Project Report "Éléments logiciels pour le traitement de données massives"

Chloé Sekkat
ENSAE & ENS Paris-Saclay
chloe.sekkat@ensae.fr

Mathilde Kaploun
ENSAE
mathilde.kaploun@ensae.fr

Abstract

This report presents the work done and the results obtained for the final project of the course "Éléments logiciels pour le traitement de données massives" at ENSAE Paris in 2021/2022. Our project is constructed around the Word2Vec model described by Mikolov et al. [3]. The goal was to code from scratch a naïve version of this model, without major tricks to make it more efficient and faster and then use a framework to build an efficient and distributed Word2Vec thanks to Pytorch. Going from the naïve implementation to one using Pytorch, we were able to divide by 10 the training time, from 31.14 seconds per epoch to 3.10 seconds when trained on a vocabulary of size 63,492.

1. Introduction

The core motivation of this project is compare the gains one can attain by using a distributed version of an algorithm on GPU hardware. First we implemented a naïve version in Python, using Numpy, of the Word2Vec Skip-gram model developed by Mikolov & al. in [3]. And compare this naïve version to a faster and more scalable version, inspired by the work of Ji & al. in [2]. Along the project we deviate a bit from the work and Ji & al. to propose our own implementation using a higher level library: Pytorch. Then we compared the two approaches (naïve vs. distributed) on various levels ranging for computational time and complexity to performances on downstream tasks. The code is fully available on Github [1].

2. Implemented algorithms

In the following section, we will briefly describe the architectures of the models we implemented. These models are based on the Word2Vec Skip-Gram Model developped by Mikolov et al. in [3] & [4].
The core idea of this model is to sample a target word w in the corpus and predict its context words given a window around w. In its simplest version, this model samples $w \in \mathbb{R}^d$, takes its context word $c \in \mathbb{R}^d$ and maximizes $\log(p(c|w))$ (Figure 1). Therefore, given T training words (size of the corpus), one wants to maximize:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-n\leq j\leq n, j\neq 0} \log p\left(c_{t+j} \mid w_t\right) \tag{1}$$

where n is the size of the window, c_i the context word at position i and w_t the target word. The conditional probability of the context word c given the target word w is defined as the softmax function:

$$p\left(c_i|w_i\right) = \frac{\exp\left(v'^{T}_{c_i} v_{w_i}\right)}{\sum_{w=1}^{W} \exp\left(v'^{T}_{w} v_{w_i}\right)} \tag{2}$$

with W the size of the vocabulary and v representing the embedding. As shown by Mikolov et al. in [4], this formulation of the problem is way too costly as the gradients computations are proportional to the number of words in the vocabulary which can grow very fast.

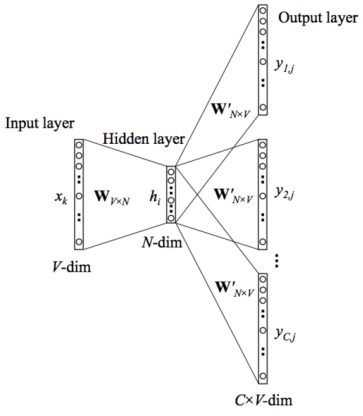

Figure 1. Architecture of the Word2Vec Skip-gram model (source: Wikipedia). Both W and W' must be learned.

Therefore when training a model with this architecture and a simple cross-entropy loss (CE), the training time becomes very long as we will show in Sub-Section 4.2.

