2 Word2Vec

1) Recall the loss function for GloVe:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{W}\sum_{j=1}^{W} f(P_{ij})(u_i^T v_j - \log P_{ij})^2$$

where P_{ij} (a scalar) is the probability that word j appears in the context of word i,
$f : \mathbb{R} \to \mathbb{R}$ is a function that gives a weight to each (i, j) pair based on its probability P_{ij},
u_i is a column vector of shape $(d \times 1)$ representing the output vector for word i, and
v_j is a column vector of shape $(d \times 1)$ representing the input vector for word j

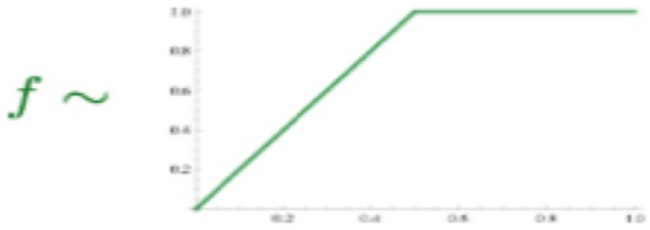
---

(i) (2 points) Calculate the gradient of the loss function with respect to input and output vectors: $\frac{\partial J(\theta)}{\partial u_i}$ and $\frac{\partial J(\theta)}{\partial v_j}$.

$\frac{\partial J(\theta)}{\partial u_i} = \sum_{j=1}^{W} f(P_{ij})(u_i^T v_j - logP_{ij})v_j$
$\frac{\partial J(\theta)}{\partial v_j} = \sum_{i=1}^{W} f(P_{ij})(u_i^T v_j - logP_{ij})u_i$

(ii) (2 points) Show what the function $f(x)$ looks like by drawing it.

Plot of $f(x)$:



(iii) (2 points) Explain in one or two sentences why should $f(x)$ have the form that you have drawn.

$f(x)$ should have the above form since we want that word pairs with higher frequency be given higher weightage but we clip it at a maximum value since we don't want most of the weightage given only to those few pairs which have very high frequency.

(iv) (2 points) Give one advantage of GloVe over Skipgram/CBOW models.

Fast training

2) (2 points) What are two ways practitioners deal with having two different sets of word vectors U and V at the end of training both Glove and word2vec?

① add them
② concatenate them.

Language Modeling

What is language modeling?

Language modeling (LM) analyzes bodies to text to provide a foundation for word prediction. These models use statistical and probabilistic techniques to determine the probability of a particular word sequence occurring in a sentence.

Language modeling is used in NLP techniques that generate written text as an output. Applications and programs with NLP-concepts rely on language models for tasks like audio-to-text conversion, sentiment analysis, speech recognition, and spelling corrections.

Language models work by determining word probabilities in an analyzed chunk of text data. Data is interpreted after being fed to a machine learning algorithm that looks for contextual rules in that given natural language (i.e. English, Japanese, Spanish).

The model then applies those rules to the input language tasks for generating predictions. It can even produce new sequences or sentences based on what it learned.

Language models are useful for both text classification and text generation. In text classification, we can use the language model's probability calculations to separate texts into different categories.

For example, if we trained a language model on spam email subject titles, the model would likely give the subject "CLICK HERE FOR FREE EASY MONEY" a relatively high probability of being spam.
In text generation, a language model completes a sentence by generating text based on the incomplete input sentence. This is the idea behind the autocomplete feature when texting on a phone or typing in a search engine. The model will give suggestions to complete the sentence based on the words it predicts with the highest probabilities.

Types of Language Models

There are two categories that Language Models fall under:

Statistical Language Models: These models use traditional statistical techniques like N-grams, Hidden Markov Models (HMM), and established linguistic rules to learn the probability distribution of words. Statistical Language Modeling involves the development of probabilistic models that can predict the next word in the sequence given the words that precede it.

Neural Language Models: These models are new players in the NLP world and have surpassed the statistical language models in their effectiveness. They use different kinds of Neural Networks to model language. The use of neural networks in the development of language models has become so popular that it is now the preferred approach for challenging tasks like speech recognition and machine translation.

Note: GPT-3 is an example of a Neural language model. BERT by Google is another popular Neural language model used in the algorithm of the search engine for next word prediction of our search query.

→ Let's say if a word doesn't exist in your test data then your probability will be 0. This can't be handled by Statistical modeling.

→ statistical language model

Limited Training Data: SLMs are trained on large amounts of text data. However, even with massive datasets, there will be many n-grams (sequences of n words) that rarely or never appear in the training data. This "out-of-vocabulary" (OOV) issue arises because the model hasn't encountered these specific word sequences before.

High-Dimensional Vectors: SLMs often represent words as high-dimensional vectors. While high dimensionality can capture complex relationships, it can also lead to sparsity in the vector space. Many dimensions might have very low values for most words, reducing the model's ability to differentiate words effectively.

→ only for vector representation

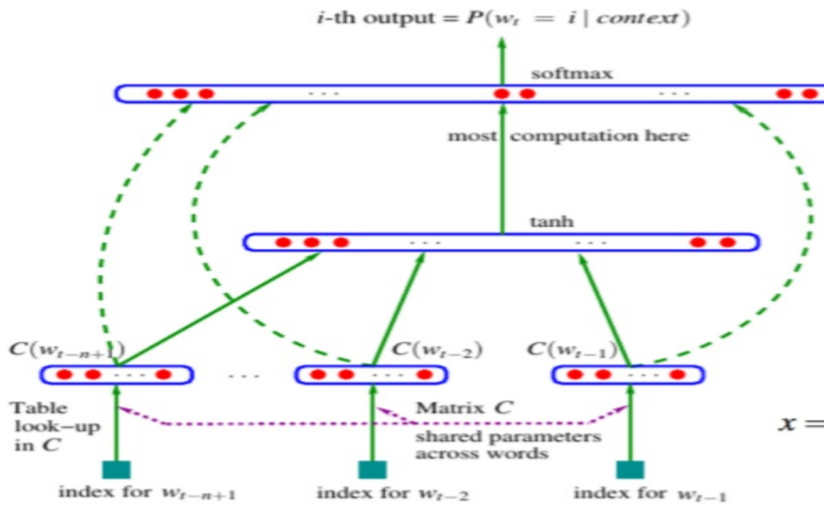
CBOW $P(w_t | w_{t-1}, w_{t+1}) = NN(w_{t-1}, w_{t+1})$

Skipgram $P(w_{t-1} | w_t) = NN(w_t)$

Neural LM $P(w_t | w_{t-1}, w_{t-2}) = NN(w_{t-1}, w_{t-2})$

→ this calculates above probability distribution & also learn word embeddings using a predictive model.

Neural Probabilistic Language Model



$$\hat{P}(w_t | w_{t-1}, \cdots w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}.$$

$$y = b + Wx + U \tanh(d + Hx)$$

$$x = (C(w_{t-1}), C(w_{t-2}), \cdots, C(w_{t-n+1})).$$

$$f(w_t, \cdots, w_{t-n+1}) = \hat{P}(w_t | w_1^{t-1}) \qquad f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$$

$$L = \frac{1}{T} \sum_t \log f(w_t, w_{t-1}, \cdots, w_{t-n+1}; \theta) + R(\theta), \qquad |V|(1 + nm + h) + h(1 + (n-1)m).$$

2. A Neural Model

The training set is a sequence $w_1 \cdots w_T$ of words $w_t \in V$, where the vocabulary V is a large but finite set. The objective is to learn a good model $f(w_t, \cdots, w_{t-n+1}) = \hat{P}(w_t | w_1^{t-1})$, in the sense that it gives high out-of-sample likelihood. Below, we report the geometric average of $1/\hat{P}(w_t | w_1^{t-1})$, also known as *perplexity*, which is also the exponential of the average negative log-likelihood. The only constraint on the model is that for any choice of w_1^{t-1}, $\sum_{i=1}^{|V|} f(i, w_{t-1}, \cdots, w_{t-n+1}) = 1$, with $f > 0$. By the product of these conditional probabilities, one obtains a model of the joint probability of sequences of words.

