Chapter 6

Language models

In probabilistic classification, the problem is to compute the probability of a label, conditioned on the text. Let's now consider the inverse problem: computing the probability of text itself. Specifically, we will consider models that assign probability to a sequence of word tokens, $\text{p}(w_1, w_2, \ldots, w_M)$, with $w_m \in \mathcal{V}$. The set $\mathcal{V}$ is a discrete vocabulary,

$$\mathcal{V} = \{\textit{aardvark}, \textit{abacus}, \ldots, \textit{zither}\}. \quad [6.1]$$

Why would you want to compute the probability of a word sequence? In many applications, the goal is to produce word sequences as output:

- In **machine translation** (chapter 18), we convert from text in a source language to text in a target language.
- In **speech recognition**, we convert from audio signal to text.
- In **summarization** (§ 16.3.4; § 19.2), we convert from long texts into short texts.
- In **dialogue systems** (§ 19.3), we convert from the user's input (and perhaps an external knowledge base) into a text response.

In many of the systems for performing these tasks, there is a subcomponent that computes the probability of the output text. The purpose of this component is to generate texts that are more **fluent**. For example, suppose we want to translate a sentence from Spanish to English.

(6.1) El cafe negro me gusta mucho.

Here is a literal word-for-word translation (a **gloss**):

(6.2) The coffee black me pleases much.

A good language model of English will tell us that the probability of this translation is low, in comparison with more grammatical alternatives,

$$\text{p}(\textit{The coffee black me pleases much}) < \text{p}(\textit{I love dark coffee}). \quad [6.2]$$

How can we use this fact? Warren Weaver, one of the early leaders in machine translation, viewed it as a problem of breaking a secret code (Weaver, 1955):

> 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.'

This observation motivates a generative model (like Naïve Bayes):

- The English sentence $\boldsymbol{w}^{(e)}$ is generated from a **language model**, $\text{p}_e(\boldsymbol{w}^{(e)})$.
- The Spanish sentence $\boldsymbol{w}^{(s)}$ is then generated from a **translation model**, $\text{p}_{s|e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)})$.

Given these two distributions, translation can be performed by Bayes' rule:

$$\text{p}_{e|s}(\boldsymbol{w}^{(e)} \mid \boldsymbol{w}^{(s)}) \propto \text{p}_{e,s}(\boldsymbol{w}^{(e)}, \boldsymbol{w}^{(s)}) \quad [6.3]$$

$$= \text{p}_{s|e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)}) \times \text{p}_e(\boldsymbol{w}^{(e)}). \quad [6.4]$$

This is sometimes called the **noisy channel model**, because it envisions English text turning into Spanish by passing through a noisy channel, $\text{p}_{s|e}$. What is the advantage of modeling translation this way, as opposed to modeling $\text{p}_{e|s}$ directly? The crucial point is that the two distributions $\text{p}_{s|e}$ (the translation model) and p_e (the language model) can be estimated from separate data. The translation model requires examples of correct translations, but the language model requires only text in English. Such monolingual data is much more widely available. Furthermore, once estimated, the language model p_e can be reused in any application that involves generating English text, including translation from other languages.

6.1 N-gram language models

A simple approach to computing the probability of a sequence of tokens is to use a **relative frequency estimate**. Consider the quote, attributed to Picasso, "*computers are useless, they can only give you answers.*" One way to estimate the probability of this sentence is,

$$\begin{aligned}&\text{p}(\textit{Computers are useless, they can only give you answers}) \\ &= \frac{\text{count}(\textit{Computers are useless, they can only give you answers})}{\text{count}(\text{all sentences ever spoken})}\end{aligned} \quad [6.5]$$

This estimator is **unbiased**: in the theoretical limit of infinite data, the estimate will be correct. But in practice, we are asking for accurate counts over an infinite number of events, since sequences of words can be arbitrarily long. Even with an aggressive upper bound of, say, $M = 20$ tokens in the sequence, the number of possible sequences is V^{20}, where $V = |\mathcal{V}|$. A small vocabularly for English would have $V = 10^5$, so there are 10^{100} possible sequences. Clearly, this estimator is very data-hungry, and suffers from high variance: even grammatical sentences will have probability zero if they have not occurred in the training data.[1] We therefore need to introduce bias to have a chance of making reliable estimates from finite training data. The language models that follow in this chapter introduce bias in various ways.

We begin with n-gram language models, which compute the probability of a sequence as the product of probabilities of subsequences. The probability of a sequence $\text{p}(\boldsymbol{w}) = \text{p}(w_1, w_2, \ldots, w_M)$ can be refactored using the chain rule (see § A.2):

$$\begin{aligned}\text{p}(\boldsymbol{w}) =& \text{p}(w_1, w_2, \ldots, w_M) && [6.6]\\ =& \text{p}(w_1) \times \text{p}(w_2 \mid w_1) \times \text{p}(w_3 \mid w_2, w_1) \times \ldots \times \text{p}(w_M \mid w_{M-1}, \ldots, w_1) && [6.7]\end{aligned}$$

Each element in the product is the probability of a word given all its predecessors. We can think of this as a *word prediction* task: given the context *Computers are*, we want to compute a probability over the next token. The relative frequency estimate of the probability of the word *useless* in this context is,

$$\begin{aligned}\text{p}(\textit{useless} \mid \textit{computers are}) &= \frac{\text{count}(\textit{computers are useless})}{\sum_{x \in \mathcal{V}} \text{count}(\textit{computers are } x)} \\ &= \frac{\text{count}(\textit{computers are useless})}{\text{count}(\textit{computers are})}.\end{aligned}$$

We haven't made any approximations yet, and we could have just as well applied the chain rule in reverse order,

$$\text{p}(\boldsymbol{w}) = \text{p}(w_M) \times \text{p}(w_{M-1} \mid w_M) \times \ldots \times \text{p}(w_1 \mid w_2, \ldots, w_M), \quad [6.8]$$

or in any other order. But this means that we also haven't really made any progress: to compute the conditional probability $\text{p}(w_M \mid w_{M-1}, w_{M-2}, \ldots, w_1)$, we would need to model V^{M-1} contexts. Such a distribution cannot be estimated from any realistic sample of text.

