APPENDIX C

Recurrent Neural Networks

So far, we have limited our attention to domains in which each output y is assumed to have been generated as a function of an associated input x, and our hypotheses have been "pure" functions, in which the output depends only on the input (and the parameters we have learned that govern the function's behavior). In the next few chapters, we are going to consider cases in which our models need to go beyond functions. In particular, behavior as a function of *time* will be an important concept:

- In *recurrent neural networks*, the hypothesis that we learn is not a function of a single input, but of the whole sequence of inputs that the predictor has received.
- In *reinforcement learning*, the hypothesis is either a *model* of a domain (such as a game) as a recurrent system or a *policy* which is a pure function, but whose loss is determined by the ways in which the policy interacts with the domain over time.

In this chapter, we introduce *state machines*. We start with deterministic state machines, and then consider recurrent neural network (RNN) architectures to model their behavior. Later, in Chapter 10, we will study *Markov decision processes* (MDPs) that extend to consider probabilistic (rather than deterministic) transitions in our state machines. RNNs and MDPs will enable description and modeling of temporally sequential patterns of behavior that are important in many domains.

C.1 State machines

A *state machine* is a description of a process (computational, physical, economic) in terms of its potential sequences of *states*.

This is such a pervasive idea that it has been given many names in many subareas of computer science, control theory, physics, etc., including: *automaton*, *transducer*, *dynamical system*, etc.

The *state* of a system is defined to be all you would need to know about the system to predict its future trajectories as well as possible. It could be the position and velocity of an object or the locations of your pieces on a game board, or the current traffic densities on a highway network.

Formally, we define a *state machine* as $(\mathcal{S}, \mathcal{X}, \mathcal{Y}, s_0, f_s, f_o)$ where

- $\mathcal{S}$ is a finite or infinite set of possible states;
- $\mathcal{X}$ is a finite or infinite set of possible inputs;

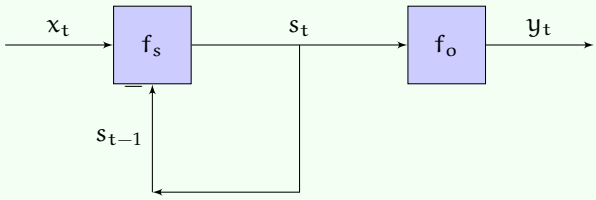
- $\mathcal{Y}$ is a finite or infinite set of possible outputs;
- $s_0 \in \mathcal{S}$ is the initial state of the machine;
- $f_s : \mathcal{S} \times \mathcal{X} \to \mathcal{S}$ is a *transition function*, which takes an input and a previous state and produces a next state;
- $f_o : \mathcal{S} \to \mathcal{Y}$ is an *output function*, which takes a state and produces an output.

The basic operation of the state machine is to start with state s_0, then iteratively compute for $t \geqslant 1$:

$$s_t = f_s(s_{t-1}, x_t) \tag{C.1}$$
$$y_t = f_o(s_t) \tag{C.2}$$

In some cases, we will pick a starting state from a set or distribution.

The diagram below illustrates this process. Note that the "feedback" connection of s_t back into f_s has to be buffered or delayed by one time step—-otherwise what it is computing would not generally be well defined.



So, given a sequence of inputs $x_1, x_2, \ldots$ the machine generates a sequence of outputs

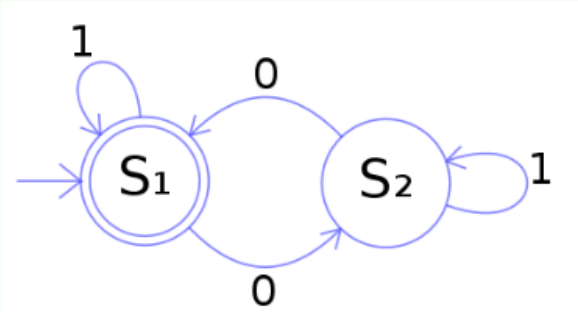
$$\underbrace{f_o(f_s(s_0, x_1))}_{y_1}, \underbrace{f_o(f_s(f_s(s_0, x_1), x_2))}_{y_2}, \ldots \ .$$

We sometimes say that the machine *transduces* sequence x into sequence y. The output at time t can have dependence on inputs from steps 1 to t.

