Chapter 2

Language Models

2.1 Motivation: Machine Translation

To start the course, we're going to look at the problem of *machine translation*, that is, translating sentences from one language to another (traditionally, a French sentence f to an English sentence e). A translation system has to try to do two things at once: first, it has to generate an e that means the same thing as f (*adequacy*), and second, it has to generate an e that is good English (*fluency*). We could produce a perfectly adequate but not fluent translation by outputting f itself; we could produce a perfectly fluent but not adequate translation by always outputting "My hovercraft is full of eels." Doing both at once is what makes the problem nontrivial.

Neural networks are rather good at combining two kinds of information like this. But older machine learning methods were not as good at doing this, so they divided the work between two submodels, in the following way. Warren Weaver first proposed, in 1947, to treat translation as a decoding problem:

> One naturally wonders if the problem of translation could conceivably be treated as a problem in cryptography. When I look at an article in Russian, I say: 'This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode.'

In other words, when a Russian speaker speaks Russian, he first thinks in English, but then, as he expresses his thoughts, they somehow become encoded in Russian. Then (leaving aside how absurd this is), the task of translating a Russian sentence f is to recover the original English sentence e that the speaker was thinking before he said it. Mathematically,

$$P(f, e) = P(e)\,P(f \mid e).$$

Then, if we are given a sentence f, we can reconstruct e by finding:

$$\begin{aligned} e^* &= \arg\max_e P(e \mid f) \\ &= \arg\max_e \frac{P(e,f)}{P(f)} \\ &= \arg\max_e P(e,f) \\ &= \arg\max_e P(e)\, P(f \mid e). \end{aligned}$$

So the model is divided into two parts. The term $P(f \mid e)$ is called the *translation model*, which says how similar the two sentences are in meaning (adequacy). The term $P(e)$ is the *language model.* It says what kinds of sentences are more likely than others (fluency).

In this chapter, we will focus on the language model. With apologies, we're going to switch notation; since we're leaving translation aside for the moment, we don't need to distinguish between languages, and we simply call the sentence $w = w_1 \cdots w_N$, where each w_i 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.)

2.2 n-gram Models

2.2.1 Definition

The simplest kind of language model is the 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 EOS, which stands for "end of sentence," so $w = w_1 \cdots w_N$ where $w_N = \texttt{EOS}$. In a 1-gram or *unigram* language model, each word is generated independently:

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

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:

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

where we now also have a special symbol BOS for the beginning of the sentence, because the first word doesn't have a real previous word to condition on.

A general n-gram language model is:

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

where we pretend that $w_t = \texttt{BOS}$ for $t \leq 0$.

2.2.2 Training

Training an n-gram model is easy. To estimate the probabilities of a unigram language model, just count the number of times each word occurs and divide it by the total number of words:

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

where Σ is the vocabulary (including $\texttt{EOS}$) and $a \in \Sigma$.

For general n-grams,

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

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

2.2.3 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. With n-gram language models, the classic solution is *smoothing*, which tries to take some probability mass away from seen n-grams and give it to unseen n-grams. 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

$$P(\text{puppy} \mid \text{pulchritudinous}) = P(\text{pachycephalosaur} \mid \text{pulchritudinous}) = 0.$$

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 $P(\text{puppy} \mid \text{pulchritudinous}) > P(\text{pachycephalosaur} \mid \text{pulchritudinous})$.

Recall that the unsmoothed probability estimate of an n-gram is

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

There are basically two ways to take probability mass away: multiply the probability by $\lambda < 1$, or subtract $d > 0$ from the numerator. Then the probability distribution doesn't sum to one, so we need to add probability mass back proportional to some other distribution $\bar{p}$. For example,

$$\text{Multiplying:} \qquad p(a \mid u) = \lambda \frac{c(ua)}{\sum\limits_{a' \in \Sigma} c(ua')} + (1 - \lambda)\bar{p}(a \mid u) \tag{2.5}$$

$$\text{Subtracting:} \qquad p(a \mid u) = \frac{\max(0, c(ua) - d)}{\sum\limits_{a' \in \Sigma} c(ua')} + \alpha\bar{p}(a \mid u). \tag{2.6}$$

where:

- λ and d must be chosen carefully; for much more information on how to do that, see the technical report by Chen and Goodman (1998).
- In (2.6), α is chosen to make the distribution sum to one; I thought it'd be simpler to omit an explicit formula.
- The distribution $\bar{p}$ is typically simpler than p. If p is an n-gram model where $n > 1$, then $\bar{p}$ is typically an $(n - 1)$-gram model. If p is a unigram model, then $\bar{p}$ is typically the uniform distribution over words.

A special case of (2.5) is simply to add 1 to the count of every n-gram:

$$p(a \mid u) = \frac{c(ua) + 1}{\sum\limits_{a' \in \Sigma} (c(ua') + 1)}, \tag{2.7}$$

which is called *add-one smoothing*. (How is this a special case of (2.5)?) In many situations (and particularly for language modeling), this is a terrible smoothing method. But it's good to know because it's so easy.

2.3 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, in any data outside the training data, *unknown* or *out-of-vocabulary* (OOV) words are rather common. Unknown words are problematic for all language models, and we have a few techniques for handling them.

2.3.1 Smoothing

Above, we saw that smoothing decreases probability estimates of seen n-grams and increases probability estimates of unseen n-grams, including zero-probability n-grams. If we add a pseudo-word `UNK` to our model's vocabulary, it'll have a count of zero, but smoothing will give it a nonzero probability.

Then, when we are using the language model to score new sentences, whenever we encounter a word that was not seen in the training data, we replace it with `UNK`, which has nonzero probability thanks to smoothing.

2.3.2 Limiting the vocabulary

A simpler method is to pretend that some word types seen in the training data are unknown. For example, we might limit the vocabulary to 10,000 word types, and all other word types are changed to `UNK`. Or, we might limit the vocabulary just to those seen two or more times, and all other word types are changed to `UNK`. When we train the language model, we treat `UNK` like any other symbol, so it gets a nonzero probability.

As above, when we are using the language model to score new sentences, whenever we encounter a word that was not seen in the training data, we replace it with `UNK`.

2.3.3 Subword segmentation

Another idea is to break words into smaller pieces, usually using a method like *byte pair encoding* (Sennrich, Haddow, and Birch, 2016). This breaks unknown words down into smaller, known, pieces. However, this only alleviates the unknown word problem; it does not solve it. So one of the above techniques is still needed.

2.4 Evaluation

Whenever we build any kind of model, we always have to think about how to evaluate it.

2.4.1 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.

2.4.2 Extrinsic evaluation

The best way to evaluate language models is extrinsically. Language models are usually used as part of some larger system, so to compare two language models, compare how much they help the larger system.

2.4.3 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:

$$L = P(w_1 \cdots w_N) \qquad \text{likelihood} \tag{2.8}$$

$$H = -\frac{1}{N} \log_2 L \qquad \text{per-word cross-entropy} \tag{2.9}$$

$$PP = 2^H \qquad \text{perplexity} \tag{2.10}$$

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, N.

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 N. 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 token `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?)

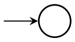
2.5 Weighted Automata

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.

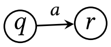
2.5.1 Finite automata

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:

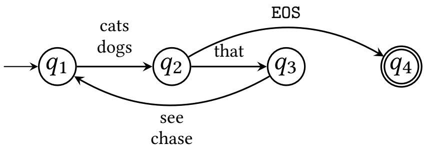


which means that if the machine is in state q and the next input symbol is a, then it can read in a and move to state r.

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:

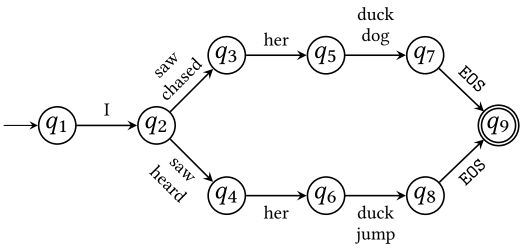
If the machine can reach the end of the string while in a final state, then it accepts the string. Otherwise, it rejects.

