CHAPTER 12

Recurrent Neural Networks

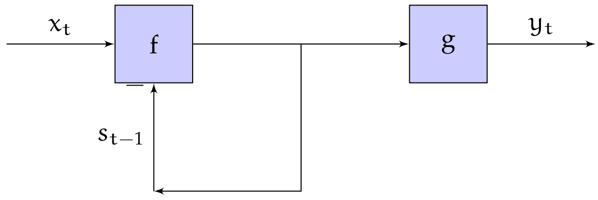
In chapter 8 we studied neural networks and how we can train the weights of a network, based on data, so that it will adapt into a function that approximates the relationship between the (x, y) pairs in a supervised-learning training set. In section 1 of chapter 10, we studied state-machine models and defined *recurrent neural networks* (RNNs) as a particular type of state machine, with a multidimensional vector of real values as the state. In this chapter, 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*.

1 RNN model

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

$$\begin{aligned} s_t &= f(s_{t-1}, x_t) \\ y_t &= g(s_t) \end{aligned}$$

as illustrated in the diagram below (remembering that there needs to be a delay on the feedback loop):



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

$$\underbrace{g(f(s_0, x_1))}_{y_1}, \underbrace{g(f(f(s_0, x_1), x_2,))}_{y_2}, \ldots \quad .$$

A *recurrent neural network* is a state machine with neural networks constituting functions f and g:

$$f(s, x) = f_1(W^{sx}x + W^{ss}s + W_0^{ss})$$
$$g(s) = f_2(W^O s + W_0^O) \quad .$$

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

We are very sorry! This course material has evolved from different sources, which used $W^T x$ in the forward pass for regular feed-forward NNs and Wx for the forward pass in RNNs. This inconsistency doesn't make any technical difference, but is a potential source of confusion.

$$x_t : \ell \times 1$$
$$s_t : m \times 1$$
$$y_t : v \times 1 \quad .$$

The weights in the network, then, are

$$W^{sx} : m \times \ell$$
$$W^{ss} : m \times m$$
$$W_0^{ss} : m \times 1$$
$$W^O : v \times m$$
$$W_0^O : v \times 1$$

with activation functions f_1 and f_2. Finally, the operation of the RNN is described by

$$s_t = f_1\left(W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}\right)$$
$$y_t = f_2\left(W^O s_t + W_0^O\right) \quad .$$

Study Question: Check dimensions here to be sure it all works out. Remember that we apply f_1 and f_2 elementwise.

2 Sequence-to-sequence RNN

Now, how can we train an RNN to model a transduction on sequences? 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 $\left[\left(x^{(1)}, y^{(1)}\right), \dots, \left(x^{(q)}, y^{(q)}\right)\right]$, where

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

- $x^{(i)}$ 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.

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 p is the predicted sequence and y is the actual one:

$$\text{Loss}_{\text{seq}}\left(p^{(i)}, y^{(i)}\right) = \sum_{t=1}^{n^{(i)}} \text{Loss}_{\text{elt}}\left(p_t^{(i)}, y_t^{(i)}\right) \quad .$$

The per-element loss function Loss_{elt} will depend on the type of y_t and what information it is encoding, in the same way as for a supervised network.. Then, letting $\theta =$

So it could be NLL, squared loss, etc.

$(W^{sx}, W^{ss}, W^O, W_0^{ss}, W_0^O)$, our overall objective is to minimize

$$J(\theta) = \sum_{i=1}^{q} \text{Loss}_{\text{seq}}\left(\text{RNN}(x^{(i)};\theta), y^{(i)}\right) \quad ,$$

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

It is typical to choose f_1 to be *tanh* but any non-linear activation function is usable. We choose f_2 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.

3 Back-propagation through time

Now the fun begins! We can find θ 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 5 we will sketch alternative methods that are in much more common use.