To solve this problem, n-gram models make a crucial simplifying approximation: they condition on only the past $n - 1$ words.

$$\text{p}(w_m \mid w_{m-1} \ldots w_1) \approx \text{p}(w_m \mid w_{m-1}, \ldots, w_{m-n+1}) \quad [6.9]$$

[1] Chomsky famously argued that this is evidence against the very concept of probabilistic language models: no such model could distinguish the grammatical sentence *colorless green ideas sleep furiously* from the ungrammatical permutation *furiously sleep ideas green colorless*.

This means that the probability of a sentence $\boldsymbol{w}$ can be approximated as

$$\mathrm{p}(w_1, \ldots, w_M) \approx \prod_{m=1}^{M} \mathrm{p}(w_m \mid w_{m-1}, \ldots, w_{m-n+1}) \quad [6.10]$$

To compute the probability of an entire sentence, it is convenient to pad the beginning and end with special symbols $\square$ and $\blacksquare$. Then the bigram ($n = 2$) approximation to the probability of *I like black coffee* is:

$$\mathrm{p}(\textit{I like black coffee}) = \mathrm{p}(I \mid \square) \times \mathrm{p}(\textit{like} \mid I) \times \mathrm{p}(\textit{black} \mid \textit{like}) \times \mathrm{p}(\textit{coffee} \mid \textit{black}) \times \mathrm{p}(\blacksquare \mid \textit{coffee}). \quad [6.11]$$

This model requires estimating and storing the probability of only V^n events, which is exponential in the order of the n-gram, and not V^M, which is exponential in the length of the sentence. The n-gram probabilities can be computed by relative frequency estimation,

$$\mathrm{p}(w_m \mid w_{m-1}, w_{m-2}) = \frac{\mathrm{count}(w_{m-2}, w_{m-1}, w_m)}{\sum_{w'} \mathrm{count}(w_{m-2}, w_{m-1}, w')} \quad [6.12]$$

The hyperparameter n controls the size of the context used in each conditional probability. If this is misspecified, the language model will perform poorly. Let's consider the potential problems concretely.

When n is too small. Consider the following sentences:

(6.3) **Gorillas** always like to groom **their** friends.

(6.4) The **computer** that's on the 3rd floor of our office building **crashed**.

In each example, the words written in bold depend on each other: the likelihood of *their* depends on knowing that *gorillas* is plural, and the likelihood of *crashed* depends on knowing that the subject is a *computer*. If the n-grams are not big enough to capture this context, then the resulting language model would offer probabilities that are too low for these sentences, and too high for sentences that fail basic linguistic tests like number agreement.

When n is too big. In this case, it is hard good estimates of the n-gram parameters from our dataset, because of data sparsity. To handle the *gorilla* example, it is necessary to model 6-grams, which means accounting for V^6 events. Under a very small vocabulary of $V = 10^4$, this means estimating the probability of 10^{24} distinct events.

These two problems point to another **bias-variance tradeoff** (see § 2.2.4). A small n-gram size introduces high bias, and a large n-gram size introduces high variance. We can even have both problems at the same time! Language is full of long-range dependencies that we cannot capture because n is too small; at the same time, language datasets are full of rare phenomena, whose probabilities we fail to estimate accurately because n is too large. One solution is to try to keep n large, while still making low-variance estimates of the underlying parameters. To do this, we will introduce a different sort of bias: **smoothing**.

6.2 Smoothing and discounting

Limited data is a persistent problem in estimating language models. In § 6.1, we presented n-grams as a partial solution. Bit sparse data can be a problem even for low-order n-grams; at the same time, many linguistic phenomena, like subject-verb agreement, cannot be incorporated into language models without high-order n-grams. It is therefore necessary to add additional inductive biases to n-gram language models. This section covers some of the most intuitive and common approaches, but there are many more (see Chen and Goodman, 1999).

6.2.1 Smoothing

A major concern in language modeling is to avoid the situation $\text{p}(\boldsymbol{w}) = 0$, which could arise as a result of a single unseen n-gram. A similar problem arose in Naïve Bayes, and the solution was **smoothing**: adding imaginary "pseudo" counts. The same idea can be applied to n-gram language models, as shown here in the bigram case,

$$\text{p}_{\text{smooth}}(w_m \mid w_{m-1}) = \frac{\text{count}(w_{m-1}, w_m) + \alpha}{\sum_{w' \in \mathcal{V}} \text{count}(w_{m-1}, w') + V\alpha}. \quad [6.13]$$

This basic framework is called **Lidstone smoothing**, but special cases have other names:

- **Laplace smoothing** corresponds to the case $\alpha = 1$.
- **Jeffreys-Perks law** corresponds to the case $\alpha = 0.5$, which works well in practice and benefits from some theoretical justification (Manning and Schütze, 1999).

To ensure that the probabilities are properly normalized, anything that we add to the numerator (α) must also appear in the denominator ($V\alpha$). This idea is reflected in the concept of **effective counts**:

$$c_i^* = (c_i + \alpha)\frac{M}{M + V\alpha}, \quad [6.14]$$

			Lidstone smoothing, $\alpha = 0.1$		Discounting, $d = 0.1$	
	counts	unsmoothed probability	effective counts	smoothed probability	effective counts	smoothed probability
impropriety	8	0.4	7.826	0.391	7.9	0.395
offense	5	0.25	4.928	0.246	4.9	0.245
damage	4	0.2	3.961	0.198	3.9	0.195
deficiencies	2	0.1	2.029	0.101	1.9	0.095
outbreak	1	0.05	1.063	0.053	0.9	0.045
infirmity	0	0	0.097	0.005	0.25	0.013
cephalopods	0	0	0.097	0.005	0.25	0.013

Table 6.1: Example of Lidstone smoothing and absolute discounting in a bigram language model, for the context $(\textit{alleged}, \_)$, for a toy corpus with a total of twenty counts over the seven words shown. Note that discounting decreases the probability for all but the unseen words, while Lidstone smoothing increases the effective counts and probabilities for *deficiencies* and *outbreak*.

