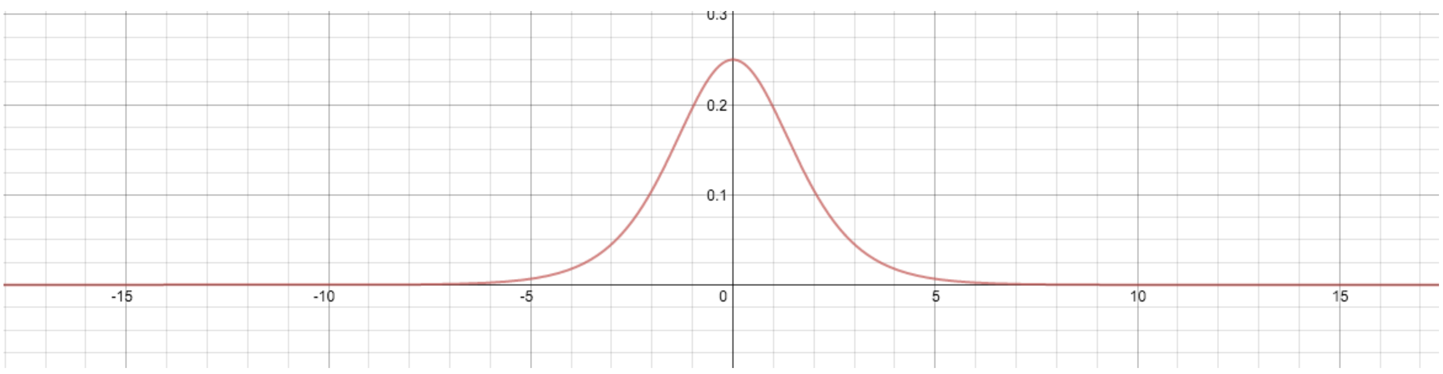
The sigmoid function is useful because it "squeezes" any input value into an output range of (0, 1) (where it asymptotes). This is perfect for representations of probabilities and classification. The sigmoid function, along with the tanh function, though, have lost popularity in recent years. Why? Because they suffer from the vanishing gradient problem!

Let's take a look at the derivative of the sigmoid function. I've precomputed it for you:

$$\frac{1}{1+e^{-\alpha}}\left[1-\frac{1}{1+e^{-\alpha}}\right]$$

In other words, simply S(1-s)

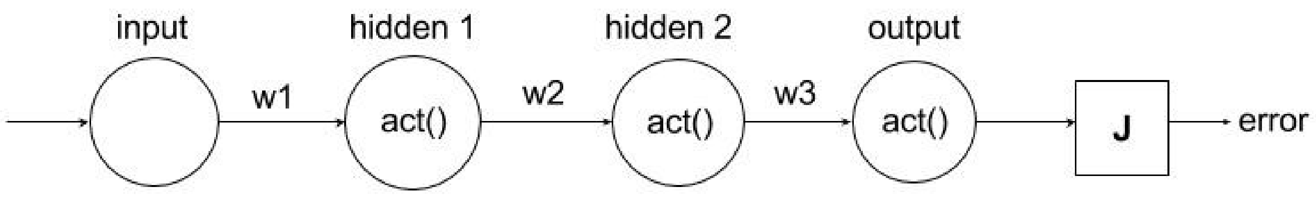
Now, let's graph that:



Courtesy of https://www.desmos.com/calculator

Looks decent, you say. Look closer. The maximum point of the function is 1/4, and the function horizontally asymptotes at 0. In other words, the output of the derivative of the cost function is always between 0 and 1/4. In mathematical terms, the range is (0, 1/4]. Keep that in mind.

Now, let's move on to the structure of a neural network and backprop and their implications on the size of gradients.



Recall this general structure for a simple, univariate neural network. Each neuron or "activity" is derived from the previous: it is the previous activity multiplied by some weight and then fed through an activation function. The input, of course, is the notable exception. The error box **J** at the end returns the aggregate error of our system. We then perform backpropagation to modify the weights through gradient descent such that the output of **J** is minimized.

To calculate the derivative to the first weight, we used the chain rule to "backpropagate" like so:

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

We then use these derivatives to iteratively make our way the minimum point using gradient descent.

