Chapter 2

Language Models

For many years, I have begun this course with the topic of *language models*, which were traditionally one component of various systems that generate text, like automatic speech recognition or machine translation. As they were not the most interesting part of these systems, I used to begin this chapter with a somewhat apologetic motivation. But now that language models (particularly GPT) have taken center stage, little motivation is needed.

The job of a language model is to estimate the probability of a sentence $w = w_1 \cdots w_T$, where each w_t is a character, or a word, or something in between. (How we cut up a sentence into segments depends on the application, but the techniques are the same in any case.) We will call each w_t a *word*, with the understanding that it might actually be bigger or smaller than a word. Let Σ, known as the vocabulary, be the (finite) set of all words that the model knows about.

2.1 n-gram Models

2.1.1 Model

The simplest kind of language model is an n-gram language model, in which each word depends on the $(n-1)$ previous words. We assume, as we will frequently do, that the string ends with a special symbol $\texttt{EOS} \in \Sigma$, which stands for "end of sentence" (and is written by some authors as `</s>`). So $w = w_1 \cdots w_T$ where $w_T = \texttt{EOS}$. In a 1-gram or *unigram* language model, each word is generated independently:

$$P(w_1 \cdots w_T) = p(w_1) \cdots p(w_T) \tag{2.1}$$

where the $p(a)$, for all $a \in \Sigma$, are the parameters of the model and $\sum_{a \in \Sigma} p(a) = 1$. The `EOS` is needed in order to make the probabilities of all sentences of all lengths sum to one. Imagine rolling a die with one word written on each face to generate random sentences; in order to know when to stop rolling, you need (at least) one face of the die to say `EOS` which means "stop rolling."

In a 2-gram or *bigram* language model (also known as a *Markov chain*), each word's probability depends on the previous word:

$$P(w_1 \cdots w_T) = p(w_1 \mid \texttt{BOS}) \left(\prod_{t=2}^{T} p(w_t \mid w_{t-1}) \right) \tag{2.2}$$

where the $p(b \mid a)$ are the parameters of the model, and, for all a, we have $\sum_{b \in \Sigma} p(b \mid a) = 1$. Since the first word doesn't have a real previous word to condition on, we condition it on a special symbol $\texttt{BOS} \in \Sigma$ (which is written by some authors as <s>).

A general n-gram language model is:

$$P(w_1 \cdots w_T) = \prod_{t=1}^{T} p(w_t \mid w_{t-n+1} \cdots w_{t-1}), \tag{2.3}$$

where we pretend that $w_t = \texttt{BOS}$ for $t \leq 0$. For example, if $n = 3$, then

$$\begin{aligned} P(\text{the cat sat on the mat}) = p(&\text{the} \mid \texttt{BOS BOS}) \cdot \\ p(&\text{cat} \mid \texttt{BOS the}) \cdot \\ p(&\text{sat} \mid \text{the cat}) \cdot \\ p(&\text{on} \mid \text{cat sat}) \cdot \\ p(&\text{the} \mid \text{sat on}) \cdot \\ p(&\text{mat} \mid \text{on the}) \cdot \\ p(&\texttt{EOS} \mid \text{the mat}). \end{aligned}$$

2.1.2 Training

Training an n-gram model is easy. First consider a unigram model ($n = 1$). For each word $a \in \Sigma$, let $c(a)$ be the number of times that a occurs. Then the probability of a is:

$$p(a) = \frac{c(a)}{\sum_{a' \in \Sigma} c(a')}$$

For a general n-gram model with $n > 1$,

$$p(a \mid u) = \frac{c(ua)}{\sum_{a' \in \Sigma} c(ua')}$$

where u ranges over $(n-1)$-grams, that is, $u \in \Sigma^{n-1}$.

2.1.3 Smoothing

A never-ending challenge in all machine learning settings is the *bias-variance* tradeoff, or the tradeoff between *underfitting* and *overfitting*. In language modeling, underfitting usually means that the model probability of a word doesn't sufficiently take into account the context of the word. For example, a unigram language model would think that "the the the" is a very good sentence. In the world of n-gram language models, the antidote to underfitting is to increase n.

Overfitting usually means that the model overestimates the probability of words or word sequences seen in data and underestimates the probability of words or word sequences not seen in data. For example, suppose that

$$\begin{aligned}
c(\text{pulchritudinous penguin}) &= 1 \\
c(\text{pulchritudinous puppy}) &= 0 \\
c(\text{pulchritudinous pachycephalosaur}) &= 0 \\
c(\text{pulchritudinous}) &= 1
\end{aligned}$$

The maximum-likelihood estimate would have

$$\begin{aligned}
P(\text{penguin} \mid \text{pulchritudinous}) &= 1 \\
P(\text{puppy} \mid \text{pulchritudinous}) &= 0 \\
P(\text{pachycephalosaur} \mid \text{pulchritudinous}) &= 0.
\end{aligned}$$

Is that a good estimate? No, because "pulchritudinous" is so rare that most words have never been seen after it. We want to take some probability mass away from $P(\text{penguin} \mid \text{pulchritudinous})$ and give it to $P(\text{puppy} \mid \text{pulchritudinous})$ and $P(\text{pachycephalosaur} \mid \text{pulchritudinous})$. Moreover, we don't have to distribute probability mass evenly. Since $P(\text{puppy}) > P(\text{pachycephalosaur})$, it's reasonable to estimate $P(\text{puppy} \mid \text{pulchritudinous}) > P(\text{pachycephalosaur} \mid \text{pulchritudinous})$.

The process of taking probability mass away from seen n-grams and giving it to unseen n-grams is called *smoothing*. The very easiest smoothing method – which is quite bad for language modeling – is called *add-one* or *Laplace* smoothing. Recall that the unsmoothed probability estimate of an n-gram is

$$p(a \mid u) = \frac{c(ua)}{\sum\limits_{a \in \Sigma} c(ua')}. \tag{2.4}$$

Then we simply add 1 to the count of every n-gram:

$$p(a \mid u) = \frac{c(ua) + 1}{\sum\limits_{a' \in \Sigma} (c(ua') + 1)} \tag{2.5}$$

so that the smoothed probability estimate of our example is

$$\begin{aligned}
P(\text{penguin} \mid \text{pulchritudinous}) &= \tfrac{1}{2} \\
P(\text{puppy} \mid \text{pulchritudinous}) &= \tfrac{1}{4} \\
P(\text{pachycephalosaur} \mid \text{pulchritudinous}) &= \tfrac{1}{4}.
\end{aligned}$$

Smoothing is a big subject; to learn more about it, please see the report by Chen and Goodman (1998).