where c_i is the count of event i, c_i^* is the effective count, and $M = \sum_{i=1}^{V} c_i$ is the total number of tokens in the dataset $(w_1, w_2, \ldots, w_M)$. This term ensures that $\sum_{i=1}^{V} c_i^* = \sum_{i=1}^{V} c_i = M$. The **discount** for each n-gram is then computed as,

$$d_i = \frac{c_i^*}{c_i} = \frac{(c_i + \alpha)}{c_i} \frac{M}{(M + V\alpha)}.$$

6.2.2 Discounting and backoff

Discounting "borrows" probability mass from observed n-grams and redistributes it. In Lidstone smoothing, the borrowing is done by increasing the denominator of the relative frequency estimates. The borrowed probability mass is then redistributed by increasing the numerator for all n-grams. Another approach would be to borrow the same amount of probability mass from all observed n-grams, and redistribute it among only the unobserved n-grams. This is called **absolute discounting**. For example, suppose we set an absolute discount $d = 0.1$ in a bigram model, and then redistribute this probability mass equally over the unseen words. The resulting probabilities are shown in Table 6.1.

Discounting reserves some probability mass from the observed data, and we need not redistribute this probability mass equally. Instead, we can **backoff** to a lower-order language model: if you have trigrams, use trigrams; if you don't have trigrams, use bigrams; if you don't even have bigrams, use unigrams. This is called **Katz backoff**. In the simple

case of backing off from bigrams to unigrams, the bigram probabilities are,

$$c^*(i,j) = c(i,j) - d \quad [6.15]$$

$$\mathrm{p}_{\text{Katz}}(i \mid j) = \begin{cases} \frac{c^*(i,j)}{c(j)} & \text{if } c(i,j) > 0 \\ \alpha(j) \times \frac{\mathrm{p}_{\text{unigram}}(i)}{\sum_{i':c(i',j)=0} \mathrm{p}_{\text{unigram}}(i')} & \text{if } c(i,j) = 0. \end{cases} \quad [6.16]$$

The term $\alpha(j)$ indicates the amount of probability mass that has been discounted for context j. This probability mass is then divided across all the unseen events, $\{i' : c(i', j) = 0\}$, proportional to the unigram probability of each word i'. The discount parameter d can be optimized to maximize performance (typically held-out log-likelihood) on a development set.

6.2.3 *Interpolation

Backoff is one way to combine different order n-gram models. An alternative approach is **interpolation**: setting the probability of a word in context to a weighted sum of its probabilities across progressively shorter contexts.

Instead of choosing a single n for the size of the n-gram, we can take the weighted average across several n-gram probabilities. For example, for an interpolated trigram model,

$$\begin{aligned} \mathrm{p}_{\text{Interpolation}}(w_m \mid w_{m-1}, w_{m-2}) = &\lambda_3 \mathrm{p}_3^*(w_m \mid w_{m-1}, w_{m-2}) \\ &+ \lambda_2 \mathrm{p}_2^*(w_m \mid w_{m-1}) \\ &+ \lambda_1 \mathrm{p}_1^*(w_m). \end{aligned}$$

In this equation, p_n^* is the unsmoothed empirical probability given by an n-gram language model, and λ_n is the weight assigned to this model. To ensure that the interpolated $\mathrm{p}(\boldsymbol{w})$ is still a valid probability distribution, the values of λ must obey the constraint, $\sum_{n=1}^{n_{\max}} \lambda_n = 1$. But how to find the specific values?

An elegant solution is **expectation-maximization**. Recall from chapter 5 that we can think about EM as learning with *missing data*: we just need to choose missing data such that learning would be easy if it weren't missing. What's missing in this case? Think of each word w_m as drawn from an n-gram of unknown size, $z_m \in \{1 \ldots n_{\max}\}$. This z_m is the missing data that we are looking for. Therefore, the application of EM to this problem involves the following **generative model**:

for Each token $w_m, m = 1, 2, \ldots, M$ **do**:
  draw the n-gram size $z_m \sim$ Categorical(λ);
  draw $w_m \sim \mathrm{p}_{z_m}^*(w_m \mid w_{m-1}, \ldots, w_{m-z_m})$.

If the missing data $\{Z_m\}$ were known, then λ could be estimated as the relative frequency,

$$\lambda_z = \frac{\text{count}(Z_m = z)}{M} \tag{6.17}$$

$$\propto \sum_{m=1}^{M} \delta(Z_m = z). \tag{6.18}$$

But since we do not know the values of the latent variables Z_m, we impute a distribution q_m in the E-step, which represents the degree of belief that word token w_m was generated from a n-gram of order z_m,

$$q_m(z) \triangleq \Pr(Z_m = z \mid \boldsymbol{w}_{1:m}; \lambda) \tag{6.19}$$

$$= \frac{\text{p}(w_m \mid \boldsymbol{w}_{1:m-1}, Z_m = z) \times \text{p}(z)}{\sum_{z'} \text{p}(w_m \mid \boldsymbol{w}_{1:m-1}, Z_m = z') \times \text{p}(z')} \tag{6.20}$$

$$\propto \text{p}^*_z(w_m \mid \boldsymbol{w}_{1:m-1}) \times \lambda_z. \tag{6.21}$$

In the M-step, λ is computed by summing the expected counts under q,

$$\lambda_z \propto \sum_{m=1}^{M} q_m(z). \tag{6.22}$$

A solution is obtained by iterating between updates to q and λ. The complete algorithm is shown in Algorithm 10.