Let’s focus on these individual derivatives:

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

With regards to the first derivative — since the output is the activation of the 2nd hidden unit, and we are using the sigmoid function as our activation function, then the derivative of the output is going to **contain** the derivative of the sigmoid function. In specific, the resulting expression will be:

$$z_1 = hidden2 * w3$$
$$\frac{\partial output}{\partial hidden2} = \frac{\partial Sigmoid(z_1)}{\partial z_1} w3$$

From the output to hidden2

The same applies for the second:

$$z_2 = hidden1 * w2$$
$$\frac{\partial hidden2}{\partial hidden1} = \frac{\partial Sigmoid(z_2)}{\partial z_2} w2$$

From hidden2 to hidden1

In both cases, the derivative contains the derivative of the sigmoid function. Now, let’s put those together:

$$\frac{\partial output}{\partial hidden2}\frac{\partial hidden2}{\partial hidden1} = \frac{\partial Sigmoid(z_1)}{\partial z_1} w3 * \frac{\partial Sigmoid(z_2)}{\partial z_2} w2$$

Recall that the derivative of the sigmoid function outputs values between 0 and 1/4. By multiplying these two derivatives together, we are multiplying two values in the range (0, 1/4]. Any two numbers between 0 and 1 multiplied with each other will simply **result in a smaller value**. For example, 1/3 × 1/3 is 1/9.

The standard approach to weight initialization in a typical neural network is to choose weights using a Gaussian with mean 0 and standard deviation 1. Hence, the weights in a neural network will also usually be between -1 and 1.

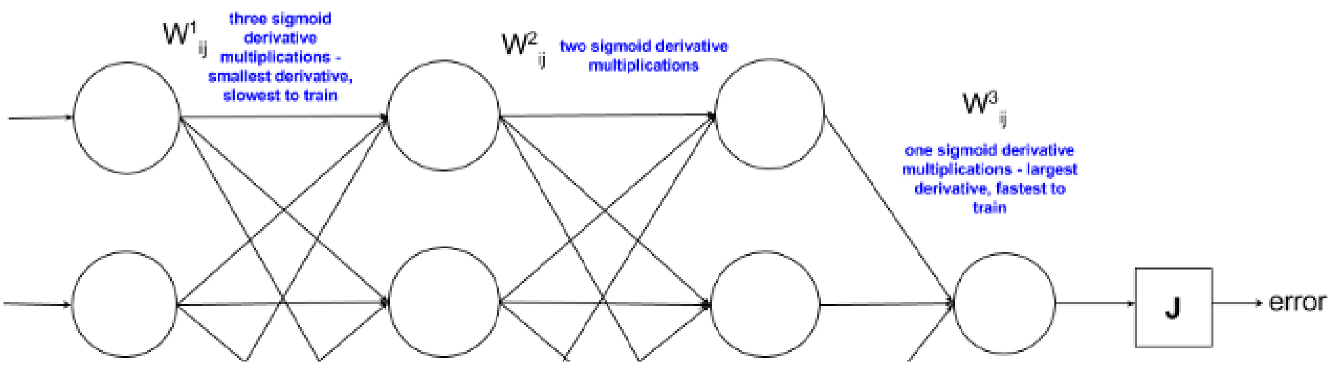
Now, look at the magnitude of the terms in our expression:

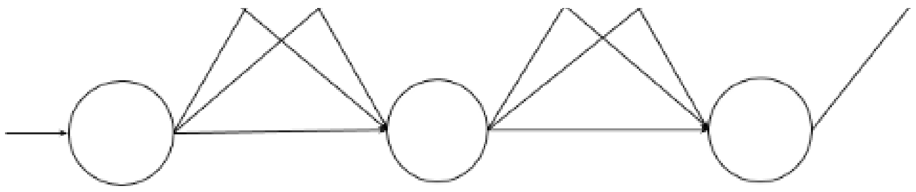
$$\frac{\partial output}{\partial hidden2}\frac{\partial hidden2}{\partial hidden1} = \overset{<1/4}{\frac{\partial Sigmoid(z_1)}{\partial z_1}}\,\overset{<1}{w3} * \overset{<1/4}{\frac{\partial Sigmoid(z_2)}{\partial z_2}}\,\overset{<1}{w2}$$

At this point, we are multiplying four values which are between 0 and 1. That will become small **very** fast. And even if this weight initialization technique is not employed, the vanishing gradient problem will most likely still occur. Many of these sigmoid derivatives multiplied together would be small enough to compensate for the other weights, and the other weights may want to shift into a range below 1.