One common form is *finite state machines*, in which $\mathcal{S}$, $\mathcal{X}$, and $\mathcal{Y}$ are all finite sets. They are often described using *state transition diagrams* such as the one below, in which nodes stand for states and arcs indicate transitions. Nodes are labeled by which output they generate and arcs are labeled by which input causes the transition.

There are a huge number of major and minor variations on the idea of a state machine. We'll just work with one specific one in this section and another one in the next, but don't worry if you see other variations out in the world!

One can verify that the state machine below reads binary strings and determines the parity of the number of zeros in the given string. Check for yourself that all input binary strings end in state S_1 if and only if they contain an even number of zeros.



All computers can be described, at the digital level, as finite state machines. Big, but finite!

Another common structure that is simple but powerful and used in signal processing and control is *linear time-invariant (LTI) systems*. In this case, all the quantities are real-valued vectors: $\mathcal{S} = \mathbb{R}^m$, $\mathcal{X} = \mathbb{R}^l$ and $\mathcal{Y} = \mathbb{R}^n$. The functions f_s and f_o are linear functions of their inputs. The transition function is described by the state matrix A and the input matrix B; the output function is defined by the output matrix C, each with compatible dimensions. In discrete time, they can be defined by a linear difference equation, like

$$s_t = f_s(s_{t-1}, x_t) = As_{t-1} + Bx_t, \tag{C.3}$$
$$y_t = f_o(s_t) = Cs_t, \tag{C.4}$$

and can be implemented using state to store relevant previous input and output information. We will study *recurrent neural networks* which are a lot like a non-linear version of an LTI system.

C.2 Recurrent neural networks

In Chapter 6, we studied neural networks and how the weights of a network can be obtained by training on data, so that the neural network will model a function that approximates the relationship between the (x, y) pairs in a supervised-learning training set. In Section C.1 above, we introduced state machines to describe sequential temporal behavior. Here in Section C.2, we explore recurrent neural networks by defining the architecture and weight matrices in a neural network to enable modeling of such state machines. Then, in Section C.3, we present a loss function that may be employed for training *sequence to sequence* RNNs, and then consider application to language translation and recognition in Section C.4. In Section C.5, we'll see how to use gradient-descent methods to train the weights of an RNN so that it performs a *transduction* that matches as closely as possible a training set of input-output *sequences*.

A *recurrent neural network* is a state machine with neural networks constituting functions f_s and f_o:

$$s_t = f_s\left(W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}\right) \tag{C.5}$$
$$y_t = f_o\left(W^{o}s_t + W_0^{o}\right) \quad . \tag{C.6}$$

The inputs, states, and outputs are all vector-valued:

$$x_t : \ell \times 1 \tag{C.7}$$
$$s_t : m \times 1 \tag{C.8}$$
$$y_t : n \times 1 \quad . \tag{C.9}$$

The weights in the network, then, are

$$W^{sx} : m \times \ell \tag{C.10}$$
$$W^{ss} : m \times m \tag{C.11}$$
$$W_0^{ss} : m \times 1 \tag{C.12}$$
$$W^{o} : n \times m \tag{C.13}$$
$$W_0^{o} : n \times 1 \tag{C.14}$$

with activation functions f_s and f_o.

Study Question: Check dimensions here to be sure it all works out. Remember that we apply f_s and f_o elementwise, unless f_o is a softmax activation.

C.3 Sequence-to-sequence RNN

Now, how can we set up an RNN to model and be trained to produce a transduction of one sequence to another? This problem is sometimes called *sequence-to-sequence* mapping. You can think of it as a kind of regression problem: given an input sequence, learn to generate the corresponding output sequence.

