Neural Word Embeddings

Johannes Gontrum

Department of Linguistics and Philology
Uppsala University

Abstract

When the *word2vec* algorithm was released in 2013, it demonstrated that word embeddings could be learned efficiently using neural network architecture. Their quality and performance of the word vectors have found applications in many fields, from machine translation to recommendation systems, while the encoded deep semantic information can help improving knowledge bases. After concentrating on the two original papers that introduced *word2vec*, Mikolov et al. (2013a) and Mikolov et al. (2013b), this survey gives a brief outlook on newer developments in this field by summarizing an approach to learn probability distributions for polysemic words by Athiwaratkun and Wilson (2017).

1 word2vec

Simple n-gram language models can be efficiently trained on huge corpora, but treat words as strictly autonomous units. While this simple view is enough for many NLP tasks, it does not allow the model to recognize more detailed semantic information. With the rising interest in machine learning and especially artificial neural networks over the last decade, new techniques have been explored to tackle these shortcomings.

Though the word2vec algorithm presented by Mikolov et al. (2013a) is by far not the first that approaches neural net language models. It was, however, the first that introduced significant performance improvements that allowed users to train accurate word vectors with comparatively low resources within a few days, compared to weeks.

1.1 Feedforward Neural Networks

The language model architecture used by Mikolov et al. (2013a) is based upon a modified feedforward neural network. The most basic neural network consists of k layers, starting with the input layer l_0, typically at least one hidden layer l_1 and an output layer l_{k-1}. The nodes in one layer i are connected with the nodes in the next layer j with the strength of their connection defined in a weight matrix $W_{i,j}$.

When using a neural network, the input nodes in the first layer are filled with values. To compute the values of the nodes in the next layer, the value of the input nodes are multiplicated with their respective weight and then summed up. As an example, we assume an input layer of

The **paper was *made* in Bohemia** I said

Figure 1: Example of a moving window around the center word "made" with size 2.

the length 3 with the nodes x_0, x_1 and x_2. The second layer consists of two nodes a_0^1 and a_1^1. The value of a_0^1 is therefore defined as $x_0 * w_{0,1}^{0,0} + x_1 * w_{0,1}^{1,0} + x_2 * w_{0,1}^{2,0}$. The second node in the second layer is calculated in a similar way.

If layer 1 is a hidden layer, an activation function is applied that transforms the value of each node by projecting it for example in the range of -1 to 1 with the sigmoid function or by replacing negative values by 0 with a rectifier. In the output layer, the values are also transformed to bring the output into a suitable form. Often the $softmax$ function is applied which changes the values in the output layer to represent a probability distribution so that the sum of the output vector will be exactly 1.

In the supervised training process of a neural network, the weight matrices between the layers are gradually changed to produce an outcome gets ever closer to the desired output. The details of the training involving backpropagation and stochastic gradient descent are beyond the scope of this word though.

In some cases, like in the later on presented one, the input vector is transformed to a projection layer using a projection matrix. Broadly speaking, projection layer (or embedding layer) and matrix behave like a hidden layer and its weight matrix, but without the non-linear activation function in the hidden layer. The values for the nodes in the layer are simply summed up, but not further transformed.

1.2 Procedure

Mikolov et al. (2013a) explore the existing approaches and discover that the hidden layer with its non-linear activation functions causes major computation complexity. Though these non-linear functions are what make deep neural networks so powerful, they decide to remove it completely since they argue that it is not necessary for a simple task like learning word embeddings. Instead, they use, a projection layer, so that their architecture contains an input layer, a projection layer, and an output layer.

Like when learning word co-occurrences, they use a moving window generate the contexts of a word. Given a window size for 2, the example in Figure 1 with the word "made" in the center, generates the training examples $(made, paper)$, $(made, was)$, $(made, in)$, and $(made, Bohemia)$.

1.2.1 Continuous Bag-of-Words Model

The first of the two approaches presented by Mikolov et al. (2013a) is the Continuous Bag-of-Words Model or CBOW. Here, the context of a word is given as an input and the neural net is trained to predict the center word. In the example in Figure 1, the input consists of the one-hot encoded words "Bohemia", "in", "paper" and "was". One-hot encoding is a technique often used to project language data into a vector space, where the vocabulary of the training corpus is saved in a sorted way. Words are then encoded by creating a vector with the size of the vocabulary