Algorithm 10 Expectation-maximization for interpolated language modeling

procedure ESTIMATE INTERPOLATED n-GRAM ($\boldsymbol{w}_{1:M}$, $\{\text{p}^*_n\}_{n \in 1:n_{\text{max}}}$)
  for $z \in \{1, 2, \ldots, n_{\text{max}}\}$ **do** ▷ Initialization
    $\lambda_z \leftarrow \frac{1}{n_{\text{max}}}$
  repeat
    for $m \in \{1, 2, \ldots, M\}$ **do** ▷ E-step
      for $z \in \{1, 2, \ldots, n_{\text{max}}\}$ **do**
        $q_m(z) \leftarrow \text{p}^*_z(w_m \mid \boldsymbol{w}_{1:m-}) \times \lambda_z$
      $\boldsymbol{q}_m \leftarrow \text{Normalize}(\boldsymbol{q}_m)$
    for $z \in \{1, 2, \ldots, n_{\text{max}}\}$ **do** ▷ M-step
      $\lambda_z \leftarrow \frac{1}{M} \sum_{m=1}^{M} q_m(z)$
  until tired
  return $\boldsymbol{\lambda}$

6.2.4 *Kneser-Ney smoothing

Kneser-Ney smoothing is based on absolute discounting, but it redistributes the resulting probability mass in a different way from Katz backoff. Empirical evidence points to Kneser-Ney smoothing as the state-of-art for n-gram language modeling (Goodman, 2001). To motivate Kneser-Ney smoothing, consider the example: *I recently visited* _. Which of the following is more likely: *Francisco* or *Duluth*?

Now suppose that both bigrams *visited Duluth* and *visited Francisco* are unobserved in the training data, and furthermore, that the unigram probability $\text{p}_1^*(\textit{Francisco})$ is greater than $\text{p}_1^*(\textit{Duluth})$. Nonetheless we would still guess that $\text{p}(\textit{visited Duluth}) > \text{p}(\textit{visited Francisco})$, because *Duluth* is a more "versatile" word: it can occur in many contexts, while *Francisco* usually occurs in a single context, following the word *San*. This notion of versatility is the key to Kneser-Ney smoothing.

Writing u for a context of undefined length, and $\text{count}(w, u)$ as the count of word w in context u, we define the Kneser-Ney bigram probability as

$$\text{p}_{KN}(w \mid u) = \begin{cases} \frac{\max(\text{count}(w,u)-d,0)}{\text{count}(u)}, & \text{count}(w,u) > 0 \\ \alpha(u) \times \text{p}_{\text{continuation}}(w), & \text{otherwise} \end{cases} \quad [6.23]$$

$$\text{p}_{\text{continuation}}(w) = \frac{|u : \text{count}(w,u) > 0|}{\sum_{w' \in \mathcal{V}} |u' : \text{count}(w',u') > 0|}. \quad [6.24]$$

Probability mass using absolute discounting d, which is taken from all unobserved n-grams. The total amount of discounting in context u is $d \times |w : \text{count}(w,u) > 0|$, and we divide this probability mass among the unseen n-grams. To account for versatility, we define the *continuation probability* $\text{p}_{\text{continuation}}(w)$ as proportional to the number of observed contexts in which w appears. The numerator of the continuation probability is the number of contexts u in which w appears; the denominator normalizes the probability by summing the same quantity over all words w'. The coefficient $\alpha(u)$ is set to ensure that the probability distribution $\text{p}_{KN}(w \mid u)$ sums to one over the vocabulary w.

The idea of modeling versatility by counting contexts may seem heuristic, but there is an elegant theoretical justification from Bayesian nonparametrics (Teh, 2006). Kneser-Ney smoothing on n-grams was the dominant language modeling technique before the arrival of neural language models.

6.3 Recurrent neural network language models

N-gram language models have been largely supplanted by neural networks. These models do not make the n-gram assumption of restricted context; indeed, they can incorporate arbitrarily distant contextual information, while remaining computationally and statistically tractable.

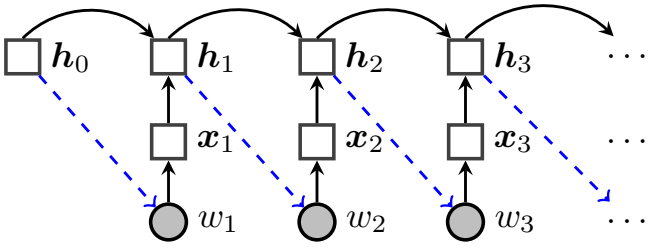

Figure 6.1: The recurrent neural network language model, viewed as an "unrolled" computation graph. Solid lines indicate direct computation, dotted blue lines indicate probabilistic dependencies, circles indicate random variables, and squares indicate computation nodes.

The first insight behind neural language models is to treat word prediction as a *discriminative* learning task.[2] The goal is to compute the probability $\text{p}(w \mid u)$, where $w \in \mathcal{V}$ is a word, and u is the context, which depends on the previous words. Rather than directly estimating the word probabilities from (smoothed) relative frequencies, we can treat treat language modeling as a machine learning problem, and estimate parameters that maximize the log conditional probability of a corpus.

The second insight is to reparametrize the probability distribution $\text{p}(w \mid u)$ as a function of two dense K-dimensional numerical vectors, $\boldsymbol{\beta}_w \in \mathbb{R}^K$, and $\boldsymbol{v}_u \in \mathbb{R}^K$,

$$\text{p}(w \mid u) = \frac{\exp(\boldsymbol{\beta}_w \cdot \boldsymbol{v}_u)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{v}_u)}, \quad [6.25]$$

where $\boldsymbol{\beta}_w \cdot \boldsymbol{v}_u$ represents a dot product. As usual, the denominator ensures that the probability distribution is properly normalized. This vector of probabilities is equivalent to applying the **softmax** transformation (see § 3.1) to the vector of dot-products,

$$\text{p}(\cdot \mid u) = \text{SoftMax}([\boldsymbol{\beta}_1 \cdot \boldsymbol{v}_u, \boldsymbol{\beta}_2 \cdot \boldsymbol{v}_u, \ldots, \boldsymbol{\beta}_V \cdot \boldsymbol{v}_u]). \quad [6.26]$$

The word vectors $\boldsymbol{\beta}_w$ are parameters of the model, and are estimated directly. The context vectors $\boldsymbol{v}_u$ can be computed in various ways, depending on the model. A simple but effective neural language model can be built from a **recurrent neural network** (RNN; Mikolov et al., 2010). The basic idea is to recurrently update the context vectors while moving through the sequence. Let $\boldsymbol{h}_m$ represent the contextual information at position m

[2] This idea predates neural language models (e.g., Rosenfeld, 1996; Roark et al., 2007).

in the sequence. RNN language models are defined,

$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m} \quad [6.27]$$

$$\boldsymbol{h}_m = \text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \quad [6.28]$$

$$\text{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}, \quad [6.29]$$

where $\boldsymbol{\phi}$ is a matrix of **word embeddings**, and $\boldsymbol{x}_m$ denotes the embedding for word w_m. The conversion of w_m to $\boldsymbol{x}_m$ is sometimes known as a **lookup layer**, because we simply lookup the embeddings for each word in a table; see § 3.2.4.

