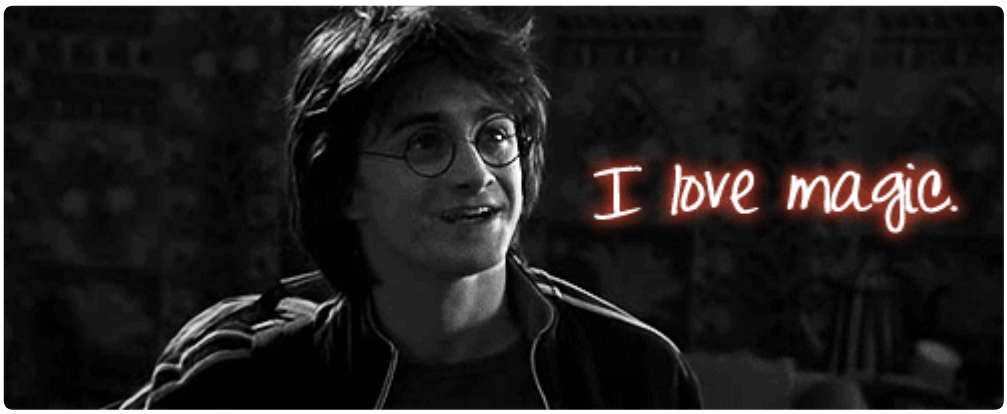
Vanishing Gradients & LSTMs

October 9th, 2016

At first, neural networks seem like black magic. They can translate languages, accurately label subsections of photos, make predictions about the stock market, play video games, generate artwork, and much, much more. Yeah, I know:

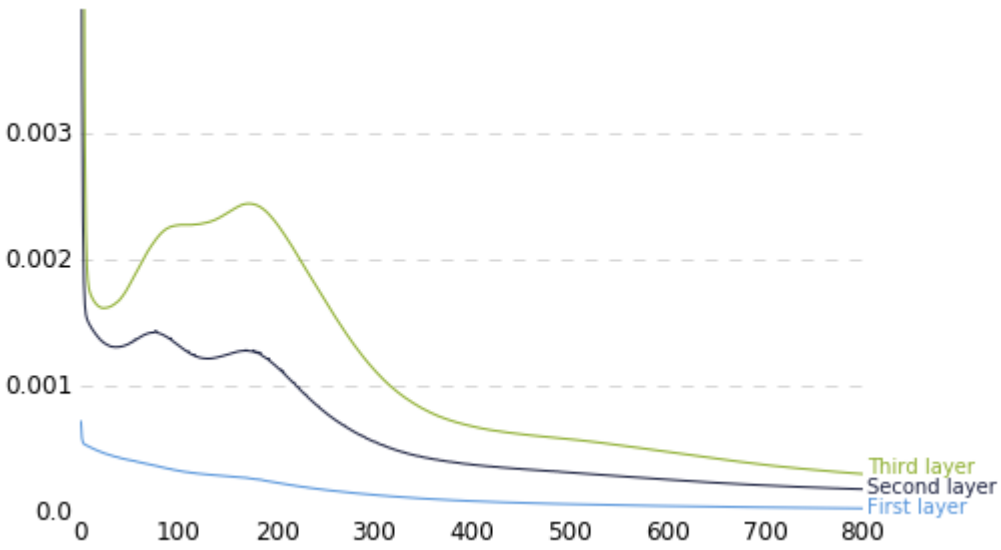


Actually, it turns out that while neural networks are sometimes intimidating structures, the mechanism for making them work is surprisingly simple: stochastic gradient descent. For each of the parameters in our network (such as weights or biases), all we have to do is calculate the derivative of the loss with respect to the parameter, and nudge it a little bit in the opposite direction.

Disappearing gradients

Stochastic gradient descent seems simple enough, but in many networks we might begin to notice something odd: the weights closer to the end of the network change a lot more than those at the beginning. And the deeper the network, the less and less the beginning layers change. This is problematic, because our weights are initialized randomly. If they're barely moving, they're never going to reach the right values, or it'll take them years.

