Lecture 16: Learning Long-Term Dependencies

Roger Grosse

1 Introduction

Last lecture, we introduced RNNs and saw how to derive the gradients using backprop through time. In principle, this lets us train them using gradient descent. But in practice, gradient descent doesn't work very well unless we're careful. The problem is that we need to learn dependencies over long time windows, and the gradients can explode or vanish.

We'll first look at the problem itself, i.e. why gradients explode or vanish. Then we'll look at some techniques for dealing with the problem — most significantly, changing the architecture to one where the gradients are stable.

1.1 Learning Goals

- Understand why gradients explode or vanish, both
  - in terms of the mechanics of computing the gradients
  - the functional relationship between the hidden units at different time steps
- Be able to analyze simple examples of iterated functions, including identifying fixed points and qualitatively determining the long-term behavior from a given initialization.
- Know about various methods for dealing with the problem, and why they help:
  - Gradient clipping
  - Reversing the input sequence
  - Identity initialization
- Be familiar with the long short-term memory (LSTM) architecture
  - Reason about how the memory cell behaves for a given setting of the input, output, and forget gates
  - Understand how this architecture helps keep the gradients stable

2 Why Gradients Explode or Vanish

Recall the encoder-decoder architecture for machine translation, shown again in Figure 1. It has to read an English sentence, store as much information as possible in its hidden activations, and output a French sentence. The information about the first word in the sentence doesn't get used in the

The encoder-decoder model was introduced in Section 14.4.2.

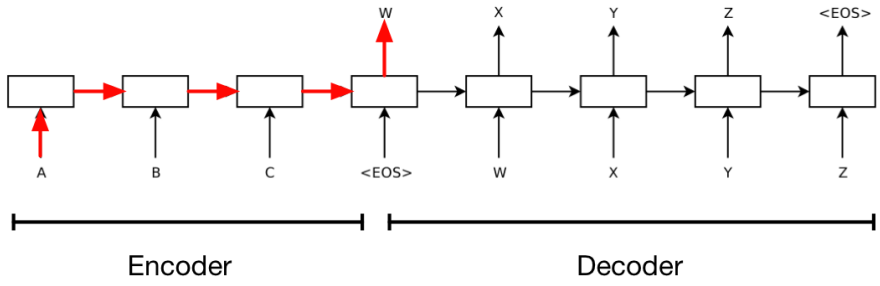
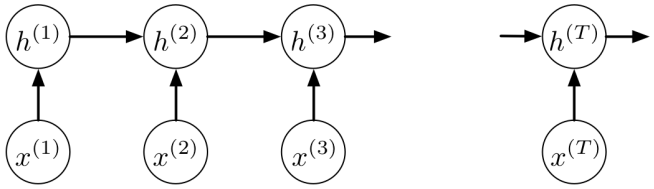


Figure 1: Encoder-decoder model for machine translation (see 14.4.2 for full description). Note that adjusting the weights based on the first input requires the error signal to travel backwards through the entire path highlighted in red.

predictions until it starts generating. Since a typical sentence might be about 20 words long, this means there's a long temporal gap from when it sees an input to when it uses that to make a prediction. It can be hard to learn long-distance dependencies, for reasons we'll see shortly. In order to adjust the input-to-hidden weights based on the first input, the error signal needs to travel backwards through this entire pathway (shown in red in Figure 1).

2.1 The mechanics of backprop

Now consider a univariate version of the encoder:



Assume we've already backpropped through the decoder network, so we already have the error signal $\overline{h^{(T)}}$. We then alternate between the following two backprop rules:[1]

$$\overline{h^{(t)}} = \overline{z^{(t+1)}}\, w$$
$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)})$$

If we iterate the rules, we get the following formula:

$$\begin{aligned}\overline{h^{(1)}} &= w^{T-1}\phi'(z^{(2)}) \cdots \phi'(z^{(T)})\overline{h^{(T)}} \\ &= \frac{\partial h^{(T)}}{\partial h^{(1)}}\overline{h^{(T)}}\end{aligned}$$

Hence, $\overline{h^{(1)}}$ is a linear function of $\overline{h^{(T)}}$. The coefficient is the partial derivative $\frac{\partial h^{(T)}}{h^{(1)}}$. If we make the simplifying assumption that the activation func-

[1] Review Section 14.3 if you're hazy on backprop through time.

tions are linear, we get

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

which can clearly explode or vanish unless w is very close to 1. For instance, if $w - 1.1$ and $T = 50$, we get $\partial h^{(T)}/\partial h^{(1)} = 117.4$, whereas if $w = 0.9$ and $T = 50$, we get $\partial h^{(T)}/\partial h^{(1)} = 0.00515$. In general, with nonlinear activation functions, there's nothing special about $w = 1$; the boundary between exploding and vanishing will depend on the values $h^{(t)}$.