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.

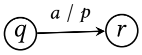
But if a state has two outgoing transitions with the same label, it is *nondeterministic* (or an NFA).



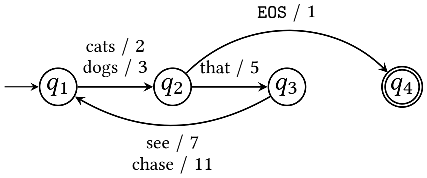
At q_2, if the next word is "saw," the NFA can go to either q_3 *or* q_4. If the sentence continues "her duck EOS," it ends up in state q_9 and accepts. If the sentence continues "her dog EOS," then the branch that was in q_3 goes to q_5, q_7, q_9 and accepts; the branch that was in q_4 goes to q_6 and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

2.5.2 Weights and probabilities

A *weighted finite automaton* adds a nonnegative real *weight* to each transition (Mohri, 1997). A transition on symbol a with weight p is written

 .

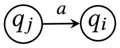
The weight of a path through a weighted FA is the product of the weights of the transitions along the path. For example, we can add weights to our example DFA:



The string "cats that chase dogs that see cats EOS" has weight $2 \cdot 5 \cdot 11 \cdot 3 \cdot 5 \cdot 7 \cdot 2 = 23100$.

In a weighted NFA, there may be more than one path that accepts a string. In that case, the weight of a string is the sum of the weights of all accepting paths of the string. A convenient way to formulate this is using matrices. Let $d = |Q|$ and number the states $q_1, \ldots, q_d$.

- Let $\mathbf{s}$ be the vector with a 1 corresponding to the start state and a 0 everywhere else.

- For each $a \in \Sigma$, we define a matrix $\mu(a) \in \mathbb{R}^{d \times d}$ such that $[\mu(a)]_{ij}$ is the weight of transition  .

- Let $\mathbf{f}$ be the vector with a 1 corresponding to the final state and a 0 everywhere else.

For example:

$$\mathbf{s} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{f} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\mu(\text{cats}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mu(\text{that}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mu(\text{see}) = \begin{bmatrix} 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{etc.}$$

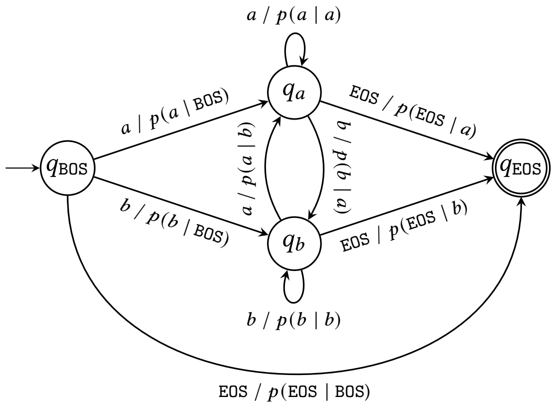
Then, the weight of a string $w = w_1 \cdots w_N$ is

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

In a *probabilistic DFA* or *NFA*, each state (except the final state) has the property that the weights of all of the outgoing transitions sum to one. This guarantees that the total weight of all strings is exactly one, that is, the automaton defines a probability distribution over all strings.

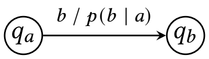
2.5.3 Language models as automata

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

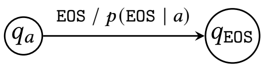


(2.11)

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



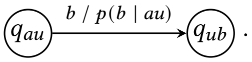
and for every state q_a, there is a transition



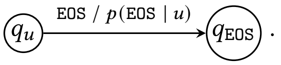
.

