CSC421/2516 Lecture 14: Exploding and Vanishing Gradients

Roger Grosse and Jimmy Ba
Last time, we saw how to compute the gradient descent update for an RNN using backprop through time.

The updates are mathematically correct, but unless we’re very careful, gradient descent completely fails because the gradients explode or vanish.

The problem is, it’s hard to learn dependencies over long time windows.

Today’s lecture is about what causes exploding and vanishing gradients, and how to deal with them. Or, equivalently, how to learn long-term dependencies.
Recall the RNN for machine translation. It reads an entire English sentence, and then has to output its French translation.

A typical sentence length is 20 words. This means there’s a gap of 20 time steps between when it sees information and when it needs it. The derivatives need to travel over this entire pathway.
Why Gradients Explode or Vanish

Recall: backprop through time

Activations:

\[ \overline{L} = 1 \]
\[ \overline{y(t)} = \overline{L} \frac{\partial L}{\partial y(t)} \]
\[ \overline{r(t)} = \overline{y(t)} \phi'(r(t)) \]
\[ \overline{h(t)} = \overline{r(t)} v + \overline{z^{(t+1)}} w \]
\[ \overline{z(t)} = \overline{h(t)} \phi'(z(t)) \]

Parameters:

\[ \overline{u} = \sum_t \overline{z(t)} x(t) \]
\[ \overline{v} = \sum_t \overline{r(t)} h(t) \]
\[ \overline{w} = \sum_t \overline{z^{(t+1)}} h(t) \]
Why Gradients Explode or Vanish

Consider a univariate version of the encoder network:

![Diagram of a univariate encoder network]

**Backprop updates:**

\[
\begin{align*}
    h^{(t)} &= z^{(t+1)} w \\
    z^{(t)} &= h^{(t)} \phi'(z^{(t)})
\end{align*}
\]

Applying this recursively:

\[
\begin{align*}
    h^{(1)} &= w^{T-1} \phi'(z^{(2)}) \cdots \phi'(z^{(T)}) h^{(T)} \\
    \text{the Jacobian } &\frac{\partial h^{(T)}}{\partial h^{(1)}}
\end{align*}
\]

With linear activations:

\[
\frac{\partial h^{(T)}}{\partial h^{(1)}} = w^{T-1}
\]

**Exploding:**

\[
\begin{align*}
    w = 1.1, T = 50 & \Rightarrow \frac{\partial h^{(T)}}{\partial h^{(1)}} = 117.4
\end{align*}
\]

**Vanishing:**

\[
\begin{align*}
    w = 0.9, T = 50 & \Rightarrow \frac{\partial h^{(T)}}{\partial h^{(1)}} = 0.00515
\end{align*}
\]
More generally, in the multivariate case, the Jacobians multiply:

\[
\frac{\partial h^{(T)}}{\partial h^{(1)}} = \frac{\partial h^{(T)}}{\partial h^{(T-1)}} \cdots \frac{\partial h^{(2)}}{\partial h^{(1)}}
\]

Matrices can explode or vanish just like scalar values, though it’s slightly harder to make precise.

Contrast this with the forward pass:
- The forward pass has nonlinear activation functions which squash the activations, preventing them from blowing up.
- The backward pass is linear, so it’s hard to keep things stable. There's a thin line between exploding and vanishing.
Why Gradients Explode or Vanish

- We just looked at exploding/vanishing gradients in terms of the mechanics of backprop. Now let’s think about it conceptually.
- The Jacobian $\frac{\partial h^{(T)}}{\partial h^{(1)}}$ means, how much does $h^{(T)}$ change when you change $h^{(1)}$?
- Each hidden layer computes some function of the previous hidden and the current input:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)})$$

- This function gets iterated:

$$h^{(4)} = f(f(f(h^{(1)}, x^{(2)}), x^{(3)}), x^{(4)}).$$

- Let’s study iterated functions as a way of understanding what RNNs are computing.
Iterated functions are complicated. Consider:

\[ f(x) = 3.5x(1 - x) \]
Iterated Functions

An aside:

- Remember the Mandelbrot set? That’s based on an iterated quadratic map over the complex plane:

\[ z_n = z_{n-1}^2 + c \]

- The set consists of the values of \( c \) for which the iterates stay bounded.

CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=321973
Let’s imagine an RNN’s behavior as a dynamical system, which has various attractors:

- Within one of the colored regions, the gradients vanish because even if you move a little, you still wind up at the same attractor.
- If you’re on the boundary, the gradient blows up because moving slightly moves you from one attractor to the other.

– Geoffrey Hinton, Coursera
Why Gradients Explode or Vanish

- Consider an RNN with tanh activation function:

  ![Diagram of RNN with tanh activation function](image)

- The function computed by the network:
Cliffs make it hard to estimate the true cost gradient. Here are the loss and cost functions with respect to the bias parameter for the hidden units:

Generally, the gradients will explode on some inputs and vanish on others. In expectation, the cost may be fairly smooth.
Keeping Things Stable

- One simple solution: gradient clipping
- Clip the gradient $\mathbf{g}$ so that it has a norm of at most $\eta$:
  
  $$ \text{if } \|\mathbf{g}\| > \eta : $$
  $$ \mathbf{g} \leftarrow \frac{\eta \mathbf{g}}{\|\mathbf{g}\|} $$

- The gradients are biased, but at least they don’t blow up.

— Goodfellow et al., Deep Learning
Keeping Things Stable

- Another trick: reverse the input sequence.

- This way, there’s only one time step between the first word of the input and the first word of the output.

- The network can first learn short-term dependencies between early words in the sentence, and then long-term dependencies between later words.
Keeping Things Stable

- Really, we’re better off redesigning the architecture, since the exploding/vanishing problem highlights a conceptual problem with vanilla RNNs.

- The hidden units are a kind of memory. Therefore, their default behavior should be to keep their previous value.
  - I.e., the function at each time step should be close to the identity function.
  - It’s hard to implement the identity function if the activation function is nonlinear!

- If the function is close to the identity, the gradient computations are stable.
  - The Jacobians $\frac{\partial h^{(t+1)}}{\partial h^{(t)}}$ are close to the identity matrix, so we can multiply them together and things don’t blow up.
Keeping Things Stable

- **Identity RNNs**
  - Use the ReLU activation function
  - Initialize all the weight matrices to the identity matrix

- Negative activations are clipped to zero, but for positive activations, units simply retain their value in the absence of inputs.

- This allows learning much longer-term dependencies than vanilla RNNs.

- It was able to learn to classify MNIST digits, input as sequence one pixel at a time!

Le et al., 2015. A simple way to initialize recurrent networks of rectified linear units.
Another architecture which makes it easy to remember information over long time periods is called **Long-Term Short Term Memory (LSTM)**

- What’s with the name? The idea is that a network’s activations are its short-term memory and its weights are its long-term memory.
- The LSTM architecture wants the short-term memory to last for a long time period.

- It’s composed of memory cells which have controllers saying when to store or forget information.
Replace each single unit in an RNN by a memory block -

\[ c_{t+1} = c_t \cdot \text{forget gate} + \text{new input} \cdot \text{input gate} \]

- \( i = 0, f = 1 \Rightarrow \text{remember the previous value} \)
- \( i = 1, f = 1 \Rightarrow \text{add to the previous value} \)
- \( i = 0, f = 0 \Rightarrow \text{erase the value} \)
- \( i = 1, f = 0 \Rightarrow \text{overwrite the value} \)

Setting \( i = 0, f = 1 \) gives the reasonable “default” behavior of just remembering things.
Long-Term Short Term Memory

- In each step, we have a vector of memory cells $c$, a vector of hidden units $h$, and vectors of input, output, and forget gates $i$, $o$, and $f$.
- There's a full set of connections from all the inputs and hiddens to the input and all of the gates:

$$
\begin{pmatrix} i_t \\ f_t \\ o_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma & \sigma & \sigma & \tanh \\ \sigma & \sigma & \sigma & \tanh \end{pmatrix} W \begin{pmatrix} y_t \\ h_{t-1} \end{pmatrix}
$$

$$
c_t = f_t \circ c_{t-1} + i_t \circ g_t
$$

$$
h_t = o_t \circ \tanh(c_t)
$$

- Exercise: show that if $f_{t+1} = 1$, $i_{t+1} = 0$, and $o_t = 0$, the gradients for the memory cell get passed through unmodified, i.e.