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:

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

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

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

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 stay constant, then we can't really change one without changing the other. In this case, we really have to think about the *total derivative*, which sums the "paths" along which r might influence V:

$$\begin{aligned}
\frac{dV}{dr} &= \frac{\partial V}{\partial r} + \frac{\partial V}{\partial h}\frac{dh}{dr} \\
&= \frac{2\pi r h}{3} + \frac{\pi r^2}{3}\frac{dh}{dr} \\
\frac{dV}{dh} &= \frac{\partial V}{\partial h} + \frac{\partial V}{\partial r}\frac{dr}{dh} \\
&= \frac{\pi r^2}{3} + \frac{2\pi r h}{3}\frac{dr}{dh}
\end{aligned}$$

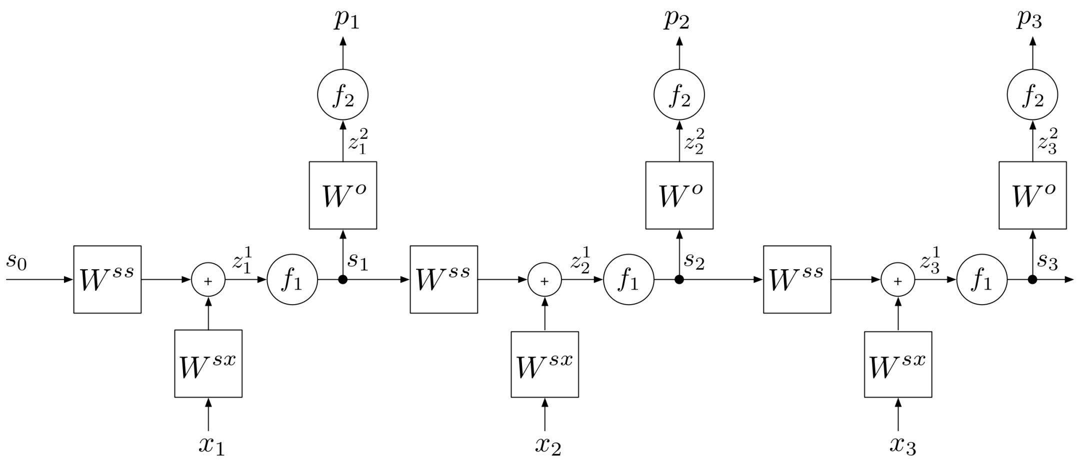
Just to be completely concrete, let's think of a right circular cone with a fixed angle $\alpha = \tan r/h$, so that if we change r or h then α remains constant. So we have $r = h \tan{}^{-}1\alpha$; let constant $c = \tan^{-1}\alpha$, so now $r = ch$. Now, we know that

$$\begin{aligned}
\frac{dV}{dr} &= \frac{2\pi r h}{3} + \frac{\pi r^2}{3}\frac{1}{c} \\
\frac{dV}{dh} &= \frac{\pi r^2}{3} + \frac{2\pi r h}{3}c
\end{aligned}$$

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 conv-net, the weights are re-used spatially, and here, they are re-used temporally.

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

$$
\begin{aligned}
z_t^1 &= W^{sx} x_t + W^{ss} s_{t-1} + W_0^{ss} \\
s_t &= f_1(z_t^1) \\
z_t^2 &= W^O s_t + W_0^O \\
p_t &= f_2(z_t^2)
\end{aligned}
$$

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

$$\frac{dL_{\text{seq}}}{dW} = \sum_{u=1}^{n} \frac{dL_u}{dW} \tag{12.1}$$

Letting $L_u = L_{\text{elt}}(p_u, y_u)$ and using the *total derivative*, which is a sum over all the ways in which W affects L_u, we have

$$= \sum_{u=1}^{n} \sum_{t=1}^{n} \frac{\partial L_u}{\partial s_t} \cdot \frac{\partial s_t}{\partial W} \tag{12.2}$$

Re-organizing, we have

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \cdot \sum_{u=1}^{n} \frac{\partial L_u}{\partial s_t} \tag{12.3}$$

Because s_t only affects $L_t, L_{t+1}, \ldots, L_n$,

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \cdot \sum_{u=t}^{n} \frac{\partial L_u}{\partial s_t}$$

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \cdot \left(\frac{\partial L_t}{\partial s_t} + \underbrace{\sum_{u=t+1}^{n} \frac{\partial L_u}{\partial s_t}}_{\delta^{s_t}} \right) \tag{12.4}$$

δ^{s_t} is the dependence of the loss on steps after t on the state at time t.