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:

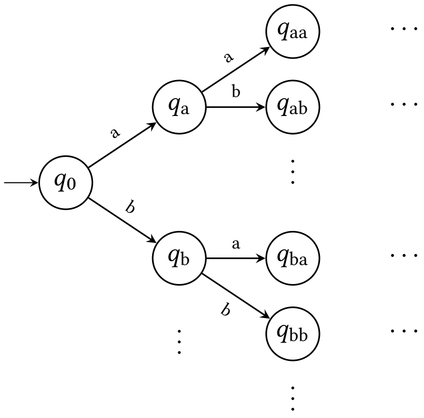
- For all $u \in \Sigma^{n-1}$, there is a state q_u.
- The start state is $q_{\texttt{BOS}^{n-1}}$.
- The accept state is $q_{\texttt{EOS}}$.
- For all $a \in \Sigma, u \in \Sigma^{n-2}, b \in \Sigma$, there's a transition



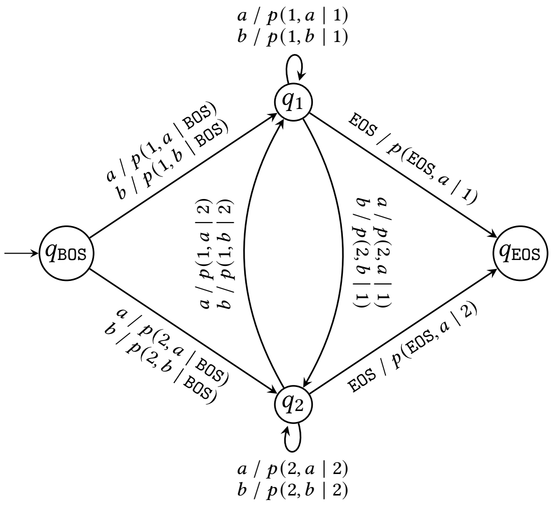
- For all $u \in \Sigma^{n-1}$, there's a transition



One can imagine designing other kinds of language models as well. For example, a *trie* is often used for storing lists like dictionaries:



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:

$$p(r, a \mid q) = t(r \mid q)\, o(a \mid r). \tag{2.12}$$

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: $q_{\texttt{BOS}}, q_1, q_2, q_1, q_1, q_{\texttt{EOS}}$ and $q_{\texttt{BOS}}, q_1, q_2, q_1, q_2, q_{\texttt{EOS}}$.

2.5.4 Training

Suppose we are given a collection $\mathcal{D}$ of strings. We also have an automaton M, and we want to learn weights for M.

If M is deterministic, this is very easy. 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

$$\text{weight}\left(\textcircled{q} \xrightarrow{a} \textcircled{r}\right) = \frac{c(q, a)}{\sum_{a'} c(q, a')}.$$

This is the weighting of M that maximizes the likelihood of $\mathcal{D}$.

If the automaton is nondeterministic, the above 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,

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

and recall that if $w = w_1 \cdots w_N \, \texttt{EOS}$,

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

The bad news is that we can't just maximize this by setting its derivatives to zero and solving for the transition weights. 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 2.6.4).

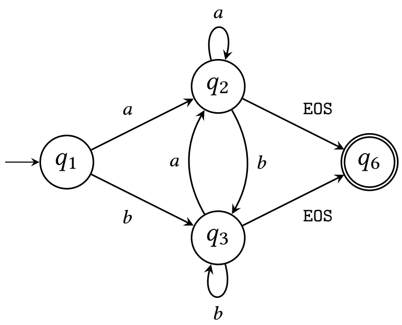
2.6 Recurrent Neural Networks

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).

Since we are done talking about n-grams, we switch to using n instead of N for the length of a string.

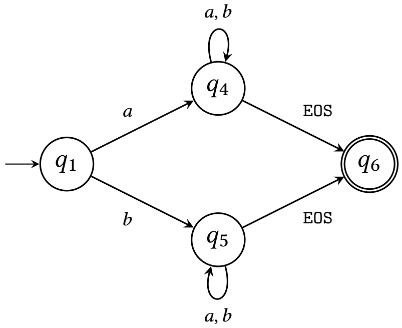
2.6.1 From finite automata...