2.2 Unknown Words

Natural languages probably don't have a finite vocabulary, and even if they do, the distribution of word frequencies has such a long tail that, if we use a language model to compute the probability of sentences in test data, words that are *unknown* to the model or *out-of-vocabulary* (OOV) will be rather common. Unknown words are problematic for all language models, and we have a few techniques for handling them.

2.2.1 Subword segmentation

We can alleviate the problem by breaking words into smaller pieces. For example, a word like *antidisestablishmentarianism* might be unknown. But if our training data contains the words

antitrust   disassemble   establishment   totalitarianism

and we break up these words into

anti-   trust   dis-   assemble   establish   -ment   totalit-   -arianism

then *anti- dis- establish -ment -arian -ism* are all known.

Currently, the most common method for subword segmentation is *byte pair encoding* (Sennrich, Haddow, and Birch, 2016); the unigram model of Kudo and Richardson (2018) is also widely used. In the limit, one could use individual characters, and even Chinese characters could be broken into sub-characters.

However, even if we use individual characters, there may still be unknown symbols. (Think, for example, about how many emojis there are.) So, what do we do with the remaining unknown symbols?

2.2.2 `UNK` replacement

The answer is simple (perhaps disappointingly so): when we encounter an unknown symbol in test data, we replace it with a special symbol $\texttt{UNK} \in \Sigma$. This raises a new question: how does the model assign a probability to `UNK`?

An unsmoothed n-gram model would assign a probability of zero, which is bad not only because it's incorrect, but because it multiplies the probability of the whole sentence by zero. That is, the sentences *The* `UNK` *saw the dog* and *The* `UNK` *saw the pulchritudinous pachycephalosaur* would have equal probability (zero). However, a smoothed n-gram model does not have this problem, because smoothing even increases the probability estimates of zero-probability events.

When smoothing is not convenient, a simpler method is to pretend that some symbols seen in the training data are unknown. For example, we might limit the vocabulary to 10,000 types, and all others are changed to `UNK`. Or, we might limit the vocabulary just to types seen two or more times, and all others are changed to `UNK`. When we train the language model, we treat `UNK` like any other symbol, so it gets a nonzero probability.

2.3 Evaluation

Whenever we build any kind of model, we always have to think about how to evaluate it. How do we evaluate language models?

2.3.1 Generating random sentences

One popular way of demonstrating a language model is using it to generate random sentences. While this is entertaining and can give a qualitative sense of what kinds of information a language model does and doesn't capture, but it is *not* a rigorous way to evaluate language models. Why? Imagine a language model that just memorizes the sentences in the training data. This model would randomly generate perfectly-formed sentences. But if you gave it a sentence w not seen in the training data, it would give w a probability of zero.

2.3.2 Extrinsic evaluation

The best way to evaluate language models is extrinsically. That is, use the language model (possibly in concert with another model) to perform some task, and evaluate on that task.

2.3.3 Intrinsic evaluation: perplexity

For intrinsic evaluation, the standard way to evaluate a language model is how well it fits some *held-out* data (that is, data that is different from the training data). There are various ways to measure this:

$$L = P(w_1 \cdots w_T) \qquad \text{likelihood} \tag{2.6}$$

$$H = -\frac{1}{N} \log_2 L \qquad \text{per-word cross-entropy} \tag{2.7}$$

$$PP = 2^H \qquad \text{perplexity} \tag{2.8}$$

We've seen likelihood already in the context of maximum-likelihood training. If a model assigns high likelihood to held-out data, that means it's generalized well. However, likelihood is difficult to interpret because it depends on w and, in particular, T.

Cross-entropy is based on the idea that any model can be used for data compression: more predictable symbols can be stored with fewer bits and less predictable symbols must be stored with more bits. If we built an ideal data compression scheme based on our model and compressed w, the total cross-entropy, $-\log_2 L$, is the number of bits we would compress w into (Shannon, 1948).

The per-word cross-entropy is the average number of bits required per word of w, which has the advantage that you can interpret it without knowing T. The best (lowest) possible per-word cross-entropy is 0, which can be achieved only if the model always knows what the next word will be. The worst (highest) possible per-word cross-entropy is $\log_2 |\Sigma|$, which means that the model doesn't know anything about what the next word will be, so the best it can do is guess randomly.

Perplexity is closely related to per-word cross-entropy; it just undoes the log. One advantage is that you can interpret it without knowing the base of the log. (A dubious advantage is that it makes small differences look large.) The best (lowest) possible perplexity is 1, and the worst (highest) possible perplexity is $|\Sigma|$.

Held-out data is always going to have unknown words, which require some special care. Above, we handled unknown words by mapping them to a special symbol `UNK`, but if we compare two language models, they must map exactly the same subset of word types to `UNK`. (If not, can you think of a way to cheat and get a perplexity of 1?)

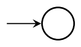
2.4 Finite Automata

If you've taken *Theory of Computing*, you should be quite familiar with finite automata; if not, you may be familiar with regular expressions, which are equivalent to finite automata. Many models in NLP can be thought of as finite automata, or variants of finite automata, including n-gram language models. Although this may feel like overkill at first, we'll soon see that formalizing models as finite automata makes it much easier to combine models in various ways.

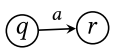
2.4.1 Definition

A *finite automaton (FA)* is an imaginary machine that reads in a string and outputs an answer, either "accept" or "reject." (For example, a FA could accept only words in an English dictionary and reject all other strings.) At any given time, the machine is in one *state* out of a finite set of possible states. It has rules, called *transitions*, that tell it how to move from one state to another.

A FA is typically represented by a directed graph. We draw nodes to represent the various states that the machine can be in. The node can be drawn with or without the state's name inside. The machine starts in the *initial state* (or *start state*), which we draw as:



The edges of the graph represent transitions, for example:

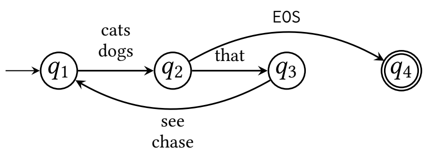


which means that if the machine is in state q and the next input symbol is a, then it can read in a and move to state r.