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

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 *future loss*

$$F_t = \sum_{u=t+1}^{n} \text{Loss}_{\text{elt}}(p_u, y_u) \quad ,$$

so

$$\delta^{s_t} = \frac{\partial F_t}{\partial s_t} \quad .$$

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

Now, working backwards,

$$\begin{aligned}
\delta^{s_{t-1}} &= \frac{\partial}{\partial s_{t-1}} \sum_{u=t}^{n} \text{Loss}_{\text{elt}}(p_u, y_u) \\
&= \frac{\partial s_t}{\partial s_{t-1}} \cdot \frac{\partial}{\partial s_t} \sum_{u=t}^{n} \text{Loss}_{\text{elt}}(p_u, y_u) \\
&= \frac{\partial s_t}{\partial s_{t-1}} \cdot \frac{\partial}{\partial s_t} \left[\text{Loss}_{\text{elt}}(p_t, y_t) + \sum_{u=t+1}^{n} \text{Loss}_{\text{elt}}(p_u, y_u) \right] \\
&= \frac{\partial s_t}{\partial s_{t-1}} \cdot \left[\frac{\partial \text{Loss}_{\text{elt}}(p_t, y_t)}{\partial s_t} + \delta^{s_t} \right]
\end{aligned}$$

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,

$$\underbrace{\frac{\partial \text{Loss}_{\text{elt}}(p_t, y_t)}{\partial s_t}}_{(m \times 1)} = \underbrace{\frac{\partial z_t^2}{\partial s_t}}_{(m \times v)} \cdot \underbrace{\frac{\partial \text{Loss}_{\text{elt}}(p_t, y_t)}{\partial z_t^2}}_{(v \times 1)} \quad ,$$

and the dependence of the state at time t on the state at the previous time, noting that we are performing an *elementwise* multiplication between W_{ss}^T and the vector of $f^{1'}$ values, $\partial s_t / \partial z_t^1$:

$$\underbrace{\frac{\partial s_t}{\partial s_{t-1}}}_{(m \times m)} = \underbrace{\frac{\partial z_t^1}{\partial s_{t-1}}}_{(m \times m)} \cdot \underbrace{\frac{\partial s_t}{\partial z_t^1}}_{(m \times 1)} = \underbrace{W^{ss\,T} * f^{1'}(z_t^1)}_{\text{not dot!}} \quad .$$

There are two ways to think about $\partial s_t / \partial z_t$: here, we take the view that it is an $m \times 1$ vector and we multiply each column of W^T by it. Another, equally good, view, is that it is an $m \times m$ diagonal matrix, with the values along the diagonal, and then this operation is a matrix multiply. Our software implementation will take the first view.

Putting this all together, we end up with

$$\delta^{s_{t-1}} = \underbrace{W^{ss\,T} * f^{1'}(z_t^1)}_{\frac{\partial s_t}{\partial s_{t-1}}} \cdot \underbrace{\left(W^{O\,T} \frac{\partial L_t}{\partial z_t^2} + \delta^{s_t} \right)}_{\frac{\partial F_{t-1}}{\partial s_t}}$$

We're almost there! Now, we can describe the actual weight updates. Using equation 12.4 and recalling the definition of $\delta^{s_t} = \partial F_t / \partial s_t$, as we iterate backwards, we can accumulate the terms in equation 12.4 to get the gradient for the whole loss:

$$\frac{dL_{seq}}{dW^{ss}} += \frac{\partial F_{t-1}}{\partial W^{ss}} = \frac{\partial z_t^1}{\partial W^{ss}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t}$$
$$\frac{dL_{seq}}{dW^{sx}} += \frac{\partial F_{t-1}}{\partial W^{sx}} = \frac{\partial z_t^1}{\partial W^{sx}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t}$$

