# Neural n-gram language model

Caio Corro

## 1 Introduction

We are interested in modeling a probability distribution over sentences, for example we may wish to compute the probability of the sentence "The dog is eating": $p(\text{"The dog is eating"})$. We call this distribution a language model. As we can't build easily this kind of distribution by hand, we will learn a parameterized distribution from a corpus of sentences. In other words, we want to learn a distribution $p_\theta(\cdot)$ that maximizes the corpus likelihood, where $\theta$ denotes the parameter of the distribution:

$$\theta^* = \arg\max_{\theta \in \Theta} \prod_{i=1}^{d} p_\theta(x^{(i)})$$

where $x^{(1)}...x^{(d)}$ are the $d$ sentences in the corpus and $\Theta$ the set of valid parameters. It often convenient to consider the log-likelihood of the corpus, which does not change the argmax because the log function is strictly increasing:

$$= \arg\max_{\theta \in \Theta} \log \prod_{i=1}^{d} p_\theta(x^{(i)})$$

Given that $\log(a \times b) = \log a + \log b$, we have:

$$= \arg\max_{\theta \in \Theta} \sum_{i=1}^{d} \log p_\theta(x^{(i)})$$

$$= \arg\min_{\theta \in \Theta} -\sum_{i=1}^{d} \log p_\theta(x^{(i)})$$

where in the last line we rewritte the maximization problem as a minimization problem. The way we compute the optimal parameters $\theta^*$ depends on how the parameterized distribution is defined. We will see two cases, a standard approach where we rely on standard categorical distributions and neural network based approach.

An obvious question to ask is why we are interested in this problem? First, there is a fundamental question: how can we learn a model of sentence distribution? This question is of great interest to analyze the ability to model sequential data. Second, there are practical applications:

- natural language generation: we want to generate sentences. In practice, this generation would be conditioned on external factors, for example generate the answer to a question or translate a sentence;
- model combination: it is very easy to build a corpus containing many sentence in a given language. However, it may be expensive to build annotated corpora for a given task. Language models can be used to improve the prediction results when the goal is to predict a sentence. For example, in machine translation we want to translate a sentence $x$ in a source language into a sentence $y$ in a target language. However, my may have only a few parallel data to train the model. Let $p(y|x)$ be the conditional probability of sentence $y$ being the translation of sentence $x$, and $p(y)$ be a language model. Predicting the translation can be casted as the following problem:

$$y^* = \arg\max_{y} \lambda p(y|x) + (1 - \lambda) p(y) \tag{1}$$

  where the term $\lambda \in [0, 1]$ balance the contribution of the two model. If $\lambda = 1$, then we only use the translation model. If $\lambda < 1$ we also take into account the language model in the target side.

## 2 Direct parameterization

The most simple (way too simple!) model would be a distribution $p_\theta(\cdot)$ which give a probability to $\frac{1}{d}$ to each sentence in the training corpus and 0 to other sentences. This model is uninteresting because it does not generalize at all to unseen sentences. Instead we are going to define the language model as a probability of a sequence of words.[1] Let $n$ be the length of a sentence composed of words $w_1, w_2...w_{n-1}, w_n$. The probability of this sentence under a parameterized distribution is written $p_\theta(w_1, w_2...w_{n-1}, w_n)$. Without any assumption, we can rewrite this distribution as a conditional distribution, that is the probability of each word is conditioned on its predecessors:

$$p_\theta(w_1, w_2...w_{n-1}, w_n) = \prod_{i=1}^{n} p_\theta(w_i|\ w_1...w_{i-1})$$

We call a model that decomposes its joint probability this way an auto-regressive model. In the most straightforward approach, $\theta$ are the parameters of categorical distributions of a word given its context (its predecessors). This approach is not feasible as such: we need one categorical distribution by possible sequence of predecessors! Neural networks, and most precisely recurrent neural networks (RNN) can help to solve this problem, but they are not the focus of this note.

Imposing a Markov property on the sequence distribution is a simple way to reduce the number of distributions to learn. The idea is that we are going to condition the probability of a word only on a limited context, the $m$ previous words. This models are qualified as memory-less as they forgot the past words. For example, if $m = 1$, the joint distribution is factorized as:[2]

$$p_\theta(w_1, w_2...w_{n-1}, w_n) = \prod_{i=1}^{n} p_\theta(w_i|\ w_{i-1})$$

This model is also called a first order Markov chain or a bigram language model as we parameterize bigram transitions. If $\mathcal{V}$ is the vocabulary, i.e. $w_i \in \mathcal{V}$, then this model parameterizes $|\mathcal{V}|$ categorical distributions, and the sample space of each distribution is $\mathcal{V}$. This means that $\theta$ is a set of parameters of size $|\mathcal{V}| \times |\mathcal{V}|$. If we set $m = 2$, the joint distribution is factorized as:

$$p_\theta(w_1, w_2...w_{n-1}, w_n) = \prod_{i=1}^{n} p_\theta(w_i|\ w_{i-2}, w_{i-1})$$

which is called a second order Markov chain or trigram language model. The number of parameters we need to parameterize a trigram language model is $|\mathcal{V}| \times |\mathcal{V}| \times |\mathcal{V}|$ We can see that the number of parameters grows exponentially with the size of the context $m$, i.e. the number of parameters in $\theta$ is $|\mathcal{V}|^{m+1}$, which can quickly become unmanageable.

In the following, we will focus on the brigram model. In this case, we have conditional distributions parameterized by a set of tensors $\theta = \{\theta^{(u)} \in \mathbb{R}^{|\mathcal{V}|}\}_{u \in \mathcal{V}}$, i.e. $\theta$ is a set of $|\mathcal{V}|$ vectors and each vector is of size $|\mathcal{V}|$. In other words, the probability of word $v \in \mathcal{V}$ being after word $u \in \mathcal{V}$ is:

$$p_\theta(v|u) = \theta_v^{(u)}$$

where we index the vector $\theta^{(u)}$ by elements of $v$. It is sometime useful to consider the vocabulary to be integers, i.e. $\mathcal{V} = \{1...|\mathcal{V}|\}$ and to assume that we mapped each word to an integer. As $p_\theta(\cdot|u)$ must be a valid probability distribution, we have following constraints on the set of parameters:[3]

- $\forall u, v : \theta_v^{(u)} \geq 0$
- $\forall u : \sum_v \theta_v^{(u)} = 1$

The set of all valid parameters is therefore $\Theta = \left\{\theta \middle| \forall u, v : \theta_v^{(u)} \geq 0 \wedge \forall u : \sum_v \theta_v^{(u)} = 1\right\}$.

As explained above, the training problem aim to compute the optimal parameters $\theta^*$ as follows:

$$\theta^* \quad = \quad \arg\min_{\theta \in \Theta} \quad -\sum_{i=1}^{d} \log p_\theta(x^{(i)})$$

[1] It is also possible to work directly on characters, but we won't consider this approach here.

[2] In practice special care should be taken wrt to the first word of the sentence: we need an initial word distribution or specific word "begin of sentence" which is used as condition for the first word. We completely ignore this practical issue in this note.

[3] Previously, we said that this distribution is a categorical distribution. In theory, in a categorical distribution, no element of the sampled space can have a probability of 0. To simplify exposition, we allow this here.

This is an easy problem which admit a simple closed form solution that has the following form:

$$\theta_v^{(u)} = \frac{\#(u,v)}{\#(u)}$$

where $\#(u,v)$ is the number of times the bigram $uv$ appears in the corpus and $\#(u)$ is the number of times the word $u$ appears in the corpus. We can prove that this is the optimal solution via the Karush-Kuhn-Tucker conditions, but this is outside the scope of this note.

With this approach, the distribution is sparse: every bigram that has not been seen in the training data has a probability of 0, i.e. $p(v|u) = 0$ if $\#(u,v)$. This can be problematic as the probability of sentence will be exactly zero if it contains such a bigram. One simple way to prevent this is smoothing by setting:

$$\theta_v^{(u)} = \frac{1 + \#(u,v)}{1 + \#(u)}$$

It is not the optimal solution of the problem above, but it prevents null probabilities!

# 3 Neural parameterization

A major issue with the model presented above is that is doesn't generalize well. For example, lets focus on adjective-noun bigrams. We would expect two probabilities $p(\text{"}cat\text{"}|\ \text{"}big\text{"})$ and $p(\text{"}cat\text{"}|\ \text{"}fat\text{"})$ to be roughly similar. However, if the data only contains bigram "big cat" but "not "fat cat", the second one will have a null probability. We could argue the more data we have, the less this problem will happen. However, dataset are always finite, and in practice this issue happens frequently. Neural parameterization of a language model is an elegant way to tackle this issue. Instead of directly parameterizing the categorical distributions, we will parameterize a neural network that computes the conditional distributions.

