ADVANCED LEARNING FOR TEXT AND GRAPH DATA

Lab session 1: word embeddings

Lecture: Prof. Michalis Vazirgiannis
Lab: Moussa Kamal Eddine and Hadi Abdine

Tuesday, November 17, 2020

This handout includes theoretical introductions, coding tasks and questions. Before the deadline, you should submit here a **.zip** file (max 10MB in size) containing a `/code/` folder (itself containing your scripts with the gaps filled) and an answer sheet named `firstname_lastname.pdf`, following the template available here, and containing your answers to the questions. Your answers should be well constructed and well justified. They should not repeat the question or generalities in the handout. When relevant, you are welcome to include figures, equations and tables derived from your own computations, theoretical proofs or qualitative explanations. **One submission is required for each student. The deadline for this lab is November 24, 2020 11:59 PM**. No extension will be granted. Late policy is as follows: $]0, 24]$ hours late $\rightarrow$ -5 pts; $]24, 48]$ hours late $\rightarrow$ -10 pts; > 48 hours late $\rightarrow$ not graded (zero).

---

1 Introduction

Compared to traditional machine learning approaches (e.g., TF-IDF + SVM) that consider words and combinations of them as unique dimensions of the feature space, deep learning models *embed* words as vectors in a low-dimensional continuous space where dimensions represent shared latent concepts. The main advantages of this approach are (1) the ability to capture the similarity between words and thus to share predictive power between them; and (2) escaping the curse of dimensionality. Word embeddings can be initialized randomly and learned during training, or be pre-trained. In NLP, pre-trained word vectors obtained with `word2vec` from very large corpora are often used. The pre-trained word vectors can then be updated during training, or be kept static.

2 Learning objective

In this lab, you will learn about one of the two `word2vec` variants: skip-gram with negative sampling. We will derive and implement the model by hand using NumPy, and train it on a subset of the Internet Movie Database (IMDB) dataset[1].
Readings: the original paper introducing skip-gram (along with the other `word2vec` variant, CBOW) is [4]. Negative sampling was proposed in a follow-up paper [5]. Helpful resources to understand how skip-gram works are [2] and subsection 3.1 of [1].

[1] `http://ai.stanford.edu/~amaas/data/sentiment/`

3 Preprocessing

Before sampling training examples, the raw data need to be cleaned. Then, each document has to be converted into a list of integers. The integers correspond to indexes in a vocabulary (dictionary) in which the most frequent word has index 1 and only the words that appear a least a certain number of times in the dataset are kept. Index 0 is reserved for out-of-vocabulary words.

Task 1
Fill the gaps in the `preprocessing.py` script to perform the steps described above.

4 Training example sampling

Skip-gram is trained on the artificial task of *context prediction*. The word vectors are learned as a *side-effect* of training (they are actually the parameters of the model). More precisely, pairs of target and context words $(t, \mathcal{C}_t)$ are sampled by sliding a window over the corpus $\{0, 1, \ldots, T\}$. For a given instantiation of the window, the word in the middle is the target word t, and the words surrounding it compose the set of positive examples $\mathcal{C}_t$. In practice, the window size is not fixed but uniformly sampled in $[1, \texttt{max\_window\_size}]$.

Task 2
Fill the first gaps in the `sample_examples()` function of the `main.py` script to perform the aforementioned steps.

5 Objective

Skip-gram is then trained to assign high probabilities to the words in $\mathcal{C}_t$, given t, i.e., to predict well the context of a given target word. This translates into the following log-likelihood:

$$\arg\max_{\theta} \sum_{t=0}^{T} \sum_{c \in \mathcal{C}_t} \log p(c|t;\theta) \tag{1}$$

The set of parameters of the model, θ, contains two matrices of word vectors, say W_t and W_c, from which are drawn the vectors of the words when they are used as targets and contexts, respectively. If we denote by V the vocabulary (unique words), and by d the dimensionality of the word embedding space, we can assume that W_t and W_c live respectively in $\mathbb{R}^{|V| \times d}$ and $\mathbb{R}^{d \times |V|}$. These two matrices are often referred to as input and output matrices, and it is common practice to use the input matrix, after training, as the final word embeddings.

Softmax vs. negative sampling. The predictions $p(c|t;\theta)\ \forall c \in \mathcal{C}_t$ are given by looking at the entries of the vector in $\mathbb{R}^{1 \times |V|}$ obtained by passing the vector $w_t \in \mathbb{R}^d$ of the target word to a simple linear layer parameterized by W_c. That is, by multiplying w_t with matrix W_c.[2] Further, to ensure that legitimate probabilities are obtained, we normalize with a softmax:

$$p(c|t;\theta) = \frac{e^{w_c \cdot w_t}}{\sum_{v \in V} e^{w_v \cdot w_t}} \tag{2}$$

[2]This operation is fully linear: no nonlinear activation function is applied (there is no hidden layer). Because of this, skip-gram cannot really be considered to be a deep learning model. The term shallow neural network, or *log-linear model*, is more appropriate. The same holds for CBOW.