The **Elman unit** defines a simple recurrent operation (Elman, 1990),

$$\text{RNN}(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \triangleq g(\boldsymbol{\Theta}\boldsymbol{h}_{m-1} + \boldsymbol{x}_m), \quad [6.30]$$

where $\boldsymbol{\Theta} \in \mathbb{R}^{K \times K}$ is the recurrence matrix and g is a non-linear transformation function, often defined as the elementwise hyperbolic tangent $\tanh$ (see § 3.1).[3] The $\tanh$ acts as a **squashing function**, ensuring that each element of $\boldsymbol{h}_m$ is constrained to the range $[-1, 1]$.

Although each w_m depends on only the context vector $\boldsymbol{h}_{m-1}$, this vector is in turn influenced by *all* previous tokens, $w_1, w_2, \ldots w_{m-1}$, through the recurrence operation: w_1 affects $\boldsymbol{h}_1$, which affects $\boldsymbol{h}_2$, and so on, until the information is propagated all the way to $\boldsymbol{h}_{m-1}$, and then on to w_m (see Figure 6.1). This is an important distinction from n-gram language models, where any information outside the n-word window is ignored. In principle, the RNN language model can handle long-range dependencies, such as number agreement over long spans of text — although it would be difficult to know where exactly in the vector $\boldsymbol{h}_m$ this information is represented. The main limitation is that information is attenuated by repeated application of the squashing function g. **Long short-term memories** (LSTMs), described below, are a variant of RNNs that address this issue, using memory cells to propagate information through the sequence without applying nonlinearities (Hochreiter and Schmidhuber, 1997).

The denominator in Equation 6.29 is a computational bottleneck, because it involves a sum over the entire vocabulary. One solution is to use a **hierarchical softmax** function, which computes the sum more efficiently by organizing the vocabulary into a tree (Mikolov et al., 2011). Another strategy is to optimize an alternative metric, such as **noise-contrastive estimation** (Gutmann and Hyvärinen, 2012), which learns by distinguishing observed instances from artificial instances generated from a noise distribution (Mnih and Teh, 2012). Both of these strategies are described in § 14.5.3.

[3] In the original Elman network, the sigmoid function was used in place of $\tanh$. For an illuminating mathematical discussion of the advantages and disadvantages of various nonlinearities in recurrent neural networks, see the lecture notes from Cho (2015).

6.3.1 Backpropagation through time

The recurrent neural network language model has the following parameters:

- $\boldsymbol{\phi}_i \in \mathbb{R}^K$, the "input" word vectors (these are sometimes called **word embeddings**, since each word is embedded in a K-dimensional space; see chapter 14);
- $\boldsymbol{\beta}_i \in \mathbb{R}^K$, the "output" word vectors;
- $\boldsymbol{\Theta} \in \mathbb{R}^{K \times K}$, the recurrence operator;
- $\boldsymbol{h}_0$, the initial state.

Each of these parameters can be estimated by formulating an objective function over the training corpus, $L(\boldsymbol{w})$, and then applying backpropagation to obtain gradients on the parameters from a minibatch of training examples (see § 3.3.1). Gradient-based updates can be computed from an online learning algorithm such as stochastic gradient descent (see § 2.6.2).

The application of backpropagation to recurrent neural networks is known as **backpropagation through time**, because the gradients on units at time m depend in turn on the gradients of units at earlier times $n < m$. Let ℓ_{m+1} represent the negative log-likelihood of word $m + 1$,

$$\ell_{m+1} = -\log \mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m). \tag{6.31}$$

We require the gradient of this loss with respect to each parameter, such as $\theta_{k,k'}$, an individual element in the recurrence matrix Θ. Since the loss depends on the parameters only through $\boldsymbol{h}_m$, we can apply the chain rule of differentiation,

$$\frac{\partial \ell_{m+1}}{\partial \theta_{k,k'}} = \frac{\partial \ell_{m+1}}{\partial \boldsymbol{h}_m} \frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}. \tag{6.32}$$

The vector $\boldsymbol{h}_m$ depends on $\boldsymbol{\Theta}$ in several ways. First, $\boldsymbol{h}_m$ is computed by multiplying $\boldsymbol{\Theta}$ by the previous state $\boldsymbol{h}_{m-1}$. But the previous state $\boldsymbol{h}_{m-1}$ also depends on $\boldsymbol{\Theta}$:

$$\boldsymbol{h}_m = g(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}) \tag{6.33}$$

$$\frac{\partial h_{m,k}}{\partial \theta_{k,k'}} = g'(x_{m,k} + \boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-1})(h_{m-1,k'} + \boldsymbol{\theta}_k \cdot \frac{\partial \boldsymbol{h}_{m-1}}{\partial \theta_{k,k'}}), \tag{6.34}$$

where g' is the local derivative of the nonlinear function g. The key point in this equation is that the derivative $\frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$ depends on $\frac{\partial \boldsymbol{h}_{m-1}}{\partial \theta_{k,k'}}$, which will depend in turn on $\frac{\partial \boldsymbol{h}_{m-2}}{\partial \theta_{k,k'}}$, and so on, until reaching the initial state $\boldsymbol{h}_0$.