A neural network is simply a parameterized function that takes some input values and compute output values. In the bigram language model case, the input will be the previous word and the output a probability distribution on the next word. The first problem we have, is how can we represent the input word? We cannot simply represent words by integers, as it is not a meaningful representation: the notion of distance between integers is unrelated to the distances between words. If "house" is mapped to 10 and "tiger" to 11, they will be almost similar inputs for the first layer of the neural network although they are highly different concepts. To bypass this issue, we represent words via embeddings. To this end, the neural network contains an embedding table that contains vectors of real values, each vector corresponding to one word. Let $\boldsymbol{E} \in \mathbb{R}^{l \times |\mathcal{V}|}$ be an embedding table. We denote $E_{.,u}$ the u-th column of $\boldsymbol{E}$, that is $E_{.,u}$ is the embedding associated with word $u$. The dimension of the embedding $l$ is an hyperparameter, typically between 100 and 300. The matrix $\boldsymbol{E}$ is a parameter of the neural network and is trained end-to-end.

We can now build a simple neural network that computes the probability $p_\theta(v|u)$:

$$\begin{aligned}
\boldsymbol{z} &= \tanh(\boldsymbol{A}^{(1)} E_{.,u} + \boldsymbol{b}^{(1)}) \\
\boldsymbol{o} &= \boldsymbol{A}^{(2)} \boldsymbol{z} + \boldsymbol{b}^{(2)} \\
p_\theta(v|u) &= \frac{\exp(o_v)}{\sum_{v'} \exp(o'_v)}
\end{aligned}$$

where $\boldsymbol{z} \in \mathbb{R}^h$ is a hidden representation of dimension $h$ and $\boldsymbol{o} \in \mathbb{R}^{|\mathcal{V}|}$ is the output logits (i.e. unconstrained weights associated with words in the vocabulary). The last line computes the probability of word $v$ using the softmax. As such, $p_\theta(v|u)$ is a valid probability distribution. Note that due the numerator being strictly positive, this distribution has full support: there no word in vocabulary with a probability of 0. The parameters of the neural network are $\theta = \{\boldsymbol{E}, \boldsymbol{A}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{A}^{(2)}, \boldsymbol{b}^{(2)}\}$ with $\boldsymbol{E} \in \mathbb{R}^{l \times |\mathcal{V}|}$, $\boldsymbol{A}^{(1)} \in \mathbb{R}^{h \times l}$, $\boldsymbol{b}^{(1)} \in \mathbb{R}^h$, $\boldsymbol{A}^{(2)} \in \mathbb{R}^{|\mathcal{V}| \times l}$, and $\boldsymbol{b}^{(2)} \in \mathbb{R}^{|\mathcal{V}| \times l}$. Note these hyperparameters are unconstrained.

Now, how could we parameterize a trigram model $p_\theta(v|u,w)$? We simply concatenate the two embeddings of word $u$ and $w$ at the input of the network![4]

$$\begin{aligned}
\boldsymbol{z} &= \tanh(\boldsymbol{A}^{(1)} [E_{.,u}; E_{.,w}] + \boldsymbol{b}^{(1)}) \\
\boldsymbol{o} &= \boldsymbol{A}^{(2)} \boldsymbol{z} + \boldsymbol{b}^{(2)} \\
p_\theta(v|u) &= \frac{\exp(o_v)}{\sum_{v'} \exp(o'_v)}
\end{aligned}$$

where $[E_{.,u}; E_{.,w}]$ denotes the concatenation of the two embeddings. the only change is that the $\boldsymbol{A}^{(1)}$ and $\boldsymbol{b}^{(1)}$ have different shapes, i.e. $\boldsymbol{A}^{(1)} \in \mathbb{R}^{h \times 2l}$ and $\boldsymbol{b}^{(1)} \in \mathbb{R}^{2l}$. Importantly, note that the same embedding matrix is used at both input positions.

[4]Here, we use the same embedding table for the two embeddings. We could use two different table, one for the word at position i-1 and one for the word at position i-2. However, it is more common to use the same table for both inputs.

# 4 Conclusion

The neural network presented in this note can be trained end-to-end on a corpus using stochastic gradient descent. Although we hope that it will capture similarities between $p(\text{"}cat\text{"}|\ \text{"}big\text{"})$ and $p(\text{"}cat\text{"}|\ \text{"}fat\text{"})$, it should be tested! The intuition is that during training, we will learn to represent "big" and "fat" with similar embeddings are they are synonymous. Note that we do not force this property, we hope that it will be true.[5] Moreover, after training, the embedding table contains vector representations of word. They can be manipulated in interesting ways (we will see that later!). One issue that our model do not solve is the memory-less problem: we cannot learn long range dependencies between words. We will later study recurrent neural network based language models that solve this issue.

[5]The reason why why we expect this to happen is related to the concept of distributional semantics, see `https://en.wikipedia.org/wiki/Distributional_semantics`