Importance sampling and Noise-Contrastive Estimation methods to train Language Models.

Mohammed LANSARI
mohammed.lansari@universite-paris-salcay.fr

Imad BOUHOU
imad.bouhou@universite-paris-salcay.fr

Ihsan ULLAH
ihsan.ullah@universite-paris-salcay.fr

Ousmane CISSE
ousmane.cisse@universite-paris-salcay.fr

Abstract :

Training neural language models is hard due to the large output that should have the same size as the vocabulary, which consequently creates another problem when the we want to compute the normalization factor.
In this work, we revisit the solutions based on Importance sampling [2], Noise-Contrastive estimation [3] and the Negative sampling method which is linked to the NCE and we compared in each solution, multiple noise distributions, and also in [3] the authors proposed to add the normalizing factor as a trainable parameter in the model, which we tested also, and trained multiple models with different initialization for the normalizing factor.
The authors [3] managed to get great results with NCE when they initialized the bias of the linear output layer with the values obtained from the $\log(Unigram)$, which we managed to reproduce in our experiments.

key words :
Importance sampling, language modeling, Noise-Contrastive estimation, Negative Sampling, Normalizing factor.

Contents

1 Introduction

In this project, we have worked on different approaches to find a workaround the computation of the normalization factor for language models, and these approaches are mainly built on the objective function with which the model will be trained.
The objective functions used are :

- Importance Sampling [2].
- Noise-Contrastive Estimation [3].
- Negative-Sampling. [3].

We denote f_θ the neural language model, so like was mentioned before, given a context S, f_θ will predict the most probable next word given a certain vocabulary, so this conditional distribution can be written like this :

$$P_\theta(w|S) = \frac{\exp(f_\theta(w,S))}{\sum\limits_{w_i \in \mathcal{V}} \exp(f_\theta(w_i,S))} = \frac{\exp(f_\theta(w,S))}{Z_\theta(S)} \tag{1}$$

As we can see in equation (1), the denominator can be huge to compute.
We denote $\tilde{p}(w|S)$ the conditional probability over the data that we want to approximate by $P_\theta(w|S)$. The neural network f_θ aims to maximize the expected value w.r.t $\tilde{p}(w|S)$ of the log-conditional distribution $P_\theta(w|S)$.

$$LL(\theta) = \mathbb{E}_{\tilde{p}(.|S)}\Big[\log P_\theta(w|S)\Big] \tag{2}$$

As presented in [3], we can recompute the gradients like this :

$$\begin{aligned}
\frac{\partial}{\partial\theta}\mathbb{E}_{\tilde{p}(.|S)}\Big[\log P_\theta(w|S)\Big] &= \frac{\partial}{\partial\theta}\sum_{w \in \mathcal{V}} \tilde{p}(w|S)\log P_\theta(w|S) \\
&= \sum_{w \in \mathcal{V}} \tilde{p}(w|S)\frac{\partial}{\partial\theta}\log P_\theta(w|S) \\
&= \mathbb{E}_{\tilde{p}(.|S)}\Big[\frac{\partial}{\partial\theta}\log P_\theta(w|S)\Big] \\
&= \mathbb{E}_{\tilde{p}(.|S)}\Big[\frac{\partial}{\partial\theta}f_\theta(w,S) - \frac{\partial}{\partial\theta}\log(Z_\theta(S))\Big] \\
&= \mathbb{E}_{\tilde{p}(.|S)}\Big[\frac{\partial}{\partial\theta}f_\theta(w,S)\Big] - \frac{\partial}{\partial\theta}\log(Z_\theta(S))
\end{aligned}$$

We know that :

$$\begin{aligned}
\frac{\partial}{\partial\theta}\log(Z_\theta(S)) &= \frac{1}{Z_\theta(S)}\frac{\partial}{\partial\theta}Z_\theta(S) \\
&= \frac{1}{Z_\theta(S)}\frac{\partial}{\partial\theta}\sum_{w_i \in \mathcal{V}}\exp(f_\theta(w_i,S)) \\
&= \frac{1}{Z_\theta(S)}\sum_{w_i \in \mathcal{V}}\exp(f_\theta(w_i,S))\frac{\partial}{\partial\theta}f_\theta(w_i,S) \\
&= \sum_{w_i \in \mathcal{V}}\frac{\exp(f_\theta(w_i,S))}{Z_\theta(S)}\frac{\partial}{\partial\theta}f_\theta(w_i,S) \\
&= \sum_{w_i \in \mathcal{V}} P_\theta(w_i|S)\frac{\partial}{\partial\theta}f_\theta(w_i,S)
\end{aligned}$$

Hence the gradients take this form :

$$\frac{\partial}{\partial\theta}\mathbb{E}_{\tilde{p}(.|S)}\Big[\log P_\theta(w|S)\Big] = \mathbb{E}_{\tilde{p}(.|S)}\Big[\frac{\partial}{\partial\theta}f_\theta(w,S)\Big] - \sum_{w_i\in\mathcal{V}} P_\theta(w_i|S)\frac{\partial}{\partial\theta}f_\theta(w_i,S) \tag{3}$$

If we consider only the case of one-sampling, the gradients will take this form (the same form in [3]) :

$$\frac{\partial}{\partial\theta}\log P_\theta(w|S) = \frac{\partial}{\partial\theta}f_\theta(w,S) - \sum_{w_i\in\mathcal{V}} P_\theta(w_i|S)\frac{\partial}{\partial\theta}f_\theta(w_i,S) \tag{4}$$

So to compute the gradients we still have to compute a summation over all possible words in the vocabulary, not to mention compute the partition function in the second term for the conditional distribution $P_\theta(w_i|S)$ $|\mathcal{V}|$ times, which is expensive and sometimes impossible.
In the following sections, we will see how Importance Sampling and NCE, simplifies the computation of the gradients in the formula (4).

