Advanced Methods in NLP Lecturer: Jonathan Berant

Home Assignment 1: Word Vectors

Due Date: *11:00pm, March 26, 2019*

In this home assignment we will implement the skip-gram model with negative sampling.

Download the data and supporting code from the following link: `https://drive.google.com/open?id=1OYIp7zQcM-8XDEtqG1W4WXv8ES-2dwTp`. To set up the environment, follow the instructions in `https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/assignment1/index.html`. (we adapted this home assignment from that class). The dataset is already there.

Submit your solution through Moodle, create a zip file named `<id1>_<id2>_<id3>.zip` (where id1 refers to the ID of the first student). This zip file for the submission should include the code and data necessary for running the tests provided out-of-the-box, as well as a written solution. Only one student needs to submit.
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 will not get points.

1 Basics

(a) Prove that softmax is invariant to constant offset in the input, that is, for any input vector $\boldsymbol{x}$ and any constant c,

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

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

$$softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Note: In practice, we make use of this property and choose $c = -max_i x_i$ when computing softmax probabilities for numerical stability (i.e., subtracting its maximum element from all elements of x).

(b) Given an input matrix of N rows and D columns, compute the softmax prediction for each row using the optimization in part (a). Write your implementation in `q1b_softmax.py`. You may test by executing `q1b_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!

(c) Derive the gradients of the sigmoid function and show that it can be rewritten as a function of the function value (i.e., in some expression where only $\sigma(x)$, but not x, is present). Assume that the input x is a scalar for this question. Recall, the sigmoid function is:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

(d) Implement the sigmoid function in `q1d_sigmoid.py` and its gradient, and test your code by running `python q1d_sigmoid.py`.

(e) To make debugging easier, we will now implement a gradient checker. Fill in the implementation for gradcheck_naive in `q1e_gradcheck.py`. Test your code using python `q1e_gradcheck.py`.

2 Word2vec

(a) Assume you are given a predicted word vector $\boldsymbol{v_c}$ corresponding to the center word c for skipgram, and the word prediction is made with the `softmax` function

$$\boldsymbol{\hat{y}_o} = p(o|c) = \frac{\exp(\boldsymbol{u_o^\intercal v_c})}{\sum_{w=1}^{W} \exp(\boldsymbol{u_w^\intercal v_c})}$$

where o is the expected word, w denotes the w-th word and u_w $(w = 1, ..., W)$ are the "output" word vectors for all words in the vocabulary.
Let's define the cross entropy function as:

$$\mathrm{J}_{\text{softmax-CE}}(o, v_c, U) = \mathrm{CE}(\boldsymbol{y}, \boldsymbol{\hat{y}}) = -\sum_i y_i \cdot \log(\hat{y}_i)$$

where the gold vector $\boldsymbol{y}$ is a one-hot vector, the softmax prediction vector $\boldsymbol{\hat{y}}$ is a probability distribution over the output space, and $\boldsymbol{U} = [u_1, u_2, ..., u_W]$ is the matrix of all the output vectors. Assume cross entropy cost is applied to this prediction, derive the gradients with respect to $\boldsymbol{v_c}$.

(b) As in the previous part, derive gradients for the "output" word vectors u_w (including u_o).

(c) Repeat part (a) and (b) assuming we are using the negative sampling loss for the predicted vector $\boldsymbol{v_c}$. Assume that K negative samples (words) are drawn, and they are $1, \ldots, K$, respectively. For simplicity of notation, assume $(o \notin \{1, \ldots, K\})$. Again, for a given word, o, denote its output vector as u_o. The negative sampling loss function in this case is:

$$\mathrm{J}_{\text{neg-sample}}(o, v_c, U) = -\log(\sigma(\boldsymbol{u_o^\intercal v_c})) - \sum_{k=1}^{K} \log(\sigma(-\boldsymbol{u_k^\intercal v_c}))$$

where σ is the sigmoid function defined in 1(c).

(d) Derive gradients for all of the word vectors for skip-gram given the previous parts and given a set of context words $[\text{word}_{c-m}, \dots, \text{word}_c, \dots, \text{word}_{c+m}]$ where m is the context size. Denote the "input" and "output" word vectors for $word_k$ as $\boldsymbol{v_k}$ and $\boldsymbol{u_k}$ respectively.