[1] https://github.com/chloeskt/word2vec_eltdm

Mikolov et al. in [4] proposed various enhancements to this baseline in order to reduce the complexity of the algorithm. We decided to integrate two of these enhancements in our training.
The first one is the subsampling of frequent words. This is based on the idea that the most frequent words are not the ones conveying the most information, hence could be removed to reduce the size of the vocabulary W and decrease the complexity. Given a threshold t (often set to $t = 1\text{e-}5$), each word w is sampled with probability $1 - \sqrt{\frac{t}{f(w)}}$) where $f(w)$ is the frequency of word w in the corpus. In their paper, Mikolov et al. proved that such a scheme does not deteriorate the quality of the embeddings (when evaluated on downstream tasks).
The second and most important one is the use of the Negative Sampling Loss (NSL). This is the core operation that makes Word2Vec very efficient. The idea is that instead of computing the probability objective over the entire vocabulary W, one can sampled K negative words that are not in the context of the target word w and only use these k negative samples and the context words to compute the objective and backpropagate. Now the objective is a bit different than before: one wants to maximize the probability of (w, c) being a pair (word, context) if needed they are, and maximize the probability of (w, w_k) not to be a pair (word, context) since they are not. This boils down to the following objective:

$$\log \sigma \left(v'^T_{c_t} v_{w_t} \right) + \sum_{i=1}^{k} \mathbb{E}_{c_i \sim P_n(w)} \left[\log \sigma \left(-v'^T_{c_i} v_{w_t} \right) \right] \quad (3)$$

with $P_n(w)$ being a well-chosen noise distribution. In their paper, Mokolov et al. found that the unigram distribution $U(w)$ to the power $\frac{3}{4}$ outperformed all others. The complexity is now of order $\mathcal{O}(d \times k \times T)$ where d is the size of the embedding (300 in our implementation). It does not depend anymore on V.

In our project, we implemented the two following architectures:

- subsampling of frequent words + baseline Word2Vec with CE loss
- subsampling of frequent words + NSL

3. Data

We used the Text8 dataset [2]. This dataset is made of Wikipedia articles in English, from before March 2006 (official dump provided by Wikipedia itself). It consists of only lowercase English characters and spaces provided as a long string. We processed the whole dataset for training, i.e. 10,566,033 words (tokens) and a vocabulary size of 63,492 words.

To process this data, we created a custom DataLoader (manually but inspired of the DataLoader class from Pytorch). Its structure is quite simple, using only lists and Numpy array; its is quite fast but could probably be fine-tuned to be even faster. However it was not the core of this project.

4. Time evaluation

In this section, we want to evaluate the speed of our models. This means that we want to estimate the gain we have when training using GPU and distributed computing.

4.1. Technical specifications

For this project, we used a computer with the following specs:

- Intel(R) Core(TM) i7-10700K CPU@3.80GHz (8 cores/16 threads)
- 32Gb of RAM (DDR4)
- NVIDIA GeForce RTX 3080 (10GB VRAM, 8704 cores@1.71GHz)

The Numpy version of the code only uses the CPUs, and only 1 core while the Pytorch version uses the GPU and all cores.

4.2. Training

This subsection focuses on the training part of the project. We trained four models (notebooks available here):

- Numpy model with Cross-Entropy Loss (referred as Numpy-CE)
- Numpy model with Negative Sampling Loss (referred as Numpy-NSL)
- Pytorch model with Cross-Entropy Loss (referred as Pytorch-CE)
- Pytorch model with Negative-Sampling Loss (referred as Pytorch-NSL)

We evaluated the time taken by each key step of these four models. To do so, we used the LINE-PROFILER library [3]. It allows to profile the time individual lines of code take to execute. We trained the models for 10 epochs each and average the execution times over these 10 epochs.
For each model, we took the following parameters:

- 10% of the text8 dataset

[2] http://mattmahoney.net/dc/textdata.html

[3] https://pypi.org/project/line-profiler/

- remove the words which appeared less than 5 times
- subsampler: each word can be discarded with probability $1 - \sqrt{\frac{t}{f(w)}})$ where t = 1e-5 and $f(w)$ is the frequency of the word w
- batch size of 256
- window size of 5
- $d = 300$, embedding dimension
- Stochastic Gradient Descent for all models, with learning rate 1e-3
- For NSL, $k = 5$ negative samples

All comparisons can be found in Tables 1, 2, 3, 4, 5, 9.

	Numpy CE	Pytorch CE
Loss forward (s per epoch)	0.097	13,12
Loss backward (s per epoch)	153.73	87,79
Model forward (s per epoch)	515.755	32.29
Update weights (s per epoch)	12.61	3,84
Total time per epoch (s)	686.024	143,77

Table 1. Training speed comparison between Numpy CE and Pytorch CE models for 4 key functions in the training loop. Results were obtained using the library LINE-PROFILER.

