Practical 2: word2vec Algorithm

Deep Natural Language Processing Class, 2023

Report due date - 21.03.2023

This practical was greatly inspired by the CS224 class from Stanford University.
The practical report (pdf + code) should be sent via Moodle/Pegaz with a standard due dates policy.

1 The reasoning behind word2vec

The main idea behind the word2vec algorithm is that a word meaning can be derived by its surrounding context, i.e. other words. 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 'wylazł'. Since the context window size is 2, the outside words are 'przesuwajac', 'odnóże', 'z', and 'wykrotu'.

The goal of the skip-gram word2vec algorithm is to accurately learn the probability distribution $P(O|C)$. Given a specific word o and a specific word c, we want to calculate $P(O = o|C = c)$, 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.

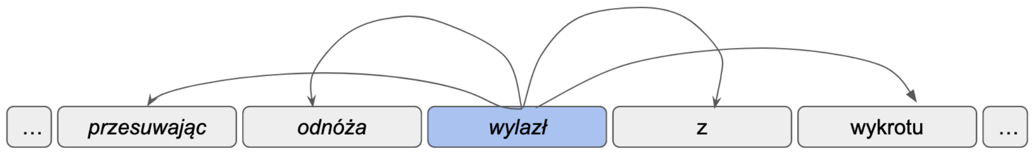

Figure 1: The word2vec skip-gram prediction model with window size 2.

In word2vec, the conditional probability distribution is given by taking vector dot-products and applying the softmax function:

$$P(O = o|C = c) = \frac{\exp(\mathbf{u}_o^T \mathbf{v}_c)}{\sum_{w \in Vocab} \exp(\mathbf{u}_w^T \mathbf{v}_c)}. \quad (1)$$

Here, $\mathbf{u}_o$ is the 'outside' vector representing outside word o, and v_c is the 'center' vector representing center word c. To contain these parameters, we have two matrices, $\mathbf{U}$ and $\mathbf{V}$. The columns of $\mathbf{U}$ are all the 'outside' vectors u_w. The columns of $\mathbf{V}$ are all of the 'center' vectors v_w. Both $\mathbf{U}$ and $\mathbf{V}$ contain a vector for every $w \in$ Vocab.

For a single pair of words c and o, the loss is given by:

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

Another way to view this loss is as the cross-entropy between the true distribution $\mathbf{y}$ and the predicted distribution $\hat{\mathbf{y}}$. The cross-entropy between two discrete probability distribution p and q is defined as $-\sum_i p_i \log q_i$. Here, both $\mathbf{y}$ and $\hat{\mathbf{y}}$ are vectors with length equal to the number of words in the vocabulary. 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 $\mathbf{y}$ is a one-hot vector with a 1 for the true outside word o, and 0 everywhere else. The predicted distribution $\hat{\mathbf{y}}$ is the probability distribution $P(O|C = c)$ given by our model in equation (1).

a) (1.5 point)

Show that the naive-softmax loss given in Equation (2) is equivalent to the cross-entropy loss between $\mathbf{y}$ and $\hat{\mathbf{y}}$; i.e., show that:

$$-\sum_{w \in Vocab} \mathbf{y}_w \log(\hat{\mathbf{y}}_w) = -\log(\hat{y}_o).$$

Note that $\mathbf{y}_w$, $\hat{\mathbf{y}}_w$ are vectors and $\hat{y}_o$ is a scalar.

b) (2 points)

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

c) (2 points)

Compute the partial derivatives of $\boldsymbol{J}_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U})$ with respect to each of the outside word vectors, $\boldsymbol{u}_w$'s. Write your answer in terms of $\mathbf{y}$, $\hat{\mathbf{y}}$, and $\mathbf{v}_c$.

d) (1.5 point)

The sigmoid function is defined as:

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

Please compute the derivative of $\sigma(x)$ with respect to x, where x is a vector.

e) (2 points)

Let's 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. For simplicity of notation we shall refer to them as $w_1, w_2, ..., w_K$ and their outside vectors as $\mathbf{u}_1, ..., \mathbf{u}_K$. Note that $o \notin w1, ..., w_K$. For a center word c and an outside word o, the negative sampling loss function is given by:

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

for a sample $w_1, ..., w_K$, where $\sigma(\cdot)$ is the sigmoid function.

Please repeat parts (b) and (c), computing the partial derivatives of $\mathbf{J}_{neg-sample}$ with respect to $\mathbf{v}_c$, with respect to $\mathbf{u}_o$, and with respect to a negative sample $\mathbf{u}_k$. Please write your answers in terms of the vectors $\mathbf{u}_o$, $\mathbf{v}_c$, and $\mathbf{u}_k$, 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 to part (d) to help compute the necessary gradients here.

f) (1.5 point)

Suppose the center word is $c = w_t$ and the context window is $[w_{tm}, ..., w_{t1}, w_t, w_{t+1}, ..., w_{t+m}]$, where m is the context window size. Recall that for the skip-gram version of word2vec, the total loss for the context window is:

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

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

1. $\partial \boldsymbol{J}_{skip-gram}(\mathbf{v}_c, w_{t-m}, ..., w_{t+m}, \mathbf{U}) / \partial \mathbf{U}$,
2. $\partial \boldsymbol{J}_{skip-gram}(\mathbf{v}_c, w_{t-m}, ..., w_{t+m}, \mathbf{U}) / \partial \mathbf{v}_c$,
3. $\partial \boldsymbol{J}_{skip-gram}(\mathbf{v}_c, w_{t-m}, ..., w_{t+m}, \mathbf{U}) / \partial \mathbf{v}_w$ when $w \neq c$.