Each derivative $\frac{\partial \boldsymbol{h}_m}{\partial \theta_{k,k'}}$ will be reused many times: it appears in backpropagation from the loss ℓ_m, but also in all subsequent losses $\ell_{n>m}$. Neural network toolkits such as Torch (Collobert et al., 2011) and DyNet (Neubig et al., 2017) compute the necessary

derivatives automatically, and cache them for future use. An important distinction from the feedforward neural networks considered in chapter 3 is that the size of the computation graph is not fixed, but varies with the length of the input. This poses difficulties for toolkits that are designed around static computation graphs, such as TensorFlow (Abadi et al., 2016).[4]

6.3.2 Hyperparameters

The RNN language model has several hyperparameters that must be tuned to ensure good performance. The model capacity is controlled by the size of the word and context vectors K, which play a role that is somewhat analogous to the size of the n-gram context. For datasets that are large with respect to the vocabulary (i.e., there is a large token-to-type ratio), we can afford to estimate a model with a large K, which enables more subtle distinctions between words and contexts. When the dataset is relatively small, then K must be smaller too, or else the model may "memorize" the training data, and fail to generalize. Unfortunately, this general advice has not yet been formalized into any concrete formula for choosing K, and trial-and-error is still necessary. Overfitting can also be prevented by **dropout**, which involves randomly setting some elements of the computation to zero (Srivastava et al., 2014), forcing the learner not to rely too much on any particular dimension of the word or context vectors. The dropout rate must also be tuned on development data.

6.3.3 Gated recurrent neural networks

In principle, recurrent neural networks can propagate information across infinitely long sequences. But in practice, repeated applications of the nonlinear recurrence function causes this information to be quickly attenuated. The same problem affects learning: backpropagation can lead to **vanishing gradients** that decay to zero, or **exploding gradients** that increase towards infinity (Bengio et al., 1994). The exploding gradient problem can be addressed by clipping gradients at some maximum value (Pascanu et al., 2013). The other issues must be addressed by altering the model itself.

The **long short-term memory** (LSTM; Hochreiter and Schmidhuber, 1997) is a popular variant of RNNs that is more robust to these problems. This model augments the hidden state $\boldsymbol{h}_m$ with a **memory cell** $\boldsymbol{c}_m$. The value of the memory cell at each time m is a gated sum of two quantities: its previous value $\boldsymbol{c}_{m-1}$, and an "update" $\tilde{c}_m$, which is computed from the current input $\boldsymbol{x}_m$ and the previous hidden state $\boldsymbol{h}_{m-1}$. The next state $\boldsymbol{h}_m$ is then computed from the memory cell. Because the memory cell is not passed through a nonlinear squashing function during the update, it is possible for information to propagate through the network over long distances.

[4] See `https://www.tensorflow.org/tutorials/recurrent` (retrieved Feb 8, 2018).

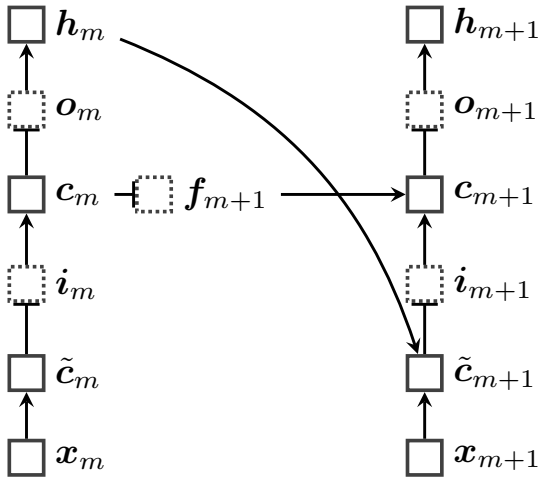

Figure 6.2: The long short-term memory (LSTM) architecture. Gates are shown in boxes with dotted edges. In an LSTM language model, each $\boldsymbol{h}_m$ would be used to predict the next word w_{m+1}.

The gates are functions of the input and previous hidden state. They are computed from elementwise sigmoid activations, $\sigma(x) = (1+\exp(-x))^{-1}$, ensuring that their values will be in the range $[0, 1]$. They can therefore be viewed as soft, differentiable logic gates. The LSTM architecture is shown in Figure 6.2, and the complete update equations are:

$$\boldsymbol{f}_{m+1} = \sigma(\boldsymbol{\Theta}^{(h\rightarrow f)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\rightarrow f)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_f) \quad \text{forget gate} \quad [6.35]$$

$$\boldsymbol{i}_{m+1} = \sigma(\boldsymbol{\Theta}^{(h\rightarrow i)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\rightarrow i)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_i) \quad \text{input gate} \quad [6.36]$$

$$\tilde{\boldsymbol{c}}_{m+1} = \tanh(\boldsymbol{\Theta}^{(h\rightarrow c)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(w\rightarrow c)}\boldsymbol{x}_{m+1}) \quad \text{update candidate} \quad [6.37]$$

$$\boldsymbol{c}_{m+1} = \boldsymbol{f}_{m+1} \odot \boldsymbol{c}_m + \boldsymbol{i}_{m+1} \odot \tilde{\boldsymbol{c}}_{m+1} \quad \text{memory cell update} \quad [6.38]$$

$$\boldsymbol{o}_{m+1} = \sigma(\boldsymbol{\Theta}^{(h\rightarrow o)}\boldsymbol{h}_m + \boldsymbol{\Theta}^{(x\rightarrow o)}\boldsymbol{x}_{m+1} + \boldsymbol{b}_o) \quad \text{output gate} \quad [6.39]$$

$$\boldsymbol{h}_{m+1} = \boldsymbol{o}_{m+1} \odot \tanh(\boldsymbol{c}_{m+1}) \quad \text{output.} \quad [6.40]$$

The operator $\odot$ is an elementwise (Hadamard) product. Each gate is controlled by a vector of weights, which parametrize the previous hidden state (e.g., $\boldsymbol{\Theta}^{(h\rightarrow f)}$) and the current input (e.g., $\boldsymbol{\Theta}^{(x\rightarrow f)}$), plus a vector offset (e.g., $\boldsymbol{b}_f$). The overall operation can be informally summarized as $(\boldsymbol{h}_m, \boldsymbol{c}_m) = \text{LSTM}(\boldsymbol{x}_m, (\boldsymbol{h}_{m-1}, \boldsymbol{c}_{m-1}))$, with $(\boldsymbol{h}_m, \boldsymbol{c}_m)$ representing the LSTM state after reading token m.

