Natural Language Processing **Tel Aviv University**

Assignment 1: Word Vectors

Due Date: *November 21, 2019* Lecturer: Jonathan Berant

0 Preliminaries

Submission Instructions Submit your solution through Moodle. Your submission should consist of a single zip file named `<id1>_<id2>.zip` (where `id1` refers to the ID of the first student). This zip file should include the code and data necessary for running the tests provided out-of-the-box, as well as a written solution, the `.ipynb` notebook from sections 1 and 3, and the generated files `saved_params_40000.npy` and `word_vectors.png`. Only one student needs to submit. If you are comfortable with LaTeX, feel free to use the supplied `written_solution_template.tex` as a basis for you written solution.

Your code will be tested on the School of Computer Science operating system, installed on nova and other similar machines. Please make sure your code runs there. If your code does not run on nova, you code will not be graded.

Acknowledgements This assignment was adapted from Stanford's CS224n course. Their contributions are greatly appreciated.

1 Count-Based Word Vectors

In this question we will use a Google Colab Notebook, which is essentially a Jupyter Notebook 'in the cloud'. Jupyter notebooks are a common and useful tool in the data science community, and Colab Notebooks add free GPU usage on top of them! If you have yet to encounter Jupyter Notebooks, here is a quick introduction: `https://www.youtube.com/watch?v=HW29067qVWk`.

To access the notebook, go to `https://drive.google.com/open?id=1Pw0LUuyzDE03_RZqjfgSaGq4IAcN-Qog`. Notice that this is a read-only version, so you'll need to copy it to your drive (`File` → `Save a copy in Drive`) and start your work from your copy. **Please do not add or delete any cells from the notebook**. For this question, please complete **Part 1: "Count-Based Word Vectors"**. The cells preceding Part 1 include the notebook's submission instructions, a quick motivation for using word vectors, and a cell that includes all the required import statements.

2 Understanding word2vec

Let's have a quick refresher on the `word2vec` algorithm. The key insight behind `word2vec` is that '*a word is known by the company it keeps*'. Concretely, suppose we have a 'center' word c and a contextual window surrounding c. We shall refer to words that lie in this contextual window as 'outside words'. For example, in Figure 1 we see that the center word c is 'banking'. Since the context window size is 2, the outside words are 'turning', 'into', 'crises', and 'as'.

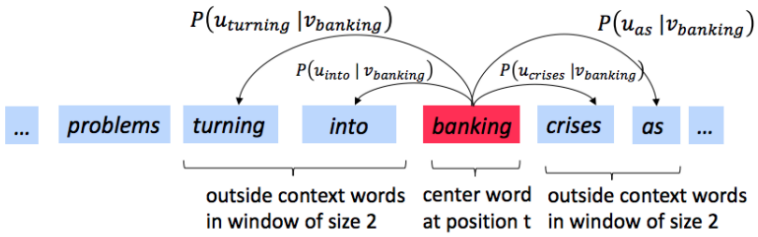

Figure 1: The word2vec skip-gram prediction model with window size 2

The goal of the skip-gram `word2vec` algorithm is to accurately learn the probability distribution $P(O \mid C)$. Given a specific word o and a specific word c, we want to calculate $P(O = o \mid C = c)$, which is the probability that word o is an 'outside' word for c, i.e., the probability that o falls within the contextual window of c. In word2vec, the conditional probability distribution is given by taking vector dot-products and applying the softmax function:

$$P(O = o \mid C = c) = \frac{\exp\left(\boldsymbol{u}_o^\intercal \boldsymbol{v}_c\right)}{\sum_{w \in \mathrm{W}} \exp\left(\boldsymbol{u}_w^\intercal \boldsymbol{v}_c\right)} \tag{1}$$

Here, W is the vocabulary, $\boldsymbol{u}_o$ is the 'outside' vector representing outside word o, and $\boldsymbol{v}_c$ is the 'center' vector representing center word c. To contain these parameters, we have two matrices, $\boldsymbol{U}$ and $\boldsymbol{V}$. The columns of $\boldsymbol{U}$ are all the 'outside' vectors $\boldsymbol{u}_w$. The columns of $\boldsymbol{V}$ are all of the 'center' vectors $\boldsymbol{v}_w$. Both $\boldsymbol{U}$ and $\boldsymbol{V}$ contain a vector for every $w \in W$[1].

Recall from lectures that, for a single pair of words c and o, the loss is given by:

$$\boldsymbol{J}_{\text{naive-softmax}}(\boldsymbol{v}_c, o, \boldsymbol{U}) = -\log P(O = o \mid C = c) \tag{2}$$

Another way to view this loss is as the cross-entropy[2] between the true distribution $\boldsymbol{y}$ and the predicted distribution $\hat{\boldsymbol{y}}$. Here, both $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are vectors with length equal to the number of words in the vocabulary ($|W|$). Furthermore, the k^{th} entry in these vectors indicates the conditional probability of the k^{th} word being an 'outside word' for the given c. The true empirical distribution $\boldsymbol{y}$ is a one-hot vector with a 1 for the true outside word o, and 0 everywhere else. The predicted distribution $\hat{\boldsymbol{y}}$ is the probability distribution $P(O \mid C = c)$ given by our model in Equation (1).