I trained a simple fully connected network to classify MNIST images to illustrate this point. Here, we can see how the gradients change over time for a network with one input layer and two hidden layers:



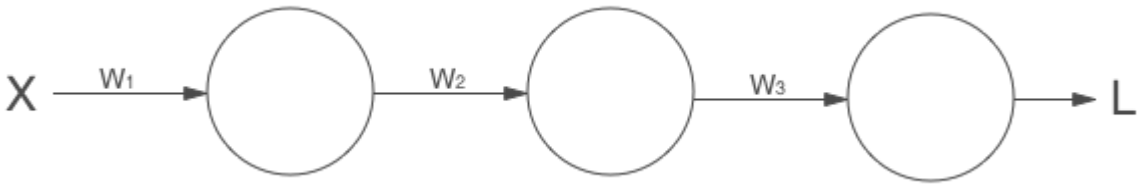
Notice how the first layer's gradients are much lower that the third layer's, which means those weights are changing by a much smaller amount. If we add more layers, the difference only gets more

dramatic. The whole rest of the network is affected by what comes out of the first layer, so if those first weights are totally wrong, our network is not going to perform well.

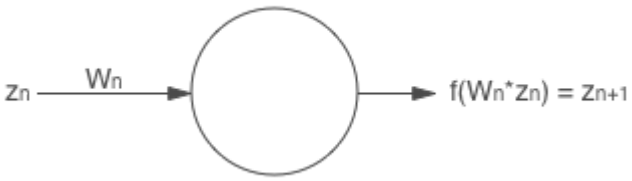
Here's an iPython notebook implementing this simple network in Tensorflow and plotting the gradients. Feel free to play with the number and size of layers to see how bad it can get.

Why do gradients vanish?

Let's imagine we have a 3-layer network, initialized with some set of weights and activations. For simplicity, let's envision that each layer has one node.

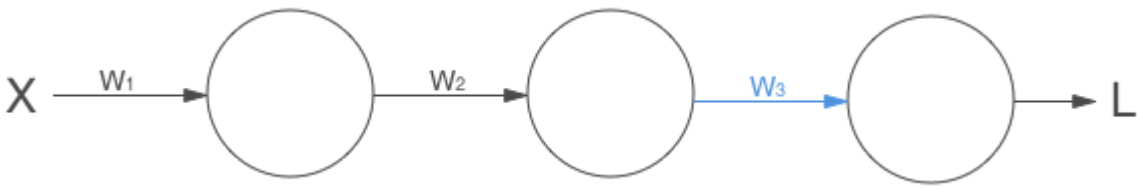


Each node is associated with an input weight, and outputs a function of that weight times the input value. For example, the first node outputs a function of $W_1 \cdot X$. In practice, this function might be a sigmoid, inverse tangent, or ReLU function, but for now we'll call the output of a node simply $f(x)$. From now on we'll refer to this output as z_n for node n:



At the end of this network we end up with a loss, or a measure the difference between what we expected to see and what our network actually outputted. This is commonly calculated with something like a cross-entropy function. For now, all we need to know is that the loss will be a function of z_3, the output of our last node.

Now let's start backpropagating. First let's improve W_3, the input weight of the last hidden layer.



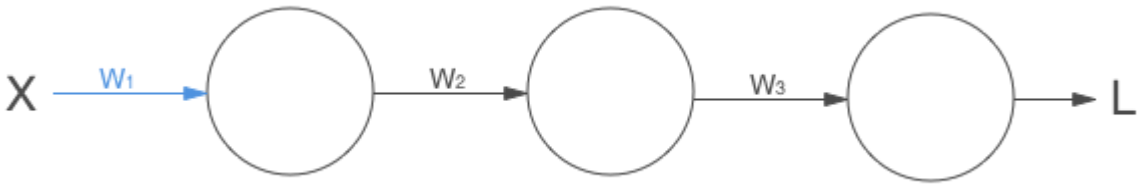
First, we need to calculate the gradient of the loss with respect to W_3, so we know in what direction to nudge that weights. If we apply the chain rule:

$$\frac{\partial \text{Loss}}{\partial W_3} = \frac{\partial \text{Loss}}{\partial f(z_3)} \cdot \frac{\partial f(z_3)}{\partial W_3} = \frac{\partial \text{Loss}}{\partial f(z_3)} \cdot f'(z_3) \cdot W_3$$

