Chapter 10

Recurrent Neural Networks

You can think of an RNN as a finite automaton or transducer whose transition function (δ) is defined by a neural network. It reads in a sequence of inputs, $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$, and computes a sequence of outputs, $\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)}$. (The superscripts are written with parentheses to make it clear that this isn't exponentiation.)

They're used for all kinds of language related things, like language modeling, speech, translation. They can be used as a kind of preprocessing step for a wide variety of tasks, because the $\mathbf{y}$'s can be seen as a representation of the words that incorporates contextual information. So if you have a bag-of-words model for (say) text classification, you can apply it to the $\mathbf{y}$'s, and now the model knows something about word order.

10.1 Definition

A so-called *simple* RNN is defined as follows:

$$\mathbf{h}^{(0)} = \mathbf{0} \tag{10.1}$$

$$\mathbf{h}^{(i)} = \text{sigmoid}(\mathbf{A}\mathbf{h}^{(i-1)} + \mathbf{B}\mathbf{x}^{(i)} + \mathbf{c}) \tag{10.2}$$

$$= 1/(1 + \exp -(\mathbf{A}\mathbf{h}^{(i-1)} + \mathbf{B}\mathbf{x}^{(i)} + \mathbf{c})) \tag{10.3}$$

$$\mathbf{y}^{(i)} = \text{softmax}(\mathbf{D}\mathbf{h}^{(i)} + \mathbf{e}) \tag{10.4}$$

that is,

$$\mathbf{s}^{(i)} = \mathbf{D}\mathbf{h}^{(i)} + \mathbf{e} \tag{10.5}$$

$$y_j^{(i)} = \frac{\exp s_j^{(i)}}{\sum_{j'} \exp s_{j'}^{(i)}}. \tag{10.6}$$

Typically, the $\mathbf{x}$'s and the $\mathbf{y}$'s are one-hot vectors.

If we want to use an RNN as a language model, we let

$$\mathbf{x}^{(1)} = \texttt{<s>} \tag{10.7}$$

$$\mathbf{x}^{(i+1)} = w_i \qquad i = 1, \ldots, n \tag{10.8}$$

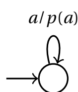
$$\mathbf{y}^{(i)} = w_i \qquad i = 1, \ldots, n \tag{10.9}$$

$$\mathbf{y}^{(n+1)} = \texttt{</s>} \tag{10.10}$$

In other words, at each time step, the network uses all the previous words to predict what the next word might be.

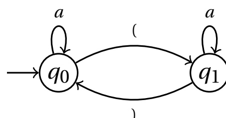
10.2 Motivation

For simplicity, let's start with a unigram language model.



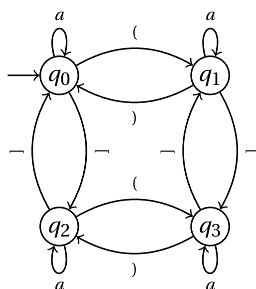
The transition labeled a is actually many transitions, one for each $a \in \Sigma$.

If you had to predict the next word in a sentence, you might use information from very far back in the string. To take a simple example, if there's a left parenthesis, it should be matched by a right parenthesis potentially very far away. So we could make more states (leaving off the probabilities for clarity):



Presumably, the probability of seeing a right parenthesis is low in q_0, but high in q_1.

And if we wanted to do something similar with left and right square brackets (sorry to focus on punctuation; there are plenty of long-distance phenomena in the words themselves, but none as clear-cut as in punctuation).



Now we have two problems. The first problem is that we are quickly going to end up with many states. If there are k flags, there are 2^k states, and even more probabilities that we have to estimate. Some of these are going to be really bad estimates, because some states will be visited very rarely.

The second problem is that coming up with ideas for structuring the state space is laborious. Can the model learn this automatically? Sure – we could just make an automaton with 1024 states and transitions between all of them (that's 1048576 transitions), and learn all the probabilities by maximum likelihood, using the Forward algorithm just as in the last few chapters. But, again, the number of transitions, and therefore the number of probabilities, would be prohibitively large.

RNNs make this easier. Each component of the $\mathbf{h}$ vectors is like one of the flags like "inside parentheses" or "inside square brackets." But instead of putting a different probability on every pair of states, an RNN defines a transition function with many fewer parameters.

10.3 Example

Figure 10.1 shows a run of RNN with 30 hidden units trained on the Wall Street Journal portion of the Penn Treebank, a common toy dataset for neural language models. When we run this model on a new sentence, we can visualize what each of its hidden units is doing at each time step. The units have been sorted by how rapidly they change.

The first unit seems to be unchanging; it serves the same purpose as the bias ($\mathbf{c}$). The second unit is blue on the start symbol, then becomes deeper and deeper red as the end of the sentence approaches. This unit seems to be measuring the position in the sentence and/or trying to predict the end of the sentence. The third unit is red for the first part of the sentence, usually the subject, and turns blue for the second part, usually the predicate. The rest of the units are unfortunately difficult to interpret. But we can see that the model is learning something about the large-scale structure of a sentence, without being explicitly told anything about sentence structure.