A training set has the form $\left[\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(q)}, y^{(q)}\right)\right]$, where

- $x^{(i)}$ and $y^{(i)}$ are length $n^{(i)}$ sequences;
- sequences in the *same pair* are the same length; and sequences in different pairs may have different lengths.

One way to think of training a sequence **classifier** is to reduce it to a transduction problem, where $y_t = 1$ if the sequence $x_1, \ldots, x_t$ is a *positive* example of the class of sequences and -1 otherwise.

Next, we need a loss function. We start by defining a loss function on sequences. There are many possible choices, but usually it makes sense just to sum up a per-element loss function on each of the output values, where g is the predicted sequence and y is the actual one:

$$\mathcal{L}_{\text{seq}}\left(g^{(i)}, y^{(i)}\right) = \sum_{t=1}^{n^{(i)}} \mathcal{L}_{\text{elt}}\left(g_t^{(i)}, y_t^{(i)}\right) \quad . \tag{C.15}$$

The per-element loss function $\mathcal{L}_{\text{elt}}$ will depend on the type of y_t and what information it is encoding, in the same way as for a supervised network.

So it could be NLL, squared loss, etc.

Then, letting $W = (W^{sx}, W^{ss}, W^{o}, W_0^{ss}, W_0^{o})$, our overall goal is to minimize the objective

$$J(W) = \frac{1}{q} \sum_{i=1}^{q} \mathcal{L}_{\text{seq}}\left(\text{RNN}(x^{(i)}; W), y^{(i)}\right) \quad , \tag{C.16}$$

where $\text{RNN}(x; W)$ is the output sequence generated, given input sequence x.

It is typical to choose f_s to be *tanh* but any non-linear activation function is usable. We choose f_o to align with the types of our outputs and the loss function, just as we would do in regular supervised learning.

Remember that it looks like a sigmoid but ranges from -1 to +1.

C.4 RNN as a language model

A *language model* is a sequence to sequence RNN which is trained on a token sequence of the form, $c = (c_1, c_2, \ldots, c_k)$, and is used to predict the next token $c_t, t \leqslant k$, given a sequence of the previous $(t-1)$ tokens:

A "token" is generally a character, common word fragment, or a word.

$$c_t = \text{RNN}\left((c_1, c_2, \ldots, c_{t-1}); W\right) \tag{C.17}$$

We can convert this to a sequence-to-sequence training problem by constructing a data set of q different (x, y) sequence pairs, where we make up new special tokens, start and end, to signal the beginning and end of the sequence:

$$x = (\langle \text{start} \rangle, c_1, c_2, \ldots, c_k) \tag{C.18}$$

$$y = (c_1, c_2, \ldots, \langle \text{end} \rangle) \tag{C.19}$$

C.5 Back-propagation through time

Now the fun begins! We can now try to find a W to minimize J using gradient descent. We will work through the simplest method, *back-propagation through time* (BPTT), in detail. This is generally not the best method to use, but it's relatively easy to understand. In Section C.6 we will sketch alternative methods that are in much more common use.

What we want you to take away from this section is that, by "unrolling" a recurrent network out to model a particular sequence, we can treat the whole thing as a feed-forward network with a lot of parameter sharing. Thus, we can tune the parameters using stochastic gradient descent, and learn to model sequential mappings. The concepts here are very important. While the details are important to get right if you need to implement something, we present the mathematical details below primarily to convey or explain the larger concepts.