We are going to go on assuming that every string ends with `EOS`, and all transitions labeled `EOS` go to the *final state* (or *accept state*), which we draw as:

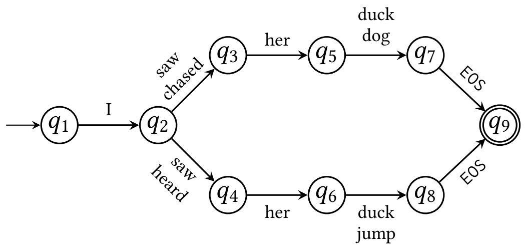
If the machine can reach the end of the string while in a final state, then it accepts the string. Otherwise, it rejects.

Here's an example of a finite automaton that generates an infinite number of grammatical noun phrases. Note that our alphabet is the set of English words, not letters.



We say that a FA is *deterministic* (or a DFA) if every state has the property that, for each label, there is exactly one outgoing transition with that label (like the above example). When a DFA reads a string, it always knows which state to go to next.

But if a state has two outgoing transitions with the same label, it is *nondeterministic* (Rabin and Scott, 1959), or is an NFA for short. For example:



At q_2, if the next word is "saw," the NFA can go to either q_3 *or* q_4. Intuitively, q_3 is the state that's looking for a noun phrase object, while q_4 is the state that's looking for a phrase called a *small clause*. If the sentence continues "her duck EOS," it ends up in state q_9 and accepts. If the sentence continues "her dog EOS," then the branch that was in q_3 goes to q_5, q_7, q_9 and accepts; the branch that was in q_4 goes to q_6 and stops. But the NFA accepts as long as at least one of its branches accepts.

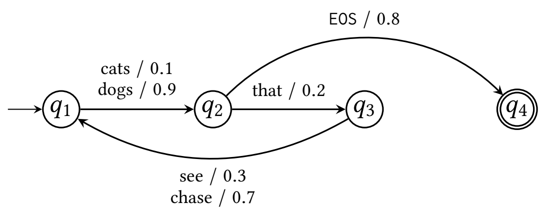
2.4.2 Probabilistic DFAs

Finite automata can accept or reject sentences, but we are interested in using them to build language models, which can assign probabilities to sentences, and in particular to each word in a sentence, given the previous words.

A *probabilistic DFA* (Mohri, 1997) attaches a probability to each transition, like this:

$$\begin{tikzcd} q \arrow[r, "a \,/\, p"] & r \end{tikzcd} .$$

For each state (except the final state), the probabilities of all of the outgoing transitions sum to one. The probability of a path through a probabilistic FA is the product of the probabilities of the transitions along the path. Then the total probability of all strings is exactly one, that is, the automaton defines a probability distribution over all strings. For example, we can add probabilities to our example DFA:



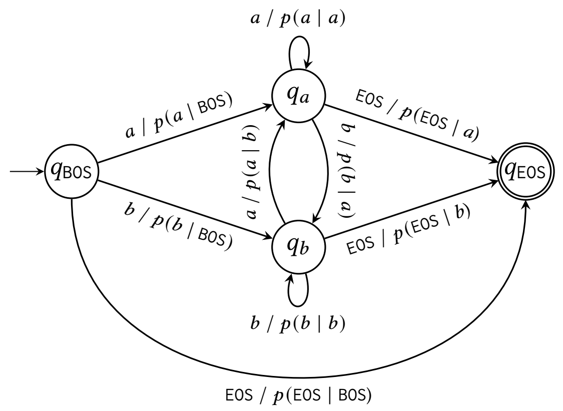
The string “cats that chase dogs that see cats EOS” has probability $0.1 \cdot 0.2 \cdot 0.7 \cdot 0.9 \cdot 0.2 \cdot 0.3 \cdot 0.1 = 7.56 \cdot 10^{-5}$.

Where do the probabilities come from? Suppose we are given a collection $\mathcal{D}$ of strings. We also have a DFA M, and we want to learn probabilities for M. So, for each string $w \in \mathcal{D}$, run M on w and count, for each state q and symbol $a \in \Sigma$, the number of times a transition $\textcircled{$q$} \xrightarrow{a} \textcircled{r}$ (for any r) is used. Call this count $c(q, a)$. Then set

$$p\Big(\textcircled{q} \xrightarrow{a} \textcircled{r}\Big) = \frac{c(q,a)}{\sum\limits_{a'} c(q,a')}.$$

This maximizes the likelihood of $\mathcal{D}$.

An n-gram language model is a probabilistic DFA with a very simple structure. A bigram model with an alphabet $\Sigma = \{a, b, \texttt{EOS}\}$ looks like this:

 (2.9)
a / $p(a \mid a)$
a / $p(a \mid \texttt{BOS})$
$\texttt{EOS}$ / $p(\texttt{EOS} \mid a)$
a / $p(a \mid b)$
b / $p(b \mid a)$
b / $p(b \mid \texttt{BOS})$
$\texttt{EOS}$ / $p(\texttt{EOS} \mid b)$
b / $p(b \mid b)$
$\texttt{EOS}$ / $p(\texttt{EOS} \mid \texttt{BOS})$
$q_{\texttt{BOS}}$
q_a
q_b
$q_{\texttt{EOS}}$

In general, we need a state for every observed context, that is, one for BOS, which we call $q_{\texttt{BOS}}$, and one for each word type $a \in \Sigma$, which we call q_a. And $q_{\texttt{EOS}}$ is an accept state. For all a, b, there is a transition

$$\textcircled{q_a} \xrightarrow{b \,/\, p(b \mid a)} \textcircled{q_b}$$

and for every state q_a, there is a transition

$$\textcircled{q_a} \xrightarrow{\texttt{EOS} \,/\, p(\texttt{EOS} \mid a)} \textcircled{$q_{\texttt{EOS}}$}\,.$$

Generalizing to n-grams, we need a state for every $(n-1)$-gram. It would be messy to actually draw the diagram, but we can describe how to construct it:

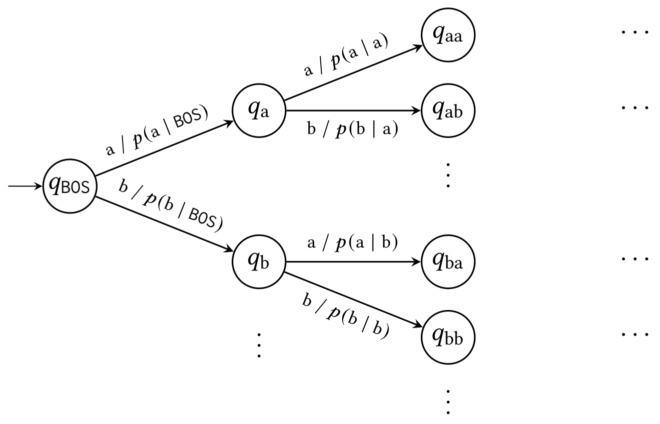
- For all $u \in \Sigma^{n-1}$, there is a state q_u.
- The start state is $q_{\texttt{BOS}^{n-1}}$.
- The accept state is $q_{\texttt{EOS}}$.
- For all $a \in \Sigma, u \in \Sigma^{n-2}, b \in \Sigma$, there's a transition

$$\begin{tikzcd} q_{au} \arrow[r, "b \mathrel{/} p(b \mid au)"] & q_{ub} \end{tikzcd} .$$

- For all $u \in \Sigma^{n-1}$, there's a transition

$$\begin{tikzcd} q_{u} \arrow[r, "\texttt{EOS} \mathrel{/} p(\texttt{EOS} \mid u)"] & q_{\texttt{EOS}} \end{tikzcd} .$$

One can imagine designing other kinds of language models as well. For example, a *trie* is often used for storing lists like dictionaries. A probabilistic trie would condition each symbol on all previous symbols:

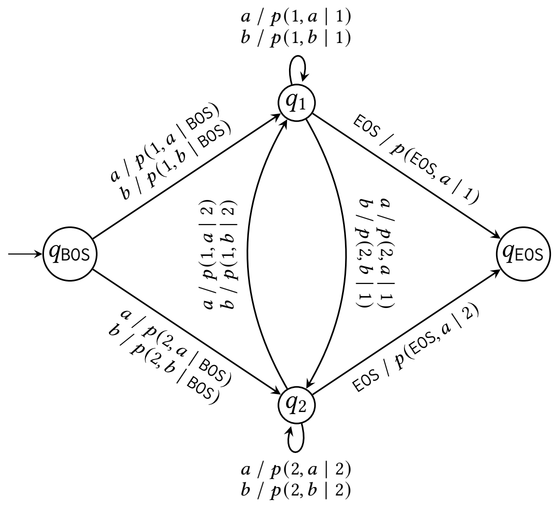


2.4.3 Probabilistic NFAs (optional)

This section is not necessary for understanding the rest of this chapter, but is historically of enough importance that I didn't want to delete it.

Just as a probabilistic DFA attaches probabilities to the transitions of a DFA, a *probabilistic NFA* attaches a probability to every transition of an NFA. Again, each state (except the final state) has the property that the probabilities of all of the outgoing transitions sum to one. Since there may be more than one path that accepts a string w, the probability of w is the sum of the probabilities of all accepting paths of w.

Here's an example of a probabilistic NFA, known as a *hidden Markov model* (HMM).



where each transition probability is defined in terms of two smaller steps:

$$p(r, a \mid q) = t(r \mid q)\, o(a \mid r). \tag{2.10}$$

Notice how a single string can have multiple accepting paths. For example, if the input symbols are English, then we could set the transition probabilities so that the NFA goes to q_1 when reading a noun and q_2 when reading a verb. In the sentence "I saw her duck," the word "duck" could be either a noun or a verb, so it would be appropriate for the NFA to have two paths that accept this sentence. Assuming that possessive "her" is a kind of noun, the two paths would be: $q_{\text{BOS}}, q_1, q_2, q_1, q_1, q_{\text{EOS}}$ and $q_{\text{BOS}}, q_1, q_2, q_1, q_2, q_{\text{EOS}}$.

How do we train a probabilistic NFA? Assume we are given a collection $\mathcal{D}$ of strings. We also have an NFA M, and we want to learn probabilities for M. The procedure we gave above for DFAs won't work. This is because, for a given string, there might be more than one path that accepts it, and we don't know which path's transitions to count. We want to maximize the log-likelihood of the training data,

$$L = \log \prod_{w \in \mathcal{D}} P(w).$$

Recall that $P(w)$ is the sum of the probabilities of all accepting paths for w. In the worst case, there could be exponentially many such paths. The good news is that $P(w)$ can be calculated efficiently. Number the states of M as $q_1, \ldots, q_d$. Let $\mathbf{s} \in \mathbb{R}^d$, $\mathbf{f} \in \mathbb{R}^d$, and $\mu\colon \Sigma \to \mathbb{R}^{d\times d}$ be such that

$$\mathbf{s}_i = \begin{cases} 1 & \text{if } q_i \text{ is the start state} \\ 0 & \text{otherwise} \end{cases}$$

and $[\mu(a)]_{ij}$ is the probability of transition $\textcircled{$q_i$} \xrightarrow{a} \textcircled{q_j}$, and if q_j is the accept state, then $\mathbf{f}_i$ is the probability of transition $\textcircled{$q_i$} \xrightarrow{\text{EOS}} \textcircled{q_j}$. Then

$$P(w) = \mathbf{f}^\top \mu(w_N) \cdots \mu(w_1)\, \mathbf{s}.$$

The bad news is that we can't just maximize L by setting its derivatives to zero and solving for the transition probabilities. Instead, we must use some kind of iterative approximation. The traditional way to do this is called *expectation-maximization*. The other way is to use gradient-based optimization, which we'll cover later when we talk about neural networks (Section 2.5.4).

2.5 Recurrent Neural Networks

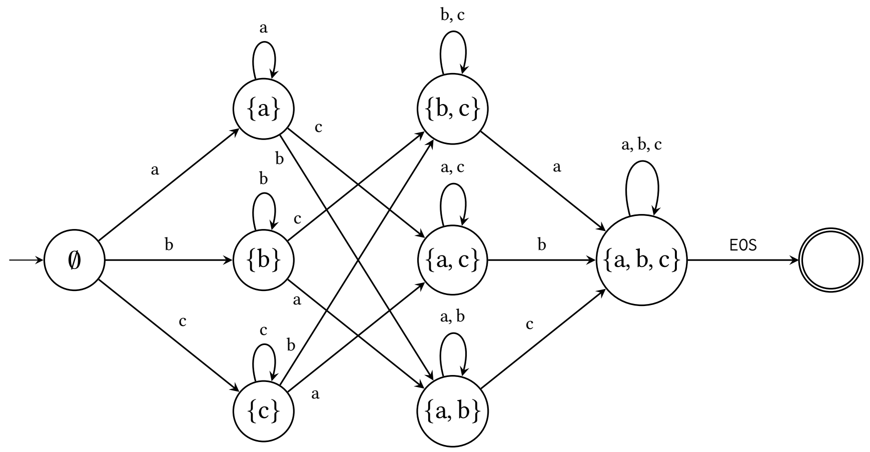
For a long time, researchers tried to find language models that were better than n-gram models and failed, but in recent years, neural networks have become powerful enough to retire n-grams at last. One way of defining a language model as a neural network is as a *recurrent neural network* (RNN).