I'd like to motivate RNNs from a historical point of view, because RNNs and finite automata have a common ancestor. Let's start with a bigram language model, M_{bigram}, without weights. To keep the diagram simple, we assume an alphabet $\Sigma = \{a, b, \texttt{EOS}\}$, and we disallow the empty string.



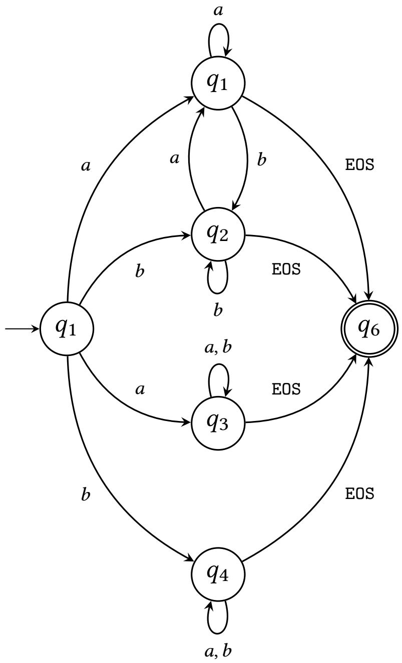
(2.13)

In contrast to the bigram model, if you, a human, had to predict the next word in a sentence, you might use information from further back in the string. For example, maybe we want the very first symbol to influence all later symbols – like if the first symbol is a Chinese character, the rest are likely to be Chinese characters, or if the first symbol is a lowercase letter, the rest are likely to be lowercase letters. Sticking with our alphabet of $\{a, b\}$ for simplicity, an automaton where all symbols depend on the first one (and only the first one) looks like this:



(2.14)

Now we'd like to combine the two models, so that each symbol can depend on both the first symbol and the previous symbol. How might we do this? One answer is to take their union (now hopefully my weird state numbering makes sense):

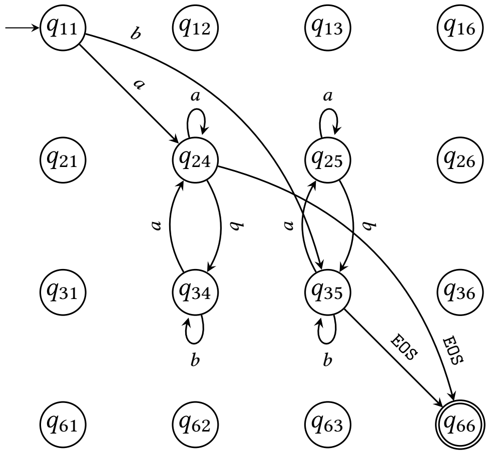


(2.15)

This new automaton likes a string if the bigram model likes it *or* the first-symbol model likes it. That may be useful in some situations: for example, if we think the input could be in English or Spanish, then it makes sense to take the union of an English model and a Spanish model. But here, we don't think that strings come in two kinds, one where each symbol depends on the previous one and one where each symbol depends on the first one.

What we really want here is to intersect them, so that the combined model likes a string if the bigram model likes it *and* the first-symbol model likes it. If you remember how this works from *Theory of Computing*, the resulting model simulates both the original automata at the same time, which requires a lot of states: it's the *product* of the number of states in the two original automata.

Here's what the intersection of our bigram and first-symbol automata looks like. You don't need to understand this; it's just here to convince you that you don't want to do this.



To get around this explosion in the number of states, we look back to the original definition of finite automata given by Kleene (1951). Under the definition you are familiar with (Rabin and Scott, 1959), an (unweighted) NFA can enter state r if it can get there by *at least one* incoming transition. We can formulate this using matrices like we did for weighted NFAs:

$$\mathbf{h}^{(0)} = \mathbf{s} \tag{2.16}$$

$$\mathbf{h}^{(t)} = \mathbb{I}[\boldsymbol{\mu}(w_t)\mathbf{h}^{(t-1)} \geq 1] \qquad t = 1, \ldots, n \tag{2.17}$$