We decompose the function $f(w_t, \cdots, w_{t-n+1}) = \hat{P}(w_t | w_1^{t-1})$ in two parts:

1. A mapping C from any element i of V to a real vector $C(i) \in \mathbb{R}^m$. It represents the *distributed feature vectors* associated with each word in the vocabulary. In practice, C is represented by a $|V| \times m$ matrix of free parameters.

2. The probability function over words, expressed with C: a function g maps an input sequence of feature vectors for words in context, $(C(w_{t-n+1}), \cdots, C(w_{t-1}))$, to a conditional probability distribution over words in V for the next word w_t. The output of g is a vector whose i-th element estimates the probability $\hat{P}(w_t = i | w_1^{t-1})$ as in Figure 1.

$$f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$$

The function f is a composition of these two mappings (C and g), with C being *shared* across all the words in the context. With each of these two parts are associated some parameters. The parameters of the mapping C are simply the feature vectors themselves, represented by a $|V| \times m$ matrix C whose row i is the feature vector $C(i)$ for word i. The function g may be implemented by a feed-forward or recurrent neural network or another parametrized function, with parameters ω. The overall parameter set is $\theta = (C, \omega)$.

Training is achieved by looking for θ that maximizes the training corpus penalized log-likelihood:

$$L = \frac{1}{T} \sum_t \log f(w_t, w_{t-1}, \cdots, w_{t-n+1}; \theta) + R(\theta),$$

where $R(\theta)$ is a regularization term. For example, in our experiments, R is a weight decay penalty applied only to the weights of the neural network and to the C matrix, not to the biases.[3]

N-gram Language Models

The n-gram model is a probabilistic language model that can predict the next item in a sequence using the (n − 1)–order Markov model. Let's understand that better with an example. Consider the following sentence:

> "I love reading blogs on Educative to learn new concepts"

A 1-gram (or unigram) is a **one-word sequence**. For the above sentence, the unigrams would simply be: "I", "love", "reading", "blogs", "on", "Educative", "and", "learn", "new", "concepts".

A 2-gram (or bigram) is a **two-word sequence of words**, like "I love", "love reading", "on Educative" or "new concepts".

Lastly, a 3-gram (or trigram) is a **three-word sequence of words**, like "I love reading", "blogs on Educative", or "learn new concepts".

An N-gram language model predicts the probability of a given N-gram within any sequence of words in the language. If we have a good N-gram model, we can predict $p(w|h)$, or the probability of seeing the word w given a history of previous words h, where the history contains n-1 words.

> **Example:** "I love reading ___". Here, we want to predict what word will fill the dash based on the probabilities of the previous words.

We must estimate this probability to construct an N-gram model. We compute this probability in two steps:

1. Apply the chain rule of probability
2. We then apply a very strong simplification assumption to allow us to compute p(w1…ws) in an easy manner.

The chain rule of probability is:

$$p(w1...ws) = p(w1).p(w2|w1).p(w3|w1w2).p(w4|w1w2w3).....p(wn|w1...wn-1)$$

> **Definition:** What is the chain rule? It tells us how to compute the joint probability of a sequence by using the conditional probability of a word given previous words.

Here, we do not have access to these conditional probabilities with complex conditions of up to n-1 words. So, how do we proceed? This is where we introduce a **simplification assumption**. We can assume for all conditions, that:

$$p(wk|w1...wk-1) = p(wk|wk-1)$$

Here, we approximate the history (the context) of the word wk by looking only at the last word of the context. This assumption is called the Markov assumption. It is an example of the Bigram model. The same concept can be enhanced further for example for trigram model the formula will be:

$$p(wk|w1...wk-1) = p(wk|wk-2wk-1)$$

These models have a basic problem: they give the probability to zero if an unknown word is seen, so the concept of smoothing is used. In smoothing we assign some probability to the unseen words. There are different types of smoothing techniques such as Laplace smoothing, Good Turing, and Kneser-ney smoothing.

Perplexity Score

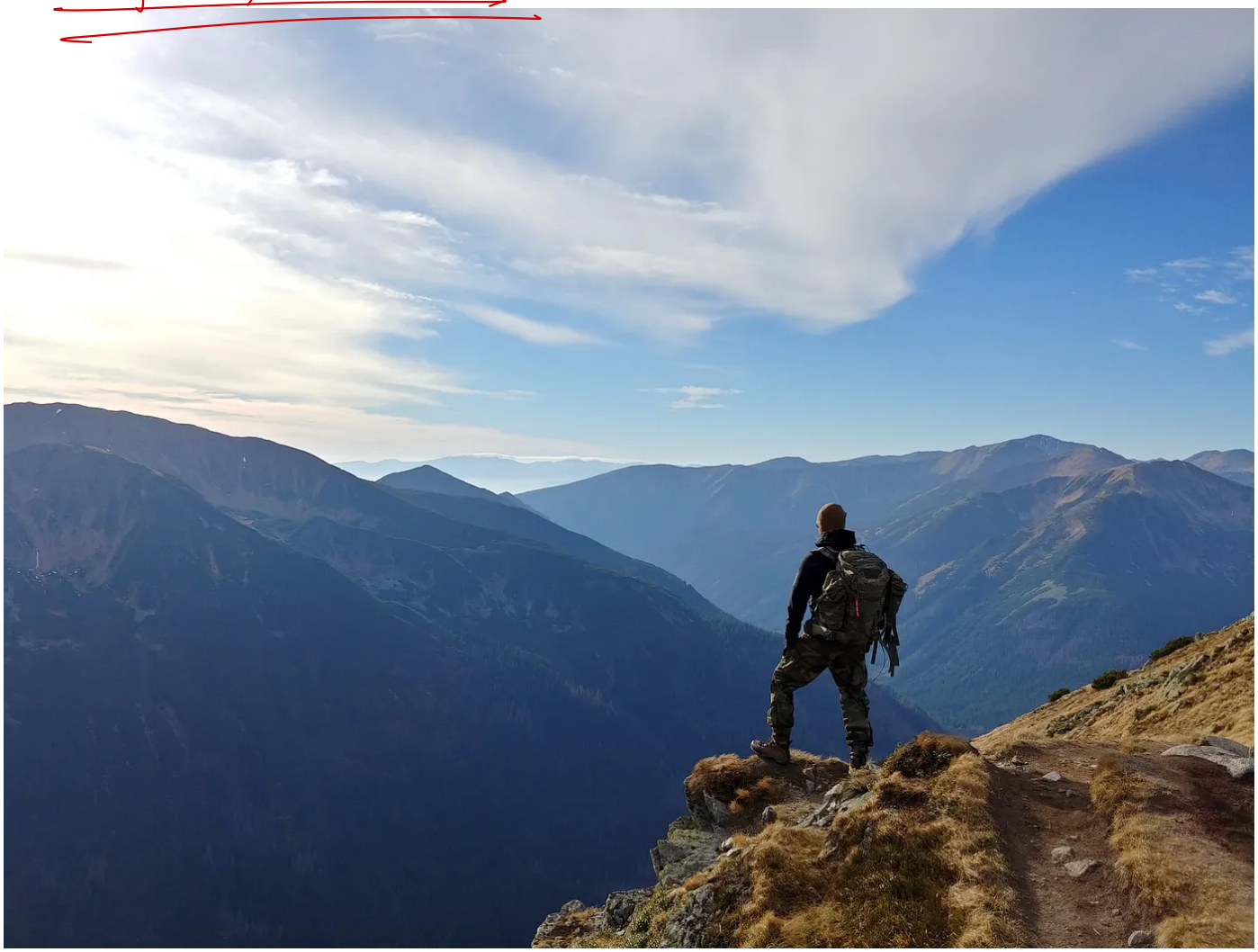


Photo by Wojciech Then on Unsplash

In general, perplexity is a measurement of how well a probability model predicts a sample. In the context of Natural Language Processing, perplexity is one way to evaluate language models.