Write your answers in terms of $\partial \boldsymbol{J}_{skip-gram}(\mathbf{v}_c, w_{t+j}, \mathbf{U}) / \partial \mathbf{U}$ and $\partial \boldsymbol{J}_{skip-gram}(\mathbf{v}_c, w_{t+j}, \mathbf{U}) / \partial \mathbf{v}_c$.

2 Building your version of word2vec

Let's implement the word2vec model and train our own word vectors with stochastic gradient descent (SGD). Before you begin, first download all the required packages through `pip install -r requirements.txt`. This guarantees that you have all the necessary packages to complete the assignment. It is also advised to finish the points a) to c) from Section 1.

a) (6 points)

First, implement the `sigmoid` function in `word2vec.py` to apply the sigmoid function to an input vector. In the same file, fill in the implementation for the `softmax` and negative sampling loss and gradient functions. Then, fill in the implementation of the loss and gradient functions for the skip-gram model. When you are done, test your implementation by running `python word2vec.py`.

b) (2 points)

Complete the implementation for your SGD optimizer in `sgd.py`. Test your implementation by running `python sgd.py`.

c) (2 points)

Now we are going to load some real data and train word vectors with everything you just implemented! We are going to use the OpenSubtitles dataset of movies translated from Georgian to Polish to train word vectors. You will need to fetch the datasets first. To do this, run `python utils/get_open_subtitles.py`. Now you can run `python run.py`.

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 in at most three sentences what you see in the plot.

3 Optimizing Neural Networks

Recall the standard Stochastic Gradient Descent update rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \boldsymbol{J}_{\text{minibatch}}(\boldsymbol{\theta})$$

where $\boldsymbol{\theta}$ is a vector containing all of the model parameters, $\boldsymbol{J}$ is the loss function, $\nabla_{\theta} \boldsymbol{J}_{\text{minibatch}}(\theta)$ is the gradient of the loss function with respect to the parameters on a minibatch of data, and α is the learning rate. Adam Optimization uses a more sophisticated update rule with two additional steps.

a) (1 point)

First, Adam uses a trick called momentum by keeping track of $\boldsymbol{m}$, a rolling average of the gradients:

$$\begin{aligned} \boldsymbol{m} &\leftarrow \beta_1 \boldsymbol{m} + (1-\beta_1) \nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{m} \end{aligned}$$

where β_1 is a hyperparameter between 0 and 1 (often set to 0.9). Briefly explain (you can just give an intuition) how using $\boldsymbol{m}$ stops the updates from varying as much and why this low variance may be helpful to learning, overall.

b) (2 points)

Adam also uses adaptive learning rates by keeping track of $\boldsymbol{v}$, a rolling average of the magnitudes of the gradients:

$$\begin{aligned}
\boldsymbol{m} &\leftarrow \beta_1 \boldsymbol{m} + (1-\beta_1)\nabla_{\boldsymbol{\theta}} \boldsymbol{J}_{\text{minibatch}}(\boldsymbol{\theta}) \\
\boldsymbol{v} &\leftarrow \beta_2 \boldsymbol{v} + (1-\beta_2)\left(\nabla_{\boldsymbol{\theta}} \boldsymbol{J}_{\text{minibatch}}(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} \boldsymbol{J}_{\text{minibatch}}(\boldsymbol{\theta})\right) \\
\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \alpha \odot \boldsymbol{m}/\sqrt{\boldsymbol{v}}
\end{aligned}$$

where $\odot$ and / denote elementwise multiplication and division (so $\boldsymbol{z} \odot \boldsymbol{z}$ is elementwise squaring) and β_2 is a hyperparameter between 0 and 1. Since Adam divides the update $\sqrt{\boldsymbol{v}}$, which of the model parameters will get larger updates? Why might this help with by learning?

*c) (2 points)

A popular way of constraining parameters is to add L_2 regularization

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \boldsymbol{J}_{\text{minibatch}}(\boldsymbol{\theta}) + \lambda ||\boldsymbol{\theta}||,$$

with λ being a hyperparameter in $[0, 1]$. Why L2 regularization can be helpful in low-data setting?

Training models with Adam and L2 (which is often the case in popular libraries) can result in some issues. The authors of an improved version of Adam (called AdamW optimizer) argue that the equivalence between L2 regularization and weight decay from the point *b*), true for SGD, does not hold for adaptive schemes. This leads to situation where L2 regularization is not effective in Adam. What is their proposal for improving the Adam update scheme?

d) (2 points)

Dropout is a regularization technique. During training, dropout randomly sets units in the hidden layer $\boldsymbol{h}$ to zero with probability p_{drop} (dropping different units each minibatch). We can write this as:

$$\boldsymbol{h}_{drop} = \boldsymbol{d} \circ \boldsymbol{h},$$

where $\boldsymbol{d} \in \{0, 1\}^{D_h}$ (D_h is the vector size of $\boldsymbol{h}$) is a mask vector where each entry is 0 with probability p_{drop} and 1 with probability $(1 - p_{\text{drop}})$.

Question 1: Why should we apply dropout during training but not during evaluation?
*Question 2: How dropout can be seen from the perspective of Bayesian theory. Answer in no more than 5 sentences (this paper could be helpful).