2.5.1 From finite automata...

I'd like to motivate RNNs from a historical point of view, because RNNs and finite automata have a common ancestor. This section is not absolutely necessarily for understanding the rest of this chapter, but I think showing the connection between finite automata and RNNs helps to explain why RNNs are so much more successful.

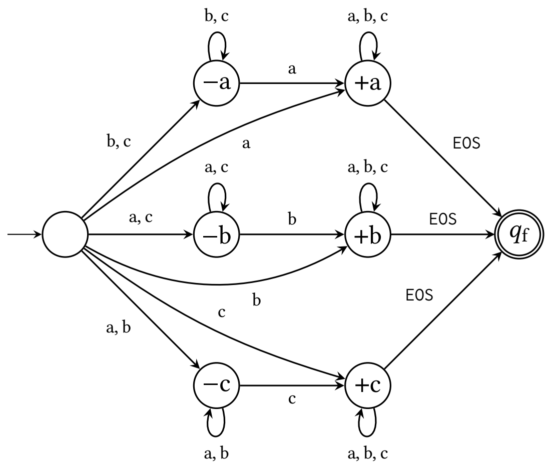
Consider the problem of writing an automaton that accepts only *pangrams*, sentences that contain all 26 letters of the English alphabet. The most famous pangram is "The quick brown fox jumped over the lazy dogs." This is an example of a problem where many constraints need to be satisfied at the same time, and is not just relevant for word games; in language modeling, there are many (soft) constraints that need to be satisfied at once, many more than 26.

A DFA for strings containing all symbols in $\{a, b, c\}$ is



We've labeled each state with the set of letters that has been seen so far. This means that in general, this DFA has $O(2^{|\Sigma|})$ states!

Recall that NFAs are equivalent to DFAs, but NFAs could be much smaller than DFAs. Is there a more efficient NFA for pangrams? Here's a wrong attempt:



The NFA can be in state −a if it hasn't read an a yet, and it can be in state +a if it has read an a. Similarly for b and c. This machine has $O(|\Sigma|)$ states, which is much better than before! The problem is that it doesn't work – because it can reach state q_f if it has read an a *or* a b *or* a c. What we really want is for it to reach state q_f if it has read an a *and* a b *and* a c. And we can get that, if we switch to a more powerful kind of automaton.

Let's define some notation. Consider an automaton M reading the string $w_1 \cdots w_T$. Number the states of M as $q_1, \ldots, q_d$. We want to define a sequence of vectors $\mathbf{h}^{(t)}$ ($t = 0, \ldots, T$) such that $\mathbf{h}^{(t)}_j = 1$ if M can be in state q_i at time t, that is, just after reading w_t, and $\mathbf{h}^{(t)}_i = 0$ otherwise. (The superscripts are written with parentheses to make it clear that this isn't exponentiation.)

For NFAs, the $\mathbf{h}^{(t)}$ are defined as follows. Initially $\mathbf{h}^{(0)}_i = 1$ iff q_i is the start state, and, for $t > 0$, $\mathbf{h}^{(t)}_j = 1$ iff there is *at least one* state q_i such that $\mathbf{h}^{(t-1)}_i = 1$ and there is a transition $q_i \xrightarrow{w_t} q_j$.

But in the earlier definition of finite automata by Kleene (1951), $\mathbf{h}^{(t)}$ was defined by an *arbitrary* function of $\mathbf{h}^{(t-1)}$ and w_t. So, in our pangram example, we can now decree that $\mathbf{h}^{(t)}_{q_\mathrm{f}} = 1$ iff $\mathbf{h}^{(t-1)}_{+\mathrm{a}} = \mathbf{h}^{(t-1)}_{+\mathrm{b}} = \mathbf{h}^{(t-1)}_{+\mathrm{c}} = 1$ and $w_t = \texttt{EOS}$. The automaton shown above, with only $O(|\Sigma|)$ states, now works.

Technically, Kleene's definition doesn't make automata any more powerful, but it does mean that they can do a lot more with the same number of states. Any automaton under Kleene's definition can be converted to an equivalent DFA, but it might have exponentially more states.

2.5.2 …to recurrent neural networks

Now we dig further back into history to *neural net(work)s*, which were introduced by McCulloch and Pitts (1943). They can be thought of as yet another choice of the recurrence for $\mathbf{h}$; Kleene gave his definition of finite automata as a generalization of McCulloch–Pitts neural networks.

Just as we previously numbered all the states, so now we number the symbols in Σ, starting from 1. The ordering is completely arbitrary. For example, if $\Sigma = \{\texttt{BOS}, \mathrm{a}, \mathrm{b}, \texttt{EOS}\}$, we could number them: $\texttt{BOS} = 1, \mathrm{a} = 2, \mathrm{b} = 3, \texttt{EOS} = 4$. If the input string is $w = w_1 \cdots w_T$, define a sequence of vectors $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$. Each vector $\mathbf{x}^{(t)}$ (for $t > 0$) encodes w_t as a *one-hot* vector, which means that $\mathbf{x}^{(t)}$ is a vector with all 0's except for a 1 corresponding to w_t. Also, $\mathbf{x}^{(0)}$ encodes BOS. For example, if $w = \mathrm{aba}\,\texttt{EOS}$, then the input vectors would be

$$\mathbf{x}^{(0)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \qquad \mathbf{x}^{(1)} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \qquad \mathbf{x}^{(2)} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \qquad \mathbf{x}^{(3)} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \qquad \mathbf{x}^{(4)} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}.$$