A language model is a probability distribution over sentences: it's both able to generate plausible human-written sentences (if it's a good language model) and to evaluate the goodness of already written sentences. Presented with a well-written document, a good language model should be able to give it a higher probability than a badly written document, i.e. it should not be "perplexed" when presented with a well-written document.

Thus, the perplexity metric in NLP is a way to capture the degree of 'uncertainty' a model has in predicting (i.e. assigning probabilities to) text.

Now, let's try to compute the probabilities assigned by language models to some example sentences and derive an intuitive explanation of what perplexity is.

Computing perplexity from sentence probabilities

Suppose we have trained a small language model over an English corpus. The model is only able to predict the probability of the next word in the sentence from a small subset of six words: *"a"*, *"the"*, *"red"*, *"fox"*, *"dog"*, and *"."*.

Let's compute the probability of the sentence *W,* which is "*a red fox.*".

The probability of a generic sentence *W*, made of the words *w1*, *w2*, up to *wn*, can be expressed as the following:

$$P(W) = P(w1, w2, ..., wn)$$

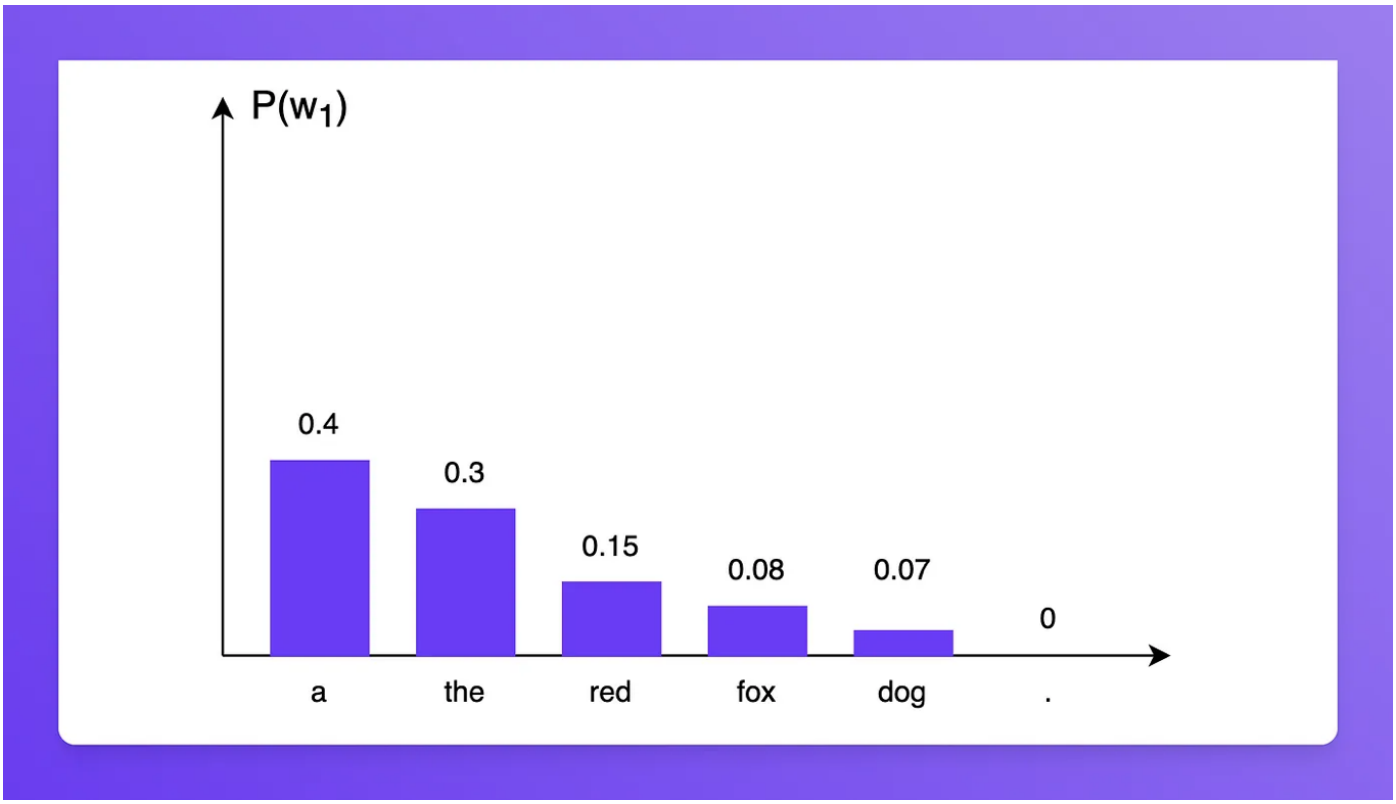
Using our specific sentence *W*, the probability can be extended as the following:

$$P(\text{"a red fox."}) =$$

$$P(\text{"a"}) * P(\text{"red"} \mid \text{"a"}) * P(\text{"fox"} \mid \text{"a red"}) * P(\text{"."} \mid \text{"a red}$$

fox”)

Suppose these are the probabilities assigned by our language model to a generic first word in a sentence:

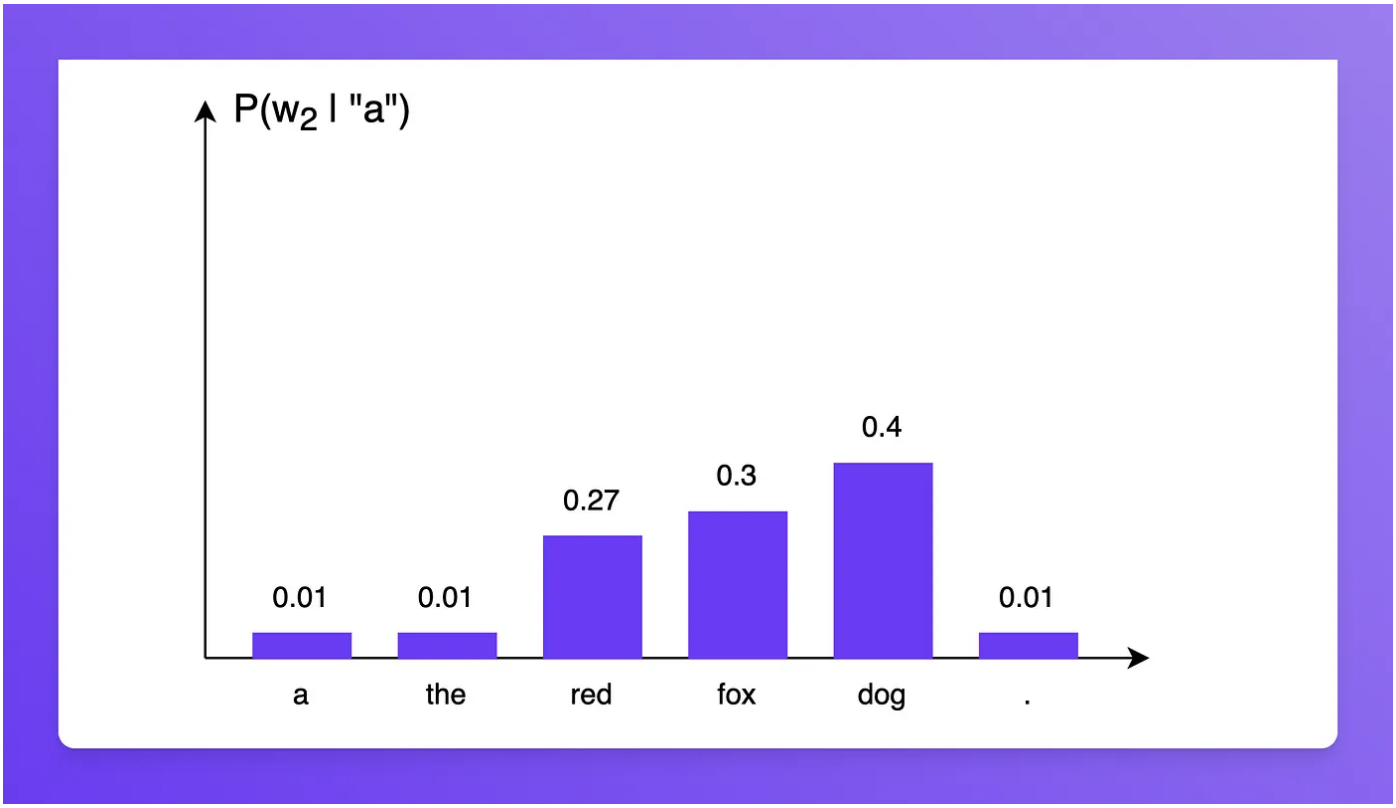


Probabilities assigned by a language model to a generic first word w1 in a sentence. Image by the author.