However, this approach is expensive because of the large size of the vocabulary V (10^5 or 10^6).

Negative sampling trick: learning to discriminate. Rather than trying to assign a probability to each single word in the vocabulary, another approach consists in performing independent discrimination tasks. In this scenario, we teach the model to distinguish between words from the true context $\mathcal{C}_t^+$ of a given target word and negative examples $\mathcal{C}_t^-$, i.e., words that are sampled at random from the vocabulary:

$$\arg\max_\theta \sum_{t=1}^{T} \Big(\sum_{c \in \mathcal{C}_t^+} \log p(c|t;\theta) + \sum_{c \in \mathcal{C}_t^-} \log\left(1 - p(c|t;\theta)\right) \Big) \tag{3}$$

We now just have independent binary classification tasks to perform. This comes with a major computational advantage: instead of multiplying w_t with the full matrix $W_c \in \mathbb{R}^{d \times |V|}$ (where $|V|$ can get very large), we now just have to compute a few dot products between w_t and the words in $\mathcal{C}_t^+ \cup \mathcal{C}_t^-$. Assuming that the labels (pos/neg) are indicated by ± 1, the individual predictions can be obtained with the sigmoid function $\sigma(x) = \frac{1}{1+e^{-w_c \cdot w_t}}$. Plugging this in Eq. 3, and using the fact that $1 - \sigma(x) = \sigma(-x)$, we obtain the loss:

$$\arg\min_\theta \sum_{t=1}^{T} \Big(\sum_{c \in \mathcal{C}_t^+} \log\left(1 + e^{-w_c \cdot w_t}\right) + \sum_{c \in \mathcal{C}_t^-} \log\left(1 + e^{w_c \cdot w_t}\right) \Big) \tag{4}$$

Note that in practice, the negative examples are not sampled uniformly but proportionally to their dampened frequency (square root of their frequency, to be exact) [1].

Task 3
Fill the second gap in the `sample_examples()` function of the `main.py` script to implement negative example sampling.

Task 4
Fill the gaps in the `compute_loss()` function of the `main.py` script to compute the loss **for one training example**.

6 Computing the gradient

Skip-gram is trained with SGD. We thus need to compute the gradient to perform updates at each training iteration. Let us consider our loss for a given training example $(t, \mathcal{C}_t^+, \mathcal{C}_t^-)$:

$$L(t, \mathcal{C}_t^+, \mathcal{C}_t^-) = \sum_{c \in \mathcal{C}_t^+} \log\left(1 + e^{-w_c \cdot w_t}\right) + \sum_{c \in \mathcal{C}_t^-} \log\left(1 + e^{w_c \cdot w_t}\right) \tag{5}$$

L is a function of the target word t and of all context words, i.e., all words in $\mathcal{C}_t^+ \cup \mathcal{C}_t^-$.

Question 1 (3 points)
Compute the partial derivatives of the loss w.r.t one positive example and one negative example, $\frac{\partial L}{\partial w_{c^+}}$ and $\frac{\partial L}{\partial w_{c^-}}$.

Correction

In Eq. 5, w_{c^+} appears in one single term of the first sum: $\log\left(1 + e^{-w_{c^+} \cdot w_t}\right)$. Using the chain rule, the partial derivative of this w.r.t. w_{c^+} (holding w_t constant) is:

$$\frac{\partial L}{\partial w_{c^+}} = \frac{1}{1 + e^{-w_{c^+} \cdot w_t}} \times -w_t e^{-w_{c^+} \cdot w_t} = \frac{-w_t}{e^{w_{c^+} \cdot w_t} + 1} \tag{6}$$

Similarly, for a negative example $w_{c^-} \in \mathcal{C}_t^-$, we obtain:

$$\frac{\partial L}{\partial w_{c^-}} = \frac{1}{1 + e^{w_{c^-} \cdot w_t}} \times w_t e^{w_{c^-} \cdot w_t} = \frac{w_t}{e^{-w_{c^-} \cdot w_t} + 1} \tag{7}$$

Question 2 (3 points)
Compute the partial derivative of the loss w.r.t. the target word, $\frac{\partial L}{\partial w_t}$.

Correction

We note that in Eq. 5, w_t appears in each term of each sum. We thus obtain:

$$\frac{\partial L}{\partial w_t} = \sum_{c \in \mathcal{C}_t^+} \frac{-w_c}{e^{w_c \cdot w_t} + 1} + \sum_{c \in \mathcal{C}_t^-} \frac{w_c}{e^{-w_c \cdot w_t} + 1} \tag{8}$$

Interpretation: the vector of each context word in W_c is updated based on w_t only, while the vector of the target word in W_t is updated based on all context words.

Task 5
Fill the gaps in the `compute_gradients()` function of the `main.py` script to compute the partial derivatives for one training example.