More generally, in the multivariate case,

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

This quantity is called the **Jacobian**. It can explode or vanish just like in the univariate case, but this is slightly more complicated to make precise. In the case of linear activation functions, $\partial \mathbf{h}^{(t+1)}/\partial \mathbf{h}^{(t)} = \mathbf{W}$, so

"Jacobian" is a general mathematical term for the matrix of partial derivatives of a vector-valued function.

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \mathbf{W}^{T-1}.$$

This will explode if the largest eigenvalue of $\mathbf{W}$ is larger than 1, and vanish if the largest eigenvalue is smaller than 1.

Contrast this with the behavior of the forward pass. In the forward pass, the activations at each step are put through a nonlinear activation function, which typically squashes the values, preventing them from blowing up. Since the backwards pass is entirely linear, there's nothing to prevent the derivatives from blowing up.

2.2 Iterated functions

We just talked about why gradients explode or vanish, in terms of the mechanics of backprop. But whenever you're trying to reason about a phenomenon, don't go straight to the equations. Instead, try to think qualitatively about what's going on. In this case, there's actually a nice interpretation of the problem in terms of the function the RNN computes. In particular, ~~each layer computes a function of the current input and the previous hidden activations,~~ i.e. $\mathbf{h}^{(t)} = f(\mathbf{x}^{(t)}, \mathbf{h}^{(t-1)})$. If we expand this recursively, we get:

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

This looks a bit like repeatedly applying the function f. Therefore, we can gain some intuition for how RNNs behave by studying **iterated functions**, i.e. functions which we iterate many times.

Iterated functions can be complicated. Consider the innocuous-looking quadratic function

$$f(x) = 3.5\, x\, (1 - x). \tag{2}$$

If we iterate this function multiple times (i.e. $f(f(f(x)))$, etc.), we get some complicated behavior, as shown in Figure 2. Another famous example of the complexity of iterated functions is the Mandelbrot set:

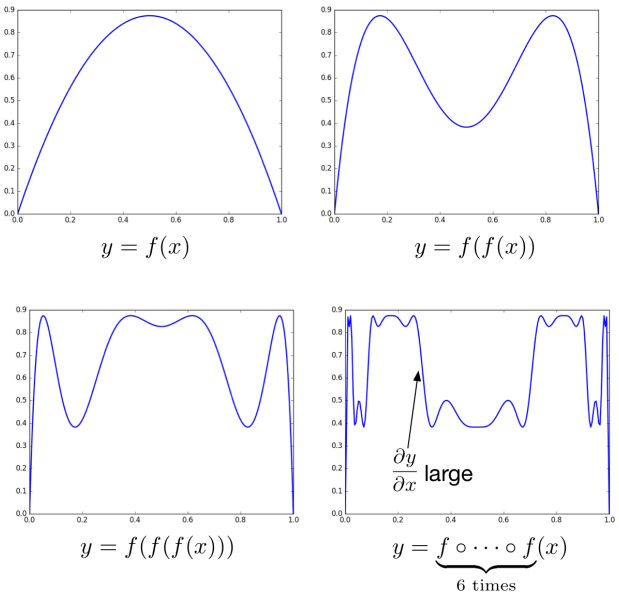
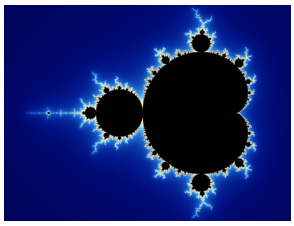


Figure 2: Iterations of the function $f(x) = 3.5\,x\,(1-x)$.



