CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Evaluating word-vector generation techniques for use in semantic role labeling with recurrent LSTM neural networks

Master's thesis in Computer Science: Algorithms, Languages and Logic

DANIEL TOOM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017:NN

Evaluating word-vector generation techniques for use in semantic role labeling with recurrent LSTM neural networks

DANIEL TOOM

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Evaluating word-vector generation techniques for use in semantic role labeling with recurrent LSTM neural networks
DANIEL TOOM

Supervisor: Richard Johansson, Department of Computer Science and Engineering
Examiner: Alexander Schliep, Department of Computer Science and Engineering

Master's Thesis 2017:NN
Department of Computer Science and Engineering
Computing Science division
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Printed by [Name of printing company]
Gothenburg, Sweden 2017

Evaluating word-vector generation techniques for use in semantic role labeling with recurrent LSTM neural networks
DANIEL TOOM
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This is an investigation of how the performance on a natural language task, semantic role labeling (SRL), is affected by how the input text is encoded. Four methods for encoding words as dense vectors of floating-point numbers are evaluated: Noise-contrastive estimation (NCE), Continuous bag-of-words (CBOW), Skip-grams and Global vectors for word representation (GloVe). Vectors are generated using a corpus of Wikipedia articles as the training set, with different values for various parameters, such as vector length, context length and training methods. In order to evaluate the generated vectors, they are then used as the input into a recurrent bi-directional neural network using LSTM neurons. This network is trained using the standard CoNLL-2005 shared task data set, and evaluated with the associated test sets. The results show that there is no large or consistent advantage to using word encodings from any of the tried methods or any of the tried parameter settings over any of the other methods or parameter settings. The SRL system that was used thus seems to be fairly robust in regard to the choice of input vectors. All methods generate vectors that outperform random vectors, however, indicating that pretraining vectors has a positive effect. The labeling accuracy is close to but slightly worse than previously published state-of-the-art performance results from the use of a similar network on the SRL task.

Keywords: word-vector generation, SRL, NCE, CBOW, skip-grams, GloVe, CoNll-2005, LSTM RNN.

Contents

List of Figures

List of Tables

1

Introduction

Natural language processing (NLP) nowadays makes heavy use of machine learning techniques, such as artificial neural networks. These techniques generally work with continuous values as input, requiring that texts be translated into a suitable representation. This thesis deals with the problem of finding suitable ways of encoding (or embedding) words as vectors of floating-point numbers. There are several commonly used methods for generating dense vector representations [1]–[3] with the goal that these should contain as much useful information for NLP tasks as possible. The use of such dense vectors has several advantages over other sparser representations, such as one-hot encoding. For instance, since the dimensionality can be much lower, processing can be more efficient. Another advantage is that if words with similar meanings are given similar representations, then a machine learning system should be able to make better use of small training data sets, since similar input texts can be dealt with in similar ways, and so generalization performance can be improved [4].

Many vector-generation techniques can be tuned by the setting of various parameters. As an example, most methods make use of a word's common contexts in a large text corpus to capture the meaning and grammatical features of words. For such methods, the length of the contexts that are considered can impact the resulting vectors. Consequently, another goal of this work is to investigate the performance impact of various settings for such parameters.

Methods for evaluating word-vector embeddings can be divided into extrinsic and intrinsic categories. Intrinsic evaluations use some low-level benchmark, often specifically designed for vector evaluation. Examples of such evaluations are comparing vectors representing synonyms and tasks where the objective is to answer questions about analogies on the form "a is to b as c is to what?" Extrinsic evaluations on the other hand take actual NLP task performance as a proxy for the performance of the word vectors. The latter approach is the one taken in this thesis.

Semantic role labeling (SRL) is the task of finding role labels for words or phrases in natural language texts and it is the extrinsic measure of quality used in this work. Semantic roles describe the semantic relation of a predicate, often a verb, to its arguments [5]. For instance, given the sentence "The thief was hit with a bat by the house owner", for the predicate "hit" the role of agent can be assigned to "the house owner" and the role of theme or patient can be assigned to "The thief" and finally, the role of instrument can be assigned to "a bat". Such labels comprise useful

information for many machine learning tasks, such as document categorization [6], question answering [7], sentiment analysis [8] and translation [9].

Recently, Zhou and Xu [10] showed that a deep-learning method can be used to solve the SRL task. Their machine learning system, using long short term memory (LSTM) artificial neural network units coupled with a conditional random field (CRF) classifier, was able to reach state-of-the-art classification performance on a standard SRL benchmark. A similar system is used here as a testbed.

1.1 Research questions and scope

The research questions that this thesis aims to answer are the following:

- Which method for word-vector generation yields the best performance in an SRL benchmark?
- How should that method's parameters be set?

This investigation is limited in scope in the following ways:

- The evaluation of the word-vectors is only performed with only one extrinsic performance measurement, the SRL task and only one benchmark for this task is used, the CoNLL-2005 shared task [11].
- The thesis work does not comprise a comparison of resource usage for the investigated methods, neither in the sense of running times nor in the sense of memory use. Such matters obviously form practical limitations for the scale of the experiments, but beyond this, no further regard is taken to resource usage.

