Homework 3: Recurrent Neural Networks for NLP

Abstract

In this homework, *Recurrent Neural Networks (RNN)* are used in order to implement a model of language. In addition to RNNs, a Word2Vec implementation is also used, with the hope of a better performance.

Code development

The first step in the assignment was the creation of a proper dataset, using Pytorch dedicated classes. The dataset itself is based on Oscar Wilde's famous '*The portrait of Dorian Gray*'. Since a Word2Vec approach was chosen, the preprocessing amounts to removing both punctuation (including the non-visible one, such as `\n`) and chapter headers, so to create a large collection of sentences. [1]

After preprocessing was over, the actual implementation of Word2Vec followed. A complete discussion of the architecture would prove too lengthy; instead, a fast description of the features will be presented.

In its most basic version, Word2Vec can be described as follows. Suppose we have a corpus of N unique words; each of them can be represented by an abstract vector of size M, called the *embedding dimension*. This vector representation is encoded by a shallow network with input layers of N units and a hidden layer of M neurons. The input layer requires the words to be encoded as one-hot vectors of size N; this results in a weight matrix of shape $(N \times M)$, where each of the M columns contains the weights for each of the hidden neurons. Indeed, this matrix can be reinterpreted as a *lookup table*: given a word in its one-hot form, its multiplication with the matrix results in selecting one row out of the N possible ones. Each of these rows has M elements, and can be interpreted as the embedding of the input word.

The version that was used in the code is actually more complex, as it is explained in the following:

- Architecture: instead of only one hidden layer, there are 2 identical embedding layers, referred to as input and output respectively. This amounts to having actually *two* $(N \times M)$ weight matrices; however, only the input weights are modified and saved.
- Loss function and word sampling: a specific loss function is used, together with some sampling techniques which aid in making the learning faster. These are called *subsampling* and *negative sampling*.
  - Subsampling: in a typical corpus, common words such as 'the', 'to' and so on will be overwhelmingly present when compared with less common ones. As a consequence, when the algorithm forms pairs of words, the common ones will

[1] Sentences were assumed to only be delimited by the characters [.!?].

appear too frequently. Subsampling consists in randomly removing frequent words, according to the following distribution, which represents the probability of keeping word w_i:

$$p(w_i) = \left(\sqrt{\frac{z(w_i)}{0.001}} + 1\right) \cdot \frac{0.001}{(w_i)}$$

where $z(w_i)$ is the fraction of w_i in the whole corpus.

– Negative sampling: a typical weight matrix for a Word2Vec is quite large; updating all weights at once, as one would do in a standard backpropagation, is not the optimal approach. With negative sampling, for each training sample, the weights pertaining to the correct word will be updated together with a small number of randomly chosen *negative samples*, that is, words which are not correct. The distribution according to which negative samples are drawn is once again dependent on how frequently the words appear. Indeed, such distribution is

$$P_n(w_i) = \frac{f(w_i)^{3/2}}{\sum_j f(w_j)^{3/2}}$$

with $f(w_i)$ being the number of times the word w_i appears.

The loss function that is used is the following, where $\log \sigma$ is the logsigmoid function:

$$J = -\left(\log\ \sigma({v'_{w_O}}^\top v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)}[\log\ \sigma(-{v'_{w_O}}^\top v_{w_I})]\right)$$

This is called *negative sampling loss*. In the first term, the two weight matrices (v'_{w_O} for the output and v_{w_I} for the input) are multiplied: this represents the loss on the actual word. The second term is a sum of k terms where the input embedding is multiplied to the output embeddings of a *negative* sample (indeed, the words are drawn from the distribution P_n described above).

After the Word2Vec step, the actual learning was carried out by means of an LSTM network. In the hopes of speeding up the learning process, the dataset structure was modified with respect to the example that was provided. Instead of dividing the corpus in chapters, a sentence-based subdivision was preferred, as it allows batches to be much larger; indeed, choosing chapters as items of the dataset only leaves roughly 20 items from which to form batches. Sentences are instead much more numerous ($\sim 10^3$), so batches can be larger. This reduces the loading times, leading to many more training samples per unit time.