This is defined in terms of a simple mapping over the complex plane:

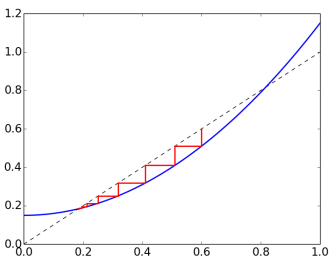
$$z_n = z_{n-1}^2 + c \tag{3}$$

If you initialize at $z_0 = 0$ and iterate this mapping, it will either stay within some bounded region or shoot off to infinity, and the behavior depends on the value of c. The Mandelbrot set is the set of values of c where it stays bounded; as you can see, this is an incredibly complex fractal.

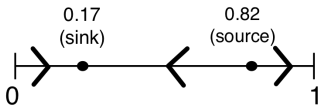
It's a bit easier to analyze iterated functions if they're monotonic. Consider the function

$$f(x) = x^2 + 0.15.$$

This is monotonic over $[0, 1]$. We can determine the behavior of repeated iterations visually:

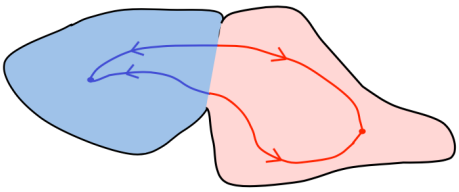


Here, the red line shows the trajectory of the iterates. If the initial value is $x_0 = 0.6$, start with your pencil at $x = y = 0.6$, which lies on the dashed line. Set $y = f(x)$ by moving your pencil vertically to the graph of the function, and then set $x = y$ by moving it horizontally to the dashed line. Repeat this procedure, and you should notice a pattern. There are some regions where the iterates move to the left, and other regions where the move to the right. Eventually, the iterates either shoot off to infinity or wind up at a **fixed point**, i.e. a point where $x = f(x)$. Fixed points are represented graphically as points where the graph of x intersects the dashed line. Some fixed points (such as 0.82 in this example) repel the iterates; these are called **sources**. Other fixed points (such as 0.17) attract the iterates; these are called **sinks**, or **attractors**. The behavior of the system can be summarized with a **phase plot**:



Observe that fixed points with derivatives $f'(x) < 1$ are sinks and fixed points with $f'(x) > 1$ are sources.

Even though the computations of an RNN are discrete, we can think of them as a sort of dynamical system, which has various attractors:



– Geoffrey Hinton, Coursera

This figure is a cartoon of the space of hidden activations. If you start out in the blue region, you wind up in one attractor, whereas if you start out in the red region, you wind up in the other attractor. If you evaluate the Jacobian $\partial \mathbf{h}^{(T)} / \partial \mathbf{h}^{(1)}$ in the interior of one of these regions, it will be close to 0, since if you change the initial conditions slightly, you still wind up at exactly the same place. But the Jacobian right on the boundary will be large, since shifting the initial condition slightly moves us from one attractor to the other.

To make this story more concrete, consider the following RNN, which uses the tanh activation function:

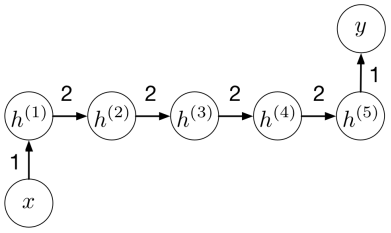


Figure 3 shows the function computed at each time step, as well as the function computed by the network as a whole. From this figure. you can see which regions have exploding or vanishing gradients.

Think about how we can derive the right-hand figure from the left-hand one using the analysis given above.

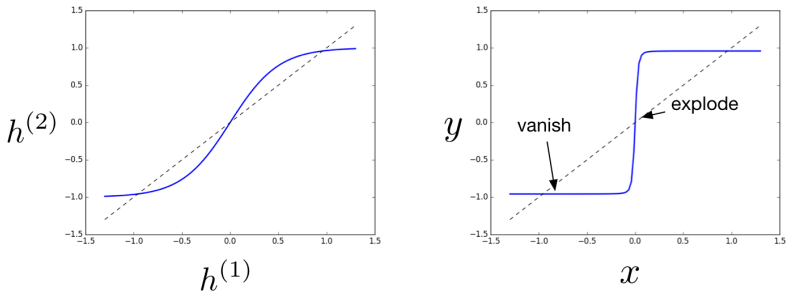


Figure 3: **(left)** The function computed by the RNN at each time step, **(right)** the function computed by the network.