1.2 Overview

The following background theory chapters contain a general introduction to word-vector generation, a description of four methods to be evaluated, a discussion about extrinsic and intrinsic testing and an introduction to SRL machine learning methods in general as well as the SRL machine learning system used by Zhou and Xu [10].

The implementation chapter covers some implementation details as well as details about which tests were run. The results chapter presents the results from the vector-generation methods one by one. In the discussion chapter, some differences between Zhou and Xu's [10] results and the results in this thesis are discussed, some comparisons to previously published evaluations are made and some suggestions for further research are given.

2

Word vectors

This chapter starts with a general discussion about and motivation for representing words as dense vectors. Next comes a description of the vector-generating algorithms that were evaluated in this work, some of which use neural network techniques. These techniques are therefore explained when needed.

2.1 Motivation for and overview of dense vector representations of words

Let us first consider some general issues regarding how to represent a word in a useful way for a machine learning system. As a first suggestion, one could imagine representing words as one-hot vectors, with a length equal to the size of the dictionary. In this case, each word would correspond to a vector with zeros in all places except one place. With very small dictionaries, this may be a feasible way to represent words but in real applications the dictionary size is in the hundreds of thousands or greater, leading to extreme data sparsity. Another downside with this strategy is that this encoding does not give a machine learning system any information about which words are similar in syntactic function and meaning. It would be better if words like "table" and "house" had representations that showed that they can be used as nouns (syntactic similarity), just as it could be useful if words like "book" and "read" had representations that hinted at their having related meanings (semantic similarity) [4]. In the following sections we give an introduction to various methods that have been tried to address these two issues.

2.2 Methods based on manual feature engineering

To solve the similar-word/dissimilar-vector problem, a traditional strategy has been to manually design word features and then use these features to encode the words to be input into a machine learning system. To elaborate, word features that may be useful for the particular machine learning task, such as part of speech, tense, number, various semantic features etc. would be manually annotated for each word. One would then form one-hot vectors for each feature, so that for instance if there

are 20 parts of speech, the part-of-speech feature vector would have 20 elements: 19 zeros and 1 one for each word. All the one-hot vectors for the different features could then be concatenated to form a representation for a word. If there are many features or features with many possible classes, these vectors may still be very sparse and high-dimensional, but similar words would now have similar encodings. The sparsity problem is exacerbated by the fact that it is often useful to encode combinations of such core features into additional features. For instance, it may be useful to have a feature that combines the part of speech information about the previous word as well as the current word in a sentence. Thus, this combined feature vector would have n^2 elements where n is the number of parts of speech [4].

To solve the sparsity problem, one solution is to "compress" the data into fewer dimensions using linear algebra matrix operations. For instance, singular value decomposition (SVD) can be used to factorize a matrix of word vectors into several smaller matrices, one of which can be used as a dense approximate representation of the initial larger one. It is also possible to use neural network techniques for this compression [4].

Deciding what features to use is a non-trivial problem, requiring trial-and-error to discover which ones works best for different natural language processing applications. There are other methods that do not require this manual feature engineering, but rather find useful features using only a corpus of text as the input. There are, for instance, methods based on cooccurrence count matrix factorizations and methods based on predicting a word from its common contexts. The next sections give brief introductions to these categories of methods.

2.3 Cooccurrence counting

In all the methods presented below, the common contexts of words in a large text corpus are used to find good vector representations for those words. The basic idea behind this is that words that are used in similar contexts probably can be expected to have similar meanings and similar syntactic features. For instance, given the context "She saw the (x) on the table.", words that could be used in the place of (x), would be words that can appear in a nominal function (function as nouns) and they would also be words denoting items that are possible to put on tables (not too large physical objects). Thus, from a word's common contexts, it should be possible to extract both syntactic and semantic information. Generally, larger contexts tend to lead to a word's semantic features having a greater importance, whereas shorter contexts tend to favor its syntactic features [12].

Contexts to be used by word-vector-generating algorithms can be configured in multiple ways, as shown in figures 2.1, 2.2, 2.3 and 2.4. The algorithms can either try to find vectors that make it possible to predict a target word from a given context, or, given a word, predict its common context words. The contexts can be either one-sided (i.e. asymmetric) or two-sided (i.e. symmetric). Note that in this thesis, the terms "word" and "token" are used interchangeably to mean the occurrence of

a string of characters, used in a text, that is processed as one unit by some learning algorithm. As seen in the figures, tokens also include punctuation and clitics.

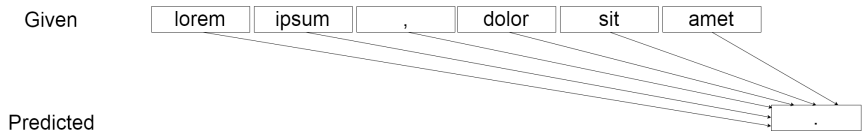

Figure 2.1: Predicting a target word from a given one-sided (asymmetric) context.

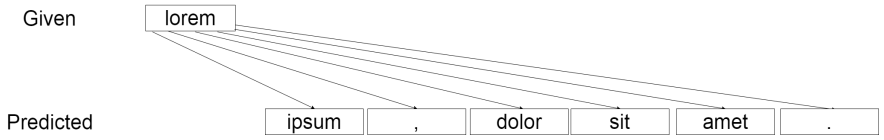

Figure 2.2: Predicting the one-sided (asymmetric) context words from a given target word.