This neural network isn't that deep. But imagine a deeper one used in an industrial application. As we backpropagate further back, we'd have many more small numbers partaking in a product, creating an even tinier gradient! Thus, with deep neural nets, the vanishing gradient problem becomes a major concern. Sidenote: we can even have exploding gradients, if the gradients were bigger than 1 (a bunch of numbers bigger than 1 multiplied together is going to give a *huge* result!) These aren't good either — they wildly overshoot.

Now, let's look at a typical ANN:


W^1_{ij}
three sigmoid derivative multiplications - smallest derivative, slowest to train
W^2_{ij}
two sigmoid derivative multiplications
W^3_{ij}
one sigmoid derivative multiplications - largest derivative, fastest to train
J
error



As you can see, the first layer is the furthest back from the error, so the derivative (using the chain rule) will be a longer expression and hence will contain more sigmoid derivatives, ending up smaller. Because of this, the first layers are the **slowest to train**. But there's another issue with this: since the latter layers (and most notably the output) is functionally dependent on the earlier layers, inaccurate early layers will cause the latter layers to simply build on this inaccuracy, corrupting the entire neural net. Take convolutional neural nets as an example; their early layers perform more high-level feature detection such that the latter layers can analyze them further to make a choice. Also, because of the small steps, gradient descent may converge at a local minimum.

This is why neural networks (especially deep ones), at first, failed to become popular. Training the earlier layers correctly was the basis for the entire network, but that proved too difficult and infeasible because of the commonly used activation functions and available hardware.

How do we solve this? Well, it's pretty clear that the root cause is the nature of the sigmoid activation function derivative. This also happened in the most popular alternative, the tanh function. Until recently, not many other activation functions were thought of / used. But now, the sigmoid and tanh functions have been declining in popularity in the light of the ReLU activation function.

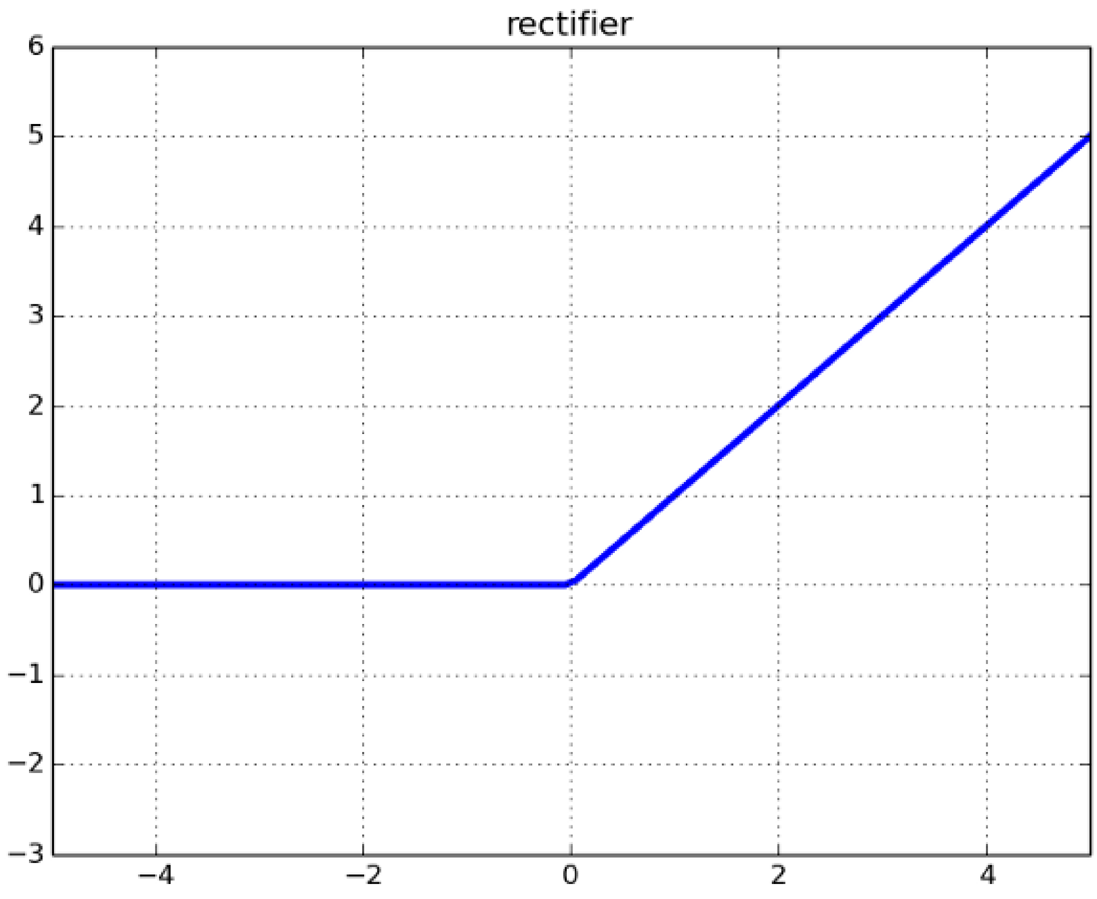
What is the ReLU — Rectified Linear Unit — function anyways? It is a piecewise function that corresponds to:

$$RELU(x) = \begin{cases} 0 \ if \ x < 0 \\ x \ if \ x >= 0 \end{cases}$$

Another way of writing the ReLU function is like so:

$$RELU(x) = \max(0, x)$$

In other words, when the input is smaller than zero, the function will output zero. Else, the function will mimic the identity function. It’s very fast to compute the ReLU function.



The “Rectified Linear Unit” acivation function — http://i.stack.imgur.com/8CGIM.png

It doesn’t take a genius to calculate the derivative of this function. When the input is smaller than zero, the output is always equal to zero, and so the rate of change of the function is zero. When the input is greater or equal to zero, the output is simply the input, and hence the derivative is equal to one:

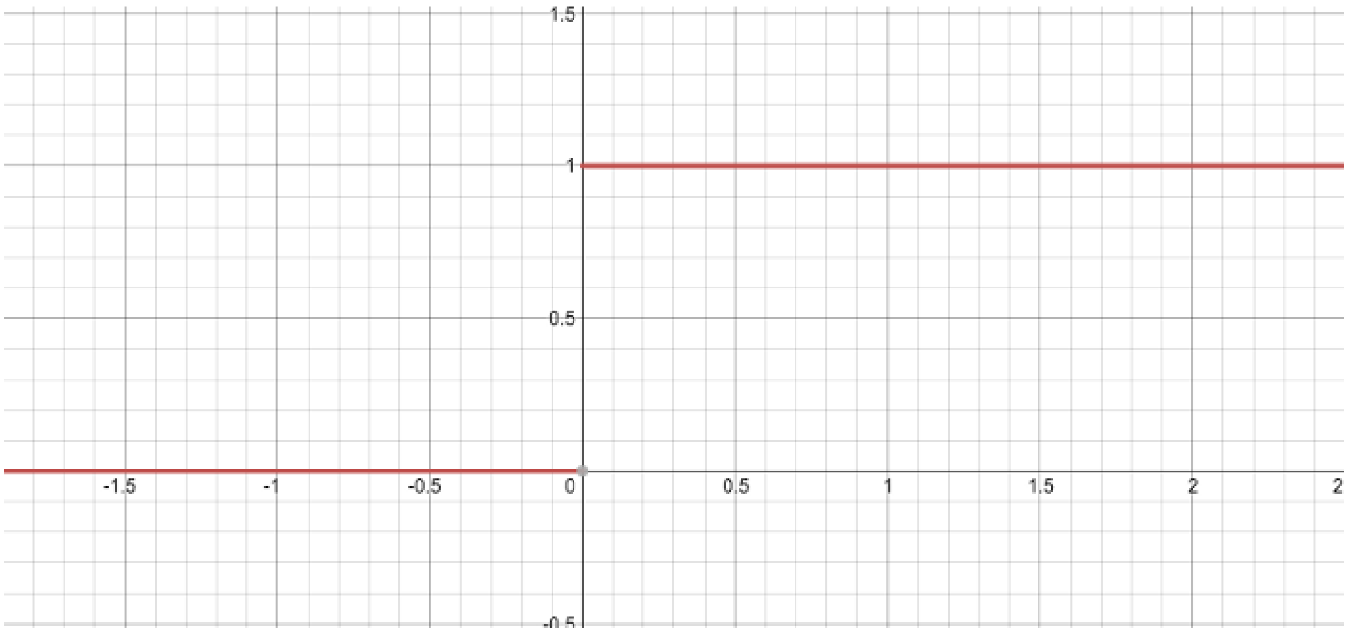
$$\frac{d}{dx}0 = 0$$

$$\frac{d}{dx}x = 1$$

$$\frac{d}{dx}RELU(x)=\begin{cases}0 & if \quad x<0\\ 1 & if \quad x>0\end{cases}$$

The function is not differentiable at zero, though

If we were to graph this derivative, it would look exactly like a typical step function:



Hey… it's a step function!