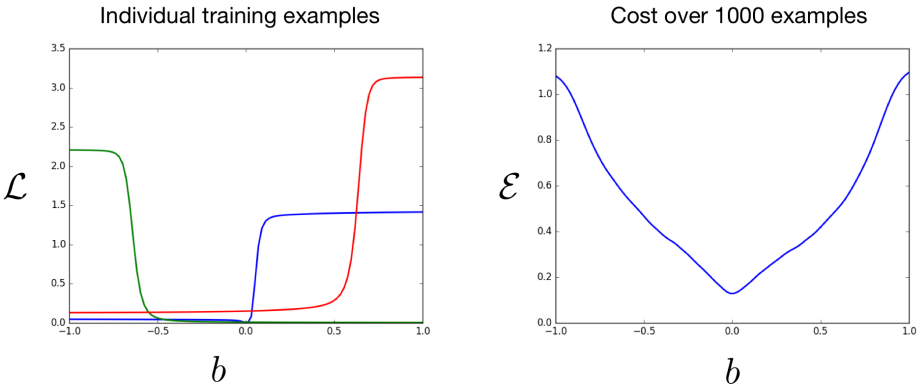


Figure 4: **(left)** Loss function for individual training examples, **(right)** cost function averaged over 1000 training examples.

This behavior shows up even if we look at the gradients with respect to parameters of the network. Suppose we define an input distribution and loss function for this network; we'll use squared error loss, but the details aren't important. Figure 4 shows the loss function for individual training examples, as well as the cost function averaged over 1000 training examples. Recall our discussion of features of the optimization landscape (plateaux, ridges, etc.). This figure shows a new one, namely **cliffs**. In this case, cliffs are a problem only for individual training examples; the cost function averaged over 1000 examples is fairly smooth. Whether or not this happens depends on the specifics of the problem.

3 Keeping Things Stable

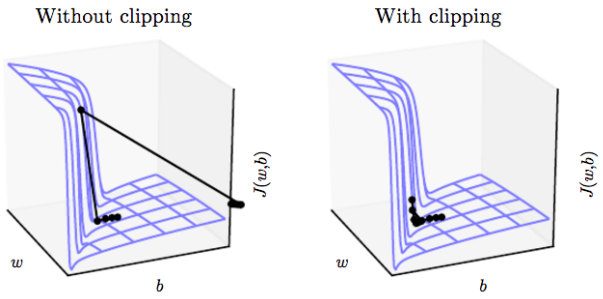
Now that we've introduced the problem of exploding and vanishing gradients, let's see what we can do about it. We'll start with some simple tricks, and then consider a fundamental change to the network architecture.

3.1 Gradient Clipping

First, there's a simple trick which sometimes helps a lot: **gradient clipping**. Basically, we prevent gradients from blowing up by rescaling them so that their norm is at most a particular value η. I.e., if $\|\mathbf{g}\| > \eta$, where $\mathbf{g}$ is the gradient, we set

$$\mathbf{g} \leftarrow \frac{\eta \mathbf{g}}{\|\mathbf{g}\|}. \tag{4}$$

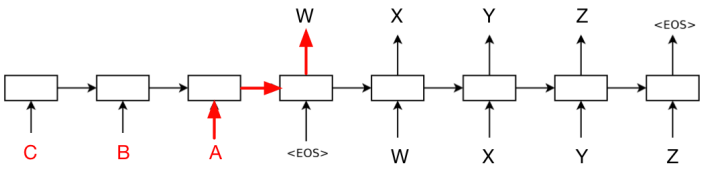
This biases the training procedure, since the resulting values won't actually be the gradient of the cost function. However, this bias can be worth it if it keeps things stable. The following figure shows an example with a cliff and a narrow valley; if you happen to land on the face of the cliff, you take a huge step which propels you outside the good region. With gradient clipping, you can stay within the valley.



— Goodfellow et al., *Deep Learning*

3.2 Input Reversal

Recall that we motivated this whole discussion in terms of the difficulty of learning long-distance dependencies, such as between the first word of the input sentence and the first word of the output sentence. What makes it especially tricky in translation is that *all* of the dependencies are long; this happens because for similar languages like English and French, the corresponding words appear in roughly the same order in both sentences, so the gaps between input and output are all roughly the sentence length. We can fix this by reversing the order of the words in the input sentence:



There's a gap of only one time step between when the first word is read and when it's needed. This means that the network can easily learn the relationships between the first words; this could allow it to learn good word representations, for instance. Once it's learned this, it can go on to the more difficult dependencies between words later in the sentences.