We'll leave $\frac{\partial \text{Loss}}{\partial f(z_3)}$ as a partial derivative that will just depend on what function we choose to calculate the loss. The important thing is that it's directly calculated from $f(z_3)$, so this will just be a constant

term in our backpropagation equations.

Now that we have this expression, we just nudge W_3 in the opposite direction, scaled by some step size. Let's skip ahead to calculating how to change our first input weight of the network:



$$\frac{\partial \text{Loss}}{\partial W_1} = \frac{\partial \text{Loss}}{\partial f(z_3)} \cdot \frac{\partial f(z_3)}{\partial f(z_2)} \cdot \frac{\partial f(z_2)}{\partial f(z1)} \cdot \frac{\partial f(z_1)}{\partial W_1}$$

$$= \frac{\partial \text{Loss}}{\partial f(z_3)} \cdot f'(z_3) \cdot W_3 \cdot f'(z_2) \cdot W_2 \cdot f'(z_1) \cdot W_1$$

(If you'd like more details on why this is true, check out thisdetailed backpropagation explanation.)

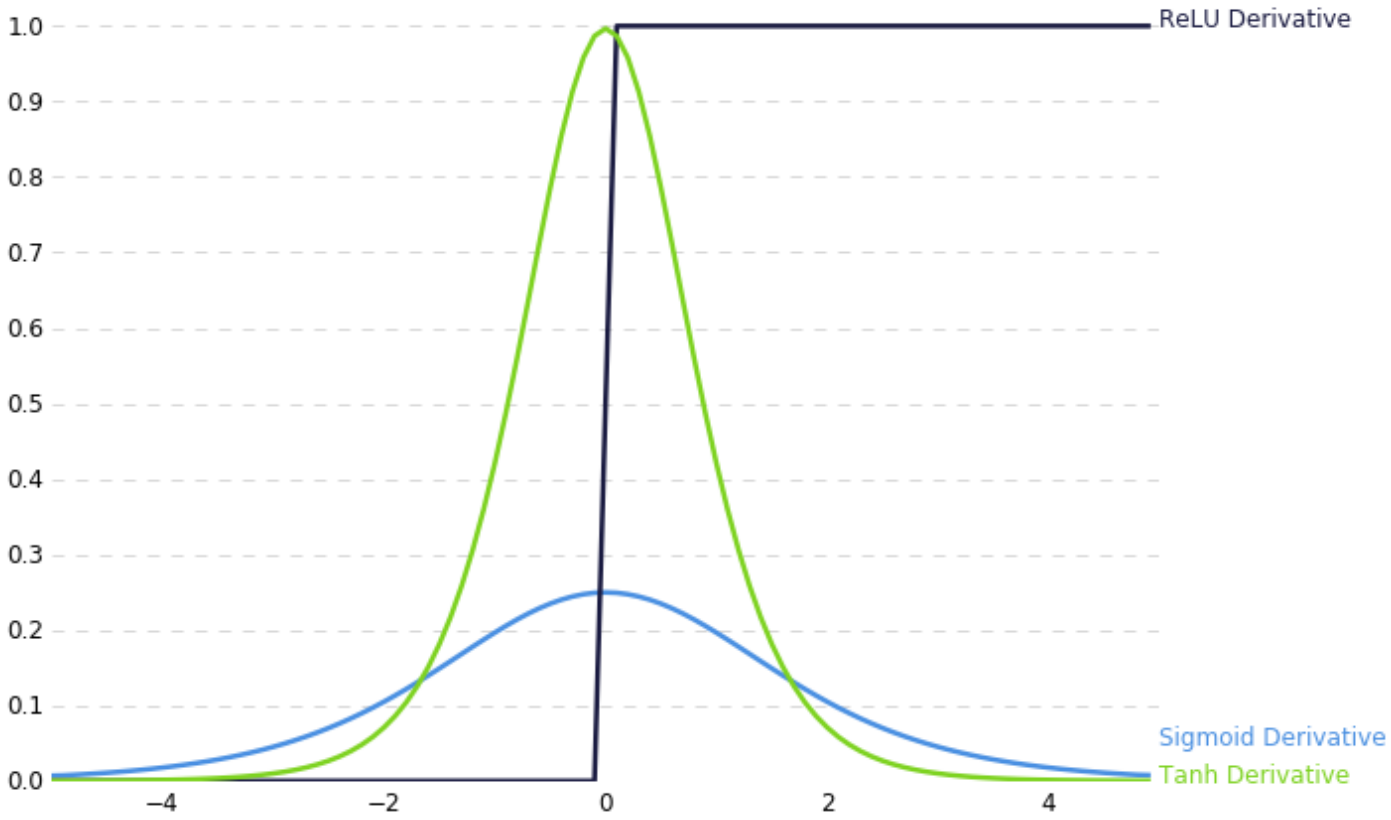
That's certainly a mouthful. Notice how many more terms of the form $f'(z) \cdot W$ we're multiplying together to get the gradient here. A typical initialization of weights is from a Gaussian with mean zero and standard deviation one, which will yield mostly weights of magnitude less than one. If $f(x)$is a sigmoid function, which is quite common, it's derivative will always be less than 0.25. That's a lot of small numbers being multiplied together, yielding a really, really small number.