At each time step t, the vector $\mathbf{h}^{(t)}$ keeps track of what states the NFA could be in. (The superscripts are written with parentheses to make it clear that this isn't exponentiation.) As before, $\mathbf{s} \in \mathbb{Z}^d$ is a vector with a 1 for the start state and a 0 for other states. And for each $a \in \Sigma$, $\boldsymbol{\mu}(a) \in \mathbb{Z}^{d \times d}$ is a matrix such that $[\boldsymbol{\mu}(a)]_{ij} = 1$ if there is a transition from q_i on a to q_j, and 0 otherwise. The notation $\mathbb{I}[\cdot]$ is 1 if the thing inside the square brackets is true and 0 otherwise; here, it formalizes the intuition that the machine can be in a state iff it can get there by *at least one* transition.

Kleene allowed $\mathbf{h}^{(t)}$ to be defined by an arbitrary function of $\mathbf{h}^{(t-1)}$ and w_t. Although his automata are no more powerful than DFAs and NFAs in the sense that they all recognize the same languages, there is a sense in which Kleene's automata are much more powerful. For example, in an NFA, every state has a threshold of 1, but what if we let different states have different thresholds?

$$\mathbf{h}^{(0)} = \mathbf{s} \tag{2.18}$$

$$\mathbf{h}^{(t)} = \mathbb{I}[\boldsymbol{\mu}(w_t)\mathbf{h}^{(t-1)} \geq \boldsymbol{\theta}] \qquad t = 1, \ldots, n \tag{2.19}$$

where $\boldsymbol{\theta}$ is a vector of thresholds; the automaton can enter state q_i if it can get there there by at least $\boldsymbol{\theta}_i$ incoming transitions. In the NFA above (2.15), we can let q_6 have a threshold of 2 (that is, $\boldsymbol{\theta}_6 = 2$). This means that a string is now accepted by the automaton iff it is accepted by *both* of the original automata. (The automaton can never be in q_2 and q_3 at once, nor q_4 and q_5.) With this little change, we get the power of intersection without the huge number of states that intersection usually creates.

2.6.2 …to recurrent neural networks

Digging further back into history, Kleene's finite automata were a generalization of neural networks, which had been defined by McCulloch and Pitts (1943). We use, however, modern notation instead of the original crazy notation.

Just as we previously numbered all the states, so now we number the symbols of the alphabet Σ, starting from 1. The ordering is completely arbitrary. For example, if $\Sigma = \{\text{a}, \text{b}, \texttt{EOS}\}$, we could number them: $\text{a} = 1, \text{b} = 2, \texttt{EOS} = 3$. If the input string is $w = w_1 \cdots w_n$, define a sequence of vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$. Each vector $\mathbf{x}^{(t)}$ 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_i. For example, if $w = \text{aba}\,\texttt{EOS}$, then the input vectors would be

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

A McCulloch-Pitts neural network is defined as follows:

$$\mathbf{h}^{(t)} = \mathbb{I}\left[\mathbf{A}\mathbf{h}^{(t-1)} + \mathbf{B}\mathbf{x}^{(t)} + \mathbf{c} \geq 0\right] \qquad t = 1, \ldots, n \tag{2.20}$$

where

$$\mathbf{h}^{(0)} \in \mathbb{Z}^d \tag{2.21}$$

$$\mathbf{A} \in \mathbb{Z}^{d \times d} \tag{2.22}$$

$$\mathbf{B} \in \mathbb{Z}^{d \times |\Sigma|} \tag{2.23}$$

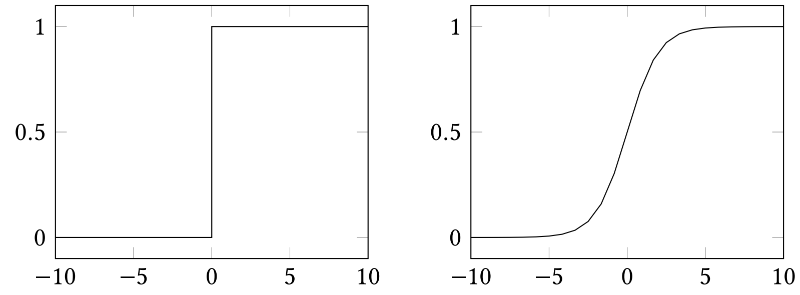
$$\mathbf{c} \in \mathbb{Z}^d \tag{2.24}$$