3.3 Identity Initialization

In general, iterated functions can have complex and chaotic behavior. But there's one particular function you can iterate as many times as you like: the identity function $f(x) = x$. If your network computes the identity function, the gradient computation will be perfectly stable, since the Jacobian is simply the identity matrix. Of course, the identity function isn't a very interesting thing to compute, but it still suggests we can keep things stable by encouraging the computations to stay close to the identity function.

The **identity RNN** architecture[2] is a kind of RNN where the activation functions are all ReLU, and the recurrent weights are initialized to the identity matrix. The ReLU activation function clips the activations to be nonnegative, but for nonnegative activations, it's equivalent to the identity function. This simple initialization trick achieved some neat results; for instance, it was able to classify MNIST digits which were fed to the network one pixel at a time, as a length-784 sequence.

3.4 Long-Term Short Term Memory

We've just talked about three tricks for training RNNs, and they are all pretty widely used. But the identity initialization trick actually gets at something much more fundamental. That is, why is it a good idea for the RNN to compute something close to the identity function? Think about how a computer works. It has a very large memory, but each instruction accesses only a handful of memory locations. All the other memory locations simply keep their previous value. In other words, if the computer's entire memory is represented as one big vector, the mapping from one time step to the next is very close to the identity function. This behavior is the most basic thing we'd desire from a memory system: the ability to preserve information over time until it's needed.

Unfortunately, the basic RNN architectures we've talked about so far aren't very good at remembering things. All of the units we've covered so far in the course consist of linear functions followed by a nonlinear activation function:

$$\mathbf{y} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b}). \tag{5}$$

For some activation functions, such as logistic or tanh, this can't even represent the identity mapping; e.g., in the network shown in Figure 3, each time step computes a function fairly close to the identity, but after just 5 steps, you get a step function.

The **Long-Term Short-Term Memory (LSTM)** architecture was designed to make it easy to remember information over long time periods until it's needed. The name refers to the idea that the activations of a network correspond to short-term memory, while the weights correspond to long-term memory. If the activations can preserve information over long distances, that makes them long-term short-term memory.

The basic LSTM unit (called a **block**) has much more internal structure than the units we've covered so far in this course. The architecture is shown in Figure 5. Each hidden layer of the RNN will be composed of many (e.g. hundreds or thousands) of these blocks.

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

- At the center is a **memory cell**, which is the thing that's able to remember information over time. It has a linear activation function, and a self-loop which is **modulated** by a **forget gate**, which takes values between 0 and 1; this means that the weight of the self-loop is equal to the value of the forget gate.
- The forget gate is a unit similar to the ones we've covered previously; it computes a linear function of its inputs, followed by a logistic activation function (which means its output is between 0 and 1). The forget gate would probably be better called a "remember gate", since if it is on (takes the value 1), the memory cell remembers its previous value, whereas if the forget gate is off, the cell forgets it.
- The block also receives inputs from other blocks in the network; these are summed together and passed through a tanh activation function (which squashes the values to be between -1 and 1). The connection from the input unit to the memory cell is gated by an **input gate**, which has the same functional form as the forget gate (i.e., linear-then-logistic).
- The block produces an output, which is the value of the memory cell, passed through a tanh activation function. It may or may not pass this on to the rest of the network; this connection is modulated by the **output gate**, which has the same form as the input and forget gates.

It's useful to summarize various behaviors the memory cell can achieve depending on the values of the input and forget gates:

input gate	forget gate	behavior
0	1	remember the previous value
1	1	add to the previous value
0	0	erase the value
1	0	overwrite the value

If the forget gate is on and the input gate is off, the block simply computes the identity function, which is a useful default behavior. But the ability to read and write from it lets it implement more sophisticated computations. The ability to add to the previous value means these units can simulate a counter; this can be useful, for instance, when training a language model, if sentences tend to be of a particular length.