2 Objectives functions

2.1 Importance Sampling

Importance sampling is a technique to approximate the partition function and the gradients. This is done by introducing a new distribution P_n from which we know that it is easy to sample.
The equation (3), can be written as following:

$$\begin{aligned}
\frac{\partial}{\partial\theta}\log P_\theta(w|S) &= \frac{\partial}{\partial\theta}f_\theta(w,S) - \sum_{w_i\in\mathcal{V}} P_\theta(w_i|S)\frac{\partial}{\partial\theta}f_\theta(w_i,S)\\
&= \frac{\partial}{\partial\theta}f_\theta(w,S) - \sum_{w_i\in\mathcal{V}} P_n(w_i)\frac{P_\theta(w_i|S)}{P_n(w_i)}\frac{\partial}{\partial\theta}f_\theta(w_i,S)\\
&= \frac{\partial}{\partial\theta}f_\theta(w,S) - \mathbb{E}_{P_n}\Big[\frac{P_\theta(w|S)}{P_n(w)}\frac{\partial}{\partial\theta}f_\theta(w,S)\Big]\\
&= \frac{\partial}{\partial\theta}f_\theta(w,S) - \frac{1}{Z_\theta(S)}\mathbb{E}_{P_n}\Big[\frac{\exp(f_\theta(w,S))}{P_n(w)}\frac{\partial}{\partial\theta}f_\theta(w,S)\Big]
\end{aligned}$$

By using the Monte-Carlo estimation, we can approximate the gradients by the following equation:

$$\frac{\partial}{\partial\theta}\log P_\theta(w|S) \approx \frac{\partial}{\partial\theta}f_\theta(w,S) - \frac{1}{k}\frac{1}{Z_\theta(S)}\sum_{\substack{i=1\\ \hat{w_i}\sim P_n}}^{k}\frac{\exp(f_\theta(\hat{w_i},S))}{P_n(\hat{w_i})}\frac{\partial}{\partial\theta}f_\theta(\hat{w_i},S) \tag{5}$$

We see that the partition function is still there even after approximating the sum of the vocabulary. The same way, we again use our new distribution P_n and rewrite the $Z_\theta(S)$ as an expectation:

$$\begin{aligned}
Z_\theta(S) &= \sum_{w_i\in\mathcal{V}}\exp(f_\theta(w_i,S))\\
&= \mathbb{E}_{P_n}\Big[\frac{\exp(f_\theta(w,S)}{P_n(w)}\Big]
\end{aligned}$$

And with Monte-Carlo estimation, we get :

$$Z_\theta(S) \approx \frac{1}{k}\sum_{\substack{i=1\\ \hat{w_i}\sim P_n}}^{k}\frac{exp(f_\theta(\hat{w_i},S))}{P_n(\hat{w_i})}$$

2.2 Noise-Contrastive Estimation

NCE was first introduced in [1], as a way to learn models that are un-normalized, by either setting the partition function to 1 or to consider it as a trainable parameter.
The full concept of NCE is based on simplifying the multi-classification problem into a binary classification by introducing new samples from a noise distribution that we will denote by P_n and we denote $\tilde{p}(w|S)$ the conditional probability over the data that we want to approximate by $P_\theta(w|S)$.
We will assume that each couple (word,context) i.e (w, S) is associated with k noise samples from P_n.
We can now consider that we have two classes for all the samples i.e if the sample comes from the noise distribution then its in class $(C = 0)$, and if the sample comes from the data available then its class is $(C = 1)$.
So $P(C, w|S)$ represents the joint probability of the word w being in the class C given a context S.
Let's see how to explicitly write this joint distribution :

$$\begin{aligned} P(C = 1, w|S) &= P(C = 1|S)\ P(w|C = 1, S) \\ &= \frac{1}{k+1}\tilde{p}(w|S) \end{aligned}$$

The same way, we get :

$$\begin{aligned} P(C = 0, w|S) &= P(C = 0|S)\ P(w|C = 0, S) \\ &= \frac{k}{k+1}P_n(w) \end{aligned}$$

So the likelihood of a given word w given a contet S will take the following form :

$$\begin{aligned} P(w|S) &= P(C = 0, w|S) + P(C = 1, w|S) \\ &= \frac{k}{k+1}P_n(w) + \frac{1}{k+1}\tilde{p}(w|S) \end{aligned}$$

Hence :

$$P(w|S) = \frac{\tilde{p}(w|S) + k\ P_n(w)}{k+1} \tag{6}$$

Let's compute the probabilities of each class given a couple (word,context) :

$$\begin{aligned} P(C = 0|w, S) &= \frac{P(C = 0, w|S)}{P(w|S)} \\ &= \frac{kP_n(w)}{k+1}\frac{k+1}{\tilde{p}(w|S) + k\ P_n(w)} \end{aligned}$$

Hence :

$$P(C = 0|w, S) = \frac{kP_n(w)}{\tilde{p}(w|S) + k\ P_n(w)} \tag{7}$$

The same way we can prove that :

$$P(C = 1|w, S) = \frac{\tilde{p}(w|S)}{\tilde{p}(w|S) + k\ P_n(w)} \tag{8}$$

The goal is to optimize a distribution with parameters θ to approximate $\tilde{p}(w|S)$, Hence we can write :

$$P_\theta(C = 0|w, S) = \frac{kP_n(w)}{P_\theta(w|S) + k\ P_n(w)} \tag{9}$$

$$P_\theta(C = 1|w, S) = \frac{P_\theta(w|S)}{P_\theta(w|S) + k\ P_n(w)} \tag{10}$$

