Chapter 4

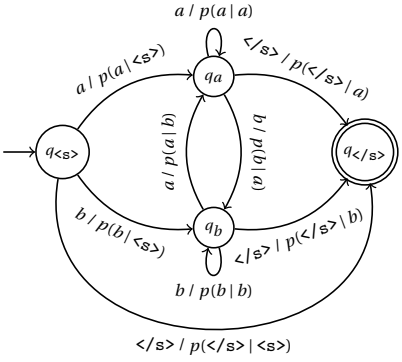
Recurrent Neural Networks

You can think of an RNN as a finite automaton 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 are often 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.

4.1 Motivation

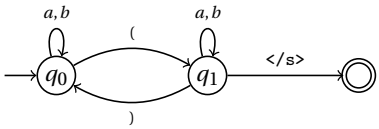
Let's start with a bigram language model. Recall that there are states $q_{\texttt{<s>}}$ and $q_{\texttt{</s>}}$, and a state q_a for every $a \in \Sigma$. We show what the transitions look like for two symbols $a, b \in \Sigma$, but you can imagine what it would look like in general.



(4.1)

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. Here's a finite automaton that keeps track of whether we're inside parentheses. Again, we just show transitions for two symbols $a, b \in \Sigma$, but you can imagine what it would look like in

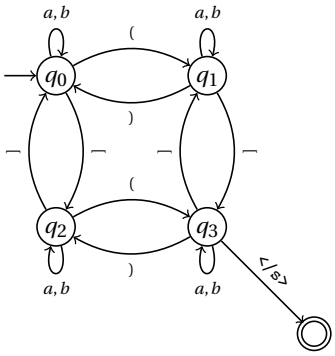
general.



(4.2)

Presumably, the probability of seeing a right parenthesis is low in q_0, but high in q_1. We can intersect the bigram automaton with this automaton, and we get a new automaton that models both bigrams and parentheses.

Now, we can make a similar automato to keep track of 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). Intersect all three automata (bigram, parentheses, square brackets) together and we get an even bigger and more powerful automaton. I can't draw the whole thing, but here's what the intersection of the parentheses and square brackets automata looks like:



(4.3)

We could dream up all kinds of features, design an automaton for each one, and intersect them all together. But then we would have two problems.

The first problem is that coming up with ideas for structuring the state space is laborious. Can the model learn this automatically? The answer is actually yes – we haven't yet covered the methods to do this, but we could just make an automaton with a bunch of states and transitions between all of them, and the model could automatically figure out what to do with them.

The second, more serious, problem is that we are quickly going to end up with many states. If there are 30 features, even if they are all just on/off features like our parentheses feature, there are $2^{30} \approx 1$ billion states, and something like $2^{60} \approx 10^{18}$ probabilities that we have to estimate. Most of these are going to be really bad estimates, because some states will be visited very rarely or not at all.

RNNs make this easier. They have a state space that is just as big as in the above thought experiment, defined by *units* that correspond to what we called "features." But they define the transition function in a different way that has many fewer parameters and can therefore be estimated much better.

4.2 Example

Figure 4.1 shows a run of a simple 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.

Other kinds of RNNs that perform better than this simple RNN have been shown to have units that perform various functions, including keeping track of parentheses (Karpathy, Johnson, and Fei-Fei, 2016).

4.3 Definition

These days most people do not implement RNNs themselves; they just one of many toolkits that do it for them. But for reference, here's a definition. A so-called *simple* RNN is defined by the equations:

$$\mathbf{h}^{(0)} = \mathbf{0} \tag{4.4}$$

$$\mathbf{h}^{(i)} = \text{sigmoid}(\mathbf{A}\mathbf{h}^{(i-1)} + \mathbf{B}\mathbf{x}^{(i)} + \mathbf{c}) \tag{4.5}$$

$$= 1/(1 + \exp -(\mathbf{A}\mathbf{h}^{(i-1)} + \mathbf{B}\mathbf{x}^{(i)} + \mathbf{c})) \tag{4.6}$$

$$\mathbf{y}^{(i)} = \text{softmax}(\mathbf{D}\mathbf{h}^{(i)} + \mathbf{e}) \tag{4.7}$$

that is,

$$\mathbf{s}^{(i)} = \mathbf{D}\mathbf{h}^{(i)} + \mathbf{e} \tag{4.8}$$

$$y_j^{(i)} = \frac{\exp s_j^{(i)}}{\sum_{j'} \exp s_j^{(i)}}. \tag{4.9}$$

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

$$\mathbf{x}^{(1)} = \texttt{<s>} \tag{4.10}$$

$$\mathbf{x}^{(i+1)} = w_i \qquad i = 1, \ldots, n \tag{4.11}$$

$$\mathbf{y}^{(i)} = w_i \qquad i = 1, \ldots, n \tag{4.12}$$

$$\mathbf{y}^{(n+1)} = \texttt{</s>} \tag{4.13}$$

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

4.4 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:

$$L = \sum_{i=1}^{n} \mathbf{c}^{(i)} \cdot \log \mathbf{y}^{(i)} \tag{4.14}$$

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 check if the norm of the gradient is bigger than 5, and if so, scale it so thats norm is just 5.