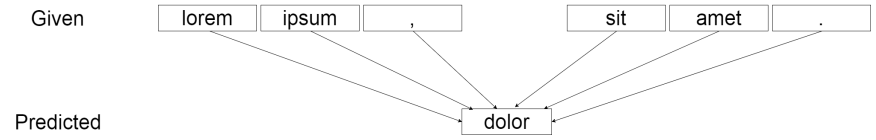

Figure 2.3: Predicting a target word from a given two-sided (symmetric) context.

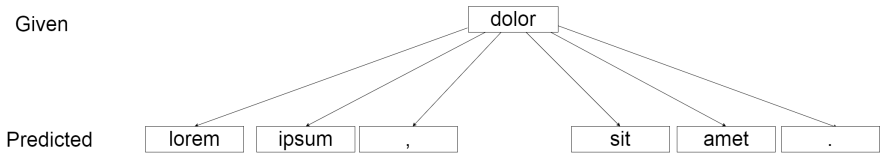

Figure 2.4: Predicting the two-sided (symmetric) context words from a given target word.

2.3.1 Matrix methods

A simple way to use cooccurrence counts is to form a word-word matrix, A, where A_{ij} is the number of times that the word i occurs in the word j's context in a large text corpus [12]. Now, the dot product of two row vectors of the matrix, normalized by the product of their lengths, constitutes a measure of the similarity of these two vectors. The more similar the directions of two vectors are, the greater the dot product becomes, and in order to reduce the effect of how common words are, the normalization is added. This value is called the cosine similarity, since it is equal to the cosine of the angle between the two vectors.

In this case too, there is the issue of sparsity, since many words never occur in each other's contexts and the vocabulary can be very large, so similar dimensionality reduction have been used for these vectors as described in section 2.2.

2.4 GloVe: Global vectors for Word Representation

A recent variation on the theme of factorizing cooccurrence matrices was suggested by Pennington, Socher and Manning [3], called Global Vectors for word representation (GloVe). The idea behind this method is that ratios of cooccurrence probabilities should be the basis for word-vector learning. They argue that if $P(x|y)$ is defined to be the probability of word x occurring in word y's context, then if the ratio $P(a|c)/P(b|c)$ is high, then c probably has a greater similarity to a than to b. Thus, if we have word vectors w for target words and $\tilde{w}$ for context words, then the model F that is trained should somehow relate the ratio $P(a|c)/P(b|c)$, which can be tabulated from a corpus, to w_a, w_b and $\tilde{w}_c$ (where w_x is the vector representation of the word x). By arguing that what matters is really the difference between the vectors w_a and w_b and that w and $\tilde{w}$ should be used in a symmetrical way in the model, they arrive at the following equation; see [3] for details.

$$w_a^T \tilde{w}_c + b_a + \tilde{b}_c - \log(P(a|c)) = 0 \tag{2.1}$$

In training, the objective is to find vectors, w_a and w_c, and scalars b_a and $\tilde{b}_c$, such that equation 2.1 is approximated as well as possible, for all possible combinations of a, b and c from the vocabulary. This is done by minimizing the squared error. Also, a weighting is multiplied with the error, so that more common words are counted more towards the total error than uncommon words, without overweighting very common words. Through testing, Pennington, Socher and Manning found that a weighting that works well is given by equation 2.2, where x_w is the occurrence count of the word w in the 6-billion-word corpus that they used for training. The parameters a and x_{max} were set to 3/4 and 100 respectively,

$$f(x_w) = (x_w/x_{max})^a, \text{ if } x_w < x_{max}, \text{ else } 1. \tag{2.2}$$

2.5 Artificial neural networks and their use in prediction-based methods for word-vector generation

Artificial neural network (ANN) techniques can be used for finding word vectors, through the task of predicting what words commonly appear in what contexts. Some training techniques used for ANNs are also used in noise-contrastive estimation (NCE), which is another vector-generation method, described in a following section. Finally, the SRL task can also be solved with ANNs. Therefore, what follows is a short introduction to the structure and training of such networks. The following description is based on Lecun, Bengio and Hinton's introduction to ANNs [13].

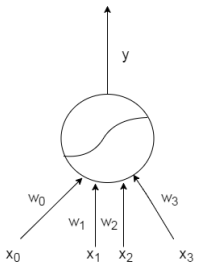

Figure 2.5: A sigmoid neural network unit with inputs x, weights w and output y.

An artificial neural network is a machine learning tool that can be used for classification tasks, where the objective is to decide which of several classes an object described by an input vector belongs to. A traditional artificial neural network is composed of several layers of neurons, where each neuron combines a number of inputs to produce an output that is passed on to the next layer. The inputs for the first layer are thus the independent variables in the classification problem, and the output from the last layer constitutes the network's classification of the input.