The LSTM outperforms standard recurrent neural networks across a wide range of problems. It was first used for language modeling by Sundermeyer et al. (2012), but can be applied more generally: the vector $\boldsymbol{h}_m$ can be treated as a complete representation of

the input sequence up to position m, and can be used for any labeling task on a sequence of tokens, as we will see in the next chapter.

There are several LSTM variants, of which the Gated Recurrent Unit (Cho et al., 2014) is one of the more well known. Many software packages implement a variety of RNN architectures, so choosing between them is simple from a user's perspective. Jozefowicz et al. (2015) provide an empirical comparison of various modeling choices circa 2015.

6.4 Evaluating language models

Language modeling is not usually an application in itself: language models are typically components of larger systems, and they would ideally be evaluated **extrinisically**. This means evaluating whether the language model improves performance on the application task, such as machine translation or speech recognition. But this is often hard to do, and depends on details of the overall system which may be irrelevant to language modeling. In contrast, **intrinsic evaluation** is task-neutral. Better performance on intrinsic metrics may be expected to improve extrinsic metrics across a variety of tasks, but there is always the risk of over-optimizing the intrinsic metric. This section discusses some intrinsic metrics, but keep in mind the importance of performing extrinsic evaluations to ensure that intrinsic performance gains carry over to real applications.

6.4.1 Held-out likelihood

The goal of probabilistic language models is to accurately measure the probability of sequences of word tokens. Therefore, an intrinsic evaluation metric is the likelihood that the language model assigns to **held-out data**, which is not used during training. Specifically, we compute,

$$\ell(\boldsymbol{w}) = \sum_{m=1}^{M} \log \mathrm{p}(w_m \mid w_{m-1}, \ldots, w_1), \tag{6.41}$$

treating the entire held-out corpus as a single stream of tokens.

Typically, unknown words are mapped to the $\langle \text{UNK} \rangle$ token. This means that we have to estimate some probability for $\langle \text{UNK} \rangle$ on the training data. One way to do this is to fix the vocabulary $\mathcal{V}$ to the $V - 1$ words with the highest counts in the training data, and then convert all other tokens to $\langle \text{UNK} \rangle$. Other strategies for dealing with out-of-vocabulary terms are discussed in § 6.5.

6.4.2 Perplexity

Held-out likelihood is usually presented as **perplexity**, which is a deterministic transformation of the log-likelihood into an information-theoretic quantity,

$$\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{\ell(\boldsymbol{w})}{M}}, \quad [6.42]$$

where M is the total number of tokens in the held-out corpus.

Lower perplexities correspond to higher likelihoods, so lower scores are better on this metric — it is better to be less perplexed. Here are some special cases:

- In the limit of a perfect language model, probability 1 is assigned to the held-out corpus, with $\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{1}{M}\log_2 1} = 2^0 = 1$.
- In the opposite limit, probability zero is assigned to the held-out corpus, which corresponds to an infinite perplexity, $\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{1}{M}\log_2 0} = 2^\infty = \infty$.
- Assume a uniform, unigram model in which $\text{p}(w_i) = \frac{1}{V}$ for all words in the vocabulary. Then,

$$\begin{aligned}
\log_2(\boldsymbol{w}) &= \sum_{m=1}^{M} \log_2 \frac{1}{V} = -\sum_{m=1}^{M} \log_2 V = -M \log_2 V \\
\text{Perplex}(\boldsymbol{w}) &= 2^{\frac{1}{M} M \log_2 V} \\
&= 2^{\log_2 V} \\
&= V.
\end{aligned}$$

  This is the "worst reasonable case" scenario, since you could build such a language model without even looking at the data.

In practice, language models tend to give perplexities in the range between 1 and V. A small benchmark dataset is the **Penn Treebank**, which contains roughly a million tokens; its vocabulary is limited to 10,000 words, with all other tokens mapped a special $\langle \text{UNK} \rangle$ symbol. On this dataset, a well-smoothed 5-gram model achieves a perplexity of 141 (Mikolov and Zweig, Mikolov and Zweig), and an LSTM language model achieves perplexity of roughly 80 (Zaremba, Sutskever, and Vinyals, Zaremba et al.). Various enhancements to the LSTM architecture can bring the perplexity below 60 (Merity et al., 2018). A larger-scale language modeling dataset is the 1B Word Benchmark (Chelba et al., 2013), which contains text from Wikipedia. On this dataset, perplexities of around 25 can be obtained by averaging together multiple LSTM language models (Jozefowicz et al., 2016).

6.5 Out-of-vocabulary words

So far, we have assumed a **closed-vocabulary** setting — the vocabulary $\mathcal{V}$ is assumed to be a finite set. In realistic application scenarios, this assumption may not hold. Consider, for example, the problem of translating newspaper articles. The following sentence appeared in a Reuters article on January 6, 2017:[5]

> The report said U.S. intelligence agencies believe Russian military intelligence, the **GRU**, used intermediaries such as **WikiLeaks**, **DCLeaks.com** and the **Guccifer** 2.0 "persona" to release emails...

Suppose that you trained a language model on the Gigaword corpus,[6] which was released in 2003. The bolded terms either did not exist at this date, or were not widely known; they are unlikely to be in the vocabulary. The same problem can occur for a variety of other terms: new technologies, previously unknown individuals, new words (e.g., *hashtag*), and numbers.

One solution is to simply mark all such terms with a special token, $\langle \text{UNK} \rangle$. While training the language model, we decide in advance on the vocabulary (often the K most common terms), and mark all other terms in the training data as $\langle \text{UNK} \rangle$. If we do not want to determine the vocabulary size in advance, an alternative approach is to simply mark the first occurrence of each word type as $\langle \text{UNK} \rangle$.