$$
\overline{c_t} = \overline{c_{t+1}}.
$$
Long-Term Short Term Memory

- Sound complicated? ML researchers thought so, so LSTMs were hardly used for about a decade after they were proposed.
- In 2013 and 2014, researchers used them to get impressive results on challenging and important problems like speech recognition and machine translation.
- Since then, they’ve been one of the most widely used RNN architectures.
- There have been many attempts to simplify the architecture, but nothing was conclusively shown to be simpler and better.
- You never have to think about the complexity, since frameworks like TensorFlow provide nice black box implementations.
Long-Term Short Term Memory

Visualizations:

http://karpathy.github.io/2015/05/21/rnn-effectiveness/
I promised you I’d explain the best ImageNet object recognizer from 2015, but that it required another idea.

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>Top-5 error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>Hand-designed descriptors + SVM</td>
<td>28.2%</td>
</tr>
<tr>
<td>2011</td>
<td>Compressed Fisher Vectors + SVM</td>
<td>25.8%</td>
</tr>
<tr>
<td>2012</td>
<td>AlexNet</td>
<td>16.4%</td>
</tr>
<tr>
<td>2013</td>
<td>a variant of AlexNet</td>
<td>11.7%</td>
</tr>
<tr>
<td>2014</td>
<td>GoogLeNet</td>
<td>6.6%</td>
</tr>
<tr>
<td>2015</td>
<td>deep residual nets</td>
<td>4.5%</td>
</tr>
</tbody>
</table>

That idea is exploding and vanishing gradients, and dealing with them by making it easy to pass information directly through a network.
Deep Residual Networks

- Recall: the Jacobian $\frac{\partial h^{(T)}}{\partial h^{(1)}}$ is the product of the individual Jacobians:

$$\frac{\partial h^{(T)}}{\partial h^{(1)}} = \frac{\partial h^{(T)}}{\partial h^{(T-1)}} \cdots \frac{\partial h^{(2)}}{\partial h^{(1)}}$$

- But this applies to multilayer perceptrons and conv nets as well! (Let $t$ index the layers rather than time.)
- Then how come we didn’t have to worry about exploding/vanishing gradients until we talked about RNNs?
  - MLPs and conv nets were at most 10s of layers deep.
  - RNNs would be run over hundreds of time steps.
  - This means if we want to train a really deep conv net, we need to worry about exploding/vanishing gradients!
Remember Homework 1? You derived backprop for this architecture:

\[
\begin{align*}
z &= W^{(1)}x + b^{(1)} \\
h &= \phi(z) \\
y &= x + W^{(2)}h
\end{align*}
\]

This is called a \textit{residual block}, and it’s actually pretty useful.

Each layer adds something (i.e. a residual) to the previous value, rather than producing an entirely new value.

Note: the network for \( \mathcal{F} \) can have multiple layers, be convolutional, etc.
Deep Residual Networks

- We can string together a bunch of residual blocks.
- What happens if we set the parameters such that $F(x^{(\ell)}) = 0$ in every layer?
  - Then it passes $x^{(1)}$ straight through unmodified!
  - This means it’s easy for the network to represent the identity function.
- Backprop:

  $$
  x^{(\ell)} = x^{(\ell+1)} + x^{(\ell+1)} \frac{\partial F}{\partial x}
  = x^{(\ell+1)} \left( I + \frac{\partial F}{\partial x} \right)
  $$

- As long as the Jacobian $\frac{\partial F}{\partial x}$ is small, the derivatives are stable.
Deep Residual Networks

- Deep Residual Networks (ResNets) consist of many layers of residual blocks.
- For vision tasks, the $F$ functions are usually 2- or 3-layer conv nets.
- Performance on CIFAR-10, a small object recognition dataset:

For a regular convnet (left), performance declines with depth, but for a ResNet (right), it keeps improving.
Deep Residual Networks

- A 152-layer ResNet achieved 4.49% top-5 error on ImageNet. An ensemble of them achieved 3.57%.
- Previous state-of-the-art: 6.6% (GoogLeNet)
- Humans: 5.1%
- They were able to train ResNets with more than 1000 layers, but classification performance leveled off by 150.
- What are all these layers doing? We don’t have a clear answer, but the idea that they’re computing increasingly abstract features is starting to sound fishy...