2.2.1 Relation between NCE and MLE:

Under this new constraints (2 classes instead of mutli-classification) the goal is to maximize the expected log-likelihood under the conditional distribution of the dataset $P(w|S)$, and we can write :

$$\mathbb{E}_{P(.|S)}\Big[\log P_\theta(C|w,S)\Big] = \sum_{w\in\mathcal{V}} P(w|S)\ \log P_\theta(C|w,S)$$

Using the equation (6), we get :

$$\begin{aligned}
\mathbb{E}_{P(.|S)}\Big[\log P_\theta(C|w,S)\Big] &= \sum_{w\in\mathcal{V}} \frac{\tilde{p}(w|S) + k\ P_n(w)}{k+1}\ \log P_\theta(C|w,S)\\
&= \frac{1}{k+1}\sum_{w\in\mathcal{V}}\Big[\tilde{p}(w|S) + k\ P_n(w)\Big]\ \log P_\theta(C|w,S)\\
&= \frac{1}{k+1}\Big(\sum_{w\in\mathcal{V}}\tilde{p}(w|S)\log P_\theta(C|w,S) + k\sum_{w\in\mathcal{V}} P_n(w)\ \log P_\theta(C|w,S)\Big)\\
&= \frac{1}{k+1}\Big(\mathbb{E}_{\tilde{p}(.|S)}\big[\log P_\theta(C|w,S)\big] + k\ \mathbb{E}_{P_n(.)}\big[\log P_\theta(C|w,S)\big]\Big)
\end{aligned}$$

The first term $\mathbb{E}_{\tilde{p}(.|S)}\big[\log P_\theta(C|w,S)\big]$ is an expectation taken w.r.t to the conditional probability over the real data, so by definition we should have $(C=1)$, and the same way for the second term, so by using the equation (9) and (10) we can write :

$$\begin{aligned}
\mathbb{E}_{P(.|S)}\Big[\log P_\theta(C|w,S)\Big] &= \frac{1}{k+1}\Big(\mathbb{E}_{\tilde{p}(.|S)}\Big[\log P_\theta(C=1|w,S)\Big] + k\ \mathbb{E}_{P_n(.)}\Big[\log P_\theta(C=0|w,S)\Big]\Big)\\
&= \frac{1}{k+1}\Big(\mathbb{E}_{\tilde{p}(.|S)}\Big[\log\frac{P_\theta(w|S)}{P_\theta(w|S)+k\ P_n(w)}\Big] + k\ \mathbb{E}_{P_n(.)}\Big[\log\frac{kP_n(w)}{P_\theta(w|S)+k\ P_n(w)}\Big]\Big)
\end{aligned}$$

For simplification, we will denote $J_\theta(S)$ such that :

$$J_\theta(S) = \mathbb{E}_{\tilde{p}(.|S)}\Big[\log\frac{P_\theta(w|S)}{P_\theta(w|S)+k\ P_n(w)}\Big] + k\ \mathbb{E}_{P_n(.)}\Big[\log\frac{kP_n(w)}{P_\theta(w|S)+k\ P_n(w)}\Big] \tag{11}$$

We can easily prove that the gradients will take the following form :

$$\begin{aligned}
\frac{\partial}{\partial\theta}J_\theta(S) &= \mathbb{E}_{\tilde{p}(.|S)}\Big[\frac{kP_n(w)}{P_\theta(w|S)+k\ P_n(w)}\frac{\partial}{\partial\theta}\log P_\theta(w|S)\Big] - k\ \mathbb{E}_{P_n(.)}\Big[\frac{P_\theta(w|S)}{P_\theta(w|S)+k\ P_n(w)}\frac{\partial}{\partial\theta}\log P_\theta(w|S)\Big]\\
&= \sum_{w\in\mathcal{V}}\tilde{p}(w|S)\frac{kP_n(w)}{P_\theta(w|S)+k\ P_n(w)}\frac{\partial}{\partial\theta}\log P_\theta(w|S) - k\sum_{w\in\mathcal{V}}P_n(w)\frac{P_\theta(w|S)}{P_\theta(w|S)+k\ P_n(w)}\frac{\partial}{\partial\theta}\log P_\theta(w|S)
\end{aligned}$$

Hence :

$$\frac{\partial}{\partial\theta}J_\theta(S) = \sum_{w\in\mathcal{V}}\frac{k\ P_n(w)}{P_\theta(w|S)+k\ P_n(w)}\Big[\tilde{p}(w|S) - P_\theta(w|S)\Big]\frac{\partial}{\partial\theta}\log P_\theta(w|S) \tag{12}$$

Let's compute this gradient when $k\to\infty$:

$$\begin{aligned}
\lim_{k\to+\infty}\frac{\partial}{\partial\theta}J_\theta(S) &= \lim_{k\to+\infty}\sum_{w\in\mathcal{V}}\frac{k\ P_n(w)}{P_\theta(w|S)+k\ P_n(w)}\Big[\tilde{p}(w|S) - P_\theta(w|S)\Big]\frac{\partial}{\partial\theta}\log P_\theta(w|S)\\
&= \sum_{w\in\mathcal{V}}\lim_{k\to+\infty}\frac{k\ P_n(w)}{P_\theta(w|S)+k\ P_n(w)}\Big[\tilde{p}(w|S) - P_\theta(w|S)\Big]\frac{\partial}{\partial\theta}\log P_\theta(w|S)\\
&= \sum_{w\in\mathcal{V}}\lim_{k\to+\infty}\frac{1}{\frac{P_\theta(w|S)}{k\ P_n(w)}+1}\Big[\tilde{p}(w|S) - P_\theta(w|S)\Big]\frac{\partial}{\partial\theta}\log P_\theta(w|S)\\
&= \sum_{w\in\mathcal{V}}\Big[\tilde{p}(w|S) - P_\theta(w|S)\Big]\frac{\partial}{\partial\theta}\log P_\theta(w|S)\\
&= \sum_{w\in\mathcal{V}}\tilde{p}(w|S)\frac{\partial}{\partial\theta}\log P_\theta(w|S) - \sum_{w\in\mathcal{V}}P_\theta(w|S)\frac{\partial}{\partial\theta}\log P_\theta(w|S)
\end{aligned}$$