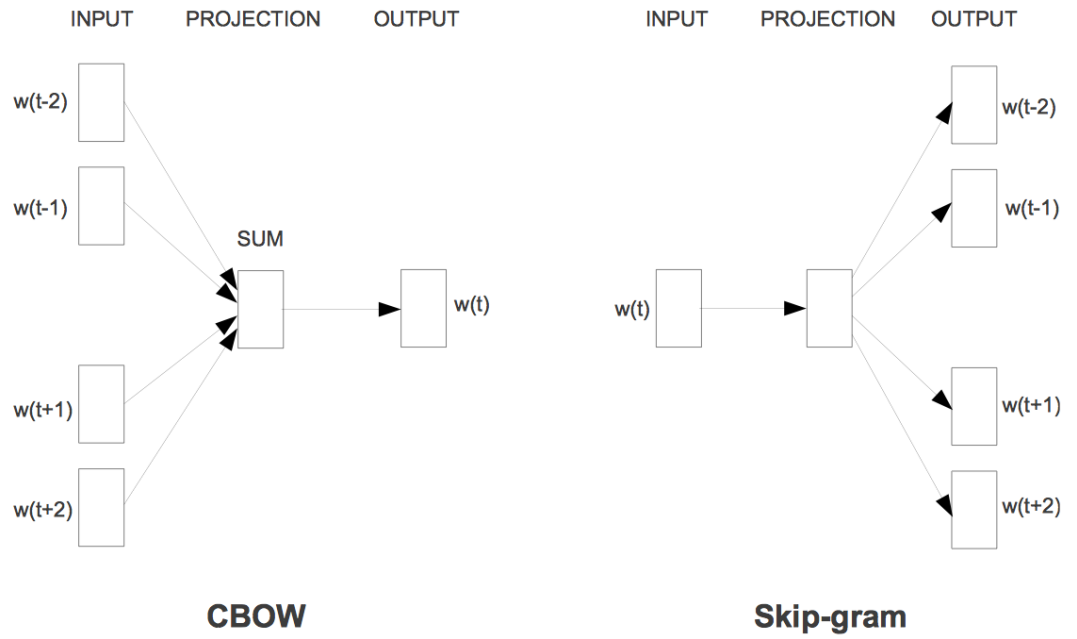

Figure 2: Illustration of the CBOW and Skip-gram architecture. Image taken directly from Mikolov et al. (2013a).

filled with zeros, except for a one that is written at the position of the encoded word. Though a potentially hundreds of thousands of dimension large vector might seem like a performance bottleneck, internally it as treated as a compressed sparse vector.

Since in the given example there are four input vectors and not only one, as usual for a neural network, the weights of the given words from projection matrix are averaged. As this procedure removes any information about the position of the word in the context, the approach is called ”bag of words” model.

During the training process, the weights inside the shared projection matrix are adapted to optimize for the correct output. This matrix has the dimensions $V \times D$ where D is the size of the projection layer and V the size of the vocabulary. The matrix can then be used as the output since it contains a vector of length D for every word in the vocabulary.

1.2.2 Skip-gram Model

The other proposed model takes a reversed approach: Instead predicting the center word, this neural network takes it as the input and tries to predict the context words, like shown in Figure 2. Though Mikolov et al. (2013a) state that this architecture increases the computational complexity compared to the CBOW model, the quality of the result also improves, as discussed later.

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

Figure 3: Probability that a training example containing the word w_i will be removed. t is a threshold, e.g. 10^{-5} as suggested by the authors and $f(w_i)$ is the frequency of the given word w_i.

1.2.3 Hierachical Softmax

In their second paper, Mikolov et al. (2013b) go into more detail about their advanced optimization techniques. One of them is to find an alternative to calculating the softmax function for the output layer. The idea behind it is the following: The output layer contains W nodes, a number that often is in the hundreds of thousands, each corresponding to a one-hot encoded word. However, during the training of a skip-gram model only the value of $2N$ of these nodes are actually important, because they represent the words in the context window. A hierarchical softmax applies a binary decision tree with W leaves to select only $log_2(W)$ important nodes and calculate their softmax values. Mikolov et al. (2013b) use a Huffman tree to speed up the lookup of the more frequent words.

1.2.4 Subsampling

To further improve the performance of their algorithm, the authors reduce the number of training examples by subsampling frequent words. They state that the most frequent words like ”a”, ”the”, ”in” etc. appear extremely often in every corpus [ZIPF] while containing much less useful information than rarer words. As an example, they state that the co-occurrence of ”the” and ”France” barely holds relevant semantic information. Therefore, they suggest removing words based on their frequency. They propose using the function in Figure 3 to compute the probability that a training example will be deleted. As expected, this technique speeds up the training process and even results in a significant improvement of accuracy.

1.2.5 Identifying Phrases

While Mikolov et al. (2013b) learned word embeddings for strictly tokenized text, they state that their model works better if phrases were left intact. For example ”New York Times” has a meaning on its own that is not composed of the words ”New”, ”York”, and ”Times”. They then use simple bigram and trigram counts to automatically identify phrases in their corpus and merge them by replacing the separating blank characters with an underscore. This process is repeated multiple times to find ever longer phrases.

1.3 Evaluation

In addition to a drastic training time improvement, Mikolov et al. (2013a) discovered that the trained word vectors also encode semantic and syntactic information. This can go so far they

even allow simple algebraic operations like $vector(\text{Germany}) - vector(\text{Berlin}) + vector(\text{France}) = vector(\text{Paris})$. They could be used for question answering systems and knowledge bases or fact checking. Even grammatical information like $vector(\text{biggest}) - vector(\text{big}) + vector(\text{small}) = vector(\text{smallest})$ are encoded in the learned word vectors.