	Numpy CE	Pytorch CE
Embedding Layer (s per epoch)	0.44	0.176
Linear Layer (s per epoch)	92.974820	24.17
Softmax (s per epoch)	421.97	7.937
Total time per epoch (s)	514.208	32.29

Table 2. Training speed comparison between Numpy CE and Pytorch CE models for the 3 main steps in the forward pass. Results were obtained using the library LINE-PROFILER.

In Table 1, we compare 4 core steps of the training of the Word2Vec with Softmax and Cross-Entropy, both in the naïve Numpy version and the Pytorch one. What is striking is that the distribution of the allocated time per step for one epoch is very different. In Numpy-CE, 75% of the computational time is taken by the forward step of the model (dot product between the matrix of parameters W and the input X and softmax), while the backward step of the loss (i.e. computing the gradients of all parameters) takes 22%. The computation of the cross-entropy loss is quite fast, taking only 0.1% of the overall time (for one epoch i.e. one loop through the entire dataset). For the Pytorch-CE, it is the reverse: the forward pass on the model only takes 24% of the time while the computation of the gradients takes 61% of the time.
This actually makes sense when looking at Table 2. The training time for one epoch is around 4.7 times faster for the Pytorch implementation. The major gain with Pytorch is for the softmax function: the one coded froms scratch using only vectorized arrays in Numpy is 52 times slower. This is due the parallelization of the computations on all the cores of the GPU, when Numpy only uses one core of the CPU. More generally it is also important to highlight that the Embedding structure proposed by Pytorch is way more efficient to retrieve the associated weights to each input than a slicing using Numpy as shown by Table 2 (0.44 seconds vs 0.17). Moreover Pytorch's Linear Layer structure is also faster than the dot product in Numpy.

	Numpy NSL	Pytorch NSL
Loss forward (% time)	3.75	0.15
Loss backward (% time)	12.70	0.70
Model forward input (% time)	0.27	0.03
Model forward output (% time)	0.27	0.02
Model forward noise (% time)	2.59	0.64
Update weights (% time)	2e-06	0.084
Total time per epoch (s)	31.04	3.15

Table 3. Training speed comparison between Numpy NSL and Pytorch NSL models for 6 key functions in the training loop. Results were obtained using the library LINE-PROFILER.

	Numpy NSL	Pytorch NSL
Forward input (s per epoch)	0.27	0.03
Forward output (s per epoch)	0.27	0.02
Foward noise (s per epoch)	2.59	0.64

Table 4. Training speed comparison between Numpy NSL and Pytorch NSL models for the 3 main steps in the forward pass. Results were obtained using the library LINE-PROFILER.

	Numpy NSL	Pytorch NSL
Multinomial sampling (s per epoch)	0.61	0.56
Retrieve noise embeddings (s per epoch)	1.958	0.03
Total time per epoch (s)	2.59	0.64

Table 5. Training speed comparison between Numpy NSL and Pytorch NSL models for the 2 main steps in the noise forward pass. Results were obtained using the library LINE-PROFILER.

	Numpy NSL	Pytorch NSL
Output loss (s per epoch)	0.43	0.05
Noise Loss (s per epoch)	3.23	0.041
Mean (s per epoch)	0.031	0.03
Total time per epoch (s)	3.74	0.15

Table 6. Training speed comparison between Numpy NSL and Pytorch NSL models for the 3 main steps in the loss forward pass (computation of the loss). Results were obtained using the library LINE-PROFILER.

The biggest improvements can be measured through a more complex training scheme: the Word2Vec using Neg-

ative Sampling Loss. The scheme is more complex conceptually but actually aims at reducing the training time by computing the loss only for the context words and the K negative samples. This should both decrease the training time and enhance the quality of the embeddings produced by the algorithms by making the target word and context words closer and pulling them further apart from the K negative samples.