from equation (1), We know that :

$$logP_\theta(w|S) = f_\theta(w,S) - log(Z_\theta(S))$$

Hence the first term can be simplified like this :

$$\begin{aligned}\sum_{w\in\mathcal{V}} \tilde{p}(w|S)\frac{\partial}{\partial\theta}(f_\theta(w,S) - log(Z_\theta(S)) &= \sum_{w\in\mathcal{V}} \tilde{p}(w|S)\frac{\partial}{\partial\theta}f_\theta(w,S) - \frac{\partial}{\partial\theta}log\ Z_\theta(S)\sum_{w\in\mathcal{V}}\tilde{p}(w|S) \\ &= \sum_{w\in\mathcal{V}} \tilde{p}(w|S)\frac{\partial}{\partial\theta}f_\theta(w,S) - \frac{\partial}{\partial\theta}log\ Z_\theta(S)\end{aligned}$$

The same thing for the second term, and we get :

$$\sum_{w\in\mathcal{V}} P_\theta(w|S)\frac{\partial}{\partial\theta}(f_\theta(w,S) - log(Z_\theta(S)) = \sum_{w\in\mathcal{V}} P_\theta(w|S)\frac{\partial}{\partial\theta}f_\theta(w,S) - \frac{\partial}{\partial\theta}log\ Z_\theta(S)$$

the term $\frac{\partial}{\partial\theta}log\ Z_\theta(S)$ will be canceled out, and we get :

$$\lim_{k\to+\infty}\frac{\partial}{\partial\theta}J_\theta(S) = \sum_{w\in\mathcal{V}} \tilde{p}(w|S)\frac{\partial}{\partial\theta}f_\theta(w,S) - \sum_{w\in\mathcal{V}} P_\theta(w|S)\frac{\partial}{\partial\theta}f_\theta(w,S)$$

Hence :

$$\lim_{k\to+\infty}\frac{\partial}{\partial\theta}J_\theta(S) = \mathbb{E}_{\tilde{p}(.|S)}\Big[\frac{\partial}{\partial\theta}f_\theta(w,S)\Big] - \sum_{w\in\mathcal{V}} P_\theta(w|S)\frac{\partial}{\partial\theta}f_\theta(w,S) \tag{13}$$

The last formula (13) is the expected gradient of the formula (4), hence when k goes to ∞, the gradient of the NCE converges to the gradient of the Maximum likelihood estimator.

2.2.2 NCE for one sampling:

For all the observations in the data (real and noise) we have :

$$\frac{\partial}{\partial\theta}J_\theta(S) = \mathbb{E}_{\tilde{p}(.|S)}\Big[\frac{kP_n(w)}{P_\theta(w|S) + k\ P_n(w)}\frac{\partial}{\partial\theta}\log P_\theta(w|S)\Big] - k\ \mathbb{E}_{P_n(.)}\Big[\frac{P_\theta(w|S)}{P_\theta(w|S) + k\ P_n(w)}\frac{\partial}{\partial\theta}\log P_\theta(w|S)\Big]$$

Now considering only one sampling (w,S) from the real data associated with $(\hat{w}_i)_{1\leq i\leq k}$ noise samples from P_n, with Monte-Carlo estimation for the second term, we will get :

$$\begin{aligned}\frac{\partial}{\partial\theta}J_\theta(w,S) &= \frac{kP_n(w)}{P_\theta(w|S) + k\ P_n(w)}\frac{\partial}{\partial\theta}\log P_\theta(w|S) - k\ \frac{1}{k}\sum_{i=1}^{k}\frac{P_\theta(w_i|S)}{P_\theta(w_i|S) + k\ P_n(w_i)}\frac{\partial}{\partial\theta}\log P_\theta(w_i|S) \\ &= \frac{kP_n(w)}{P_\theta(w|S) + k\ P_n(w)}\frac{\partial}{\partial\theta}\log P_\theta(w|S) - \sum_{i=1}^{k}\frac{P_\theta(\hat{w}_i|S)}{P_\theta(\hat{w}_i|S) + k\ P_n(\hat{w}_i)}\frac{\partial}{\partial\theta}\log P_\theta(\hat{w}_i|S)\end{aligned}$$

In these gradients, we still have to compute the partition function in P_θ.
What the authors suggested is to consider that : $\forall\ S\ Z_\theta(S) = 1$, and train the model to learn a self-normalized distribution.
They also suggested to add : $\forall\ S\ Z_\theta(S)$ as a trainable parameter for the model, and consider one additional parameter Z_c i.e the distribution P_θ will become :

$$P_\theta(w|S) = \exp\Big[f_\theta(w,S) - \log(Z_c)\Big]$$

The parameter Z_c will shift the output of the language model by $\log(Z_c)$, and will be learned in parallel.
The authors tested multiple initialization values for Z_c which we will revisit in the experiments section.