But is often better to make distinctions about the likelihood of various unknown words. This is particularly important in languages that have rich morphological systems, with many inflections for each word. For example, Portuguese is only moderately complex from a morphological perspective, yet each verb has dozens of inflected forms (see Figure 4.3b). In such languages, there will be many word types that we do not encounter in a corpus, which are nonetheless predictable from the morphological rules of the language. To use a somewhat contrived English example, if *transfenestrate* is in the vocabulary, our language model should assign a non-zero probability to the past tense *transfenestrated*, even if it does not appear in the training data.

One way to accomplish this is to supplement word-level language models with **character-level language models**. Such models can use n-grams or RNNs, but with a fixed vocabulary equal to the set of ASCII or Unicode characters. For example, Ling et al. (2015) propose an LSTM model over characters, and Kim (2014) employ a convolutional neural network. A more linguistically motivated approach is to segment words into meaningful subword units, known as **morphemes** (see chapter 9). For example, Botha and Blunsom

[5] Bayoumy, Y. and Strobel, W. (2017, January 6). U.S. intel report: Putin directed cyber campaign to help Trump. *Reuters*. Retrieved from `http://www.reuters.com/article/us-usa-russia-cyber-idUSKBN14Q1T8` on January 7, 2017.

[6] `https://catalog.ldc.upenn.edu/LDC2003T05`

(2014) induce vector representations for morphemes, which they build into a log-bilinear language model; Bhatia et al. (2016) incorporate morpheme vectors into an LSTM.

Additional resources

A variety of neural network architectures have been applied to language modeling. Notable earlier non-recurrent architectures include the neural probabilistic language model (Bengio et al., 2003) and the log-bilinear language model (Mnih and Hinton, 2007). Much more detail on these models can be found in the text by Goodfellow et al. (2016).

Exercises

1. Prove that n-gram language models give valid probabilities if the n-gram probabilities are valid. Specifically, assume that,

$$\sum_{w_m}^{\mathcal{V}} \mathrm{p}(w_m \mid w_{m-1}, w_{m-2}, \ldots, w_{m-n+1}) = 1 \quad [6.43]$$

   for all contexts $(w_{m-1}, w_{m-2}, \ldots, w_{m-n+1})$. Prove that $\sum_{\boldsymbol{w}} \mathrm{p}_n(\boldsymbol{w}) = 1$ for all $\boldsymbol{w} \in \mathcal{V}^*$, where p_n is the probability under an n-gram language model. Your proof should proceed by induction. You should handle the start-of-string case $\mathrm{p}(w_1 \mid \underbrace{\square, \ldots, \square}_{n-1})$, but you need not handle the end-of-string token.

2. First, show that RNN language models are valid using a similar proof technique to the one in the previous problem.

   Next, let $\mathrm{p}_r(\boldsymbol{w})$ indicate the probability of $\boldsymbol{w}$ under RNN r. An ensemble of RNN language models computes the probability,

$$\mathrm{p}(\boldsymbol{w}) = \frac{1}{R} \sum_{r=1}^{R} \mathrm{p}_r(\boldsymbol{w}). \quad [6.44]$$

   Does an ensemble of RNN language models compute a valid probability?

3. Consider a unigram language model over a vocabulary of size V. Suppose that a word appears m times in a corpus with M tokens in total. With Lidstone smoothing of α, for what values of m is the smoothed probability greater than the unsmoothed probability?

4. Consider a simple language in which each token is drawn from the vocabulary $\mathcal{V}$ with probability $\frac{1}{V}$, independent of all other tokens.

Given a corpus of size M, what is the expectation of the fraction of all possible bigrams that have zero count? You may assume V is large enough that $\frac{1}{V} \approx \frac{1}{V-1}$.

5. Continuing the previous problem, determine the value of M such that the fraction of bigrams with zero count is at most $\epsilon \in (0, 1)$. As a hint, you may use the approximation $\ln(1 + \alpha) \approx \alpha$ for $\alpha \approx 0$.

6. In real languages, words probabilities are neither uniform nor independent. Assume that word probabilities are independent but not uniform, so that in general $\text{p}(w) \neq \frac{1}{V}$. Prove that the expected fraction of unseen bigrams will be higher than in the IID case.

7. Consider a recurrent neural network with a single hidden unit and a sigmoid activation, $h_m = \sigma(\theta h_{m-1} + x_m)$. Prove that if $|\theta| < 1$, then the gradient $\frac{\partial h_m}{\partial h_{m-k}}$ goes to zero as $k \to \infty$.[7]

8. **Zipf's law** states that if the word types in a corpus are sorted by frequency, then the frequency of the word at rank r is proportional to r^{-s}, where s is a free parameter, usually around 1. (Another way to view Zipf's law is that a plot of log frequency against log rank will be linear.) Solve for s using the counts of the first and second most frequent words, c_1 and c_2.

9. Download the wikitext-2 dataset.[8] Read in the training data and compute word counts. Estimate the Zipf's law coefficient by,

$$\hat{s} = \exp\left(\frac{(\log \boldsymbol{r}) \cdot (\log \boldsymbol{c})}{||\log \boldsymbol{r}||_2^2}\right), \quad [6.45]$$

where $\boldsymbol{r} = [1, 2, 3, \ldots]$ is the vector of ranks of all words in the corpus, and $\boldsymbol{c} = [c_1, c_2, c_3, \ldots]$ is the vector of counts of all words in the corpus, sorted in descending order.

Make a log-log plot of the observed counts, and the expected counts according to Zipf's law. The sum $\sum_{r=1}^{\infty} r^s = \zeta(s)$ is the Riemann zeta function, available in python's `scipy` library as `scipy.special.zeta`.

10. Using the Pytorch library, train an LSTM language model from the Wikitext training corpus. After each epoch of training, compute its perplexity on the Wikitext validation corpus. Stop training when the perplexity stops improving.

[7] This proof generalizes to vector hidden units by considering the largest eigenvector of the matrix $\boldsymbol{\Theta}$ (Pascanu et al., 2013).

[8] Available at `https://github.com/pytorch/examples/tree/master/word_language_model/data/wikitext-2` in September 2018. The dataset is already tokenized, and already replaces rare words with ⟨UNK⟩, so no preprocessing is necessary.