A McCulloch–Pitts neural network is defined as follows:

$$\mathbf{h}^{(-1)} = \mathbf{0} \tag{2.11}$$
$$\mathbf{h}^{(t)} = \mathbb{I}\left[\mathbf{A}\mathbf{h}^{(t-1)} + \mathbf{B}\mathbf{x}^{(t)} + \mathbf{c} \geq 0\right] \qquad t = 0, \ldots, T \tag{2.12}$$

where

$$\mathbf{A} \in \mathbb{Z}^{d \times d} \tag{2.13}$$
$$\mathbf{B} \in \mathbb{Z}^{d \times |\Sigma|} \tag{2.14}$$
$$\mathbf{c} \in \mathbb{Z}^{d} \tag{2.15}$$

In our pangram example, we wanted to let $\mathbf{h}^{(t)}_{q_\mathrm{f}} = 1$ iff $\mathbf{h}^{(t-1)}_{+\mathrm{a}} = \mathbf{h}^{(t-1)}_{+\mathrm{b}} = \mathbf{h}^{(t-1)}_{+\mathrm{c}} = 1$ and $w_t = \texttt{EOS}$. Can we do this in a McCulloch–Pitts neural network? Yes, if we set:

$$\begin{aligned} \mathbf{A}_{q_\mathrm{f},+\mathrm{a}} &= 1 \\ \mathbf{A}_{q_\mathrm{f},+\mathrm{b}} &= 1 \\ \mathbf{A}_{q_\mathrm{f},+\mathrm{c}} &= 1 \\ \mathbf{B}_{q_\mathrm{f},\texttt{EOS}} &= 1 \\ \mathbf{c} &= -4 \end{aligned}$$

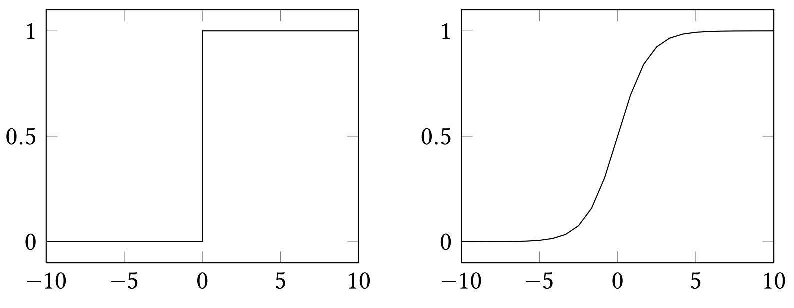
The components of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{c}$ are the parameters of the model. As we will see, we will learn parameter values by gradient ascent, which maximizes a function by climbing uphill. But the step function, $\mathbb{I}[\cdot \geq 0]$, is not so good for climbing. In a so-called *simple* or *Elman recurrent neural network* (Elman, 1990), the step function, $\mathbb{I}[\cdot \geq 0]$, is replaced with the *sigmoid* function,

$$\mathrm{sigmoid}(z) = \frac{1}{1 + \exp(-z)}$$

which is a smooth version of the step function (see Figure 2.1).

See Figure 2.2 for a picture of an RNN. From the one-hot vectors $\mathbf{x}^{(t)}$, the RNN computes a sequence of vectors:

$$\mathbf{h}^{(-1)} = \mathbf{0} \tag{2.16}$$
$$\mathbf{h}^{(t)} = \mathrm{sigmoid}(\mathbf{A}\mathbf{h}^{(t-1)} + \mathbf{B}\mathbf{x}^{(t)} + \mathbf{c}) \qquad t = 0, \ldots, T \tag{2.17}$$



(a) step (b) sigmoid

Figure 2.1: The step function (a) is 0 for negative values and 1 for positive values, while the sigmoid function (b) is 0 for very negative values, 1 for very positive values, and smooth in between.

$$\begin{tikzcd}
\boxed{\mathbf{y}^{(1)}} & \boxed{\mathbf{y}^{(2)}} & \boxed{\mathbf{y}^{(3)}} & \boxed{\mathbf{y}^{(4)}} \\
\boxed{\mathbf{h}^{(0)}} \arrow[u] \arrow[r] & \boxed{\mathbf{h}^{(1)}} \arrow[u] \arrow[r] & \boxed{\mathbf{h}^{(2)}} \arrow[u] \arrow[r] & \boxed{\mathbf{h}^{(3)}} \arrow[u] \\
\boxed{\mathbf{x}^{(0)}} \arrow[u] & \boxed{\mathbf{x}^{(1)}} \arrow[u] & \boxed{\mathbf{x}^{(2)}} \arrow[u] & \boxed{\mathbf{x}^{(3)}} \arrow[u]
\end{tikzcd}$$

Figure 2.2: A simple RNN, shown for a string of length $T = 4$ (including EOS). Each rectangle is a vector that is either input to or computed by the network.

where

$$\mathbf{A} \in \mathbb{R}^{d\times d} \tag{2.18}$$
$$\mathbf{B} \in \mathbb{R}^{d\times|\Sigma|} \tag{2.19}$$
$$\mathbf{c} \in \mathbb{R}^{d} \tag{2.20}$$

are parameters of the model, which will be learned during the training process, as described in Section 2.5.4.

At each time step, the RNN makes a prediction about the next symbol:

$$\mathbf{y}^{(t)} = \text{softmax}(\mathbf{D}\mathbf{h}^{(t-1)} + \mathbf{e}) \qquad t = 1, \ldots, T \tag{2.21}$$

where

$$\mathbf{D} \in \mathbb{R}^{|\Sigma|\times d} \tag{2.22}$$
$$\mathbf{e} \in \mathbb{R}^{|\Sigma|} \tag{2.23}$$

are more parameters of the model. See Section 1.6 for a definition of the softmax function. The vector $\mathbf{y}^{(t)}$ is a probability distribution over Σ, that is, we estimate

$$P(w_t \mid w_1 \cdots w_{t-1}) \approx \mathbf{y}^{(t)}_{w_t} = \mathbf{x}^{(t)} \cdot \mathbf{y}^{(t)}.$$

Since each $\mathbf{x}^{(t)}$ is a one-hot vector, dotting it with another vector selects a single component from that other vector, which in this case is the probability of w_t.

For example, if after reading $w_1 = \text{a}$, we have

$$\mathbf{y}^{(2)} = \begin{bmatrix} 0.6 \\ 0.2 \\ 0.4 \end{bmatrix},$$

that means

$$\begin{aligned} P(w_2 = \text{a} \mid w_1 = \text{a}) &= 0.6 \\ P(w_2 = \text{b} \mid w_1 = \text{a}) &= 0.2 \\ P(w_2 = \texttt{EOS} \mid w_1 = \text{a}) &= 0.4. \end{aligned}$$

Simple RNNs are certainly not the only kinds of RNNs; the RNNs most commonly used in NLP today are based on *long-short term memory* (LSTM) (Hochreiter and Schmidhuber, 1997).