As can be seen from the chart, the probability of “*a*” as the first word of a sentence is:

P(“a”) = 0.4

Next, suppose these are the probabilities given by our language model to a generic second word that follows “*a*”:



Probabilities assigned by a language model to a generic second word w2 in a sentence. Image by the author.

The probability of "*red*" as the second word in the sentence after "*a*" is:

$$P(\text{“red”} \mid \text{“a”}) = 0.27$$

Similarly, these are the probabilities of the next words:

Finally, the probability assigned by our language model to the whole sentence "*a red fox.*" is:

$$P(\text{“a red fox.”}) =$$

$$P(\text{“a”}) * P(\text{“red”} \mid \text{“a”}) * P(\text{“fox”} \mid \text{“a red”}) * P(\text{“.”} \mid \text{“a red fox”})$$

$$= 0.4 * 0.27 * 0.55 * 0.79$$

$$= 0.0469$$

It would be nice to compare the probabilities assigned to different sentences to see which sentences are better predicted by the language model. However, since the probability of a sentence is obtained from a product of probabilities, the longer is the sentence the lower will be its probability (since it's a product of factors with values smaller than one). We should find a way of measuring these sentence probabilities, without the influence of the sentence length.

This can be done by normalizing the sentence probability by the number of words in the sentence. Since the probability of a sentence is obtained by multiplying many factors, we can average them using the geometric mean.

Let's call *Pnorm(W)* the normalized probability of the sentence *W*. Let *n* be the number of words in *W*. Then, applying the geometric mean:

$$Pnorm(W) = P(W) ^ (1 / n)$$

Using our specific sentence "*a red fox.*":

$$Pnorm(\text{“a red fox.”}) = P(\text{“a red fox”}) ^ (1 / 4) = 0.465$$

Great! This number can now be used to compare the probabilities of sentences with different lengths. The higher this number is over a well-written sentence, the better is the language model.

So, what does this have to do with perplexity? Well, perplexity is just the reciprocal of this number.

Let's call *PP(W)* the perplexity computed over the sentence *W*. Then:

$$PP(W) = 1 / Pnorm(W)$$

$$= 1 / (P(W) ^ (1 / n))$$

$$= (1 / P(W)) ^ (1 / n)$$

Which is the formula of perplexity. Since perplexity is just the reciprocal of the normalized probability, the lower the perplexity over a well-written sentence the better is the language model.

Let's try computing the perplexity with a second language model that

assigns equal probability to each word at each prediction. Since the language models can predict six words only, the probability of each word will be 1/6.

$$P(\text{“a red fox.”}) = (1/6) \hat{} 4 = 0.00077$$

$$Pnorm(\text{“a red fox.”}) = P(\text{“a red fox.”}) \hat{} (1/4) = 1/6$$

$$PP(\text{“a red fox”}) = 1 / Pnorm(\text{“a red fox.”}) = 6$$

…which, as expected, is a higher perplexity than the one produced by the well-trained language model.

Perplexity and Entropy

Perplexity can be computed also starting from the concept of Shannon entropy. Let’s call *H(W)* the entropy of the language model when predicting a sentence *W*. Then, it turns out that:

$$PP(W) = 2 \hat{} (H(W))$$

This means that, when we optimize our language model, the following sentences are all more or less equivalent:

- We are maximizing the normalized sentence probabilities given by the language model over well-written sentences.
- We are minimizing the perplexity of the language model over well-

3.1 N-Grams

Let's begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is "*its water is so transparent that*" and we want to know the probability that the next word is *the*:

$$P(\textit{the}|\textit{its water is so transparent that}). \tag{3.1}$$

One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see *its water is so transparent that*, and count the number of times this is followed by *the*. This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(\textit{the}|\textit{its water is so transparent that}) = \frac{C(\textit{its water is so transparent that the})}{C(\textit{its water is so transparent that})} \tag{3.2}$$

With a large enough corpus, such as the web, we can compute these counts and estimate the probability from Eq. 3.2. You should pause now, go to the web, and compute this estimate for yourself.

While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions

of the example sentence may have counts of zero on the web (such as "*Walden Pond's water is so transparent that the*"; well, *used to* have counts of zero).

Similarly, if we wanted to know the joint probability of an entire sequence of words like *its water is so transparent*, we could do it by asking "out of all possible sequences of five words, how many of them are *its water is so transparent*?" We would have to get the count of *its water is so transparent* and divide by the sum of the counts of all possible five word sequences. That seems rather a lot to estimate!

For this reason, we'll need to introduce more clever ways of estimating the probability of a word w given a history h, or the probability of an entire word sequence W. Let's start with a little formalizing of notation. To represent the probability of a particular random variable X_i taking on the value "the", or $P(X_i = \text{"the"})$, we will use the simplification $P(the)$. We'll represent a sequence of n words either as $w_1 \dots w_n$ or $w_{1:n}$ (so the expression $w_{1:n-1}$ means the string $w_1, w_2, \dots, w_{n-1}$). For the joint probability of each word in a sequence having a particular value $P(X_1 = w_1, X_2 = w_2, X_3 = w_3, \dots, X_n = w_n)$ we'll use $P(w_1, w_2, \dots, w_n)$.

Now, how can we compute probabilities of entire sequences like $P(w_1, w_2, \dots, w_n)$? One thing we can do is decompose this probability using the **chain rule of probability**:

$$\begin{aligned} P(X_1 \dots X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_{1:2}) \dots P(X_n|X_{1:n-1}) \\ &= \prod_{k=1}^{n} P(X_k|X_{1:k-1}) \end{aligned} \tag{3.3}$$

Applying the chain rule to words, we get

$$\begin{aligned} P(w_{1:n}) &= P(w_1)P(w_2|w_1)P(w_3|w_{1:2}) \dots P(w_n|w_{1:n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_{1:k-1}) \end{aligned} \tag{3.4}$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. Equation 3.4 suggests that we could estimate the joint probability of an entire sequence of words by multiplying together a number of conditional probabilities. But using the chain rule doesn't really seem to help us! We don't know any way to compute the exact probability of a word given a long sequence of preceding words, $P(w_n|w_{1:n-1})$. As we said above, we can't just estimate by counting the number of times every word occurs following every long string, because language is creative and any particular context might have never occurred before!