Instead of $\boldsymbol{\mu}(w_t)\mathbf{h}^{(t-1)}$, we now have two separate terms $\mathbf{A}\mathbf{h}^{(t-1)} + \mathbf{B}\mathbf{x}^{(t)}$. This is smaller than an NFA ($d \times d + d \times |\Sigma|$ weights instead of $|\Sigma| \times d \times d$) and less powerful (since there is no direct dependence between q and a). We also now have the term $\mathbf{c}$, which plays the same role as $\boldsymbol{\theta}$ from before and, as we've seen, lets us do powerful things like simulate intersection.

As we will see, we will learn weights for neural networks 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,

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

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



(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.

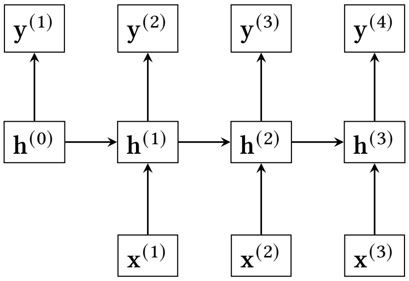


Figure 2.2: A simple RNN, shown for a string of length $n = 4$ (including EOS). Each rectangle is a vector that is either input to or computed by the network.

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:

$$\mathbf{h}^{(t)} = \text{sigmoid}(\mathbf{A}\mathbf{h}^{(t-1)} + \mathbf{B}\mathbf{x}^{(t)} + \mathbf{c}) \qquad t = 1, \ldots, n \tag{2.25}$$

where

$$\mathbf{h}^{(0)} \in \mathbb{R}^{d} \tag{2.26}$$

$$\mathbf{A} \in \mathbb{R}^{d \times d} \tag{2.27}$$

$$\mathbf{B} \in \mathbb{R}^{d \times |\Sigma|} \tag{2.28}$$

$$\mathbf{c} \in \mathbb{R}^{d} \tag{2.29}$$

are parameters of the model, which will be learned during the training process, as described in Section 2.6.4.

At each time step, the RNN can make a prediction about the next symbol (unlike a probabilistic automaton, in which the prediction of the next symbol is coupled with the decision of the next state):

$$\mathbf{y}^{(t)} = \text{softmax}(\mathbf{D}\mathbf{h}^{(t-1)} + \mathbf{e}) \qquad t = 1, \ldots, n \tag{2.30}$$

where

$$\mathbf{D} \in \mathbb{R}^{|\Sigma| \times d} \tag{2.31}$$

$$\mathbf{e} \in \mathbb{R}^{|\Sigma|} \tag{2.32}$$

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).