2.5.3 Example

Figure 2.3 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; maybe it's useful for other units to compute their values. The second unit is blue on the start symbol, then becomes

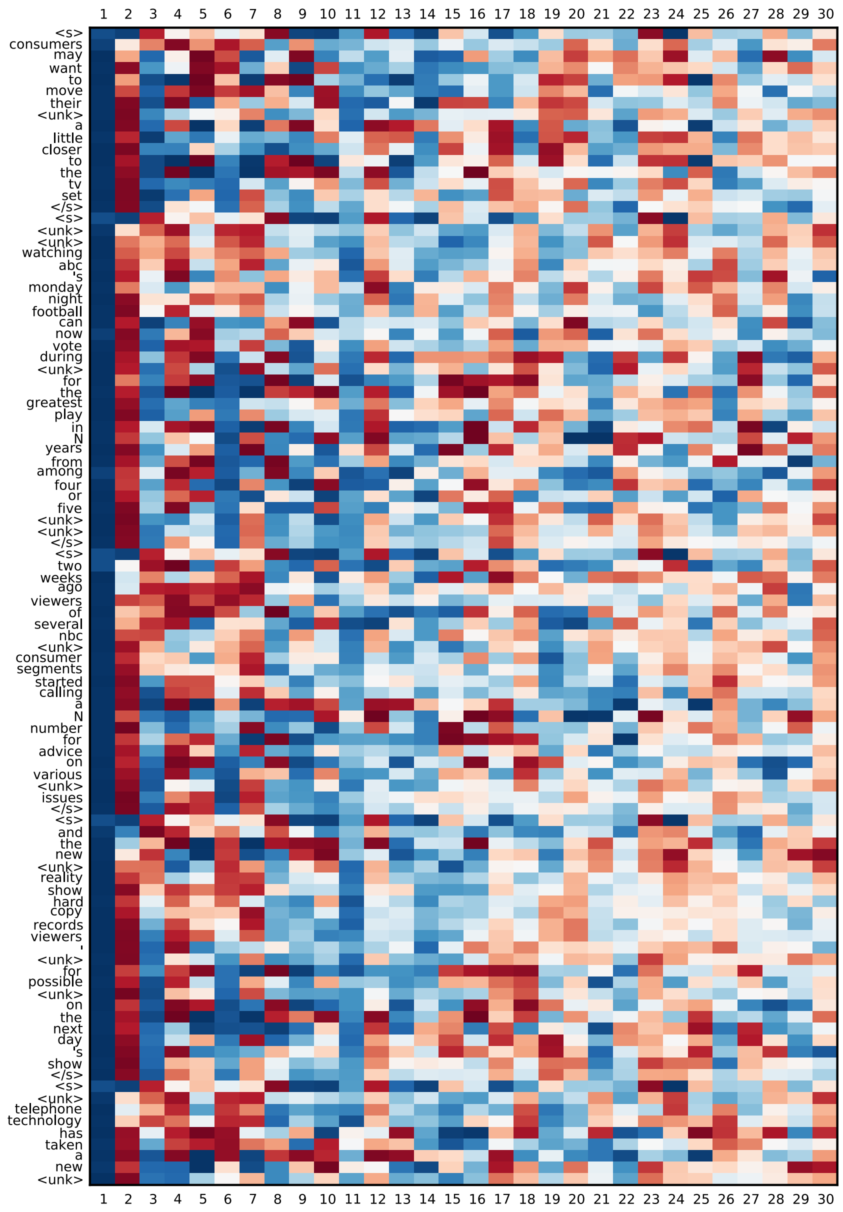


Figure 2.3: Visualization of a simple RNN language model on English text.

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.

LSTM RNNs, which perform better than this simple RNN, have many more units with interpretable functions on natural language (Karpathy, Johnson, and Fei-Fei, 2016).

2.5.4 Training

We are given a set $\mathcal{D}$ of training sentences, each of which can be converted into a sequence of vectors, $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$. We write $\boldsymbol{\theta}$ for the collection of all the parameters of the model, flattened into a single vector: $\boldsymbol{\theta} = (\mathbf{A}, \mathbf{B}, \mathbf{c}, \mathbf{D}, \mathbf{e})$. For each training example and each time step t, the RNN predicts the probability of word w_t as a vector $\mathbf{y}^{(t)}$.

During training, our goal is to find the parameter values that maximize the log-likelihood,[1]

$$
\begin{aligned}
L(\boldsymbol{\theta}) &= \log \prod_{w \in \mathcal{D}} P(w; \boldsymbol{\theta}) && (2.24)\\
&= \sum_{w \in \mathcal{D}} \log P(w; \boldsymbol{\theta}) && (2.25)\\
&= \sum_{w \in \mathcal{D}} \sum_{t=1}^{n} \mathbf{x}^{(t)} \cdot \log \mathbf{y}^{(t)}. && (2.26)
\end{aligned}
$$

To maximize this function, there are lots of different methods. We're going to look at the easiest (but still very practical) method, *stochastic gradient ascent*.[2] This algorithm goes back to the perceptron (Rosenblatt, 1958), which was a shallow trainable neural network, and the backpropagation algorithm (Rumelhart, Hinton, and Williams, 1986). Imagine that the log-likelihood is an infinite, many-dimensional surface. Each point on the surface corresponds to a setting of $\boldsymbol{\theta}$, and the altitude of the point is the log-likelihood for that setting of $\boldsymbol{\theta}$. We want to find the highest point on the surface. We start at an arbitrary location and then repeatedly move a little bit in the steepest uphill direction.

In pseudocode, gradient ascent looks like this:

```
initialize parameters $\boldsymbol{\theta}$ randomly
repeat
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla L(\boldsymbol{\theta})$
until done
```

[1]Since "likelihood," "log-likelihood," and "loss function" all start with L, it's common to write L for all three. Here, it stands for "log-likelihood."

[2]If we're minimizing a function, then we use stochastic gradient *de*scent, and this is the name that the method is more commonly known by.

The randomness of the initialization is important, because there are many situations where if two parameters are initialized to the same value, they'll always have the same value and therefore be redundant.