The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can **approximate** the history by just the last few words.

bigram

The **bigram** model, for example, approximates the probability of a word given all the previous words $P(w_n|w_{1:n-1})$ by using only the conditional probability of the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

$$P(\text{the}|\text{Walden Pond's water is so transparent that}) \tag{3.5}$$

we approximate it with the probability

$$P(\text{the}|\text{that}) \tag{3.6}$$

When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1}) \tag{3.7}$$

Markov

The assumption that the probability of a word depends only on the previous word is called a **Markov** assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the **n-gram** (which looks $n-1$ words into the past).

n-gram

Let's see a general equation for this n-gram approximation to the conditional probability of the next word in a sequence. We'll use N here to mean the n-gram size, so $N = 2$ means bigrams and $N = 3$ means trigrams. Then we approximate the probability of a word given its entire context as follows:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) \tag{3.8}$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by substituting Eq. 3.7 into Eq. 3.4:

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{3.9}$$

maximum likelihood estimation

normalize

How do we estimate these bigram or n-gram probabilities? An intuitive way to estimate probabilities is called **maximum likelihood estimation** or **MLE**. We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and **normalizing** the counts so that they lie between 0 and 1.[1]

For example, to compute a particular bigram probability of a word w_n given a previous word w_{n-1}, we'll compute the count of the bigram $C(w_{n-1}w_n)$ and normalize by the sum of all the bigrams that share the same first word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \tag{3.10}$$

We can simplify this equation, since the sum of all bigram counts that start with a given word w_{n-1} must be equal to the unigram count for that word w_{n-1} (the reader should take a moment to be convinced of this):

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.11}$$

Let's work through an example using a mini-corpus of three sentences. We'll first need to augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word. We'll also need a special end-symbol. </s>[2]