The output of a neuron in a traditional artificial neural network with n scalar inputs, $x_i, i \in [1, n]$, is described by equation 2.3 and a pictorial representation of such a neuron is shown in figure 2.5. As can be seen, the scalar product of the input vector and a vector of weights, w, is computed (each neuron in each layer has its own weights), and the result is then fed into a non-linear function, σ. It can easily be shown that if there is no such non-linearity, the addition of more neurons to a layer in a neural network does not increase the expressiveness of that layer. Some possible choices for this non-linearity are the sigmoid function in equation 2.4, the tanh function in equation 2.5 or the rectified linear function in equation 2.6. Two of these choices for non-linearities also have the effect of squashing the output into a limited range — $(0, 1)$ for the sigmoid and $(-1, 1)$ for the tanh function. The main advantage of the rectified linear function is that its derivative does not go to zero for large inputs, which is very helpful in training networks with many layers. If there is a connection between every neuron in layer n to every neuron in layer $n+1$ then these layers are said to be fully connected, which is the common case in the applications used in this thesis.

$$y = \sigma(\sum_{i=1}^{n} w_i x_i) \tag{2.3}$$

$$\sigma_{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

$$\sigma_{tanh}(x) = \frac{e^{-2x} + 1}{e^{-2x} - 1} \tag{2.5}$$

$$\sigma_{relu}(x) = x \text{ if } x > 0, \text{ else } 0 \tag{2.6}$$

In the simplest case, the output of the network is a binary classification. That is, the purpose of the network is to conclude whether some input data belongs to one of two classes. As an example, one might imagine a network designed to classify houses as overpriced or not given certain input statistics about their sales. The last layer of the network could in such a network consist of a single sigmoid or tanh neuron whose output would constitute the classification. A value closer to the maximum output could be taken to mean that the network classifies the house as more probable to be overpriced whereas a value closer to the minimum output could be taken to mean that the network classifies the house as less likely to be overpriced.

If there are more than two classes in the learning problem, there would be one neuron for each class in the final layer and their outputs would be fed through a normalization, such that the sum of all outputs equal 1. In this case, the outputs can be understood as probabilities for the different classes. This normalization is called softmax, and is shown in equation 2.7, where c is a class, C is the set containing all possible classes and y_x is the output from the neuron corresponding to class x.

$$P(c) = \frac{y_c}{\sum_{i \in C} y_i} \tag{2.7}$$

2.5.1 Training neural networks

In order to find optimal values for the weights to use in a neural network, a training set with correctly prelabeled data is used. The purpose of the training is to minimize the error on this training set, measured by some function of the output of the network and the correct labels, called the loss function.

To do this, the algorithms forward/backward propagation are commonly used together with stochastic gradient descent (SGD). The first step is to feed a batch of training examples through the network and compare the output from the network with the correct output using the loss function. This is the forward propagation part of the algorithm. The next step is to change the weights of the network so that the error in the classification is reduced. The gradient of this error with respect to the weights is calculated and an average is taken over the training examples. This is called the backward propagation part of the algorithm, since the gradients in any layer can be calculated using the chain rule for derivatives if the error gradient with respect to that layer's output is known. So, for each layer starting at the last layer, the gradient is calculated with respect to that layer's weights and then the error gradient with respect to that layer's inputs is propagated backwards. Finally, the weights are updated in the direction of diminishing errors, as given by the gradient.

2.5.1.1 Using Adagrad or Adadelta for learning rate adjustments

An important parameter in SGD is the step size, which decides the size of the update steps in the direction of the error gradients. If this is too small, then training will

take longer than necessary and if it is too large, the algorithm may take too large a step and skip over an optimal configuration of the weights. Simply stated, it would be preferable if the step sizes were large when the training had just begun and smaller as the training went on and the model closed in on an optimal configuration of weights. A method for doing this without manually setting learning rates and which has been used in language tasks is called Adagrad, short for Adaptive gradient algorithm.

The idea behind Adagrad is to maintain histories of previous gradients and adjust the learning rate according to these [14]. If there have been many large updates to the weights in the past, then the learning rate is reduced for these weights. This is done by maintaining a squared sum or previous gradients for all the weights (see equations 2.8 and 2.9, where T is the number of training batches so far, f is the function being optimized, w_t is a vector of the weights with respect to which the gradient is taken, λ is the nominal learning rate (a "master" learning rate), z is a "fudge factor", used to prevent numerical instability if g should be very small and $\odot$ is component-wise multiplication.)

$$g_T = \sum_{t=1}^{T} (\nabla_{w_t} f) \odot (\nabla_{w_t} f) \tag{2.8}$$

$$w_T = w_{T-1} - \lambda \frac{\nabla_{w_T} f}{\sqrt{g + z}} \tag{2.9}$$

A problem with Adagrad is that it reduces the rate of learning fast, and the rate reduction caused by Adagrad can never diminish as learning goes on. Thus, the learning rate for often-updated parameters can become very low, very much reducing the impact of late training examples as compared to early ones. A modified version of Adagrad can reduce this problem, by decaying the rate adjustment factor for each update of the weights. This is a description of a simplification of a method called Adadelta. Only g is changed from Adagrad, and the new version is shown in equation 2.10, where d is a decay factor that controls how fast old gradients are forgotten.