2.3 The Negative Sampling loss

In this section, we will present another approach that was talked about in the paper but was not the main focus.
Like in the NCE and IS, we need a noise distribution that we will denote by P_n, and each couple of (word,context) (w, S) is associated with $(\hat{w}_i)_{1\leq i\leq k}$ noise samples from P_n.
If we go back to the equation (11):

$$J_\theta(S) = \mathbb{E}_{\tilde{p}(.|S)}\big[\log \frac{P_\theta(w|S)}{P_\theta(w|S) + k\ P_n(w)}\big] + k\ \mathbb{E}_{P_n(.)}\big[\log \frac{k\ P_n(w)}{P_\theta(w|S) + k\ P_n(w)}\big]$$

We denote $R_\theta(w, S) = \log\left[\frac{P_\theta(w|S)}{k\ P_n(w)}\right]$, and we rewrite the previous equation like this:

$$\begin{aligned} J_\theta(S) &= \mathbb{E}_{\tilde{p}(.|S)}\Big[\log \frac{e^{R_\theta(w,S)}}{1 + e^{R_\theta(w,S)}}\Big] + k\ \mathbb{E}_{P_n(.)}\Big[\log \frac{1}{e^{R_\theta(w,S)} + 1}\Big] \\ &= \mathbb{E}_{\tilde{p}(.|S)}\Big[\log \sigma(R_\theta(w,S))\Big] + k\ \mathbb{E}_{P_n(.)}\Big[\log \sigma(-R_\theta(w,S))\Big] \end{aligned}$$

where σ is the sigmoid function.
At this point, we can train the neural language model to approximate R_θ, and when we take one sampling from the real data associated with $(\hat{w}_i)_{1\leq i\leq k}$ noise samples from P_n, we will get this objective function:

$$\begin{aligned} J_\theta(w,S) &= \log \sigma(R_\theta(w,S)) + k\frac{1}{k}\sum_{i=1}^{k} \log \sigma(-R_\theta(\hat{w}_i, S)) \\ &= \log \sigma(R_\theta(w,S)) + \sum_{i=1}^{k} \log \sigma(-R_\theta(\hat{w}_i, S)) \end{aligned}$$

At the end of the training, the model will have learned the distribution R_θ from which we can extract P_θ w.r.t to the noise distribution P_n.

3 Dataset and Data preprocessing

The dataset used in this project is the "Penn Treebank Corpus" [4]. The dataset consists of delimited and normalized word sequences (sentences) with replacement of unknown words with $< unk >$. The dataset is divided into 3 parts (training, validation and test), but we only used the training set which counts to 42068 sequences that will be used to train the model.

- The data is in the form of sentences with variable lengths, so we start by constructing the vocabulary V from the training data and we add two tokens $< bos >$ and $< eos >$ that indicates the end or the beginning of sentence.
  The neural language model can only interpret numbers so we construct two dictionaries: word-to-index and index-to-word to transform each word into an integer that will be the input of the neural language model, and also we keep track of the total words in the vocabulary.
- The next step is to convert the sentences into tensors, so each sentence will be represented by a series of integers where each integer represents a word in the dictionaries defined before.
- The tensor of each sentence consists of $< bos >$ token followed by the indexes of words in the sentence and the end token is $< eos >$. We also create a binary mask for each sentence for the purpose of keeping track of the words that really belong to the sentence in case we decide to pad the sentences to be able to use large batches to train the model.

4 Model

In this section, we describe the model that was used.
The model consists of three components as described below (we take a batch size of 1 and number of layers 1 to have a more readable description) :

1. x $\leftarrow$ Embedding(x)
2. out, $(h_n, c_n) \leftarrow$ LSTM(x)
3. out $\leftarrow$ Linear(out)

where :

- $x \in \mathbf{R}^L$ is a sentence of length L
- Embedding(.) is the embedding function from Pytorch which returns the embedding vectors associated to the input. In this case, Embedding is a function from $\mathbf{R}^L$ to $\mathbf{R}^{L\times E}$ where E is the embedding size.
- LSTM(.) is the LSTM function from Pytorch which returns the output of the model "out", the final hidden state "h_n" and the final cell state "c_n".
  In this case, LSTM is a function from $\mathbf{R}^{L\times E}$ to $(\mathbf{R}^{L\times H}, \mathbf{R}^H, \mathbf{R}^H)$ where H is the hidden size.
  We will only use the first output of the LSTM function which has dimension of $\mathbf{R}^{L\times H}$.
- Linear(.) is the linear function from Pytorch which applies a linear transformation to the given input.
  In this case, Linear(.) is a function from $\mathbf{R}^{L\times H}$ to $\mathbf{R}^{L\times V}$ where V is the vocabulary size.

4.1 Model evaluation with Perplexity

We evaluate the performance of the various objective functions that were used to train LSTM model through their losses and perplexities.
Perplexity is an intrinsic evaluation measure of the language model itself, independent of the specific tasks for which it will be used. It is defined as the inverse probability of the test set normalized by the number of words and a better model is the one that has a higher probability to the test set. Therefore a lower perplexity indicates a better model.
By considering a sequence of words $\mathbf{W} = \big[w_1, ..., w_N\big]$ of length $\mathbf{N}$, we can define the preplexity like this:

$$\mathbf{PP}(\mathbf{W}) = \sqrt[N]{\frac{1}{P(w_1, w_2, w_3, ..., w_N)}}$$

Using the chain rule we can expand the probability of $\mathbf{W}$

$$\begin{aligned}\mathbf{PP}(\mathbf{W}) &= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1, w_2, ..., w_{i-1})}} \\ &= \exp\Big[-\frac{1}{N}\sum_{i=1}^{N} \log P(w_i|w_1, w_2, ..., w_{i-1})\Big]\end{aligned}$$

The probability $\mathbf{P}(w_i|w_1, w_2, ..., w_{i-1})$ is the probability of the word w_i being the next after the sequence $\big[w_1, w_2, ..., w_{i-1}\big]$.
So when the model takes a sequence $\mathbf{W} = \big[w_1, ..., w_N\big]$ of length $\mathbf{N}$, for each word it will predict the next word, i.e :

- For w_1 the model will predict the next word and the model has to assign the highest probability for w_2 being the next word.
- For w_2 the model has to assign the highest probability for w_3 being the next word and the same for the rest.