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>
```

[1] For probabilistic models, normalizing means dividing by some total count so that the resulting probabilities fall between 0 and 1.

[2] We need the end-symbol to make the bigram grammar a true probability distribution. Without an end-symbol, instead of the sentence probabilities of all sentences summing to one, the sentence probabilities for all sentences *of a given length* would sum to one. This model would define an infinite set of probability distributions, with one distribution per sentence length. See Exercise 3.5.

Here are the calculations for some of the bigram probabilities from this corpus

$P(\texttt{I}|\texttt{<s>}) = \frac{2}{3} = .67$ $P(\texttt{Sam}|\texttt{<s>}) = \frac{1}{3} = .33$ $P(\texttt{am}|\texttt{I}) = \frac{2}{3} = .67$

$P(\texttt{</s>}|\texttt{Sam}) = \frac{1}{2} = 0.5$ $P(\texttt{Sam}|\texttt{am}) = \frac{1}{2} = .5$ $P(\texttt{do}|\texttt{I}) = \frac{1}{3} = .33$

For the general case of MLE n-gram parameter estimation:

$$P(w_n|w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1}\ w_n)}{C(w_{n-N+1:n-1})} \tag{3.12}$$

relative frequency

Equation 3.12 (like Eq. 3.11) estimates the n-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a **relative frequency**. We said above that this use of relative frequencies as a way to estimate probabilities is an example of maximum likelihood estimation or MLE. In MLE, the resulting parameter set maximizes the likelihood of the training set T given the model M (i.e., $P(T|M)$). For example, suppose the word *Chinese* occurs 400 times in a corpus of a million words like the Brown corpus. What is the probability that a random word selected from some other text of, say, a million words will be the word *Chinese*? The MLE of its probability is $\frac{400}{1000000}$ or .0004. Now .0004 is not the best possible estimate of the probability of *Chinese* occurring in all situations; it might turn out that in some other corpus or context *Chinese* is a very unlikely word. But it is the probability that makes it *most likely* that Chinese will occur 400 times in a million-word corpus. We present ways to modify the MLE estimates slightly to get better probability estimates in Section 3.5.

Matching genres and dialects is still not sufficient. Our models may still be subject to the problem of **sparsity**. For any n-gram that occurred a sufficient number of times, we might have a good estimate of its probability. But because any corpus is limited, some perfectly acceptable English word sequences are bound to be missing from it. That is, we'll have many cases of putative "zero probability n-grams" that should really have some non-zero probability. Consider the words that follow the bigram *denied the* in the WSJ Treebank3 corpus, together with their counts:

denied the allegations:	5
denied the speculation:	2
denied the rumors:	1
denied the report:	1

But suppose our test set has phrases like:

denied the offer
denied the loan

Our model will incorrectly estimate that the $P(\text{offer}|\text{denied the})$ is 0!

These **zeros**—things that don't ever occur in the training set but do occur in the test set—are a problem for two reasons. First, their presence means we are underestimating the probability of all sorts of words that might occur, which will hurt the performance of any application we want to run on this data.

Second, if the probability of any word in the test set is 0, the entire probability of the test set is 0. By definition, perplexity is based on the inverse probability of the

given 3 sentences

"JOHN READ MOBY DICK"

"MARY READ A DIFFERENT BOOK"

"SHE READ A BOOK BY CHER"

Let's Calculate

$$P(\text{JOHN READ A BOOK})$$

$$P(\text{JOHN} \mid \langle BOS \rangle) = \frac{c(\langle BOS \rangle \text{ JOHN})}{\sum_w c(\langle BOS \rangle\ w)} = \frac{1}{3}$$

$$P(\text{READ} \mid \text{JOHN}) = \frac{c(\text{JOHN READ})}{\sum_w c(\text{JOHN } w)} = \frac{1}{1}$$

$$P(A \mid \text{READ}) = \frac{c(\text{READ } A)}{\sum_w c(\text{READ } w)} = \frac{2}{3}$$

$$P(\text{BOOK} \mid A) = \frac{c(A \text{ BOOK})}{\sum_w c(A\ w)} = \frac{1}{2}$$

$$P(\langle EOS \rangle \mid \text{BOOK}) = \frac{c(\text{BOOK} \langle EOS \rangle)}{\sum_w c(\text{BOOK } w)} = \frac{1}{2}$$

$$P(\text{JOHN READ A BOOK}) = P(\text{JOHN} \mid \langle BOS \rangle)\, P(\text{READ} \mid \text{JOHN})\, P(A \mid \text{READ})\, P(\text{BOOK} \mid A)\, P(\langle EOS \rangle \mid \text{BOOK})$$

$$= \frac{1}{3} \times 1 \times \frac{2}{3} \times \frac{1}{2} \times \frac{1}{2} \approx 0.06$$

1.2 Smoothing

Now, consider the sentence CHER READ A BOOK. We have

$$p(\text{READ} \mid \text{CHER}) = \frac{c(\text{CHER READ})}{\sum_w c(\text{CHER } w)} = \frac{0}{1}$$

giving us $p(\text{CHER READ A BOOK}) = 0$. Obviously, this is an underestimate for the probability of CHER READ A BOOK as there is *some* probability that the sentence occurs. To show why it is important that this probability should be given a nonzero value, we turn to the primary application for language models, *speech recognition*. In speech recognition, one attempts to find the sentence s that maximizes $p(s|A) = \frac{p(A|s)p(s)}{p(A)}$ for a given acoustic signal A. If $p(s)$ is zero, then $p(s|A)$ will be zero and the string s will never be considered as a transcription, regardless of how unambiguous the acoustic signal is. Thus, whenever a string s such that $p(s) = 0$ occurs during a speech recognition task, an error will be made. Assigning all strings a nonzero probability helps prevent errors in speech recognition.

Smoothing is used to address this problem. The term *smoothing* describes techniques for adjusting the maximum likelihood estimate of probabilities (as in equations (2) and (4)) to produce more accurate probabilities. The name *smoothing* comes from the fact that these techniques tend to make distributions more uniform, by adjusting low probabilities such as zero probabilities upward, and high probabilities downward. Not only do smoothing methods generally prevent zero probabilities, but they also attempt to improve the accuracy of the model as a whole. Whenever a probability is estimated from few counts, smoothing has the potential to significantly improve estimation.

To give an example, one simple smoothing technique is to pretend each bigram occurs once more than it actually does (Lidstone, 1920; Johnson, 1932; Jeffreys, 1948), yielding

$$p(w_i|w_{i-1}) = \frac{1 + c(w_{i-1}w_i)}{\sum_{w_i}[1 + c(w_{i-1}w_i)]} = \frac{1 + c(w_{i-1}w_i)}{|V| + \sum_{w_i} c(w_{i-1}w_i)} \quad (5)$$

Smoothing

What do we do with words that are in our vocabulary (they are not unknown words) but appear in a test set in an unseen context (for example they appear after a word they never appeared after in training)? To keep a language model from assigning zero probability to these unseen events, we'll have to shave off a bit of probability mass from some more frequent events and give it to the events we've never seen. This modification is called smoothing or discounting. In this section and the fol- lowing ones we'll introduce a variety of ways to do smoothing: Laplace (add-one) smoothing, add-k smoothing, stupid backoff, and Kneser-Ney smoothing.

Laplace Smoothing!

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.23}$$

For add-one smoothed bigram counts, we need to augment the unigram count by the number of total word types in the vocabulary V:

$$P_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{\sum_w (C(w_{n-1}w)+1)} = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V} \tag{3.24}$$

Example: In the Berkeley Restaurant Project, we have the following counts:

$$C(\texttt{want}) = 927$$
$$C(\texttt{want to}) = 608$$
$$C(\texttt{want want}) = 0$$
$$|\mathcal{V}| = 1446$$

Estimate the probabilities for $P(\texttt{to}|\texttt{want})$ and $P(\texttt{want}|\texttt{want})$ with and without add-one smoothing.

$$P_{\text{MLE}}(\texttt{to}|\texttt{want}) = \frac{608}{927} = 0.6559$$
$$P_{\text{MLE}}(\texttt{want}|\texttt{want}) = \frac{0}{927} = 0$$
$$P_{+1}(\texttt{to}|\texttt{want}) = \frac{608+1}{927+1446} = 0.2566$$
$$P_{+1}(\texttt{want}|\texttt{want}) = \frac{1}{927+1446} = 0.0004$$

Problem: We often steal way too much! This is because the $|\mathcal{V}|$ in the denominator can completely overpower $C(w_{t-1})$. In the example above, we had a probability go from 0.6559 to 0.2566!

Disadvantage of add-one smoothing is

→ takes away too much probability mass from seen events.

→ Assign too much total probability mass to unseen events.

3.5.2 Add-k smoothing

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events. Instead of adding 1 to each count, we add a fractional count k (.5? .05? .01?). This algorithm is therefore called **add-k smoothing**.

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+k}{C(w_{n-1})+kV} \tag{3.26}$$

Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a **devset**. Although add-k is useful for some tasks (including text classification), it turns out that it still doesn't work well for language modeling, generating counts with poor variances and often inappropriate discounts (Gale and Church, 1994).

3.5.3 Backoff and Interpolation

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on. If we are trying to compute $P(w_n|w_{n-2}w_{n-1})$ but we have no examples of a particular trigram $w_{n-2}w_{n-1}w_n$, we can instead estimate its probability by using the bigram probability $P(w_n|w_{n-1})$. Similarly, if we don't have counts to compute $P(w_n|w_{n-1})$, we can look to the unigram $P(w_n)$.

In other words, sometimes using **less context** is a good thing, helping to generalize more for contexts that the model hasn't learned much about. There are two ways to use this n-gram "hierarchy". In **backoff**, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order n-gram. By contrast, in **interpolation**, we always mix the probability estimates from all the n-gram estimators, weighting and combining the trigram, bigram, and unigram counts.

In simple linear interpolation, we combine different order n-grams by linearly interpolating them. Thus, we estimate the trigram probability $P(w_n|w_{n-2}w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a

λ:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1 P(w_n) \\ &\quad +\lambda_2 P(w_n|w_{n-1}) \\ &\quad +\lambda_3 P(w_n|w_{n-2}w_{n-1})\end{aligned} \tag{3.27}$$

The λs must sum to 1, making Eq. 3.27 equivalent to a weighted average:

$$\sum_i \lambda_i = 1 \tag{3.28}$$

Back-off

In back-off we use the highest-order model if the count is not zero, otherwise we back off to a lower-order model.

We need to discount the higher-order models in order to spread mass to the lower-order models. And then we need to make sure the probabilities sum to 1.

Back-off N-gram model:

$$P_{\text{BO}}(w_t|w_{t-N+1:t-1}) = \begin{cases} P_d(w_t|w_{t-N+1:t-1}) & \text{if } C(w_{t-N+1:t}) > 0 \\ \alpha(w_{t-N+1:t}) P_{\text{BO}}(w_t|w_{t-N+2:t-1}) & \text{if } C(w_{t-N+1:t}) = 0 \end{cases}$$

where

- $P_d(w_t|w_{t-N+1:t-1})$ is some discounted N-gram model.
- The back-off weights $\alpha(w_{t-N+1:t})$ are such that the probability sum to 1.

Good Turing smoothing:

<s> I am here </s>
<s> who am I </s>
<s> I would like </s>

Computing N_c

I	3	frequency of frequency
am	2	$N_1 = 4$
here	1	$N_2 = 1$
who	1	$N_3 = 1$
would	1	
like	1	

Idea:

→ Reallocating the probability mass of n-gram that occurs r+1 times in the training data to the n-grams that occurs r times.

→ In particular, reallocate the probability mass of n-grams that were seen once to the n-grams that were never seen.

for each count c, an adjusted count c* is

$$c^* = \frac{(c+1)\, N_{c+1}}{N_c}$$

N_c = the number of n-grams exactly seen c times.

distribute probability mass

N_c → No. of words occurs c times

N_{c+1} → No of words occurs c+1 times

$$c^* = \frac{(c+1)\, N_{c+1}}{N}$$

P^*_{GT} (things with frequency c) = $\frac{c^*}{N}$

distributing this probability mass that occurs c times.

If there are N_c words.

so, each of them will get

$$\frac{(c+1)\, N_{c+1}}{N N_c}$$

effective count

$$\Rightarrow \frac{c^*}{N} = \frac{(c+1)\, N_{c+1}}{N_c}$$

if c = 0 that means it will distribute the probability using the probabilities that occurs 1.

P^*_{GT} (things with frequency c) = $\frac{N_1}{N}$ where

N denotes the total no of bigrams that actually occurs in train data.

3. (a) In the table below you are given the bag of words in 4 documents and the label of the document (spam or not spam).

Document	Bag of words	Label
1	{price, weight, loss, vitamin, discount}	spam
2	{vitamin, weight, discount, sad}	spam
3	{loss, sad, sorry}	not spam
4	{weight, foundation, price}	not spam

i. Calculate the parameters of the Naive-Bayes model from the data given (use Laplace - that is add 1 - smoothing for zeroes).

Solution:

The Naive Bayes parameters are: $P(spam)$, $P(non-spam)$ and $P(w_i|c)$ for each $w_i \in \mathcal{V}$ and $c \in \{\text{spam, non-spam}\}$. These probabilities have to be estimated from the learning set $\mathcal{L}$. We have $|\mathcal{L}| = 4$ where 2 documents each are spam and non-spam. So, $P(spam) = P(non-spam) = \frac{1}{2}$. To calculate class conditional term probabilities for each $w_i \in \mathcal{V}$ conditioned on the class label we use MLE estimates. We must handle the case when a particular term does not occur at all in a particular class leading to a conditional probability of 0 which will further lead to a 0 probability when we try to predict the label for a test document that contains that term.

So, let us assume that we start with a uniform prior ϵ over $\mathcal{V}$ for each class. Then we count occurrences for each term in $\mathcal{V}$ conditioned on the class and add it to the prior. The table below shows this for the given data:

BoW index	discount	foundation	loss	price	sad	sorry	vitamin	weight
Prior	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
Spam	$2+\epsilon$	ϵ	$1+\epsilon$	$1+\epsilon$	$1+\epsilon$	ϵ	$2+\epsilon$	$2+\epsilon$
Non-spam	ϵ	$1+\epsilon$	$1+\epsilon$	$1+\epsilon$	$1+\epsilon$	$1+\epsilon$	ϵ	$1+\epsilon$

The class conditional probabilties in each case are obtained by dividing each value by the sum of the values in each row for that class - that is:

$9+8\epsilon$ for spam and $6+8\epsilon$ for non-spam. One can choose any value for ϵ. If ϵ is 1 we have Laplace or add 1 smoothing. So, using $\epsilon = 1$ the class conditional probabilities are:

BoW index	discount	foundation	loss	price	sad	sorry	vitamin	weight
Spam	$\frac{3}{17}$	$\frac{1}{17}$	$\frac{2}{17}$	$\frac{2}{17}$	$\frac{2}{17}$	$\frac{1}{17}$	$\frac{3}{17}$	$\frac{3}{17}$
Non-spam	$\frac{1}{14}$	$\frac{2}{14}$	$\frac{2}{14}$	$\frac{2}{14}$	$\frac{2}{14}$	$\frac{2}{14}$	$\frac{1}{14}$	$\frac{2}{14}$

ii. What are the probabilities for the labels *spam* and *non spam* for the document with the following bag of words:

{`weight, price, discount, foundation, loss`}?

Solution:

Let d be the test document then:

$$P(spam|d) = \frac{1}{2} \times \frac{3}{17} \times \frac{2}{17} \times \frac{3}{17} \times \frac{1}{17} \times \frac{2}{17} = \frac{36}{17^5}$$

$$P(non\text{-}spam|d) = \frac{1}{2} \times \frac{2}{14} \times \frac{2}{14} \times \frac{1}{14} \times \frac{2}{14} \times \frac{2}{14} = \frac{16}{14^5}$$

what is the final label for the document?

→ $P(non\text{-}spam|d)$ is highest.

RNN

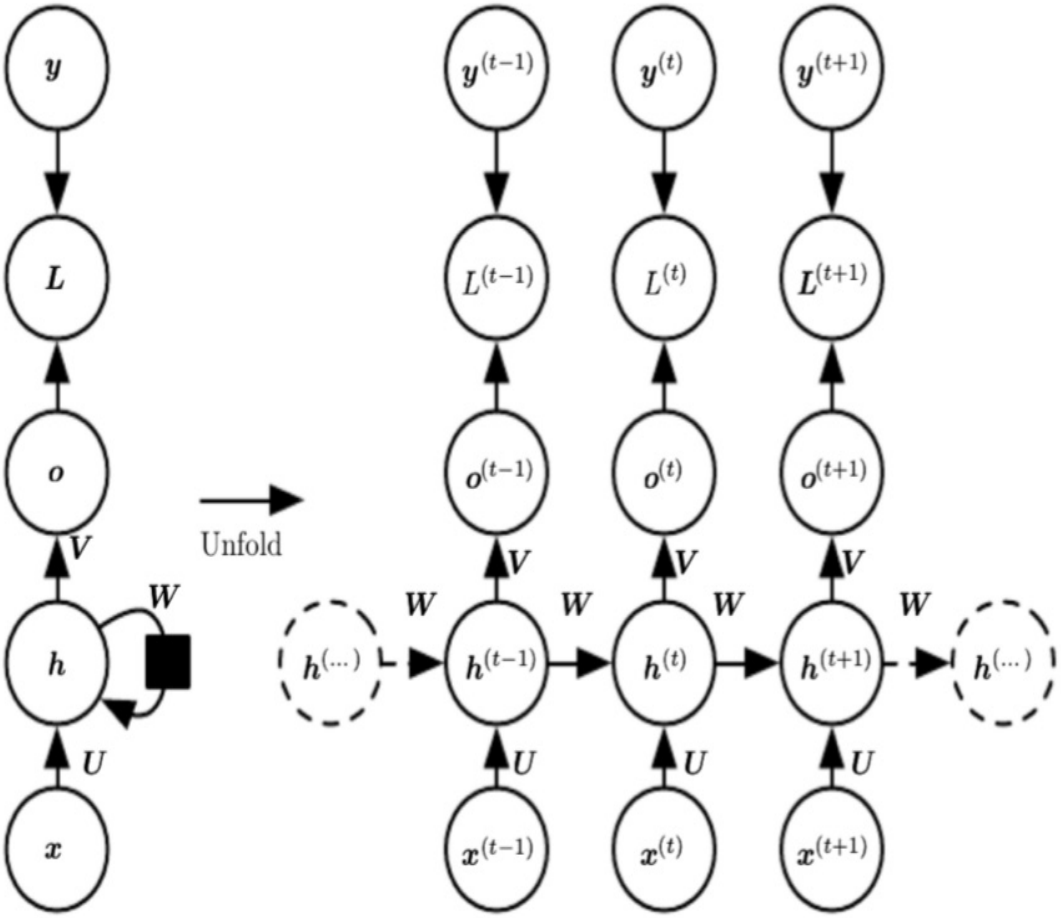
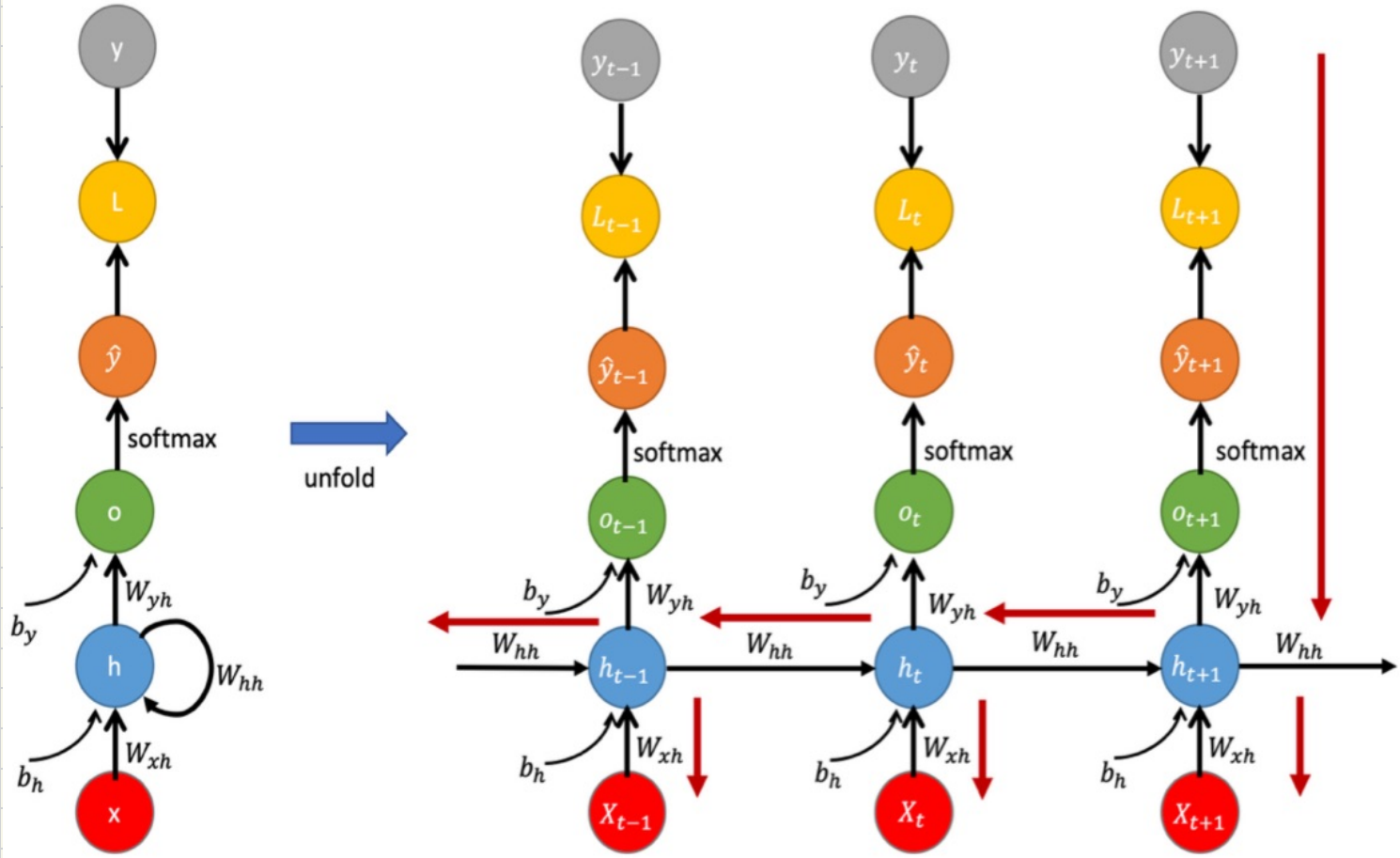


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. A loss L measures how far each $\boldsymbol{o}$ is from the corresponding training target $\boldsymbol{y}$. When using softmax outputs, we assume $\boldsymbol{o}$ is the unnormalized log probabilities. The loss L internally computes $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ and compares this to the target $\boldsymbol{y}$. The RNN has input to hidden connections parametrized by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parametrized by a weight matrix $\boldsymbol{W}$, and hidden-to-output connections parametrized by a weight matrix $\boldsymbol{V}$. Equation 10.8 defines forward propagation in this model. *(Left)* The RNN and its loss drawn with recurrent connections. *(Right)* The same seen as a time-unfolded computational graph, where each node is now associated with one particular time instance.



Just like for feedforward neural networks, we can compute a recurrent layer's output in one shot for a whole mini-batch by placing all the inputs at time step t in an input matrix X_t:

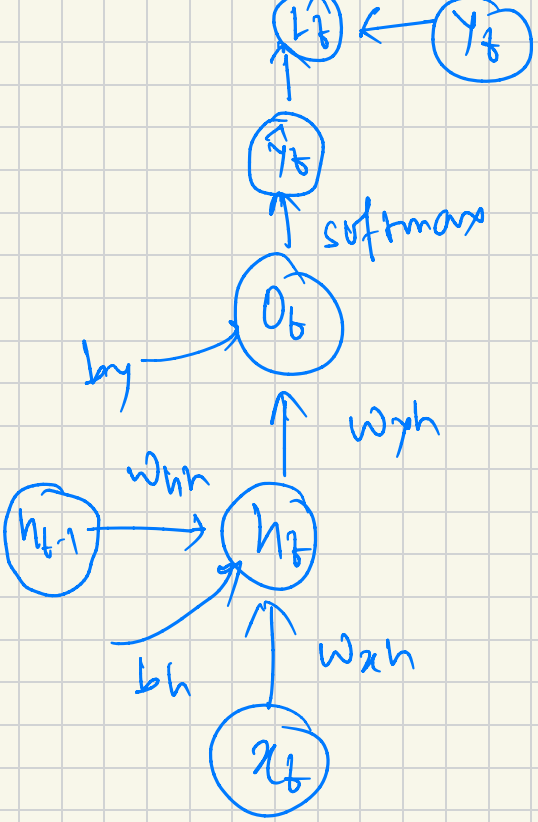
$$
\begin{aligned}
h_t &= tanh(X_t \cdot W_{xh} + h_{t-1} \cdot W_{hh} + b_h) \\
&= \phi_h([X_t h_{t-1}] \cdot W + b_h) \\
o_t &= h_t \cdot W_{yh} + b_y \\
\hat{y}_t &= softmax(o_t)
\end{aligned}
$$

1. The weight matrices W_{xh} and W_{yh} are often concatenated vertically into a single weight matrix W of shape $(n_{inputs} + n_{neurons}) \times n_{neurons}$.
2. The notation $[X_t h_{t-1}]$ represents the horizontal concatenation of the matrices X_t and h_{t-1}, shape of $m \times (n_{inputs} + n_{neurons})$

Let's denote m as the number of instances in the mini-batch, $n_{neurons}$ as the number of neurons, and n_{inputs} as the number of input features.

1. X_t is an $m \times n_{inputs}$ matrix containing the inputs for all instances.
2. h_{t-1} is an $m \times n_{neurons}$ matrix containing the hidden state of the previous time-step for all instances.
3. W_{xh} is an $n_{inputs} \times n_{neurons}$ matrix containing the connection weights between input and the hidden layer.
4. W_{hh} is an $n_{neurons} \times n_{neurons}$ matrix containing the connection weights between two hidden layers.
5. W_{yh} is an $n_{neurons} \times n_{neurons}$ matrix containing the connection weights between the hidden layer and the output.
6. b_h is a vector of size $n_{neurons}$ containing each neuron's bias term.
7. b_y is a vector of size $n_{neurons}$ containing each output's bias term.
8. y_t is an $m \times n_{neurons}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch

NOTE: At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.



Let's assume we pick a vocabulary = 8000

$x_t \in \mathbb{R}^{8000}$ $o_t \in \mathbb{R}^{8000}$ $h_t \in \mathbb{R}^{100}$

$W_{xh} \in \mathbb{R}^{100 \times 8000}$

$W_{hh} \in \mathbb{R}^{100 \times 100}$

$W_{yh} \in \mathbb{R}^{8000 \times 100}$

total parameter we need to learn

$= XH + YH + H^2$

$= 8000 \times 100 + 8000 \times 100 + (100)^2$

$= 1610000$

So, the dimensions also tell us the bottleneck of our model. As x_t is one-hot vector, multiplying it with W_{xh} is same as selecting a column of W_{xh}, so we don't need to perform the full multiplication. Then the biggest multiplication in our network is

Why. That's why we keep our vocabulary size small if possible.

Training RNN

9.1.2 Training (Why Backpropagation can't be used)

As with feedforward networks, we'll use a training set, a loss function, and backpropagation to obtain the gradients needed to adjust the weights in these recurrent networks. As shown in Fig. 9.2, we now have 3 sets of weights to update: **W**, the weights from the input layer to the hidden layer, **U**, the weights from the previous hidden layer to the current hidden layer, and finally **V**, the weights from the hidden layer to the output layer.

Fig. 9.4 highlights two considerations that we didn't have to worry about with backpropagation in feedforward networks. First, to compute the loss function for the output at time t we need the hidden layer from time $t-1$. Second, the hidden layer at time t influences both the output at time t and the hidden layer at time $t+1$ (and hence the output and loss at $t+1$). It follows from this that to assess the error accruing to $\mathbf{h}_t$, we'll need to know its influence on both the current output *as well as the ones that follow*.

Tailoring the backpropagation algorithm to this situation leads to a two-pass algorithm for training the weights in RNNs. In the first pass, we perform forward inference, computing $\mathbf{h}_t$, $\mathbf{y}_t$, accumulating the loss at each step in time, saving the value of the hidden layer at each step for use at the next time step. In the second phase, we process the sequence in reverse, computing the required gradients as we go, computing and saving the error term for use in the hidden layer for each step backward in time. This general approach is commonly referred to as **backpropagation through time** (Werbos 1974, Rumelhart et al. 1986, Werbos 1990).

We need to calculate 5 derivatives w.r.t loss

$\frac{\partial L}{\partial W_{yh}}$, $\frac{\partial L}{\partial W_{hh}}$, $\frac{\partial L}{\partial W_{xh}}$, $\frac{\partial L}{\partial b_y}$, $\frac{\partial L}{\partial b_h}$

While calculating $\frac{\partial L}{\partial W_{hh}}$ we face problem with traditional Backpropagation.

Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps. So, slightly modified version of Backpropagation called backpropagation through time (BPTT) is being used.

Let's calculate these derivatives one by one.

$$\text{Loss } L(\hat{Y}, Y) = \sum_{t=1}^{T} L_t(\hat{Y}_t, Y_t)$$

$$= -\sum_{t=1}^{T} Y_t \log \hat{Y}_t$$

$$= -\sum_{t=1}^{T} Y_t \log\left[\text{softmax}(O_t)\right]$$

$$\boxed{\frac{\partial L}{\partial W_{yh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_{yh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{Y}_t} \cdot \frac{\partial \hat{Y}_t}{\partial O_t} \left(\frac{\partial O_t}{\partial W_{yh}}\right) \to h_t}$$

$$= \sum_{t=1}^{T} (\hat{Y}_t - Y_t) \otimes h_t$$

$$\frac{\partial L}{\partial b_y} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{Y}_t} \cdot \frac{\partial \hat{Y}_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial b_y} = \sum_{t=1}^{T} (\hat{Y}_t - Y_t)$$

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial \hat{Y}_t} \frac{\partial \hat{Y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}}$$

we know $h_t = \tanh\left(W_{xh}^T x_t + W_{hh}^T \cdot h_{t-1} + b_h\right)$

$$\Rightarrow \frac{\partial L_t}{\partial \hat{Y}_t} \cdot \frac{\partial \hat{Y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial W_{hh}}$$

Thus, at timestamp t, we can compute the gradient & further go through time till 1 to compute overall gradient w.r.t W_{hh}.