You could also imagine the opposite scenario, where if our weights happened to reach larger values, we'd be multiplying a lot of big numbers, and the gradient would explode rather than vanish. Neither of these are particularly appealing situations, so now let's take a look at how we can get these pesky gradients under control.

1. Activation Functions

Remember that our vanishing gradient was arising from multiplying lots of $f'(z) \cdot W$ terms. This gives us some insight into why certain functions f (called activation functions) might work better than others for combatting this problem.

For example, while the derivative of a sigmoid function is < 0.25 everywhere, making each term even smaller, the derivative of the ReLU function is one at every point above zero, creating a more stable network. This is also one of the reasons why the inverse tangent activation function is sometimes preferred over the sigmoid.



2. Clipping Gradients

Pascanu et. al. provide a simple solution for exploding gradients: just scale them down whenever they pass above a certain threshold. See their paper for a more detailed geometrical interpretation of why this is an okay thing to do during stochastic gradient descent!

3. LSTMs

As you might imagine, the vanishing gradient becomes a very important issue the deeper a network gets. One genre of networks that tend to be very deep are recurrent neural networks (RNNs). RNNs are used to model time-dependent data, like words in a sentence. We feed in words one by one, and the nodes in the network store their state at one timestep and use it to inform the next timestep.

If we think about each timestep as a layer, with weights going from one timestep to the next (this is often referred to as "unraveling" an RNN), we can see that our network will be at least as deep as the number of timesteps. When it comes to sentences, paragraphs, or other timeseries data, these sequences we're feeding in can be very long, so we face the same problems that a very deep neural network would.

The first word fed into an RNN is equivalent to the first layer in the simple neural network from above. If we're experiencing a vanishing gradient, the weights at the beginning of the network change less and less, and the RNN becomes worse at modeling long-term dependencies. If we're predicting words of a sentence, the first word in the sentence might actually be really important context for predicting a word at the end, so we don't want to lose that information.