$$g_T = g_{T-1} d + (1 - d)(\nabla_{w_T} f) \odot (\nabla_{w_T} f) \tag{2.10}$$

2.5.1.2 Adam — Adaptive moments

Another alternative method for updating variables in SGD is called Adam [15], or Adaptive moment estimation. This method keeps track of moving averages of the first and second moments of the gradients and uses these values in its update rule for the parameters. That means that if a certain parameter's derivative has had a very similar value, close to say a, in several consecutive time steps, then the estimated first moment (the mean of those derivatives) will be close to a for that variable and the estimated second moment (the variance of the derivative) will shrink towards

0. In this case it is considered safe to increase the update rate for this variable. On the other hand, if the derivative of a certain variable with regard to the error function has had values both positive and negative and with a large range in several consecutive time steps, then the first moment will be close to 0, and the second moment will be large. In this case, the rate of update for this parameter should be kept more conservative. Thus, the update rule for the Adam method is given in equation 2.11 where m_1 and m_2 are moving averages of the first and second moments of the gradient respectively, λ is the learning rate and z is a fudge factor to avoid numerical instability.

$$w_T = w_{T-1} - \lambda \frac{\nabla_{w_T} f \odot m_1}{\sqrt{m_2} + z} \tag{2.11}$$

2.5.1.3 Avoiding overfitting — regularization and early stopping

A problem in most machine learning applications is known as overfitting — when a learnt model is so detailed and/or powerful that it is able to predict correct labels for a training data set very well, but the model is not able to get generalized good classification performance on data outside of the training set. One way to reduce this problem is by penalizing models that make use of many model parameters to very well fit to the training data. This can be done by regularization, where having many high values for model weights are penalized in the loss function. For instance, if L2 regularization is used, then the L2 norm of the model's weight vector is added to the loss function (weighted by a factor).

Another way to reduce overfitting is by early stopping. In early stopping, a part of the training data is not actually used for training the model, but as a check to see whether further training increases the generalized classification performance of the model. The system is trained until the classification performance on this part of the training set does not increase or starts to drop.

2.6 Continuous bag-of-words

Returning to the discussion on word-vector-generation, the following sections present two methods based on neural network techniques. In the continuous bag-of-words (CBOW) model, the objective is to predict a target word when given a context. Mikolov, Corrado, Chen and Dean [2] presented an efficient method to perform this task. The network that they used consisted of three layers: input, projection and output, as shown in figure 2.6.

The first layer is the input layer with one-hot representations of the context words. Thus, the size of each input context vector is equal to the size of the dictionary.

The middle layer is a projection layer, which can be thought of as a look-up table

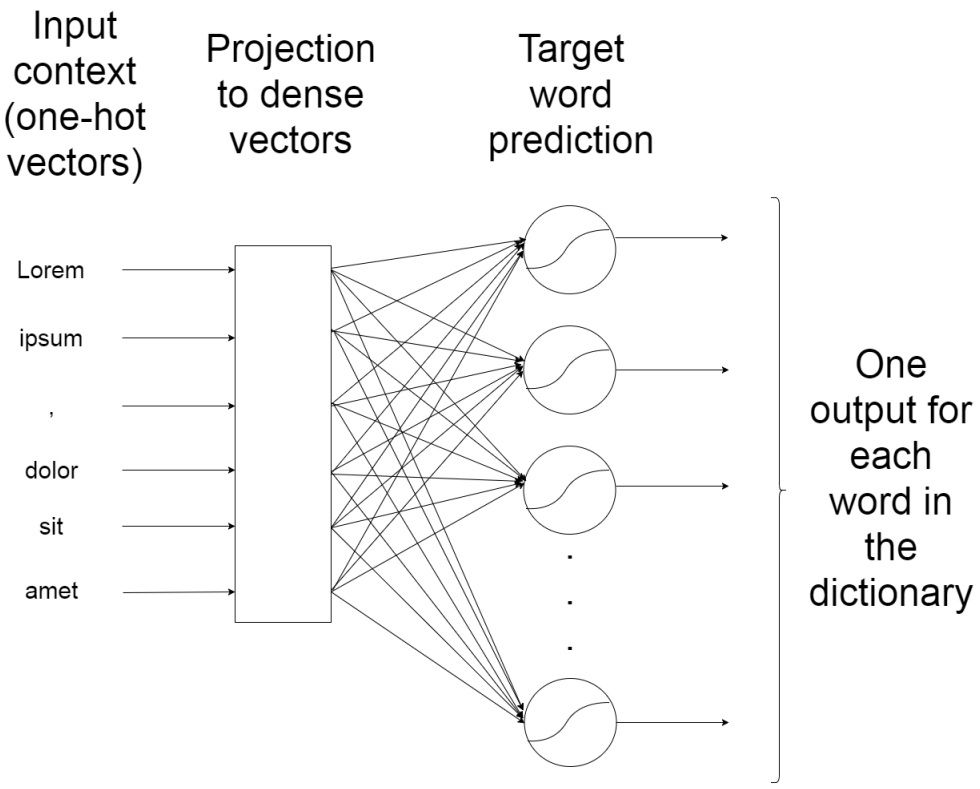

Figure 2.6: Continuous bag-of-words network. Note that the projection layer also includes the summing of all the looked-up vectors.