The negative average of the probabilities predicted by the model represents the cross-entropy by definition.
To facilitate the calculation of the perplexity we will use its formulation as the exponential of the cross-entropy:

$$\mathbf{PP}(\mathbf{W}) = \exp\big[\mathbf{H}(\mathbf{W})\big]$$

Where $\mathbf{H}(\mathbf{W})$ is the cross-entropy.

We can detail how is the cross-entropy computed in practice, and we denote $y_{true} = \left[y_{true}^{(1)}, y_{true}^{(2)}, .., y_{true}^{(N)}\right]$ the one-hot encoded labels for each word in the sequence $\mathbf{W}$, such that $y_{true}^{(j)}$ is a binary vector where the entries are equal to 0 except the entry at the index of the word w_j, and the size of the $y_{true}^{(j)}$ is equal to $|\mathcal{V}|$.
We denote $y_{pred} = \left[y_{pred}^{(1)}, y_{pred}^{(2)}, .., y_{pred}^{(N)}\right]$ the predicted probabilities by the model $\mathbf{P}$.

For one word, the cross-entropy takes the following form :

$$CE(y_{true}^{(i)}, y_{pred}^{(i)}) = -\sum_{k=1}^{|\mathcal{V}|} y_{true}^{(i)}(k) \log \left[y_{pred}^{(i)}(k)\right]$$

In reality, the sum simplifies to one term, because $y_{true}^{(i)}$ is binary and only contains one value that's equal to 1.
In practice, we just use $y_{true}^{(i)}$ as index to recover the predicted probability from $y_{pred}^{(i)}$ instead of one-hot encoding $y_{true}^{(i)}$ to avoid high usage of memory in case we have many classes which is case because in this situation we have the size of the vocabulary as a number of classes.
For a sequence of words $\mathbf{W} = \left[w_1, ..., w_N\right]$ of length $\mathbf{N}$, it's like taking the average over the words in the sequence.

$$\mathbf{H}(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} CE(y_{true}^{(i)}, y_{pred}^{(i)}) = -\frac{1}{N}\sum_{i=1}^{N}\left[\sum_{k=1}^{|\mathcal{V}|} y_{true}^{(i)}(k) \log \left[y_{pred}^{(i)}(k)\right]\right]$$

4.2 Model Training

We have trained our model with Importance Sampling loss by choosing two different P_n noise distributions i.e. uniform and unigram categorical distributions.

We train with NCE loss with uniform and unigram distributions P_n and with trainable and non-trainable log partition function.
In some cases, we also initialize the bias of the linear layer of the model with $\log(P_n)$. We also have tried different values of k i.e. 100 and 500.

For Negative Sampling loss, we train the model with a unigram distribution P_n and with $k = 500$.

The hyper-parameters used for training are shown in Table 1.
Results of all the trainings are shown in Table 2.
The evolution of cross-entropy in each training are given in Figure 1a and Figure 1b shows losses with trainable $\log(Z_c)$.
Pytorch implementation of each loss function is shown in Appendix A-C.

vocabulary size	$\|V\|$
embeddding size	128
hidden size	128
number of layers	1
learning rate	For IS = 1e-4 otherwise 1e-3
number of epochs	50
batch size	512
optimizer	Adam

Table 1: Hyper-parameters for training

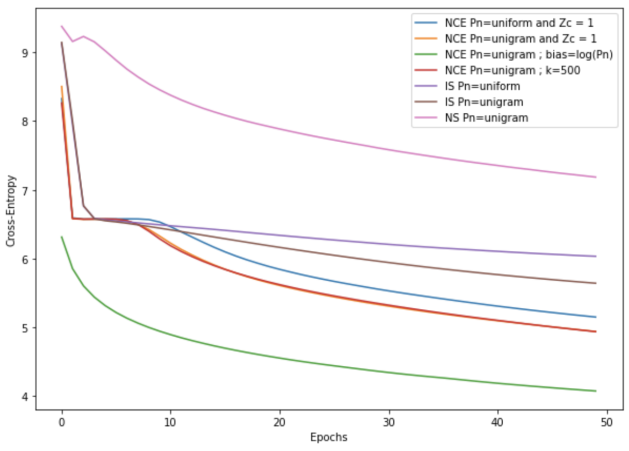
Training Method	Distribution P_n	Train CE	Train Perplexity	Trainable $log(Z_c)$	K-value	Test CE	Test Perplexity
Importance sampling	unigram	5.6389	281.15	✗	500	5.6272	277.88
Importance sampling	uniform	6.0316	416.38	✗	500	5.9979	402.57
NCE	unigram	4.9358	139.19	✗	100	5.1208	167.47
NCE	uniform	5.1461	171.77	✗	100	5.2605	192.57
NCE + bias init=$\log(P_n)$	**unigram**	**4.0689**	**58.49**	✗	100	**4.9777**	**145.14**
NCE	unigram	4.9332	138.83	✗	500	5.1137	166.29
Negative Sampling	unigram	7.1849	1319.34	✗	500	7.5719	1942.92
NCE	**unigram**	**4.4666**	**87.06**	$\log Z = \log(\lvert V\rvert)$	**100**	**4.9535**	**141.66**
NCE	unigram	4.9443	140.38	$\log Z = 0$	100	5.1231	167.86
NCE	unigram	4.7370	114.09	$\log Z = 5$	100	5.0324	153.30
NCE	unigram	4.5460	94.26	$\log Z = 10$	100	4.9787	145.28
NCE	unigram	4.7470	115.24	$\log Z = 15$	100	5.0320	153.24
NCE	unigram	4.9455	140.54	$\log Z = 20$	100	5.1278	168.64

Table 2: Results of Training with different loss functions, initialization of the bias of linear layer and different initialization values of $\log(Z_c)$ when it is trainable for NCE loss. (The symbol ✗ means that the partition function was set to 1 and was not trainable, i.e the model will learn a self-normalized distribution.