So, it's solved! Our derivatives will no longer vanish, because the activation function's derivative isn't bounded by the range (0, 1).

ReLUs have one caveat though: they "die" (output zero) when the input to it is negative. This can, in many cases, completely block backpropagation because the gradients will just be zero after one negative value has been inputted to the ReLU function. This would also be an issue if a large negative bias term / constant term is learned — the weighted sum fed into neurons may end up being negative because the positive weights cannot compensate for the significance of the bias term. Negative weights also come to mind, or negative input (or some combination that gives a negative weighted sum). The dead ReLU will hence output the same value for almost all of your activities — zero. ReLUs cannot "recover" from this problem because they will not modify the weights in anyway, since not only is the output for any negative input zero, the **derivative** is too. No updates will be made to modify the (for example) bias term to be a lesser magnitude of negative such that the neural net can escape from corruption of the entire network. It doesn't happen *all* that often that the weighted sum ends up negative, though; and we can indeed initialize weights to be only positive and/or normalize input between 0 and 1 if we are concerned about the chance of an issue like this occurring.

EDIT: As it turns out, there are some extremely useful properties of gradients dying off. In particular, the idea of "sparsity". So, many times, backprop being blocked can actually be an *advantage*. More on that in this StackOverflow answer:

How does rectilinear activation function solve the vanishing gradient problem in neural networks?

Cross Validated is a question and answer site for people interested in statistics, machine learning, data analysis...

stats.stackexchange.com

A "leaky" ReLU solves this problem. Leaky Rectified Linear Units are ones that have a very small gradient instead of a zero gradient when the input is negative, giving the chance for the net to continue its learning.

$$\varepsilon = 0.01$$
$$RELU(x) = \max(\varepsilon x, x)$$

EDIT: Looks like values in the range of 0.2–0.3 are more common than something like 0.01.

Instead of outputting zero when the input is negative, the function will output a very flat line, using gradient ε. A common value to use for ε is 0.01. The resulting function is represented in following diagram:

As you can see, the learning will be small with negative inputs, but will exist nonetheless. In this sense, leaky ReLUs do not die.

However, ReLUs/leaky ReLUs aren't necessarily always optimal — results with them have been inconsistent (maybe because, due to the small constant, in certain cases this could cause vanishing gradients again? — that being said, dead units again don't happen all that often). Another notable issue is that, because the output of ReLU isn't bounded between 0 and 1 or -1 and 1 like tanh/sigmoid are, the activations (values in the neurons in the network, not the gradients) can in fact explode with extremely deep neural networks like recurrent neural networks. During training, the whole network becomes fragile and unstable in that, if you update weights in the wrong direction even the slightest, the activations can blow up. Finally, even though the ReLU derivatives are either 0 or 1, our overall derivative expression contains the weights multiplied in. Since the weights are generally initialized to be < 1, this could contribute to vanishing gradients.

So, overall, it's not a black and white problem. ReLUs still face the vanishing gradient problem, it's just that they often face it to a *lesser degree*. In my opinion, the vanishing gradient problem isn't binary; you can't solve it with one technique, but you can delay (in terms of how deep we can go before we start suffering again) its impact.

You may also be wondering: by the way, ReLUs don't squeeze values into a probability, so why are they so commonly used? One can easily stick a sigmoid/logistic activity to the end of a neural net for a binary classification scenario. For multiple outputs, one could use the softmax function to create a probability distribution that adds to 1.

Conclusion

This article was a joy to write because it was my first step into the intricacies and practice of Machine Learning, rather than just looking at theoretical algorithms and calculations. The vanishing gradient problem was a major obstacle for the success of deep learning, but now that we've overcome it through multiple different techniques in weight initialization (which I talked less about today), feature preparation (through batch normalization — centering all input feature values to zero), and activation functions, the field is going to thrive — and we've already seen it with recent innovations in game AI (AlphaGo, of course!). Here's to the future of multi-layered neural networks! 🍻

Machine Learning Artificial Intelligence Today I Learned