where a vector representation of the target word is looked up. In a bag-of-words model, the sum or average of the vectors from the context is computed and passed on to the next layer. See equation 2.12 where W is a matrix containing the vector representations, x_t are the input one-hot representations for all T context words. The size (in number of parameters) of the middle layer is the size of the dictionary times the size of the vector representations. However, since only one vector is used per input word, the amount of computation required in this layer is still small. The word vector representations in this layer (W) constitute the output when the training is done.

$$y = \sum_{t=1}^{T} W x_t \tag{2.12}$$

The final layer is a prediction/output layer where the target word is predicted. One could imagine this (fully connected) layer implemented with one neuron per word in the dictionary, and one could use softmax to make predictions as to which word is the most probable target word. However, this would require a number of computations of the order of nv where n is the number of words in the context and v is the size of the dictionary. Since the size of the dictionary can be very large (hundreds of thousands or even millions of words), this would be very inefficient.

One way of avoiding the inefficiency of calculating the full softmax in the final layer

is to use what is called negative sampling [16]. In negative sampling, not all neurons in the output layer are used, but only a randomly sampled subset. For instance, given the context "she walked into the" — "and ordered a drink" and the actual target word "bar", all taken from a corpus, the algorithm could generate a number of random noise target words, like "and", "from" or "home" etc. The objective for the learning algorithm is then to distinguish between only the sampled words and the actual target word.

The expression in 2.13 gives the objective for negative sampling in CBOW for one training example. Just as in GloVe, each word x has two vectors: one target vector representation (w_x) from the output layer and one context vector representation ($\tilde{w}_x$) from the middle projection layer. In 2.13, w_O is the vector for the target word, $\tilde{w_I}$ is the sum or average of the context vectors. $\mathbb{E}$ is the expectation and P_n is the distribution used in the sampling for the k noise sample vectors, w_x.

$$\log \sigma(w_O^T \tilde{w_I}) + \sum_{x=1}^{k} \mathbb{E}_{w_x \sim P_n}[\log \sigma(-w_x^T \tilde{w}_I)] \tag{2.13}$$

Another way to avoid the inefficiency of a large final layer in the network is to use a hierarchical softmax [16] as illustrated in figure 2.7. In the hierarchical softmax, like with negative sampling, only a few of the neurons in the output layer need to be involved in the training for each training example. However, the interpretation of the outputs is different from the methods described in the previous paragraphs. In order to understand the hierarchical softmax, one can imagine the final output layer laid out as a binary tree, where each leaf corresponds to an output word and each non-leaf corresponds to a neuron in the final layer. Now, only the nodes on the path from the root of the tree to the leaf corresponding to the correct output word need to be used for any given training example. The output of a neuron on the path can be either closer to 1, suggesting that the "left" child is the likeliest or closer to 0, suggesting that the "right" child is the likeliest. Thus, one can multiply the outputs along the path (or 1 - the outputs in the case of the "left" child being correct) in order to calculate the predicted probability of the leaf node being in the context of the target word. When training, similarly, only the weights associated with these nodes need to be updated. Typically, the binary tree that is used is built from the Huffman encoding of words given the words' occurrence counts in a large corpus, leading to short paths for common words and longer paths for uncommon words.

The expression in 2.14 gives the objective for hierarchical softmax for a training example. Here, n_O is the leaf node for target word O, $h(n_O)$ is the set of vectors (weights) in the neurons on the path to n_O, and the function $f(n_O, w)$ is equal to -1 if n_O is in the "left" subtree under w and it is equal to 1 if n_O is in the "right" subtree of w.