(a) Equation (1) uses the softmax function. Prove that softmax is invariant to constant offset in the input, i.e prove that for any input vector $\boldsymbol{x}$ and any constant c,

$$\text{softmax}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} + c)$$

where $\boldsymbol{x} + c$ means adding the constant c to every dimension of $\boldsymbol{x}$. Remember that

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp\left(x_i\right)}{\sum_j \exp\left(x_j\right)}$$

(b) Show that the naive softmax loss given in Equation (2) is the same as the cross-entropy loss between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$, i.e., show that

$$-\sum_{w \in \mathrm{W}} y_w \log(\hat{y}_w) = -\log(\hat{y}_o) \tag{3}$$

Your answer should be one line.

[1] Assume that every word in our vocabulary is matched to an integer number k. $\boldsymbol{u}_k$ is both the k^{th} column of $\boldsymbol{U}$ and the 'outside' word vector for the word indexed by k. $\boldsymbol{v}_k$ is both the k^{th} column of $\boldsymbol{U}$ and the 'center' word vector for the word indexed by k. **In order to simplify notation we shall interchangeably use k to refer to the word and the index-of-the-word**.

[2] The Cross Entropy Loss between the true (discrete) probability distribution p and another distribution q is $-\sum_i p_i \log(q_i)$.

(c) Compute the partial derivative of $\boldsymbol{J}_{\text{naive-softmax}}(\boldsymbol{v}_c, o, \boldsymbol{U})$ with respect to $\boldsymbol{v}_c$. Please write you answer in terms of $\boldsymbol{y}$, $\hat{\boldsymbol{y}}$ and $\boldsymbol{U}$.

(d) Compute the partial derivative of $\boldsymbol{J}_{\text{naive-softmax}}(\boldsymbol{v}_c, o, \boldsymbol{U})$ with respect to each of the 'outside' word vectors, $\boldsymbol{u}_w$'s. There will be two cases: when $w = o$, the true 'outside' word vector, and when $w \neq o$, for all other words. Please write you answer in terms of $\boldsymbol{y}$, $\hat{\boldsymbol{y}}$ and $\boldsymbol{v}_c$.

(e) The sigmoid function is given by

$$\sigma(\boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{x})} = \frac{\exp(\boldsymbol{x})}{\exp(\boldsymbol{x})+1}$$

Please compute the derivative of $\sigma(\boldsymbol{x})$ with respect to $\boldsymbol{x}$ where $\boldsymbol{x}$ is a vector. Try to express it in terms of $\sigma(\boldsymbol{x})$ and constants only.

(f) Now we shall consider the negative sampling loss, which is an alternative to the naive softmax loss. Assume that K negative samples (words) are drawn from the vocabulary W. For simplicity of notation we shall refer to them as $w_1, w_2, \ldots, w_K$ and to their corresponding outside vectors as $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_K$. Note that $o \notin \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_K\}$. For a center word c and an outside word o, the negative sampling loss function is given by:

$$\boldsymbol{J}_{\text{neg-sample}}(\boldsymbol{v}_c, o, \boldsymbol{U}) = -\log(\sigma(\boldsymbol{u}_o^\intercal \boldsymbol{v}_c))) - \sum_{k=1}^{K} \log(\sigma(-\boldsymbol{u}_k^\intercal \boldsymbol{v}_c))$$

for a sample $w_1, w_2, \ldots, w_K$, where $\sigma(\cdot)$ is the sigmoid function[3].

Please repeat parts (b) and (c), computing the partial derivatives of $\boldsymbol{J}_{\text{neg-sample}}$ with respect to $\boldsymbol{v}_c$, with respect to $\boldsymbol{u}_o$, and with respect to a negative sample $\boldsymbol{u}_k$. Please write your answers in terms of the vectors $\boldsymbol{u}_o$, $\boldsymbol{v}_c$ and $\boldsymbol{u}_k$, where $k \in [1, K]$. After you've done this, describe with one sentence why this loss function is much more efficient to compute than the naive softmax loss. Note, you should be able to use your solution of (e) to help compute the necessary gradients here.

(g) Suppose the center word is $c = w_t$ and the context window is $[w_{t-m}, \ldots, w_{t-1}, w_t, w_{t+1}, \ldots, w_{t+m}]$, where m is the context window size. Recall that for the skip-gram version of `word2vec`, the total loss for the context window is:

$$\boldsymbol{J}_{\text{skip-gram}}(\boldsymbol{v}_c, w_{t-m}, \ldots, w_{t+m}, \boldsymbol{U}) = \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \boldsymbol{J}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U})$$

Here, $\boldsymbol{J}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U})$ represents an arbitrary loss term for the center word $c = w_t$ and outside word w_{t+j}. $\boldsymbol{J}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U})$ could be $\boldsymbol{J}_{\text{naive-softmax}}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U})$ or $\boldsymbol{J}_{\text{neg-sample}}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U})$, depending on your implementation.