Calculus reminder: total derivative Most of us are not very careful about the difference between the *partial derivative* and the *total derivative*. We are going to use a nice example from the Wikipedia article on partial derivatives to illustrate the difference. The volume of a circular cone depends on its height and radius:

$$V(r, h) = \frac{\pi r^2 h}{3} \ . \tag{C.20}$$

The partial derivatives of volume with respect to height and radius are

$$\frac{\partial V}{\partial r} = \frac{2\pi r h}{3} \quad \text{and} \quad \frac{\partial V}{\partial h} = \frac{\pi r^2}{3} \ . \tag{C.21}$$

They measure the change in V assuming everything is held constant except the single variable we are changing. Now assume that we want to preserve the cone's proportions in the sense that the ratio of radius to height stays constant. Then we can't really change one without changing the other. In this case, we really have to think about the *total derivative*. If we're interested in the total derivative with respect to r, we sum the "paths" along which r might influence V:

$$\frac{dV}{dr} = \frac{\partial V}{\partial r} + \frac{\partial V}{\partial h}\frac{dh}{dr} \tag{C.22}$$

$$= \frac{2\pi r h}{3} + \frac{\pi r^2}{3}\frac{dh}{dr} \tag{C.23}$$

Or if we're interested in the total derivative with respect to h, we consider how h might influence V, either directly or via r:

$$\frac{dV}{dh} = \frac{\partial V}{\partial h} + \frac{\partial V}{\partial r}\frac{dr}{dh} \tag{C.24}$$

$$= \frac{\pi r^2}{3} + \frac{2\pi r h}{3}\frac{dr}{dh} \tag{C.25}$$

Just to be completely concrete, let's think of a right circular cone with a fixed angle $\alpha = \tan r/h$, so that if we change r or h then α remains constant. So we have $r = h \tan^{-1} \alpha$; let constant $c = \tan^{-1} \alpha$, so now $r = ch$. Thus, we finally have

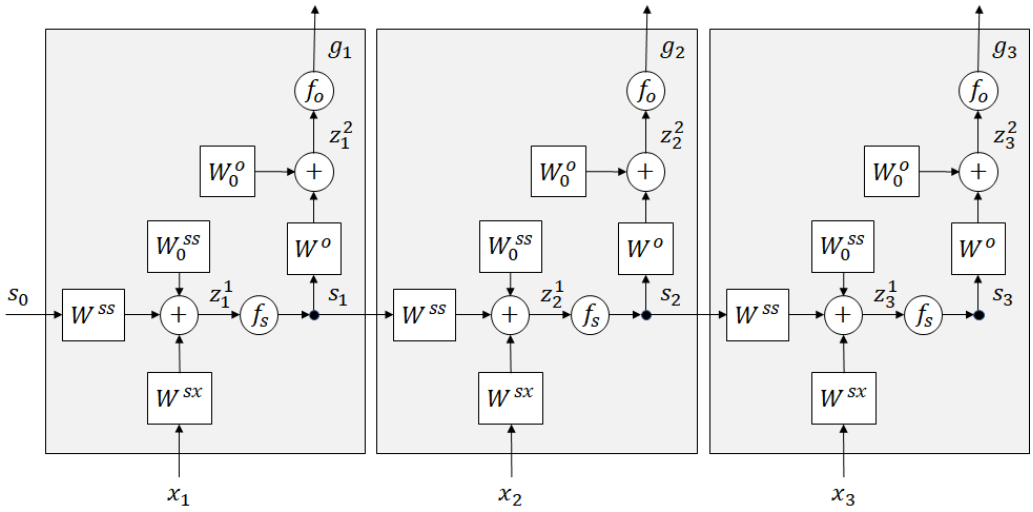
$$\frac{dV}{dr} = \frac{2\pi r h}{3} + \frac{\pi r^2}{3}\frac{1}{c} \tag{C.26}$$

$$\frac{dV}{dh} = \frac{\pi r^2}{3} + \frac{2\pi r h}{3}c \ . \tag{C.27}$$

The BPTT process goes like this:

(1) Sample a training pair of sequences (x, y); let their length be n.

(2) "Unroll" the RNN to be length n (picture for $n = 3$ below), and initialize s_0:



Now, we can see our problem as one of performing what is almost an ordinary back-propagation training procedure in a feed-forward neural network, but with the difference that the weight matrices are shared among the layers. In many ways, this is similar to what ends up happening in a convolutional network, except in the convnet, the weights are re-used spatially, and here, they are re-used temporally.