To evaluate the qualitative performance, the researchers created a test set with semantic and syntactic pairs. Given one word, a model should predict the other using the methods mentioned above. They state that a 100% accuracy is almost impossible to archive because they only evaluate if the closest word is exactly the one that should be predicted. Example for semantic pairs is capital cities: (Athens, Greece), (Oslo, Norway or Man-Woman: (brother, sister), (grandson, granddaughter). On the syntactic side, they use e.g. past tense: (walking, walked), (swimming, swam).

1.3.1 Model architecture

Trained on the same data and using 640-dimensional word vectors, the authors compare their Skip-gram and CBOW models against state-of-the-art architectures using feedforward-neural network language models (NNLM) and recurrent neural network language models (RNNLM). In the category of semantic accuracy, Skip-gram outperforms not only the old models RNNLM (9%) and NNLM (23%) but even vastly the CBOW architecture (24%) by reaching an accuracy of 55%.

In the category of syntactic accuracy, CBOW scores highest with 64%, followed directly by skip-gram with a score 59%. NNLM and RNNLM reach an accuracy of 54% and 36% respectively.

They also compared their best models, trained on a subset of the Google News corpus, against publicly available word vectors. With a total score of 53.3 (average over the syntactic and semantic accuracy), Skip-gram again outperformed not only CBOW (36.1) but also doubled the performance of the best competitive word embeddings with a score of 24.6.

2 Distributed Word Representations

Traditional word embeddings like the previously discussed word2vec map words into a vector space and group words with similar semantics together. Representing a word by a single point, however, is a very narrow approach. To offer word embeddings that feature concepts like entailment and a measure of uncertainty, probabilistic word embeddings have been proposed. Here, a word is represented as a Gaussian probability distribution. While its mean vector is equal to the conventional word vectors, it also includes a covariance matrix that describes how widespread a word is in a vector space. This allows us to investigate entailment properties like $music \models jazz$, where the wide distribution of "music" includes the embedding of "jazz", a subcategory of music.

While Athiwaratkun and Wilson (2017) summarize that existing approaches work reasonably well, the probability distribution is often skewed and distorted for polysemic words. In the popular example "bank", one needs two separate distributions: one that is near finance words

and one in the area of "river". Squeezing it into a single distribution causes its mean vector to be in between the two words and its variance to be high.

2.1 Procedure

To represent multiple senses of a word, Athiwaratkun and Wilson (2017) learn a Gaussian mixture model. The model is initialized with a number of K components that correspond to the number of senses that a word is supposed to have. Broadly speaking, each component has its own probability distribution around its own mean vector. Merging them together into a mixture then creates a probability function with multiple maxima.

Though Athiwaratkun and Wilson (2017) are not completely clear about the exact learning algorithm, it is largely built upon an adapted version of the previously discussed Skip-gram model that learns to predict the context of a given word. To learn the values of the mixture model, they propose an energy-based maximum margin objective that aims to maximize the similarity between words of a sentence. As training examples, they not only use the already presented word pairs of a sentence like $(made, in)$ in Figure 1 but also negative examples. To not only teach the energy function positives (similar words), but also unrelated ones, they randomly sample other words from the corpus that are not present in the context and use them as negative examples.

To determine the most relative meanings when evaluating similarity, Athiwaratkun and Wilson (2017) propose multiple measures that compare all sense of the given words. They use for example the Expected Likelihood Kernel, which computes the inner product between both distributions, or the Maximum Cosine Similarity. Here the cosine similarity between all components of the given words are measured and their maximum returned.

2.2 Evaluation

In addition to hyperparameters like the size of the context window and the dimensions of the vector space, the multi-modal modal also requires a pre-defined value for K, the number of components of the mixture model. This also implies that every word, in the corpus will have exactly K different senses, even stopwords like "the" or "in". Though a more dynamic approach would be cleaner and speed up application, the authors argue that having multiple distributions for a non-polysemic word does not harm the qualitative performance, as the similarity function simply choose the best distribution for the given case.

They evaluate their model on the most common word similarity datasets that contain word pairs and a human annotated score on how similar they are. With one exception, their model performs better than their competition, though they notice differences concerning the used similarity function. Overall, however, the Maximum Cosine Similarity performs best.

In another test, Athiwaratkun and Wilson (2017) evaluate their multimodal modal on an entailment dataset to see, how good the word distributions encode information like $vehicle \models aircraft$. They compare their results against a self-trained model with only one component. Overall the multimodal model performs better, though the improvements in F_1 score are in the area of a few percentage points. Given their arguments about gaining more information for polysemous words, this improvement seems rather small.

References

Ben Athiwaratkun and Andrew Gordon Wilson. Multimodal word distributions. *arXiv preprint arXiv:1704.08424*, 2017.

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

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, 2013b.