$$\sum_{w \in h(n_O)} \log \sigma(f(n_O, w) w^T \tilde{w}_I) \tag{2.14}$$

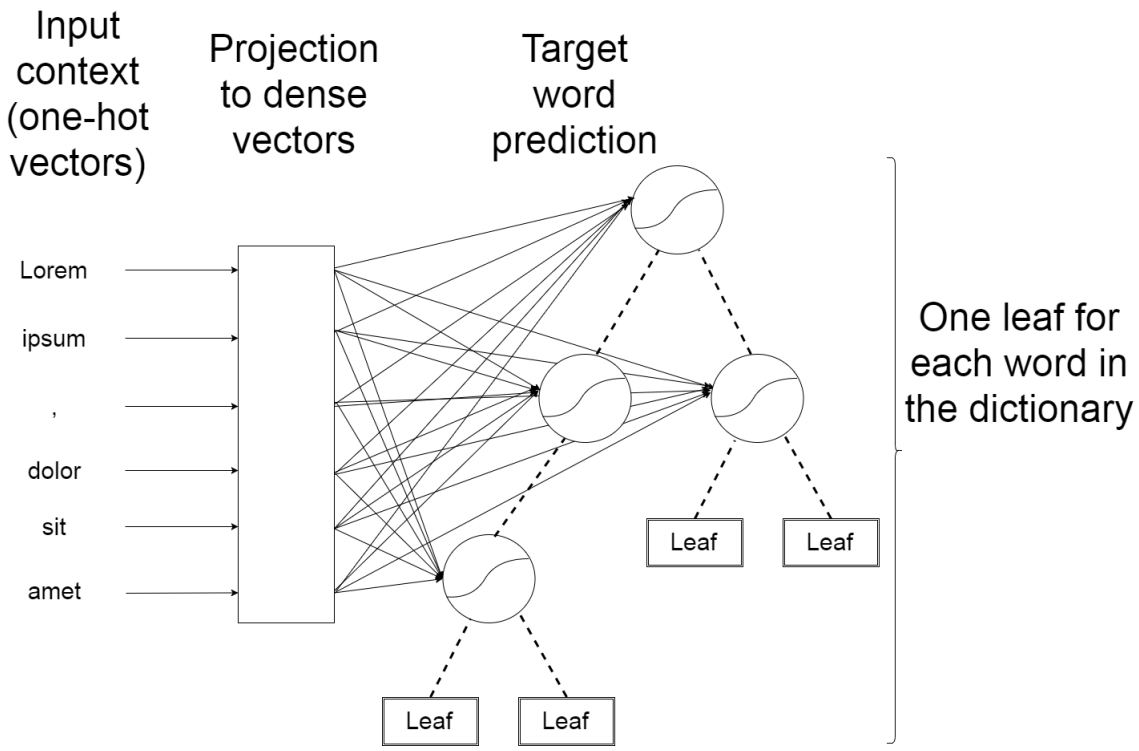

Figure 2.7: Continuous bag-of-words network with hierarchical softmax. Note that the projection layer also includes the summing of all the looked-up vectors.

2.7 Skip-gram

In the same article that presented the CBOW method described above, Mikolov, Corrado, Chen and Dean [2] also suggest another method: the skip-gram method. Here, a very similar neural network is used as in the CBOW method, but instead the objective in the training is to predict words in common contexts of a given word. A constant k is chosen, denoting the number of words to be predicted for each training example. These k words are chosen among the context words for the training example — thus, the method does not (necessarily) train on the full contexts of each example. The probability of context words being chosen can be skewed so that words closer to the given word are chosen more often.

The network used for this method also has three layers: input, projection and output. The input layer again consists of one-hot vectors, but this time for each training example there is only one one-hot vector for the given word whose context is to be predicted. The middle layer is a projection layer, just as in the CBOW method. Finally, the last layer is a hierarchical softmax layer or a softmax layer with negative sampling, just as in the skip-gram method.

2.8 Another prediction-based method: A log-bilinear model with noise-contrastive estimation

A final method, noise-contrastive estimation (NCE), for finding vector representations for words was described by Mnih and Kavukcouglu [1]. This method shares many of its basic principles with the CBOW and skip-gram methods described above, but it allows some additional parameters not available in the CBOW or skip-gram methods. NCE is also based on either predicting a word, given that word's common contexts in a corpus, or, given a word, predicting that word's common contexts. In this thesis, only the first of these options is explored for the NCE method.

In NCE, just like the other methods, each word has two vector representations: one context representation, $\tilde{w}$, and one target representation, w, with w used as the output of the algorithm after the training. Furthermore, each word also has a scalar value, b, that is connected to how common the word is in the corpus. Finally, there are vectors c that function as weights that multiply the context vectors. Thus, c can be used to find which positions and which elements in the vectors of the context are more or less important for predicting the target word. When these parameters are combined in the way described in equation 2.15, a score is produced (note that $\tilde{W}$ in this equation is actually all the context word vectors), which signifies how well the word represented by w fits the context represented by the vectors $\tilde{w} \in \tilde{W}$.

$$s(w, \tilde{W}) = (\sum_{i=1}^{N} c_i \odot \tilde{W}_i)^T w + b \tag{2.15}$$

If one wants to calculate how probable a word is according to a model (an assignment of values to the parameters for an entire dictionary) in a certain context, one can then calculate the above score for all words in the dictionary, and then use a softmax (look back to equation 2.7) to get a probability score. In order to improve such a model, one can then use stochastic gradient descent to update the weights so that the probabilities match the actual word cooccurrence counts in a text corpus.

However, with most real-world use-cases, just as with skip-gram or CBOW, the softmax sum would be prohibitively expensive to compute naively since the dictionaries are too large. Instead, a method called noise-contrastive estimation (NCE), similar to the negative sampling of [2] can be used to train the log-bilinear model. In NCE, random sampling is used, where random noise target words are generated and compared to an actual target word, given a context from a corpus. Next, NCE updates the parameters of the model such that the score for the actual target word is increased and the score for the noise targets are reduced. This is done with gradient descent, where the gradient is calculated based on what scores the model assigns for the target word and the noise words occurring in the context. (For details for how this gradient is calculated, see [1].)

The actual objective for NCE is given in 2.16. Here P_n is the noise distribution, $P_d^{\tilde{W}}$

is the trained model's distribution of words in the context $\tilde{W}$.

$$\mathbb{E}_{w \sim P_d^{\tilde{W}}}[\log(\Delta s(w, \tilde{W})] + k\mathbb{E}_{w \sim P_n}[\log(1 - \Delta s(w, \tilde{W})] \tag{2.16}$$

$$\Delta s(w, \tilde{W}) = s(w, \tilde{W}) - \log(kP_n(w)) \tag{2.17}$$

Mnih and Kavukcouglu found experimentally that using Adagrad with NCE and SGD reduced the learning time and that it was possible to find good word vectors in hours or days on a single CPU core with corpora with sizes in the tens or hundreds of million words. However, to be able to do this, they simplified the Adagrad algorithm to only keep track of one adjusted learning rate for each word in the dictionary, rather than one such rate for each element of each vector for each word.