As shown in Table 3, both for Numpy-NSL and Pytorch-NSL the training time has been drastically decreased compared to their CE version. What's more, the Pytorch version is 10 times faster than the Numpy one. The key step which makes the Pytorch version way more faster is the computation of the gradients. Indeed, it takes 12.70s per epoch in Numpy-NSL while only taking .7s in Pytorch-NSL. This is important to note that thanks to autograd, the computations of the gradients are done automatically by Pytorch, under the hood. Whereas in Numpy, one has to manually code the computations required. This leaves room for major improvements, especially since Pytorch uses all the GPU's cores.

In the forward step of the loss computation (Table 9), one can compare the execution time of each line for Numpy-NSL and Pytorch-NSL. It is worth noticing that Numpy implementation of matrix-matrix product (np.matmul) is slower and cannot be done by batch compared to Pytorch's batch matrix-matrix operator (torch.bmm). This makes the computations of the loss more efficient since it can be easily parallelized and can handle 3D arrays/tensors (needed for a straightforward implementation of the Negative Sampling loss).

Again, by looking at Tables 4 and 5, one can see that the data structures and classes provided by Pytorch are more efficient than the coded-from-scratch counterparts. Combined with the use of a GPU and the parallelization scheme, we can reach faster training (with also bigger/more complex models).

We would like to highlight that even though not represented in this report, the same analysis was conducted using only the CPU and the Pytorch versions of the models were always faster (around 3 to 5 times faster).

More extensive speed comparisons, for each key function, can be found in the source code in txt format.

5. Performance evaluation

5.1. Selected models

We use a set of hyperparameters inspired from [2], who use the parameter settings of BIDMatch.

As for the learning rates we have chosen them based on empirical performance, as the ones used in [2] did not perform as well in our context.

	Hyperparameters
Embedding size	300
Window	5
Negative Sample Size	5
Vocabulary size	1,115,011 words

Table 7. Common BIDMatch Hyperparameters

	Learning Rate
Numpy CE	0.001
Pytorch CE	0.003
Numpy NSL	0.03
Pytorch NSL	0.003

Table 8. Learning rates of best-performing models

5.2. Qualitative evaluation

In order to evaluate qualitatively the performance of our models, we are going to test whether they embed words considered synonyms in the wordnet dataset closer than random words. Very simply we take the average cosine similary between a word's embedding and its synonym's embedding through one of the models. Then we compare this average similarity with the one we found with random words. If our model place synonyms closer than random words, they have learned something.

We cannot really evaluate the absolute quality of the embedding this way, only tell whether they are completely random or not. However we can compare our two models. For this comparison we are only going to use the best performing Pytorch and Numpy model respectively in order to give them a fair evaluation.

	synonyms	random
Pytorch	0.116991	0.020535
Numpy	0.117375	0.020560

Table 9. Average cosine similarity between synonyms and random words for the best performing Pytorch and Numpy models

From these results we can see that both models have learned something. Unsurprisingly, they present very similar performance. Seeing as they implement the same algorithm and have been trained on the exact same dataset, this was to be expected. The difficulty of implementing gradient descent through Numpy (compared to just using the autograd in Pytorch) doesn't seem to have affected the quality of the results at all.

6. Conclusion

To conclude, leveraging the automatic parallelization and full use of multi-core GPU implemented under the hood by Pytorch, allows us to reach a ten times faster training speed compared to our Numpy implementation of Word2Vec. The use of Negative Sampling Loss improves

for both models the learning speed as well as the final performance.

However this difference in implementation (Numpy vs Pytorch) doesn't affect the quality of predictions since both models perform similarly with regards to the distance between synonyms and random words.

Further speed improvements could be reached through asynchronous training as developed by Anand & al [1].

References

[1] Avishek Anand, Megha Khosla, Jaspreet Singh, Jan-Hendrik Zab, and Zijian Zhang. Asynchronous training of word embeddings for large text corpora, 2018. 5

[2] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. Parallelizing word2vec in shared and distributed memory, 2016. 1, 4

[3] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. 1

[4] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality, 2013. 1, 2