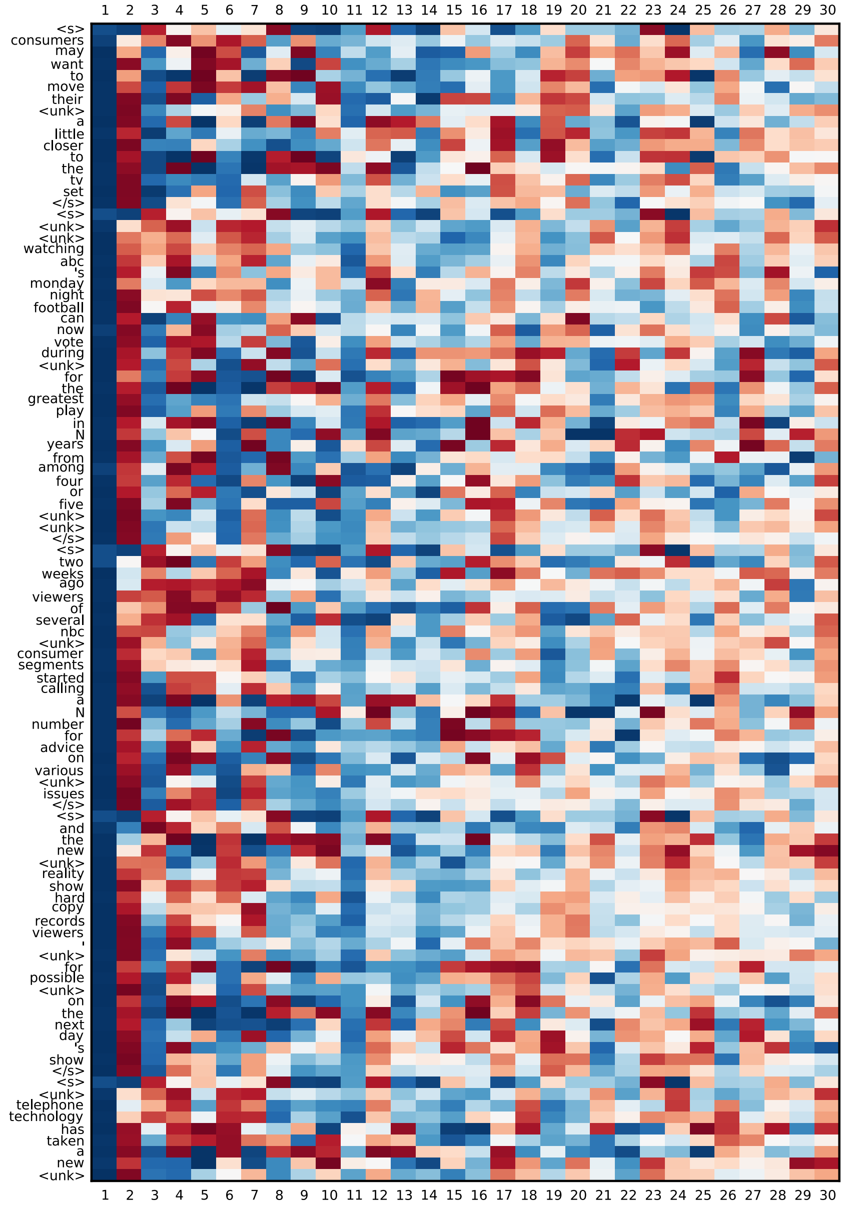


Figure 2.3: Visualization of a simple RNN language model on English text.

2.6.3 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 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).

2.6.4 Training

We are given a set $\mathcal{D}$ of training examples w, each of which can be converted into a sequence of vectors, $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$. We write $\boldsymbol{\theta}$ for the collection of all the parameters of the model, flattened into a single vector: $\boldsymbol{\theta} = (\mathbf{h}^{(0)}, \mathbf{A}, \mathbf{B}, \mathbf{c}, \mathbf{D}, \mathbf{e})$. For each training example and each time step i, the RNN predicts the probability of word i as a vector $\mathbf{y}^{(i)}$.

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

$$\begin{aligned} L(\boldsymbol{\theta}) &= \log \prod_{w \in \mathcal{D}} P(w; \boldsymbol{\theta}) && (2.33) \\ &= \sum_{w \in \mathcal{D}} \log P(w; \boldsymbol{\theta}) && (2.34) \\ &= \sum_{w \in \mathcal{D}} \sum_{t=1}^{n} \mathbf{x}^{(t)} \cdot \log \mathbf{y}^{(t)}. && (2.35) \end{aligned}$$

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

[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.

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
```

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,

$$L_w(\boldsymbol{\theta}) = \log P(w; \boldsymbol{\theta}) \tag{2.36}$$

$$= \sum_{t=1}^{n} \mathbf{x}^{(t)} \cdot \log \mathbf{y}^{(t)}. \tag{2.37}$$

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 two advantages and one disadvantage 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.

– Because the per-sentence log-likelihoods are only an approximation to the full log-likelihood, the updates can temporarily take us in the wrong direction.

The next section talks about one way to mitigate this disadvantage.

2.6.5 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 development or 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 or minibatches.

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).

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.

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.

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.