(3) Do the *forward pass*, to compute the predicted output sequence g:

$$z_t^1 = W^{sx} x_t + W^{ss} s_{t-1} + W_0^{ss} \tag{C.28}$$

$$s_t = f_s(z_t^1) \tag{C.29}$$

$$z_t^2 = W^o s_t + W_0^o \tag{C.30}$$

$$g_t = f_o(z_t^2) \tag{C.31}$$

(4) Do *backward pass* to compute the gradients. For both W^{ss} and W^{sx} we need to find

$$\frac{d\mathcal{L}_{seq}(g, y)}{dW} = \sum_{u=1}^{n} \frac{d\mathcal{L}_{elt}(g_u, y_u)}{dW}$$

Letting $\mathcal{L}_u = \mathcal{L}_{elt}(g_u, y_u)$ and using the *total derivative*, which is a sum over all the ways in which W affects $\mathcal{L}_u$, we have

$$= \sum_{u=1}^{n} \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \frac{\partial \mathcal{L}_u}{\partial s_t}$$

Re-organizing, we have

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \sum_{u=1}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t}$$

Because s_t only affects $\mathcal{L}_t, \mathcal{L}_{t+1}, \ldots, \mathcal{L}_n$,

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \sum_{u=t}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t}$$

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \left(\frac{\partial \mathcal{L}_t}{\partial s_t} + \underbrace{\sum_{u=t+1}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t}}_{\delta^{s_t}} \right) . \tag{C.32}$$

where δ^{s_t} is the dependence of the future loss (incurred after step t) on the state S_t. We can compute this backwards, with t going from n down to 1. The trickiest part is figuring out how early states contribute to later losses. We define the *future loss* after step t to be

That is, δ^{s_t} is how much we can blame state s_t for all the future element losses.

$$F_t = \sum_{u=t+1}^{n} \mathcal{L}_{\text{elt}}(g_u, y_u) , \tag{C.33}$$

so

$$\delta^{s_t} = \frac{\partial F_t}{\partial s_t} . \tag{C.34}$$

At the last stage, $F_n = 0$ so $\delta^{s_n} = 0$.

Now, working backwards,

$$\delta^{s_{t-1}} = \frac{\partial}{\partial s_{t-1}} \sum_{u=t}^{n} \mathcal{L}_{\text{elt}}(g_u, y_u) \tag{C.35}$$

$$= \frac{\partial s_t}{\partial s_{t-1}} \frac{\partial}{\partial s_t} \sum_{u=t}^{n} \mathcal{L}_{\text{elt}}(g_u, y_u) \tag{C.36}$$

$$= \frac{\partial s_t}{\partial s_{t-1}} \frac{\partial}{\partial s_t} \left[\mathcal{L}_{\text{elt}}(g_t, y_t) + \sum_{u=t+1}^{n} \mathcal{L}_{\text{elt}}(g_u, y_u) \right] \tag{C.37}$$

$$= \frac{\partial s_t}{\partial s_{t-1}} \left[\frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial s_t} + \delta^{s_t} \right] \tag{C.38}$$

Now, we can use the chain rule again to find the dependence of the element loss at time t on the state at that same time,

$$\underbrace{\frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial s_t}}_{(m \times 1)} = \underbrace{\frac{\partial z_t^2}{\partial s_t}}_{(m \times \nu)} \underbrace{\frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial z_t^2}}_{(\nu \times 1)} , \tag{C.39}$$

and the dependence of the state at time t on the state at the previous time,

$$\underbrace{\frac{\partial s_t}{\partial s_{t-1}}}_{(m \times m)} = \underbrace{\frac{\partial z_t^1}{\partial s_{t-1}}}_{(m \times m)} \underbrace{\frac{\partial s_t}{\partial z_t^1}}_{(m \times m)} = W^{ss\,\mathsf{T}} \frac{\partial s_t}{\partial z_t^1} \tag{C.40}$$