7 Performing updates

The model is trained via stochastic gradient descent. The representation of each word is updated based on its corresponding partial derivative. For instance, for a positive word at iteration n, we perform the following update:

$$w_{c^+_{n+1}} = w_{c^+_n} - \gamma_n \frac{\partial L}{\partial w_{c^+_n}} \tag{9}$$

where γ_n is the learning rate at iteration n. Note that we use γ_n and not γ because the learning rate is not fixed, but annealed during training. The schedule we use is $\gamma_n = \frac{\gamma_0}{1 + \text{decay} \times n}$, where $\text{decay} = 10^{-6}$.

Task 6
Fill the gaps in the nested for loop of the `main.py` script to perform the updates.

8 Sanity checking

Task 7
Train the model for several epochs. Monitor your loss to verify that the model learns. Then, fill the gaps at the end of the `main.py` script to compute some similarities and visualize the learned embeddings.

Question 3 (4 points)
Observe and interpret your similarity values and your plot. What can you say about the embedding space?

Correction
In terms of cosine similarity, semantically close words have vectors that are more similar than unrelated words. For example, `cosine(movie,film)` = 0.99 (almost equal vectors) while `cosine(movie,banana)` = 0.08 (almost orthogonal vectors).

By visualizing the embedding space, we can see that words are grouped based on semantic relatedness (synonymy), e.g., "woman", "women", "wife", "mother", but also based on syntactic relatedness (same role), e.g., "also", "instead", "would", "could", "least", "worst", etc. Some words are also grouped together because they very often follow each other in the corpus, e.g., "special" and "effects".

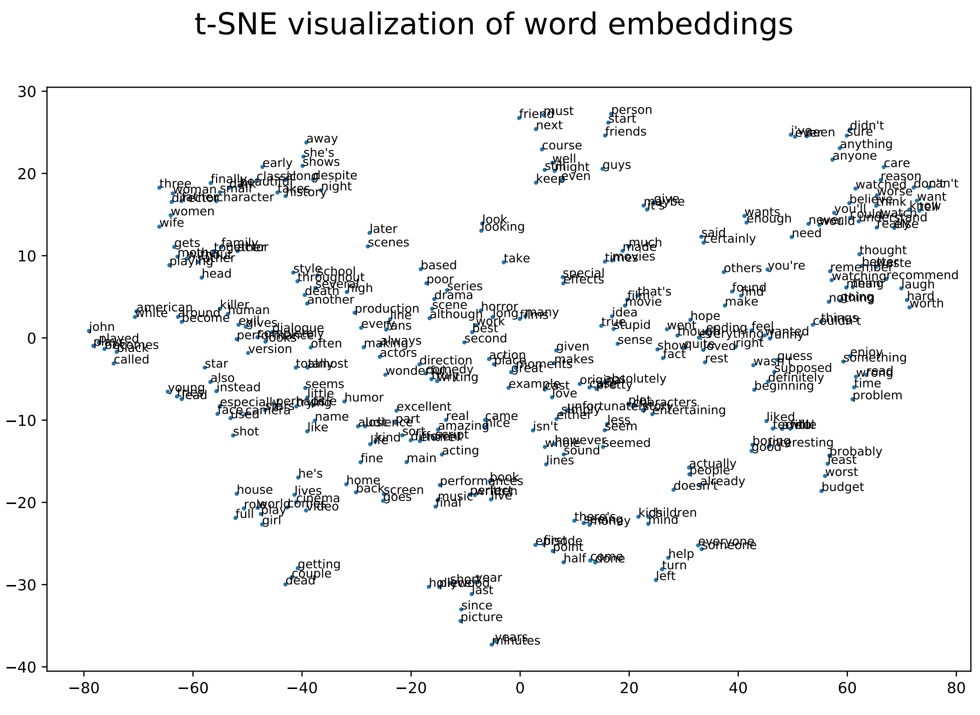

Figure 1: t-SNE visualization of the word embedding space.

---

9 Follow-up

Question 4 (10 points)
What changes should we make to our pipeline (preprocessing and training) to learn document vectors jointly with the word vectors? Base your answer on [3].

Correction

There are several ways to do that. One option is to replace the input matrix W_t by a document vector matrix W_d, and to predict the context simply from the document vectors. This is the *distributed bag-of-words* variant proposed in [3]. The W_c matrix, after training, can be used as the word embeddings.

Another option consists in having both W_t and W_d, and for each window, using the target word vector and document vector in turn to predict the context. This approach is implemented in gensim[3]. While it takes more time to run, the word embeddings learned via this approach (W_t, or even W_c), might be of better quality as word vectors are compared against word vectors (whereas, in the previous approach, word vectors are only compared to document vectors).

Note that with the two approaches above, word vectors are shared across all documents in the corpus, whereas each document vector is only used, of course, for the windows sampled from the corresponding document. So, during training, document vectors learn sort of a 'summary' of their documents. Also, in both cases, word and document vectors live in the same space, which provides a nice way to sanity check the results.

In terms of preprocessing, the only required change is that we need to keep track of the document each window was sampled from. This can be taken care of when sampling the windows at the beginning of each epoch (by modifying the `sample_examples()` function).

References

[1] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[2] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[3] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[4] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[5] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[3]`https://stackoverflow.com/questions/44011706/what-is-different-between-doc2vec-models-when-the`
`44013893`