As for the RNN itself, the main modification was on the loss function. In the seq-2-seq implementations, the loss function was based on the one-hot-encoding of each character. In a Word2Vec based implementation, however, word vectors are no longer orthogonal, so a categorical loss function is to be avoided. Instead, a loss that takes advantage of the vector nature of the encodings seems the wiser choice: indeed, similar words will be hopefully 'closer'. The attempt is then to find a function which can capture this high-dimensional distance: that will be the loss function. The following two choices were explored:

- Angle-loss: if one interprets the words as vectors in an M dimensional space, where M is the embedding dimension, then similar vectors should in principle share their direction. The angle loss tries to capture this similarity in direction:

$$L_{angle}(o,l) = \frac{\sum_{n=1}^{k} 1 - \frac{|o_n \cdot l_n|}{\|o_n\| \|l_n\|}}{k}$$

  where o, l are the batch-predictions and the batch-labels respectively, and k is the number of words per batch.

  The ratio between the scalar product and the norms of the output predictions and the correct labels results in the computation of the cosine of the angle between the two. This quantity is the shifted and rescaled in order to lie in the interval [0,1].

- Distance-loss: a more naive choice, this function simply computes the distance of the output from the labels:

$$L_{dist}(o,l) = \frac{\sum_{n=1}^{k} \|o_n - l_n\|}{k}$$

  The main problem of this approach lies in the high dimension of the embedding space, which makes the computation of distances a less effective tool.

After the RNN training, the last step was to modify the chapter generation script in order for it to take a list of words as input and to generate an appropriate prediction as output. Two main problems had to be faced:

- Input words: contrary to a character-based approach, the network needs input words to be actually present in the dictionary that was created by Word2Vec. Indeed, no representation was created for words outside the training corpus. If an unknown word is provided to such a network, an error arises. To avoid this, in case of an unknown word being provided as seed, the 'closest word' in the dictionary was used, based on the Levenshtein distance[2]. This allows the network to use any sequence of letters as a valid seed.

- Prediction: while the one-hot nature of a character based approach makes prediction easy, as the output of the network uniquely identifies one letter, a vector approach requires the output to be associated to an actual word. To do so, the M dimensional output was compared to the M dimensional encodings of the words in the dictionary, and this distance was used as a metric, on the basis of which the actual output could be chosen (either by taking the argmax or by drawing from a softmax).

Results

The training of the Word2Vec was carried out with embedding dimension $M = 256$, while the dictionary dimension was of the order of 7000 words. The same embedding was used with both losses.

[2]The Levenshtein distance between two strings counts the minimum number of deletions, insertions or letter changes that allow to turn the first string into the second (or viceversa).

For the training of the RNN, both the aforementioned losses were employed (of course, separately). However, at training time, both the losses did not seem to meaningfully decrease. Furthermore, neither managed to produce satisfying results: despite the high amount of training samples, at the moment of generating the chapter, the networks do not manage to output even slightly meaningful sequences of words.

Indeed, the resulting word sequences strongly depend on the chosen drawing method. As explained above, the output vector of the RNN is compared with the encodings for all the words in the corpus. An M dimensional distance is then computed, and based on these values one can either:

- choose the argmin: this leads to a small selection of words being repeated over and over;
- take all distances, compute the softmax distribution over them and then draw words according to it. This leads to an almost completely random sequence of words.

Some examples follow. The seed is underlined. Three points indicate that the chapter generation was interrupted early (mostly because of uninteresting patterns repeating over and over).

- `it was a dark and rainy night creep and the triangular of the attempted of the of of the of of of of the of of of of of the of of of of the of of of the of...`

  Hyperparameters: argmax generation, angle loss.

- `it was a dark and rainy night ear you creep to be examinations clatter clatter examinations to be creep examinations clatter clatter clatter to be triangular to be creep to be examinations clatter clatter examinations to be creep examinations clatter clatter clatter to be triangular to be creep to be examinations...`

  Hyperparameters: argmax generation, distance loss.

- `it was a dark and rainy night directors madrid miniatures royalties hand noon carlington affection amazement restored cookery piccadilly lilas forgiven chest wizen la wrong whistle torch selects dare theory sterile preceding alliance chopin...`

  Hyperparameters: softmax generation, angle loss.

- `it was a dark and rainy night established proud interest dies idolatry burgundy ruined guide smoking educated crocus horrid am margaret pomegranates doorway steaming dread principle company packed displayed tightening webster edward lithe chemical...`

  Hyperparameters: softmax generation, distance loss.

The reasons for this lack of success may be due to a low number of epochs or to a problem in the high dimensionality of the embedding space. This in particular may be the reason for the random-like drawing of words in the softmaxed prediction: all words are somewhat 'equidistant' in the embedding space, leading to all words being effectively equal at the time of drawing them.

Comments

Despite the efforts, the results were all but encouraging. Varying the parameters seems the only feasible solution, but the computational weight of the model makes this a rather daunting perspective.