Note that $\partial s_t / \partial z_t^1$ is formally an $m \times m$ diagonal matrix, with the values along the diagonal being $f_s'(z_{t,i}^1)$, $1 \leqslant i \leqslant m$. But since this is a diagonal matrix, one could represent it as an $m \times 1$ vector $f_s'(z_t^1)$. In that case the product of the matrix $W^{ss\,T}$ by the vector $f_s'(z_t^1)$, denoted $W^{ss\,T} * f_s'(z_t^1)$, should be interpreted as follows: take the first column of the matrix $W^{ss\,T}$ and multiply each of its elements by the first element of the vector $\partial s_t / \partial z_t^1$, then take the second column of the matrix $W^{ss\,T}$ and multiply each of its elements by the second element of the vector $\partial s_t / \partial z_t^1$, and so on and so forth ...

Putting this all together, we end up with

$$\delta^{s_{t-1}} = \underbrace{W^{ss\,T} \frac{\partial s_t}{\partial z_t^1}}_{\frac{\partial s_t}{\partial s_{t-1}}} \underbrace{\left(W^{o\,T} \frac{\partial \mathcal{L}_t}{\partial z_t^2} + \delta^{s_t} \right)}_{\frac{\partial F_{t-1}}{\partial s_t}} \tag{C.41}$$

We're almost there! Now, we can describe the actual weight updates. Using Eq. C.32 and recalling the definition of $\delta^{s_t} = \partial F_t / \partial s_t$, as we iterate backwards, we can accumulate the terms in Eq. C.32 to get the gradient for the whole loss.

$$\frac{d\mathcal{L}_{seq}}{dW^{ss}} = \sum_{t=1}^{n} \frac{d\mathcal{L}_{elt}(g_t, y_t)}{dW^{ss}} = \sum_{t=1}^{n} \frac{\partial z_t^1}{\partial W^{ss}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t} \tag{C.42}$$

$$\frac{d\mathcal{L}_{seq}}{dW^{sx}} = \sum_{t=1}^{n} \frac{d\mathcal{L}_{elt}(g_t, y_t)}{dW^{sx}} = \sum_{t=1}^{n} \frac{\partial z_t^1}{\partial W^{sx}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t} \tag{C.43}$$

We can handle W^o separately; it's easier because it does not affect future losses in the way that the other weight matrices do:

$$\frac{d\mathcal{L}_{seq}}{dW^o} = \sum_{t=1}^{n} \frac{d\mathcal{L}_t}{dW^o} = \sum_{t=1}^{n} \frac{\partial \mathcal{L}_t}{\partial z_t^2} \frac{\partial z_t^2}{\partial W^o} \tag{C.44}$$

Assuming we have $\frac{\partial \mathcal{L}_t}{\partial z_t^2} = (g_t - y_t)$, (which ends up being true for squared loss, softmax-NLL, etc.), then

$$\underbrace{\frac{d\mathcal{L}_{seq}}{dW^o}}_{v \times m} = \sum_{t=1}^{n} \underbrace{(g_t - y_t)}_{v \times 1} \underbrace{s_t^T}_{1 \times m} . \tag{C.45}$$

Whew!

Study Question: Derive the updates for the offsets W_0^{ss} and W_0^o.

C.6 Vanishing gradients and gating mechanisms

Let's take a careful look at the backward propagation of the gradient along the sequence:

$$\delta^{s_{t-1}} = \frac{\partial s_t}{\partial s_{t-1}} \left[\frac{\partial \mathcal{L}_{elt}(g_t, y_t)}{\partial s_t} + \delta^{s_t} \right] . \tag{C.46}$$

Consider a case where only the output at the end of the sequence is incorrect, but it depends critically, via the weights, on the input at time 1. In this case, we will multiply the loss at step n by

$$\frac{\partial s_2}{\partial s_1}\frac{\partial s_3}{\partial s_2}\cdots\frac{\partial s_n}{\partial s_{n-1}} \quad . \tag{C.47}$$

In general, this quantity will either grow or shrink exponentially with the length of the sequence, and make it very difficult to train.

