Vanishing Gradients in Vanilla RNNs

Aaron Avram

June 24 2025

Introduction

In this writeup I will describe why Vanilla RNNs suffer from vanishing gradients, then work through the most popular solution: LSTM cells.

Vanishing Gradients

In the basic formulation of a Recurrent Neural Network, we have:

$$\begin{aligned} z^{(t)} &= Wh^{(t-1)} + Ux^{(t)} + b \\ h^{(t)} &= \tanh(z^{(t)}) \end{aligned}$$

Where W and U are matrices and b is a vector. We choose tanh here for convenience but other activation functions may be used, and the next results all follow regardless. For now, only consider W and h discarding U, x and b as this result comes about independently of their influence.

Say we want to find the Jacobian of $h^{(t)}$ with respect to $h(0)$. I claim that the formula is given by $J_t := J(h^{(t)}) = \prod_{k=1}^{t} D_k W$ where $D_k =$diag$[\tanh'(z^{(k)})]$, and this is easy to prove by induction:

Proof. **Base Case**: $t = 1$, consider $h^{(1)} = \tanh(Wh^{(0)})$. Clearly, $J(z^1)(h^{(0)}) = W$, and by the chain rule:

$$\begin{aligned} (J_t)_{ij} &= \frac{\partial h_i^{(1)}}{\partial z_i^{(1)}} \frac{\partial z_i^{(1)}}{\partial h_{ij}^{(0)}} \\ &= \tanh'(z^{(1)}) W_{ij} \end{aligned}$$

So $J_1 = D_1 W$, as required.

Inductive Step. Suppose for some $t \in \mathbb{N}$, $J_t = \prod_{k=1}^{t} D_k W$. Now consider $t + 1$:

$$h^{(t+1)} = \tanh(Wh^{(t)})$$

First let us find $J(h^{(t+1)})(h^{(t)})$:

$$J(h^{(t+1)})(h^{(t)}) = \left[J(h^{(t+1)})(z^{(t+1)})\right]\left[J(z^{(t+1)})(h^{(t)})\right]$$

If we let $h^{(t)}$ act as $h^{(0)}$ in the base case, it is clear that:

$$J(h^{(t+1)})(h^{(t)}) = D_{t+1}W$$

And so:

$$J_{t+1} = \prod_{k=1}^{t+1} D_k W$$

As required. $\square$

Now we want to analyze the norm of this Jacobian matrix to understand the behaviour of weight gradients long term. We will let $||.||$ denote the spectral norm. First consider positive W. This means W admits an spectral decomposition, $W = CVC^T$ and $||W|| = ||V|| = \lambda_{max}(V)$. Notice that D_t is a diagonal matrix so its norm is the absolute value of its largest entry. However, since this entry is the product of an elementwise application of the tanh derivative, it is equal to $1 - \tanh^2(x)$ for some $x \in \mathbb{R}$, but since tanh is bounded by $(-1, 1)$, its derivative is bounded by $[0, 1)$. Thus, $0 \leq ||D_t|| < 1$, $\forall t \in \mathbb{N}$. Next consider $D := \{||D_t|| : t \in \mathbb{N}\}$ and define $\delta_i := \inf D$ and $\delta_s := \sup D$. Thus for all t, $0 \leq \delta_i \leq ||D_t|| \leq \delta_s \leq 1$. We can consider $||J_t||$

$$\begin{aligned}
||J_t|| &= ||\prod_{k=1}^{t} D_k W|| \\
&= \prod_{k=1}^{t}(||D_k||)(||W||) \\
&= ||W||^t \prod_{k=1}^{t}(||D_k||) \\
(\delta_i||W||)^t \leq ||J_t|| &\leq (\delta_s||W||)^t
\end{aligned}$$

It is clear that if $(\delta_i||W||)^t$ explodes, so does $||J_t||$ leading to exploding gradients and if $(\delta_s||W||)^t$ vanishes, so does $||J_t||$ leading to vanishing gradients. Now that leaves us a single case. $(\delta_i||W||)^t$ vanishes or goes to 1 and $(\delta_s||W||)^t$ explodes or goes to 1, then: $||W||\delta_i \leq 1$ and $|W||\delta_s \geq 1$. For this we will assume that $\delta_i > 0$ as otherwise this would imply that the hidden states vanish, which is a useless scenario. Then we can bound $||W||$ with $1 \leq \frac{1}{\delta_s} \leq ||W|| \leq \frac{1}{\delta_i}$. In this case, the terms δ_i, δ_s only introduce a bounded multiplicative effect. Thus the size of $||W||$ is ultimately be the major factor that will affect the gradient behaviour. So, we consider a simpler RNN model, with the non-linearities removed. The gradient behaviour here captures the gradient behaviour in the more general case due to the dominating effects of $||W||$. So consider: $h^{(t)} = W^t h^{(0)}$, meaning we now have $J_t = W^t$. And by the spectral decomposition:

$$\begin{aligned}
W^t &= (CVC^T)^t \\
&= CV^tC^T
\end{aligned}$$

Thus the behaviour of gradient is mostly captured by V^t as C is unitary. But, as t increases, all entries of V that are not equal to 1, either vanish or explode. Thus as t increases the largest eigenvalue of V will dominate gradient behaviour and eigenvalues less than 1 will vanish, which diminishes the expressive power of the network over longer sequences. The only case where V remains stable is if it is also unitary. However, this restricts the space of possible values for W far too much to be used in practice and there is no guarantee that this property will be preserved.

Now in the general case, if W is not positive, we can apply the polar decomposition to see that $W = UR$, and since U is unitary the magnitude of W's gradients will be largerly determined by R, which is the case we have analyzed above. Thus, it is clear that in Vanilla Recurrent Neural Networks, gradient instability easily appears, especially as sequence length increases.

A Solution: LSTM

The main issue with Vanilla RNNs is that they do not have selective memory. Thus we want to find a way for RNNs to learn which updates to its hidden state are important, so that it does not get flooded with large amounts of information that weakens gradient expressiveness. One of the most successful solutions was the Long Short-Term memory cell. First, informally, what we are trying to do is learn three gates which filter what our RNN remembers, these will apply elementwise to the state and output values of each cell with the filters as arrays of the same dimension as the state value with values between 0 and 1. The first of these gates is the forget gate, $f^{(t)}$, the gate at step t, which is a filter for how important the memory of the previous state was. If the forget gate has mostly values close to 1, then the new cell keeps most of the previous state's memory and if it is close to 0, it forgets most of it. Next we have $g^{(t)}$ the input gate, which filters how much of the new candidate memory based on the tth element in the sequence should be kept. Finally, we have an output gate $o^{(t)}$, which learns how much of the memory at step t to pass on to the next step.

Now the cell is a bit more complex than a Vanilla RNN cell, it passes on two hidden state values between time steps. One is the familiar output unit $h^{(t)}$ and the other is a memory unit $c^{(t)}$. The idea being that the value of $h^{(t)}$ is the output representation which is evolved and updated through the LSTM and $c^{(t)}$ is a unit that maintains important information used to update the output value. This way the cell memory and output is decoupled which allows the memory to be streamlined to maximize how efficient the model stores past information. For the specifics, each unit and gate has its own weight matrices and bias vector. (Note that below σ will be used to denote a generic sigmoidal function applied elementwise).

$$\begin{aligned} f^{(t)} &= \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f) \\ g^{(t)} &= \sigma(W_g h^{(t-1)} + U_g x^{(t)} + b_g) \\ o^{(t)} &= \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o) \end{aligned}$$

Now we can consider our new candidate memory, which is the new information the RNN processes at this time step calculated with the original core Vanilla cell:

$$\tilde{c}^{(t)} = \tanh(W h^{(t-1)} + U x^{(t)} + b)$$

Now we update our hidden state memory at step t, $c^{(t)}$:

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + g^{(t)} \odot \tilde{c}^{(t)}$$

Finally we formulate the hidden output unit:

$$h^{(t)} = \tanh(c^{(t)}) \odot o^{(t)}$$

With this, we pass on $c^{(t)}$, $h^{(t)}$ through the RNN and with this we have developed the theory of an LSTM. Now for why this solves the vanishing gradient problem, the intuitive reason is that it updates our hidden state memory additively instead of multiplicatively. This difference helps regulate gradient flow. In fact if we consider the gradients of one memory unit with respect to the previous unit, we can see that the model now learns exactly how the gradients are passed throughout the network. Specifically:

$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + g^{(t)} \odot \tilde{c}^{(t)}$$
$$\nabla_{c^{(t-1)}}(c^{(t)}) = f^{(t)}$$

Thus our forget gate controls how much information is passed between cell memory units. Applying the chain rule:

$$\nabla_{c^{(0)}}(c^{(t)}) = \prod_{k=1}^{t} f^{(k)}$$

Note that this is a Pi product of elementwise multiplication not matrix multiplication. But notice, that now our gradient will flow if all the values of f are close to 1 in a certain entry. Thus our model learns exactly when to let gradients flow. I will implement an LSTM but not in this project.