Write down three partial derivatives:

(i) $\partial \boldsymbol{J}_{\text{skip-gram}}(\boldsymbol{v}_c, w_{t-m}, \ldots, w_{t+m}, \boldsymbol{U})/\partial \boldsymbol{U}$

(ii) $\partial \boldsymbol{J}_{\text{skip-gram}}(\boldsymbol{v}_c, w_{t-m}, \ldots, w_{t+m}, \boldsymbol{U})/\partial \boldsymbol{v}_c$

(iii) $\partial \boldsymbol{J}_{\text{skip-gram}}(\boldsymbol{v}_c, w_{t-m}, \ldots, w_{t+m}, \boldsymbol{U})/\partial \boldsymbol{v}_w$ when $w \neq c$

Write your answers in terms of $\partial \boldsymbol{J}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U})/\partial \boldsymbol{U}$ and $\partial \boldsymbol{J}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U})/\partial \boldsymbol{v}_c$. This is very simple, each solution should be one line.

Once you're done: *Given that you computed the derivatives of $\boldsymbol{J}(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U})$ with respect to all the model parameters $\boldsymbol{U}$ and $\boldsymbol{V}$ in parts (a) to (c), you have now computed the derivatives of the full loss function $\boldsymbol{J}_{skip\text{-}gram}$ with respect to all parameters. You're ready to implement* `word2vec`*!*

[3]The loss function here is the negative of what Mikolov et al. had in their original paper, because we are minimizing rather than maximizing in our assignment code. Ultimately, this is the same objective function.

3 Implementing word2vec

In this part you will implement the word2vec model and train your own word vectors with stochastic gradient descent (SGD). Before you begin, first run the following commands within the assignment directory in order to create the appropriate conda virtual environment. This guarantees that you have all the necessary packages to complete the assignment.

```
conda env create --file env.yml
conda activate nlp-hw1
```

Once you are done with the assignment you can deactivate this environment by running:

```
conda deactivate
```

Note: you are not required to use conda in this assignment, but the usage of virtual environment managers (such as conda) is highly recommended and is standard practice in both the industry and the academia. If you are unfamiliar with conda, check out `https://docs.conda.io/projects/conda/en/latest/user-guide/getting-started.html`.

To start, download the data and supporting code from the following link: `https://drive.google.com/open?id=1vFW0jr6Mk4CSSwTZDArATRoox2ZwwBiG`.

(a) Implement the `softmax` function in the module `q3a_softmax.py`. Note that in practice, for numerical stability, we make use of the property we proved in question 2.a and choose $c = -\max_i x_i$ when computing softmax probabilities (i.e., subtracting its maximum element from all elements of x). You can test your implementation by running `python q3a_softmax.py`.

Note: The provided tests are not exhaustive. Later parts of the assignment will reference this code so it is important to have a correct implementation. Your implementation should also be efficient and vectorized whenever possible (i.e., use numpy matrix operations rather than `for` *loops). A non-vectorized implementation will not receive full credit!*

(b) To make debugging easier, we will now implement a gradient checker. Fill in the implementation for the `gradcheck_naive` function in the module `q3b_gradcheck`. You can test your implementation by running `python q3b_gradcheck.py`.

(c) Fill in the implementation for `naive_softmax_loss_and_gradient`, `neg_sampling_loss_and_gradient`, and `skipgram` in the module `q3c_word2vec.py`. You can test your implementation by running `python q3c_word2vec.py`. Verify that your results are approximately equal to the expected results.

(d) Complete the implementation for the SGD optimizer in the module `q3d_sgd.py`. You can test your implementation by running `python q3d_sgd.py`.

(e) Show time! Now we are going to load some real data and train word vectors with everything you just implemented! We are going to use the Stanford Sentiment Treebank (SST) dataset to train word vectors, and later apply them to a simple sentiment analysis task. There is no additional code to write for this part; just run `python q3e_run.py`.

Note: The training process may take a long time depending on the efficiency of your implementation. Plan accordingly!

After 40,000 iterations, the script will finish and a visualization for your word vectors will appear. It will also be saved as `word_vectors.png` in your project directory. **Include the plot in your homework write up**. Briefly explain what you notice in the plot. Are there any reasonable clusters/trends? Are the word vectors as good as you expected? If not, what do you think could make them better?

4 Exploring word2vec Embeddings

Return to the colab notebook from Section 1 and complete "Part 2: Prediction-Based Word Vectors". Note that this notebook does not use the word vectors you computed in Section 3, but different pre-trained word vectors which were trained for longer, and on much more data.