When we implement an LSTM, we have a bunch of vectors at each time step, representing the values of all the memory cells and each of the gates. Mathematically, the computations are as follows:

$$\begin{pmatrix} \mathbf{i}^{(t)} \\ \mathbf{f}^{(t)} \\ \mathbf{o}^{(t)} \\ \mathbf{g}^{(t)} \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{x}^{(t)} \\ \mathbf{h}^{(t-1)} \end{pmatrix} \tag{6}$$

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \circ \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \circ \mathbf{g}^{(t)} \tag{7}$$

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \circ \tanh(\mathbf{c}^{(t)}). \tag{8}$$

Here, (6) uses a shorthand for applying different activation functions to different parts of the vector. Observe that the blocks receive signals from

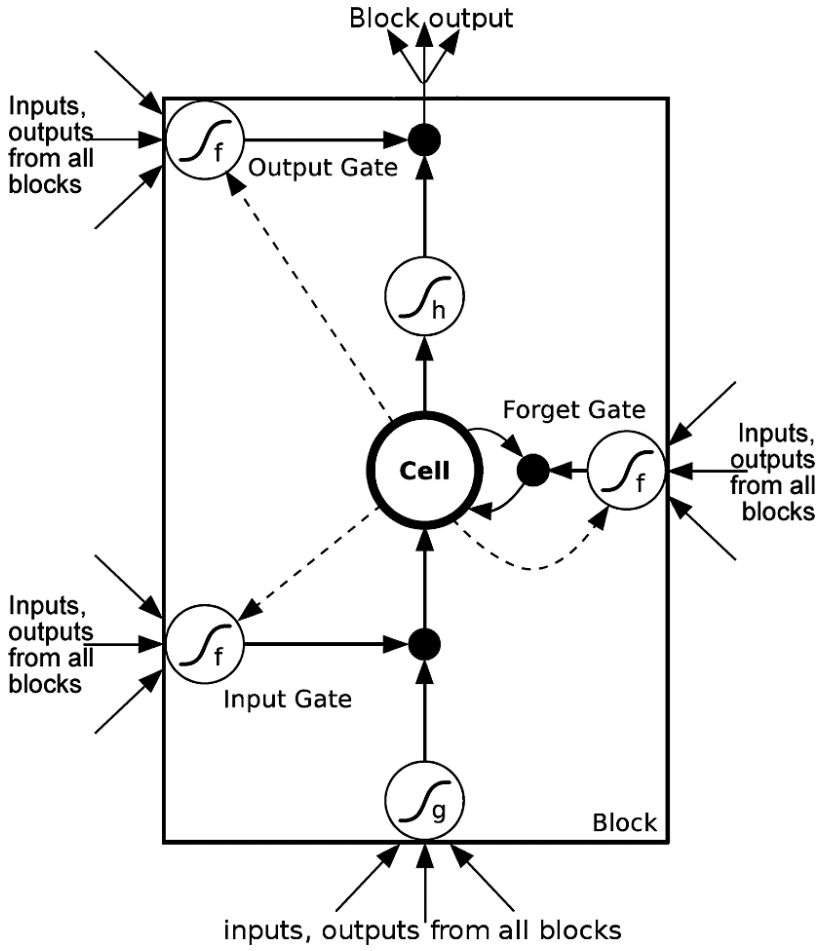


Figure 5: The LSTM unit.

the current inputs and the previous time step's hidden units, just like in standard RNNs. But the network's input $\mathbf{g}$ and the three gates $\mathbf{i}$, $\mathbf{o}$, and $\mathbf{f}$ have independent sets of incoming weights. Then (7) gives the update rule for the memory cell (think about how this relates to the verbal description above), and (8) defines the output of the block.

For homework, you are asked to show that if the forget gate is on and the input and output gates are off, it just passes the memory cell gradients through unmodified at each time step. Therefore, the LSTM architecture is resistant to exploding and vanishing gradients, although mathematically both phenomena are still possible.

If the LSTM architecture sounds complicated, that was the reaction of machine learning researchers when it was first proposed. It wasn't used much until 2013 and 2014, when resesarchers achieved impressive results on two challenging and important sequence prediction problems: speech-to-text and machine translation. Since then, they've become one of the most widely used RNN architectures; if someone tells you they're using an RNN, there's a good chance they're actually using an LSTM. There have been many attempts to simplify the architecture, and one particular variant called the gated recurrent unit (GRU) is fairly widely used, but so far nobody has found anything that's both simpler and at least as effective across the board. It appears that most of the complexity is probably required. Fortunately, you hardly ever have to think about it, since LSTMs are implemented as a black box in all of the major neural net frameworks.