Study Question: The last time we talked about exploding and vanishing gradients, it was to justify per-weight adaptive step sizes. Why is that not a solution to the problem this time?

An important insight that really made recurrent networks work well on long sequences is the idea of *gating*.

C.6.1 Simple gated recurrent networks

A computer only ever updates some parts of its memory on each computation cycle. We can take this idea and use it to make our networks more able to retain state values over time and to make the gradients better-behaved. We will add a new component to our network, called a *gating network*. Let g_t be a $m \times 1$ vector of values and let W^{gx} and W^{gs} be $m \times l$ and $m \times m$ weight matrices, respectively. We will compute g_t as

It can have an offset, too, but we are omitting it for simplicity.

$$g_t = \text{sigmoid}(W^{gx}x_t + W^{gs}s_{t-1}) \tag{C.48}$$

and then change the computation of s_t to be

$$s_t = (1 - g_t) * s_{t-1} + g_t * f_s(W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}) \ , \tag{C.49}$$

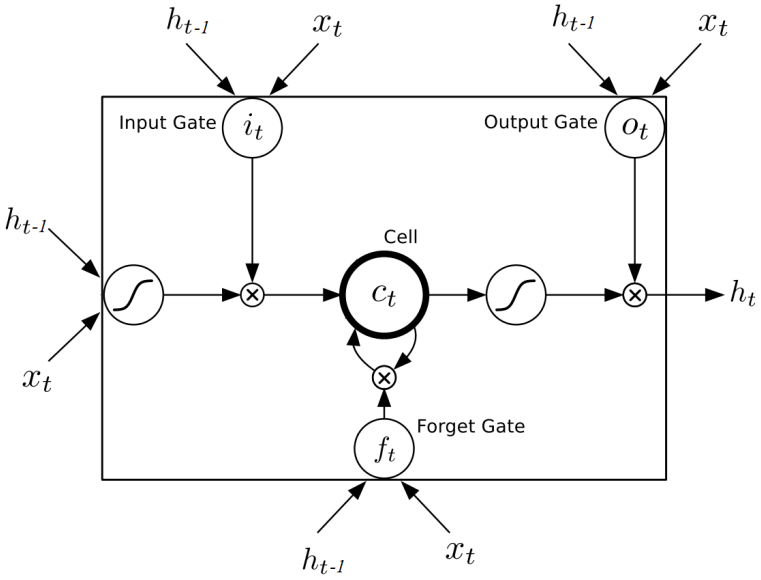
where $*$ is component-wise multiplication. We can see, here, that the output of the gating network is deciding, for each dimension of the state, how much it should be updated now. This mechanism makes it much easier for the network to learn to, for example, "store" some information in some dimension of the state, and then not change it during future state updates, or change it only under certain conditions on the input or other aspects of the state.

Study Question: Why is it important that the activation function for g be a sigmoid?

C.6.2 Long short-term memory

The idea of gating networks can be applied to make a state machine that is even more like a computer memory, resulting in a type of network called an LSTM for "long short-term memory." We won't go into the details here, but the basic idea is that there is a memory cell (really, our state vector) and three (!) gating networks. The *input* gate selects (using a "soft" selection as in the gated network above) which dimensions of the state will be updated with new values; the *forget* gate decides which dimensions of the state will have its old values moved toward 0, and the *output* gate decides which dimensions of the state will be used to compute the output value. These networks have been used in applications like language translation with really amazing results. A diagram of the architecture is shown below:

Yet another awesome name for a neural network!


h_{t-1}
x_t
h_{t-1}
x_t
Input Gate
i_t
Output Gate
o_t
h_{t-1}
Cell
c_t
h_t
x_t
Forget Gate
f_t
h_{t-1}
x_t