The function ∇L is the gradient of L and gives the direction, at $\boldsymbol{\theta}$, that goes uphill the steepest. These days, it's uncommon to need to figure out what the gradient is by hand, because there are numerous automatic differentiation packages that do this for you.

The *learning rate* $\eta > 0$ controls how far we move at each step. (What happens if η is too small? too big?) To guarantee convergence, η should decrease over time (for example, $\eta = 1/t$), but it's also common in practice to leave it fixed. See below for another common trick.

In *stochastic* gradient ascent, we work on just one sentence at a time. Let $L_w(\boldsymbol{\theta})$ be the log-likelihood of just one sentence, that is,

$$L_w(\boldsymbol{\theta}) = \log P(w; \boldsymbol{\theta}) \tag{2.27}$$

$$= \sum_{t=1}^{n} \mathbf{x}^{(t)} \cdot \log \mathbf{y}^{(t)}. \tag{2.28}$$

It could be thought of as an approximation to the full log-likelihood $L(\boldsymbol{\theta})$. Then stochastic gradient ascent goes like this:

```
initialize parameters $\boldsymbol{\theta}$ to random numbers
repeat
    for each sentence $w$ do
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla L_w(\boldsymbol{\theta})$
    end for
until done
```

Each pass through the training data is called an *epoch*. This method has several advantages compared to full gradient ascent:

- Computing the gradient for one sentence uses much less memory.
- Updating the model after every sentence instead of waiting until the end of the data means that the model can get better faster.
- The per-sentence log-likelihoods are only an approximation to the full log-likelihood. This may seem like a disadvantage, because the updates can temporarily take us in the wrong direction. But there's some evidence that this actually improves generalization.

2.5.5 Tricks

There are a number of tricks that are important for training well. This is not a complete list, but these are the most essential and/or easiest tricks.

Validation. The above pseudocode doesn't specify how to choose the learning rate η or when to stop. There are many ways to do this, but one tried-and-true method is to look at the score (likelihood or some other metric) on held-out data (also known as validation data). At the end of each epoch, run on the validation data and compute the score. If it got worse, multiply the learning rate by $\frac{1}{2}$ and continue. Usually, the validation score will start to go up again. If the learning rate goes below some threshold (say, after a certain number of halvings), stop training.

Shuffling. Because stochastic gradient ascent updates the model based on one sentence at a time, it will have a natural tendency to remember the recent sentences most. To mitigate this effect, before each epoch, randomly shuffle the order of the training sentences.

Gradient clipping. When using SGA on RNNs, 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. There are fancier learning methods than SGA that alleviate this problem (currently, the most popular is probably Adam), but for SGA, the simplest fix is *gradient clipping*: just check if the norm of the gradient is bigger than 5, and if so, scale it so that its norm is just 5. The PyTorch function `torch.nn.utils.clip_grad_norm_` does this for you.

Minibatching. To speed up training and/or to reduce random variations between sentences, it's standard to train on a small number (10–1000) of sentences at a time instead of a single sentence at a time. If we can process the sentences in one minibatch in parallel, we get a huge speedup. For example, if the model contains the matrix-vector product $\mathbf{Ah}$ where $\mathbf{A}$ is a parameter matrix and $\mathbf{h}$ is a vector that depends on the input sentence, then with minibatching, $\mathbf{h}$ becomes a matrix (one row for each sentence), and $\mathbf{Ah}$ can become a matrix-matrix product, which is much faster than a bunch of matrix-vector products. You just have to make sure that the indices match up correctly: $\mathbf{hA}^\top$ or, in PyTorch, $\mathbf{A}$ @ $\mathbf{h}[:, :, \text{None}]$.

However, a major nuisance is that the sentences are all different lengths. The typical solution goes like this:

- Sort all the sentences by length.
- Divide up the sentences into minibatches. Because of the sorting, each minibatch contains sentences with similar lengths.
- In each minibatch, equalize the lengths of sentences by appending a special symbol `PAD`.
- When computing L, mask out the `PAD` symbols to avoid biasing the model towards predicting `PAD` (not to mention wasting training time).

We do not expect you to implement minibatching in the homework assignments.

2.6 Transformers? (not yet)

RNNs have largely been displaced by transformers (Vaswani et al., 2017). But to introduce transformers, we are going to take a detour to the problem that transformers were originally invented for, which is machine translation.

Bibliography

Chen, Stanley F. and Joshua Goodman (1998). *An Empirical Study of Smoothing Techniques for Language Modeling*. Tech. rep. TR-10-98. Harvard University Center for Research in Computing Technology. URL: `http://nrs.harvard.edu/urn-3:HUL.InstRepos:25104739`.

Elman, Jeffrey L. (1990). "Finding Structure in Time". In: *Cognitive Science* 14, pp. 179–211.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Karpathy, Andrej, Justin Johnson, and Li Fei-Fei (2016). "Visualizing and Understanding Recurrent Neural Networks". In: *Proc. ICLR*. URL: `https://arxiv.org/abs/1506.02078`.

Kleene, S. C. (1951). *Representation of Events in Nerve Nets and Finite Automata*. Tech. rep. RM-704. RAND. URL: `https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM704.pdf`.

Kudo, Taku and John Richardson (Nov. 2018). "SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing". In: *Proc. EMNLP: System Demonstrations*, pp. 66–71. DOI: `10.18653/v1/D18-2012`. URL: `https://aclanthology.org/D18-2012`.

McCulloch, Warren S. and Walter Pitts (1943). "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5, pp. 115–133. URL: `https://doi.org/10.1007/BF02478259`.

Mohri, Mehryar (1997). *Finite-State Transducers and Language and Speech Processing*.

Rabin, M. O. and D. Scott (1959). "Finite Automata and Their Decision Problems". In: *IBM Journal of Research and Development* 3.2, pp. 114–125. URL: `https://doi.org/10.1147/rd.32.0114`.

Rosenblatt, F. (1958). "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological Review* 65.6, pp. 386–408.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323, pp. 533–536.

Sennrich, Rico, Barry Haddow, and Alexandra Birch (2016). "Neural Machine Translation of Rare Words with Subword Units". In: *Proc. ACL*, pp. 1715–1725. DOI: `10.18653/v1/P16-1162`.

Shannon, C. E. (1948). "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3, pp. 379–423.

Vaswani, Ashish et al. (2017). "Attention is All You Need". In: *Proc. NeurIPS*, pp. 5998–6008. URL: `https://papers.nips.cc/paper/7181-attention-is-all-you-need`.