10.4 Training

We are given a set of training examples, each of which is a sequence of input vectors, $\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(n)}$, and a sequence of output vectors, $\mathbf{c}^{(1)},\ldots,\mathbf{c}^{(n)}$ (the $\mathbf{c}$ stands for "correct"). Usually the $\mathbf{c}$ vectors are one-hot vectors. For each such training example, the RNN outputs a sequence of vectors, $\mathbf{y}^{(1)},\ldots,\mathbf{y}^{(n)}$. Each $\mathbf{y}$ vector can be interpreted as a vector of probabilities.

The objective function (for a single example) that we want to increase is the log-likelihood:

$$L = \sum_{i=1}^{n} \mathbf{c}^{(i)} \cdot \log \mathbf{y}^{(i)} \tag{10.11}$$

where the log is elementwise and $\cdot$ is a vector dot product (inner product). We train using stochastic gradient ascent as usual.

However, a common problem is known as the *vanishing gradient* problem and its evil twin, the *exploding gradient* problem. What happens is that L is a a very long chain of functions (n times a constant). When we differentiate L, then by the chain rule, the partial derivatives are products of the partial derivatives of the functions in the chain. Suppose these partial derivatives are small numbers (less than 1). Then the product of many of them will be a vanishingly small number, and the gradient update will not have very much effect.

Or, suppose these partial derivatives are large numbers (greater than 1). Then the product of many of them will explode into a very large number, and the gradient update will be very damaging. This is definitely the more serious problem, and preventing it is important. The simplest fix is just to clip each partial derivative to the interval $[-5, 5]$.

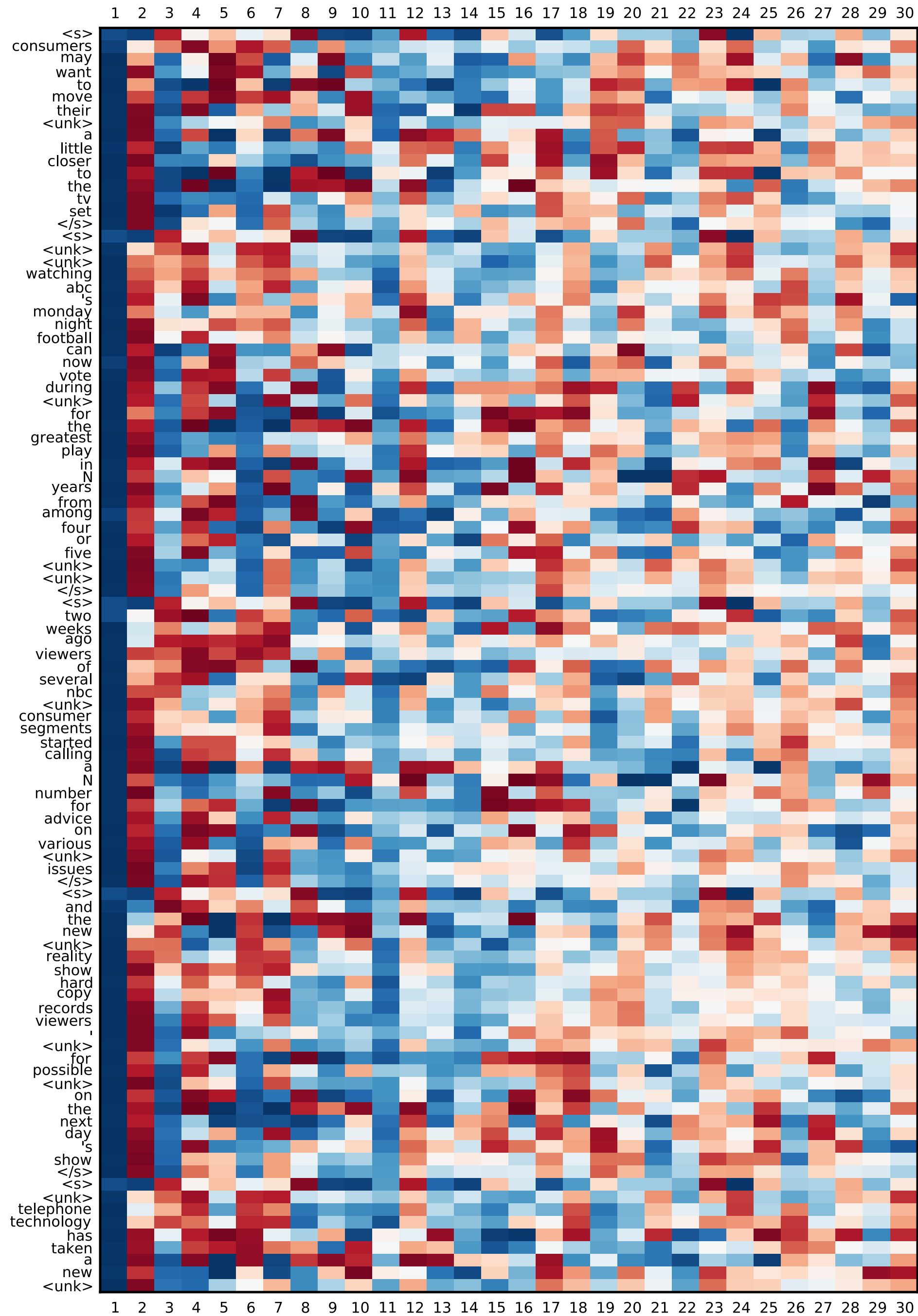


Figure 10.1: Visualization of a simple RNN language model on English text.