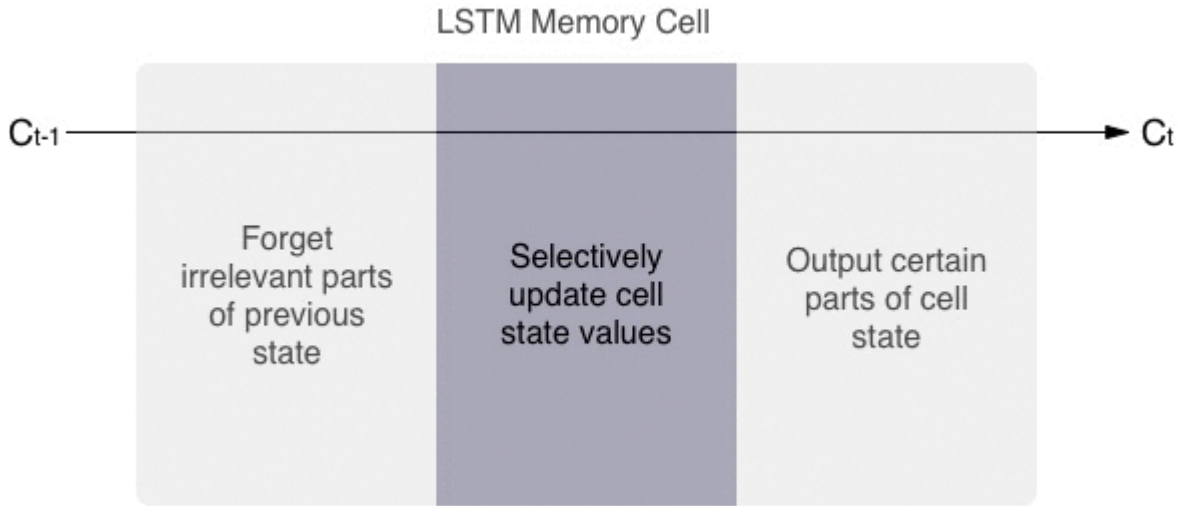
LSTMs (Long Short-Term Memory Networks) are a special subset of RNNs that are able to deal with remembering information for much longer periods of time. The idea behind an LSTM is actually really simple! Rather than each hidden node being simply a node with a single activation function, each node is a memory cell that can store other information. Specifically, it maintains its own cell state. Normal RNNs take in their previous hidden state and the current input, and output a new hidden state. An LSTM does the same, except it also takes in its old cell state and outputs its new cell state.

So what's this magic that goes on inside an LSTM memory cell? Let's split it into three main steps.

1. We decide what from the previous cell state is worth remembering, and tell the cell state to **forget the stuff we decide is irrelevant**.

2. We **selectively update** the cell state based on the new input we've just seen.

3. We **selectively decide what part of the cell state we want to output** as the new hidden state.



This is all achieved by a few simple gates: the **forget gate**, the **input gate**, and the **output gate**.

Let's go through the steps with a specific example: translating the English sentence *When we go to France, you speak English but I speak French* to the French sentence *Quand nous allons à France, tu parles Anglais mais je parle Français*.

Forget Gate
In the first step, a function of the previous hidden state and the new input passes through the forget gate, letting us know what is probably irrelevant and can be taken out of our cell state. The forget gate will output values close to 1 for parts of the cell state we wish to completely keep, and zero for values we'd like to totally get rid of.

Let's say we're feeding in the example English sentence from above and see the word "you". Now we might like to forget the "we" that appeared previously, since the next verb will likely be conjugated according to the new subject "you".