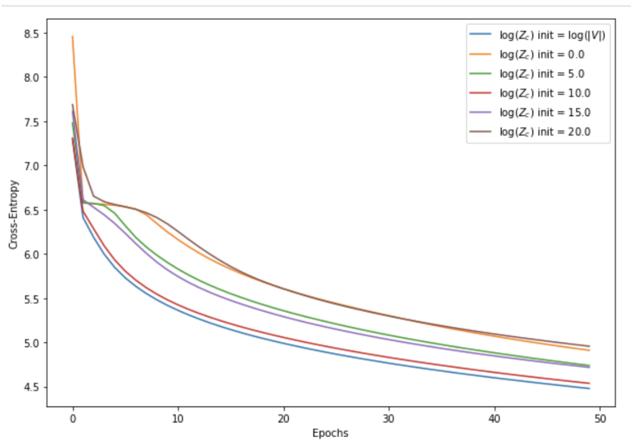
The Table 2 sums up most of the experiments that were performed using the objective functions and varying multiple parameters.
The models that had the best results in the table are the following :

- The model that was trained with the NCE with a bias of the output linear layer initialized with $\log(P_n)$ with P_n is the unigram distribution .
- The model that was trained with the NCE with a trainable partition function initialized with $\log(Z_c) = log(|\mathcal{V}|)$

During the training of the models with NCE and setting $\log(Z_c) = 0$ pushed the model to learn a self-normalized distribution and we supervised during the training the sum of the exponential of the outputs by printing it, and it's always close 1.
The Negative sampling method could not match the results of the other experiments but we believe that it only needs a higher number of epochs to improve, which we can observe in the curves in the figures below.



(a) Cross-entropy values during training without trainable $\log(Z_c) = 0$ for NCE and different distributions P_n.



(b) Cross-entropy values during training with trainable partition function $\log(Z_c)$ with different initialization values for NCE.

The Figure (1a) shows the changes in the cross-entropy as we train the models with Importance Sampling, Noise Contastive Estimation and Negative Sampling losses.
We can see that the cross-entropy decreases as the number of epochs increase, but at the beginning it drops very fast and then stays stable for 5 or 6 epochs before it starts decreasing again.

Figure (1b) shows the evolution of cross-entropy loss for NCE with trainable partition function and different initialization values for $\log(Z_c)$, and we can see that almost all the curves have the same behavior and eventually they will converge towards the same optimum, but some of them might take bigger number of epochs and the others will converge very fast, for example when the log-partition function is initialized by $\log |\mathcal{V}|$ the convergence is faster.

5 Conclusion

In this work, we have tried multiple methods to train neural language models like Importance Sampling [2] and Noise-Contrastive estimation [3] to avoid computing the partition function of the denominator in the learned conditional probability by the neural network.
We have trained multiple models with different parameters, and we managed to reproduce almost the same results as the paper, for example, the best model with the best low perplexity values were obtained with the models with the NCE when the bias of the output linear layer is initialized with the $\log(P_n)$ such that P_n is the unigram distribution.
The models that were trained with the NCE with a partition function fixed to 1, managed to learn a self-normalized distribution which was presented in the paper also.

Overall the generated sentences from the trained models using different loss functions are not satisfactory. Sometimes the model generates sentences in which the start of the sentence makes sense but the end of the sentence consists of a repeated word. In some cases, the model generates text with just one word repeated multiple times.

More experiments could be done with bi-grams to check how well the models learn bi-grams and then generates bi-grams instead of unigrams. The text generation can be explored more by Training on one language e.g. English to learn context and then generate sentences in another language e.g. French which is closer in structure to the training language.
We could also to try to combine the NCE loss alongside the initialization of the bias of the output linear layer with the log-unigram distribution and try a different noise distribution where sampled words can only occur once in the whole sample even though we believe this is expected to yield bad results, we can also try to explore how to train the noise distribution in parallel with the model, but the problem that could be faced in this case is which distribution to choose for the noise distribution.

Before we proofed that the gradients of the NCE will converge the gradients of MLE as k grows, which makes us believe that NCE should be the first objective function to use to train language models, and this appears also in the experiments that we did where we can see clearly that the NCE outperformed all the rest of the methods especially when the bias of output of linear layer initialized with log-unigram or using a trainable partition function initialized $\log Z_c = \log |\mathcal{V}|$

6 Bibliography

[1] Gutmann and Hyvarinen. *Noise-contrastive estimation: A new estimation principle for unnormalized statistical models.* 2010.

[2] Yoshua Bengio Jean-Sébastien Senécal. *Adaptive Importance Sampling to Accelerate Training of a Neural Probabilistic Language Model.* 2003.

[3] Matthieu Labeau and Alexandre Allauzen. "Learning with Noise-Contrastive Estimation: Easing training by learning to scale". In: *Proceedings of the 27th International Conference on Computational Linguistics.* Santa Fe, New Mexico, USA: Association for Computational Linguistics, Aug. 2018, pp. 3090–3101. URL: `https://aclanthology.org/C18-1261`.

