CSC321 Lecture 16: Learning Long-Term Dependencies

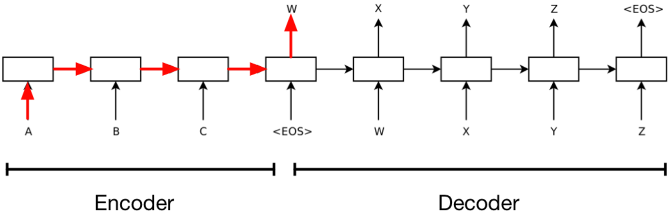
Roger Grosse

Overview

- Yesterday, 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.

Why Gradients Explode or Vanish

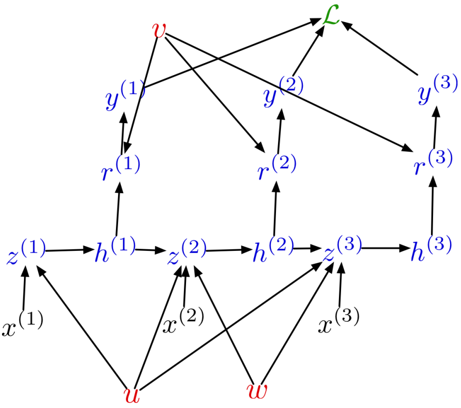
- Recall the RNN for machine translation. It reads an entire English sentence, and then has to output its French translation.



- 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:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y^{(t)}} = \overline{\mathcal{L}} \frac{\partial \mathcal{L}}{\partial y^{(t)}}$$

$$\overline{r^{(t)}} = \overline{y^{(t)}} \, \phi'(r^{(t)})$$

$$\overline{h^{(t)}} = \overline{r^{(t)}} \, v + \overline{z^{(t+1)}} \, w$$

$$\overline{z^{(t)}} = \overline{h^{(t)}} \, \phi'(z^{(t)})$$

Parameters:

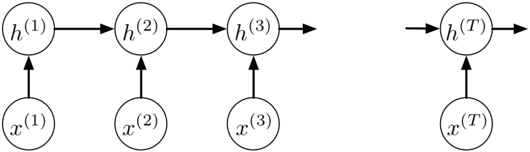
$$\overline{u} = \sum_t \overline{z^{(t)}} \, x^{(t)}$$

$$\overline{v} = \sum_t \overline{r^{(t)}} \, h^{(t)}$$

$$\overline{w} = \sum_t \overline{z^{(t+1)}} \, h^{(t)}$$

Why Gradients Explode or Vanish

Consider a univariate version of the encoder network:



Backprop updates:

$$\overline{h^{(t)}} = \overline{z^{(t+1)}}\, w$$

$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)})$$

Applying this recursively:

$$\overline{h^{(1)}} = \underbrace{w^{T-1} \phi'(z^{(2)}) \cdots \phi'(z^{(T)})}_{\text{the Jacobian } \partial h^{(T)}/\partial h^{(1)}} \overline{h^{(T)}}$$

With linear activations:

$$\partial h^{(T)} / \partial h^{(1)} = w^{T-1}$$

Exploding:

$$w = 1.1,\ T = 50 \quad \Rightarrow \quad \frac{\partial h^{(T)}}{\partial h^{(1)}} = 117.4$$

Vanishing:

$$w = 0.9,\ T = 50 \quad \Rightarrow \quad \frac{\partial h^{(T)}}{\partial h^{(1)}} = 0.00515$$

error signal is a linear function of error signal from child

Why Gradients Explode or Vanish

- More generally, in the multivariate case, the Jacobians multiply:

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

- 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 $\partial \mathbf{h}^{(T)} / \partial \mathbf{h}^{(1)}$ means, how much does $h^{(T)}$ change when you change $\mathbf{h}^{(1)}$?
- Each hidden layer computes some function of the previous hiddens and the current input:

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

- This function gets iterated:

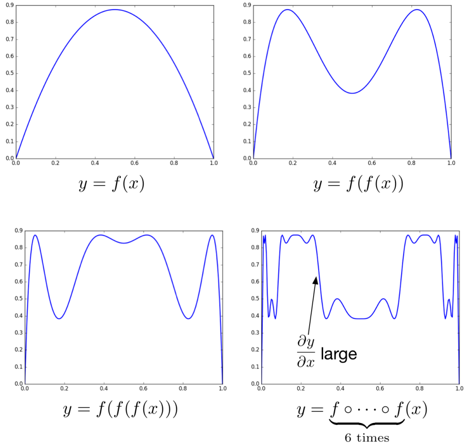
$$\mathbf{h}^{(4)} = f(f(f(\mathbf{h}^{(1)}, \mathbf{x}^{(2)}), \mathbf{x}^{(3)}), \mathbf{x}^{(4)}).$$

- Let's study iterated functions as a way of understanding what RNNs are computing.

Iterated Functions

- Iterated functions are complicated. Consider:

$$f(x) = 3.5\,x\,(1-x)$$



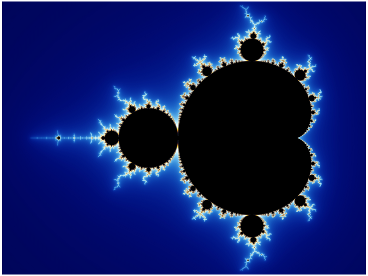
Iterated Functions

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.

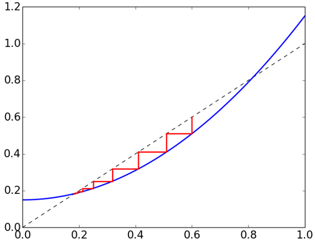


Iterated Functions

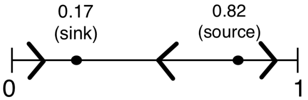
Consider the following iterated function:

$$x_{t+1} = x_t^2 + 0.15.$$

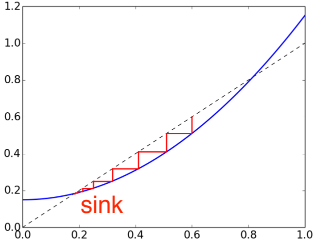
We can determine the behavior of repeated iterations visually:



The behavior of the system can be summarized with a phase plot:



Iterated Functions

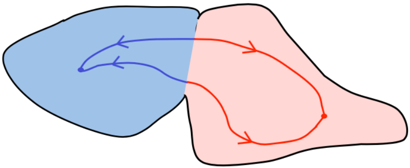


Some observations:

- Fixed points of f correspond to points where f crosses the line $x_{t+1} = x_t$.
- Fixed points with $f'(x_t) > 1$ correspond to sources.
- Fixed points with $f'(x_t) < 1$ correspond to sinks.

Why Gradients Explode or Vanish

- Let's imagine an RNN's behavior as a dynamical system, which has various attractors:

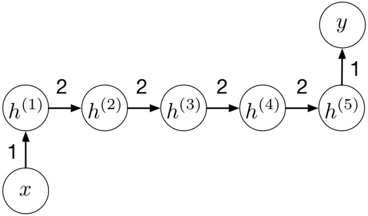


– Geoffrey Hinton, Coursera

- 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.

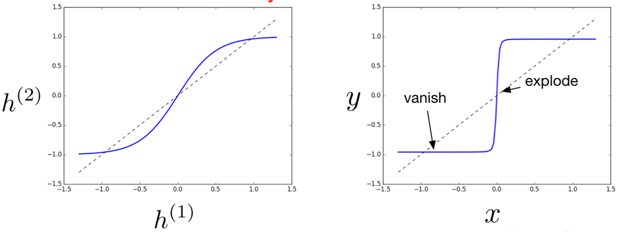
Why Gradients Explode or Vanish

- Consider an RNN with tanh activation function:



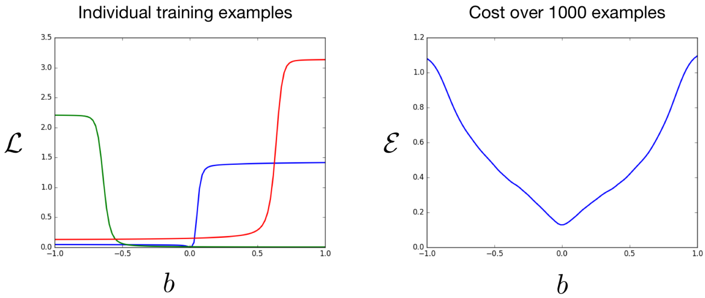
- The function computed by the network:

y is an iterated function over x many times



Why Gradients Explode or Vanish

- 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:

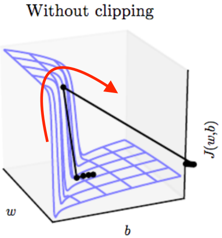


- Generally, the gradients will explode on some inputs and vanish on others. In expectation, the cost may be fairly smooth.

Keeping Things Stable

- One simple solution: gradient clipping
- Clip the gradient $\mathbf{g}$ so that it has a norm of at most η:

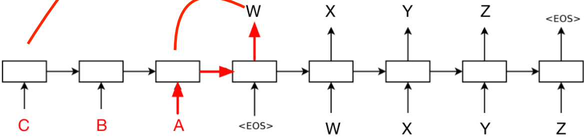
  if $\|\mathbf{g}\| > \eta$:
$$\mathbf{g} \leftarrow \frac{\eta \mathbf{g}}{\|\mathbf{g}\|}$$
- The gradients are biased, but at least they don't blow up.



— 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 $\partial \mathbf{h}^{(t+1)} / \partial \mathbf{h}^{(t)}$ 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!

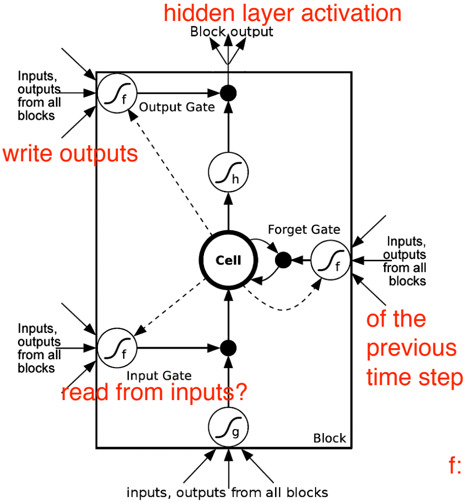
Le et al., 2015. A simple way to initialize recurrent networks of rectified linear units.

Long-Term Short Term Memory

- 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.

Long-Term Short Term Memory

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: input, f:forget

- $i = 0, f = 1 \Rightarrow$ remember the previous value identity function
- $i = 1, f = 1 \Rightarrow$ add to the previous value
- $i = 0, f = 0 \Rightarrow$ erase the value
- $i = 1, f = 0 \Rightarrow$ overwrite the value

Setting $i = 0, f = 1$ gives the reasonable "default" behavior of just remembering things.

f: logistic

Long-Term Short Term Memory

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

3 gates and input to cell g

$$\begin{pmatrix} \mathbf{i}_t \\ \mathbf{f}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{y}_t \\ \mathbf{h}_{t-1} \end{pmatrix}$$

input

hidden activation

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t$$

output

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t)$$

- Exercise: show that if $\mathbf{f}_{t+1} = 1$, $\mathbf{i}_{t+1} = 0$, and $\mathbf{o}_t = 0$, the gradients for the memory cell get passed through unmodified, i.e.

$$\overline{\mathbf{c}_t} = \overline{\mathbf{c}_{t+1}}.$$

verify identity function

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/`