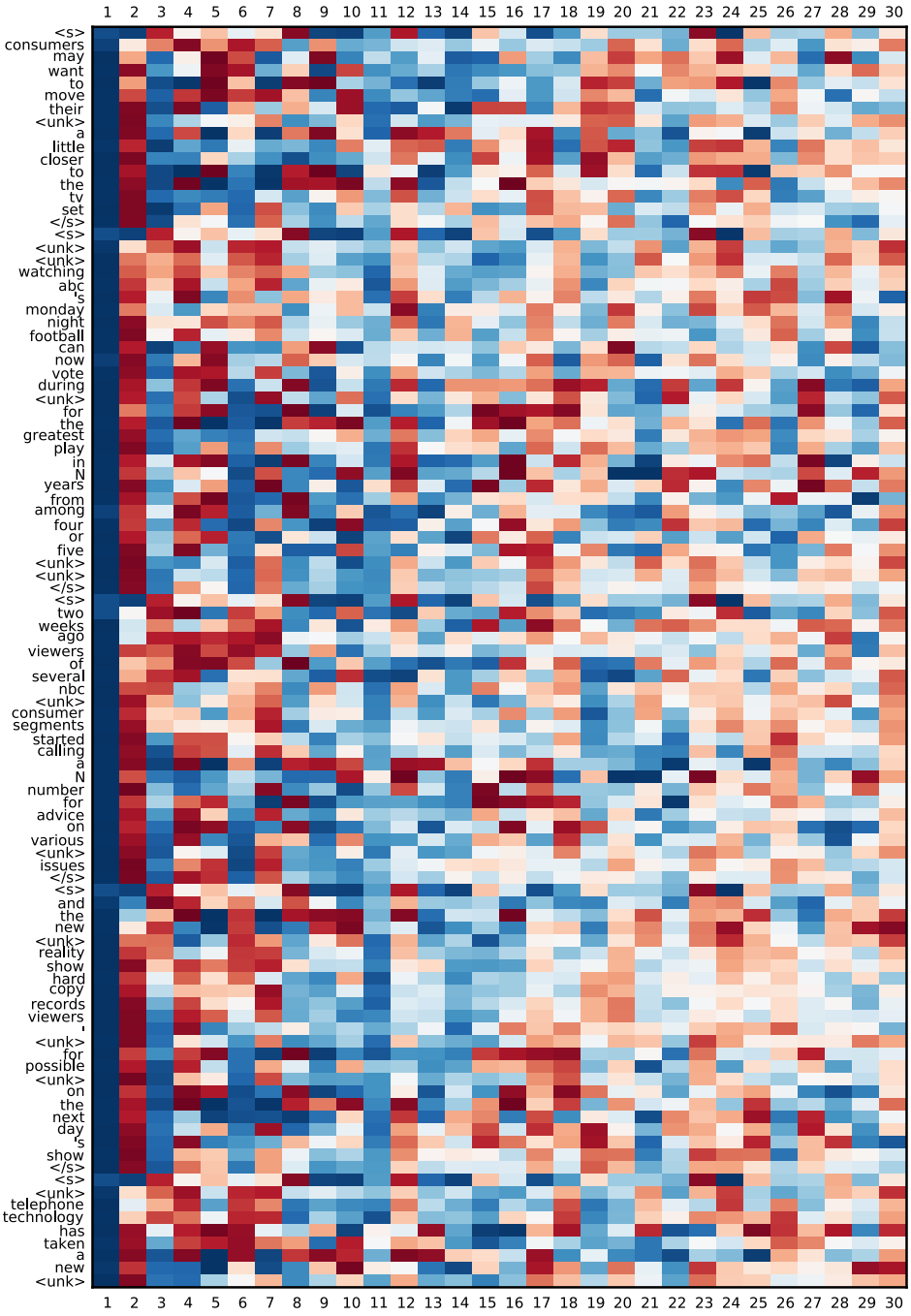


Figure 4.1: Visualization of a simple RNN language model on English text.

Bibliography

Karpathy, Andrej, Justin Johnson, and Li Fei-Fei (2016). “Visualizing and Understanding Recurrent Neural Networks”. In: *Proc. ICLR*.