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

$$\frac{\partial L_{seq}}{\partial W^O} = \sum_{t=1}^{n} \frac{\partial L_t}{\partial W^O} = \sum_{t=1}^{n} \frac{\partial L_t}{\partial z_t^2} \cdot \frac{\partial z_t^2}{\partial W^O}$$

Assuming we have $\frac{\partial L_t}{\partial z_t^2} = (p_t - y_t)$, (which ends up being true for squared loss, softmax-NLL, etc.), then on each iteration

$$\underbrace{\frac{\partial L_{seq}}{\partial W^O}}_{v\times m} += \underbrace{(p_t - y_t)}_{v\times 1} \cdot \underbrace{s_t^T}_{1\times m}$$

Whew!

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

4 Training a language model

A *language model* is just trained on a set of input sequences, $(c_1^{(i)}, c_2^{(i)}, \ldots, c_{n^i}^{(i)})$, and is used to predict the next character, given a sequence of previous tokens:

A "token" is generally a character or a word.

$$c_t = \text{RNN}(c_1, c_2, \ldots, c_{t-1})$$

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

$$x = (\langle\text{start}\rangle, c_1, c_2, ;c_n)$$
$$y = (c_1, c_2, \ldots, \langle\text{end}\rangle)$$

5 Vanishing gradients and gating mechanisms

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

$$\delta^{s_{t-1}} = \frac{\partial s_t}{\partial s_{t-1}} \cdot \left[\frac{\partial \text{Loss}_{\text{elt}}(p_t, y_t)}{\partial s_t} + \delta^{s_t}\right] .$$

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

$$\frac{\partial s_2}{\partial s_1} \cdot \frac{\partial s_3}{\partial s_2} \cdots \frac{\partial s_n}{\partial s_{n-1}} .$$

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*.

5.1 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.

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

and then change the computation of s_t to be

$$s_t = (1 - g_t) * s_{t-1} + g_t * f_1(W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}) \ ,$$

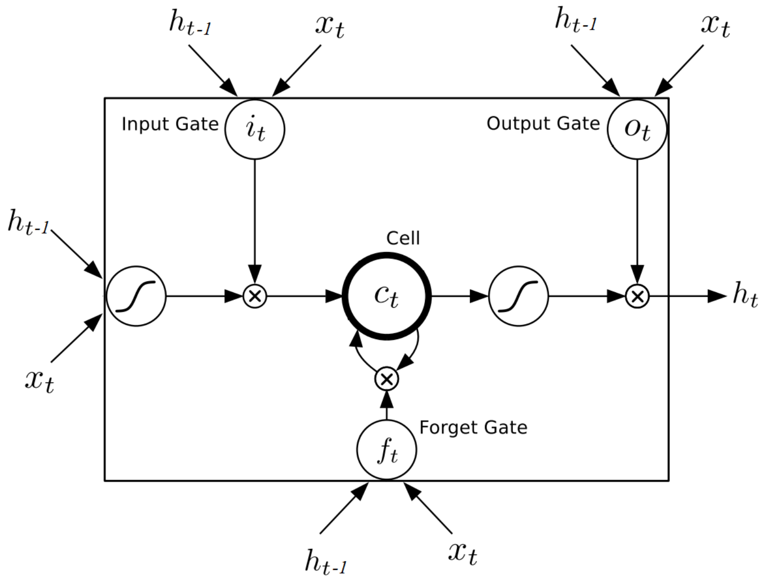
where $*$ is component-wise multiplication. We can see, here, that the output of the gating network is deciding, for each dimension of the state, how much it should be updated now. This mechanism makes it much easier for the network to learn to, for example, "store" some information in some dimension 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?

5.2 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!


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