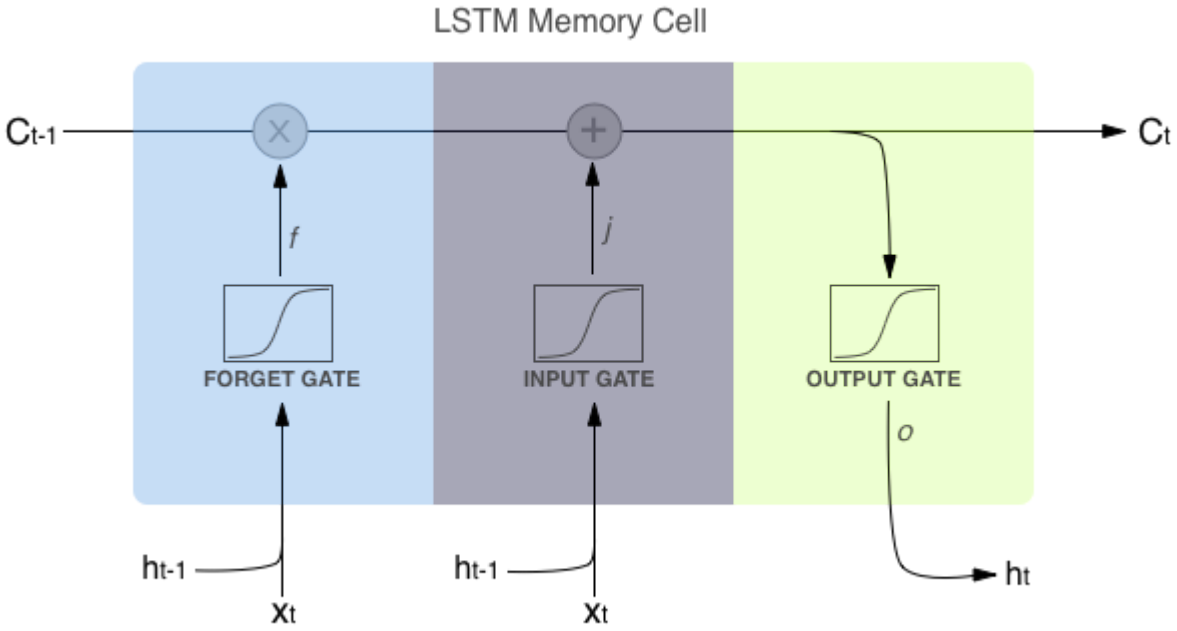
Input Gate
In the second step, a function of the inputs passes through the input gate and is added to the cell state to update it. Following our scenario from above, we might want to add information to the cell state about the new word "you" we've just seen -- for example, the fact that it's a subject, singular and second person.

Output Gate
In the final step, the output gate decides what values from our cell state we are going to add to the hidden state output.

In our example, if we expect the next word will be a verb, we might output the information about the current subject that will important for conjugating the verb -- for example, the fact that "you" is singular and second person.

At the same time, we can continue to hold onto things in the cell state that we think might be important not at the next time step, but at some point much later along -- like the fact that the

sentence is set in France. This information might not be relevant to the verb appearing next, but it will be helpful information for the end of the sentence, which is about speaking French. The ability to preserve information in the cell state for long stretches of time is a big part of what makes LSTMs special.



The cell state allows an LSTM to surpass the vanishing gradient problem for two main reasons.

First: Remember how the sigmoid activation function always has a derivative less than 0.25? So when we multiply together all those $\mathrm{f}^{'}(\mathrm{x}) \cdot \mathrm{W}$ terms, our gradient just vanishes away? Well, if we look at the cell state in an LSTM, the only thing we're multiplying it by is the output of the forget gate, so we can think of f as the weights for the cell state. In that case, what's the activation function? Technically, there isn't one, besides the identity function itself. The derivative of the identity function is, conveniently, always one. **So if f = 1, information from the previous cell state can pass through this step unchanged**.

Second: There is one more step, located in the center of the diagram, where we adjust the cell state. Notice that we adjust the cell state by adding some function of the inputs. When we backpropagate and take the derivative of $\mathrm{C_t}$ with respect to $\mathrm{C_{t-1}}$, **this added term just disappears**!

So, because the forget gate is essentially the weights *and*activation function for the cell state, and because the LSTM can learn to set that forget gate to one for important things in the cell state, information can pass through unchanged.

Because of their ability to capture long-term dependencies, LSTMs have gained a lot of popularity recently. For some awesome applications, see Google's <a href-"https://research.googleblog.com/2016/09/a-neural-network-for-machine.html">machine translation technology, recent work on image generation, or the use of LSTMs to try and diagnose patients based on clinical timeseries. To learn more about how LSTMs can be extended to work even better, check out attention and memory networks.

Hopefully, this post has given you some insight into the difficulty of modeling long-term dependencies in data and why LSTMs work so well. Reach out with any questions to hsuresh@mit.edu!