Hint: feel free to use $F(o, \boldsymbol{v_c})$ (where o is the expected word) as a placeholder for the $J_{softmax\text{-}CE}(o, v_c \dots)$ or $J_{neg\text{-}sample}(o, v_c \dots)$ cost functions in this part – you'll see that this is a useful abstraction for the coding part. That is, your solution may contain terms of the form $\frac{\partial F(o, \boldsymbol{v_c})}{\partial \dots}$ Recall that for skip-gram, the cost for a context centered around c is:

$$\sum_{-m \le j \le m, j \ne 0} F(w_{c+j}, v_c)$$

(e) In this part you will implement the word2vec models and train your own word vectors with stochastic gradient descent (SGD). First, write a helper function to normalize rows of a matrix in `q2e_word2vec.py`. In the same file, fill in the implementation for the softmax and negative sampling cost and gradient functions. Then, fill in the implementation of the cost and gradient functions for the skip-gram model. When you are done, test your implementation by running `python q2e_word2vec.py`.

(f) Complete f implementation for your SGD optimizer in `q2f_sgd.py`. Test your implementation by running `python q2f_sgd.py`.

(g) In this part you will implement the k-nearest neighbors algorithm, which will be used for analysis. The algorithm receives a vector, a matrix and an integer k, and returns k indices of the matrix's rows that are closest to the vector. Use the cosine similarity as a distance metric (`https://en.wikipedia.org/wiki/Cosine_similarity`). Fill the implementation of the algorithm in `q2g_knn.py`.

Note: for this part a naive implementation will be accepted.

(h) 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. There is no additional code to write for this part; just run `python run.py`.

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

When the script finishes, a visualization for your word vectors will appear. It will also be saved as `word_vectors.png` in your project directory. In addition, the script should print the nearest neighbors of a few words (using the knn function you implemented in 2(g)). Include the plot and the nearest neighbors lists in your homework write up, and briefly explain those results.

3 Anologies

This part will use the following notebook as a utility.

`https://colab.research.google.com/drive/1L8tstTTUeDsL3lrdkNuP6tMAL_dwQS7G`

The notebook loads a model that was trained on a large dataset. We can therfore ask questions that relate to its performance.

(a) A polysemous word is a word that has more than one meaning. Find a polysemous word (for example, “leaves” or “scoop”) such that the top-10 most similar words (according to cosine similarity) contains related words from *both* meanings. For example, “leaves” might have both “vanishes” and “trunk” in the top-10, and ”scoop” might have both “cone” and “newspaper”.

You will probably need to try several polysemous words before you find one. Please state the polysemous word you discovered and the multiple meanings that occur in the top-10. Why do you think many of the polysemous words you tried didn’t work?

Note: You should use the `most_similar` *function to get the top-10 most similar words. This function ranks all other words in the vocabulary with respect to their cosine similarity to the given word.*

(b) When considering Cosine Similarity, it’s often more convenient to think of Cosine Distance, which is simply 1 - Cosine Similarity. Find three words (w_1, w_2, w_3) where w_1 and w_2 are synonyms and w_1 and w_3 are antonyms, but $CosineDistance(w_1, w_3) < CosineDistance(w_1, w_2)$. For example, $w_1 =$”happy” is closer to $w_3 =$”sad” than to $w_2 =$”cheerful”.
Once you have found your example, please give a possible explanation for why this counter-intuitive result may have happened. You should use the the **distance** function here in order to compute the cosine distance between two words.

(c) Word2Vec vectors have been shown to **sometimes** exhibit the ability to solve analogies. As an example, for the analogy ”man:king :: woman:x”, what is x?

In the utility notebook, we show how to use word vectors to find x. The `most_similar` method can be applied by running

`most_similar(positive=['king', 'woman'], negative=['man'])` to compute `nearest-neightbor`$(w_{\text{king}} - w_{man} + w_{\text{woman}})$. The function will returned a ranked list of words based on similarity to the computed vector. Hopefully in this case, the answer will be “queen”.

Find 2 examples of analogies that hold according to these vectors. i.e., the intended word is ranked at the top-3. In your solution please state the full analogy in the form x:y :: a:b. If you believe the analogy is complicated, explain why the analogy holds in one or two sentences. *Note: You may have to try several analogies to find one that works!*

(d) Find an example of an analogy that does **not** hold according to these vectors. In your solution, state the intended analogy in the form x:y :: a:b, and state the (incorrect) value of b according to the word vectors.