[4] Tomas Mikolov. *PTB dataset.* URL: `https://github.com/townie/PTB-dataset-from-Tomas-Mikolov-s-webpage/tree/master/data`.

A Importance Sampling Code

```python
class ImportanceSamplingLoss(nn.Module):
    def __init__(self, Q, Ns=500):
        super(ImportanceSamplingLoss, self).__init__()

        # initialize the noise distribution Q
        self.Q = Q

        # initialize number of samples (to be used in approximation)
        self.Ns = Ns

    def forward(self, outputs, targets):

        bs = outputs.size(0)


        rows = torch.tensor( [list(range(bs))] * self.Ns)
        rows = torch.transpose(rows , 0  , 1)

        # sample from the Noise distribution Q
        wi = self.Q.sample(torch.Size([self.Ns * bs]))

        # Get probability for w_i
        q  = self.Q.probs[wi].to(device)

        # Reshape the sample and probability
        wi = wi.reshape(-1,self.Ns)
        q  = q.reshape(-1,self.Ns)

        # Compute Z = Sum_k(exp(s_theata)/q)
        Z = (torch.exp(outputs[rows, wi])/q).mean(-1)

        # Sample from Q
        wi = self.Q.sample(torch.Size([self.Ns * bs]))
        # Get probability of w_i
        q  = self.Q.probs[wi].to(device)

        # Reshape the sample and probability
        wi = wi.reshape(-1,self.Ns)
        q  = q.reshape(-1,self.Ns)

        # Compute the first term in the Sum over k in the loss
        cte  = torch.exp(outputs[rows ,wi])/q

        # Combine the terms inside the sum over k in the loss
        cte1 = (cte.detach() * outputs[rows ,wi]).mean(-1)

        rows = torch.tensor(list(range(bs)))

        # compute the first term of the loss
        out  = outputs[rows,targets]

        # combine all the terms in the loss
        loss = - (out - cte1/Z.detach()).mean()

        return loss, torch.log(Z)
```

B NCE Code

```python
class NCELoss(nn.Module):
    def __init__(self, Q, Ns=100, train_partition = False, log_Z = 0.0):
        super(NCELoss, self).__init__()

        # initialize the noise distribution Q
        self.Q = Q

        # initialize number of samples (to be used in approximation)
        self.Ns = Ns

        # true when we want to train Z as a parameter
        if train_partition:
            self.log_Z = nn.Parameter(torch.tensor([float(log_Z)]))
        else :
            self.log_Z = torch.tensor([0.0]).to(device)

    def forward(self, outputs, targets):

        bs = outputs.size(0)

        outputs = outputs - self.log_Z

        rows = torch.tensor( [list(range(bs))])

        # compute exponential of score function:  p_theta = exp(s_theta)
        exp_out = torch.exp(outputs[rows,targets]).reshape(-1)

        # sample from the Noise distribution Q
        wi = self.Q.sample(torch.Size([ self.Ns * bs ]))

        # Get probability for w_i
        q = self.Q.probs[wi].to(device)

        # Reshape the sample and probability
        wi = wi.reshape(-1,self.Ns)
        q = q.reshape(-1,self.Ns)


        rows = torch.tensor( [list(range(bs))] * self.Ns)
        rows = torch.transpose(rows , 0  , 1)


        # compute the numerator of the first term of loss = kQ(w)
        kP_n = self.Ns * self.Q.probs[targets].to(device)

        # compute the first term of loss = [kQ(w)/ (kQ(w)+p_theta)] * log(p_theta)
        first_term = (kP_n/(kP_n + exp_out )).detach() * torch.log( exp_out )

        # compute the numerator inside the sum of second term of loss
        s_exp = torch.exp(outputs[rows,wi])

        # compute the full last term of the loss = Sum_k (p_theta/(p_theta + k*q))*
    log(p_theta)
        second_term = ( (s_exp/(s_exp + self.Ns * q)).detach()*torch.log(s_exp)).sum
    (-1)

        # combine the first and second terms of loss
        loss = -(first_term - second_term).mean()

        return loss, self.log_Z
```

C Negative Sampling Code

```python
class NEGLoss(nn.Module):
    def __init__(self, Q, Ns=100):
        super(NEGLoss, self).__init__()

        # initialize the noise distribution Q
        self.Q = Q

        # initialize number of samples (to be used in approximation)
        self.Ns = Ns

        # we do not want to compute log_Z
        self.log_Z = torch.tensor([0.0]).to(device)

    def forward(self, outputs, targets):


        bs = outputs.size(0)
        rows = torch.tensor( [list(range(bs))])

        # sample from the Noise distribution Q
        wi = self.Q.sample(torch.Size([self.Ns * bs]))

        # Get probability for w_i
        q  = self.Q.probs[wi].to(device)

        # Reshape the sample and probability
        wi = wi.reshape(-1,self.Ns)
        q  = q.reshape(-1,self.Ns)

        # Compute the first term of Negative sampling loss = log(sigmoid(s_theta))
        loss_first = torch.log(torch.sigmoid(outputs[rows , targets]))

        rows = torch.tensor( [list(range(bs))] * self.Ns)
        rows = torch.transpose(rows, 0, 1)

        # Compute the second term of the loss = sum over k log(sigmoid(-s_theta))
        loss_second = torch.log(torch.sigmoid(-outputs[rows, wi])).sum(-1)

        #combine the 1st and 2nd terms of the loss
        loss = -(loss_first + loss_second).mean()

        return loss, self.log_Z
```