3

Evaluating the performance of word vectors

There are several ways to measure the performance of techniques such as those described in the previous chapter and the vector representations that they generate. The approach taken in this thesis is to use the performance in a natural language processing task, semantic role labeling, as a proxy for the performance of the vector generation techniques. This is called an extrinsic measurement strategy, but there are also more direct, intrinsic techniques, which have more commonly been used for such evaluations [17]. As background, and to give some results for comparison to the results in this thesis, some intrinsic evaluation methods and results are described in the following. After that and as a conclusion to this chapter follows a description of the architecture of the SRL system of Zhou and Xu, a similar system to the one used in this work [10].

3.1 Intrinsic measures

Intrinsic measures of word vector quality consider some "simple" test of semantic or syntactic similarities between words. Four groups of such measures used by Schnabel, Labutov, Mimno and Joachims [17] are 1) relatedness tests, where human rankings of similarity between words are compared to some measure of similarity between the generated vectors, 2) analogy test, where the task is to answer questions like "a is to b as c is to what?", 3) categorization, where the objective is to cluster vectors into groups where all words should be related to each other and 4) selectional preference tests, where the objective is to say if a noun is commonly used as the subject or object of a given verb.

In a comparison using 14 tests from all these four categories evaluating the representations generated by the techniques CBOW, GloVe and some other vector factorization methods, CBOW showed better performance than all other methods in most of the tests. However, in one relatedness test and in the selectional preference tests, either GloVe or another matrix method had better results.

Another investigation of word vector performance was made by Baroni, Dinu and Kruszewski [18], using intrinsic measurement techniques very similar to the ones

used by Schnabel et al. They trained 48 CBOW models (the word model is here used to mean a set of word vectors) using various parameter settings and 36 models using different count-matrix methods. In all the tests, CBOW outperformed the matrix-based models. This was true, both if the best model for each subtask was considered, or if only one model was considered across all subtasks.

3.2 Extrinsic measures

Schnabel et al [17] also performed some extrinsic tests: noun phrase chunking and sentiment classification. Chunking [19] involves finding boundaries for phrases such as noun phrases or prepositional phrases in text. Sentiment classification means to decide whether the attitude expressed in a text is positive or negative. In [17], the texts that were used were movie reviews.

For the noun chunking task, the differences between the methods were all within half a percent of each other in classification performance, with one of the matrix techniques achieving the best results and CBOW and GloVe in the bottom positions. In the sentiment classification task, however, CBOW reached the best results. The conclusion drawn in [17] is that the intrinsic measures do not always predict the performance in the extrinsic measures.

Noting that there is a dearth of extrinsic evaluations, with intrinsic evaluations being much more common, Nayak, Angeli and Manning [20] created a tool for extrinsic evaluation, comprising six different language tasks: part-of-speech tagging, chunking, named-entity recognition (classifying an entity as a person, location, organization or other), sentiment classification, question classification (classifying questions by the type of answer that is expected) and a task that involved understanding the meaning of whole phrases. Their article only presented the method of evaluation however, and no actual benchmark results.

The small number of published extrinsic evaluations, and the apparent weak correlation between intrinsic and extrinsic evaluation performance, indicate that further extrinsic evaluations may shed more light on the strengths and weaknesses of various word vector generating techniques. Using SRL as a testbed for word vectors is novel and interesting, not just because of the general lack of extrinsic evaluations, however, but also because extrinsic testing has generally used less complex tasks than SRL.

The next section describes semantic role labeling and how it is used for extrinsic performance evaluation in this work.

Input 1	He	opened (P)	the	door	,	entering	the	house	.
Output 1	Agent	Predicate	Patient						
Input 2	He	opened	the	door	,	entering (P)	the	house	.
Output 2	Agent					Predicate	Location		

Table 3.1: An example of SRL tags for an input sentence with two predicates. The predicate label is given in the input, shown as (P). A separate set of labels is generated for each predicate of the sentence. Empty columns denote a null tag.

3.3 The SRL task

This section describes machine learning strategies for semantic role labeling. This task deals with finding the semantic relationship in a sentence between a predicate and the other phrases in the sentence. The predicate is generally a verb, but can also be other word classes with action semantics. The other phrases can be tagged with roles such as agent (somebody who volitionally and intentionally performs or initiates an action), patient (the entity affected by the action and made to change state in some way), location (place of the action), instrument (a tool used to perform the action) etc. Note that these definitions are not meant to be precise, but to give a prototypical description of the roles [5]. Not all words need to have a semantic relationship to the predicate, and those that do not should be given a null tag.

As an example of a role labeling, see table 3.1. As can be seen in the example, a sentence may contain more than one predicate, and each such predicate will get its own role labels.

Traditional strategies have generally involved manually engineered features. That is, researchers have by hand designed a set of features describing the words of a text, to be used as input to some machine learning algorithm. A brief introduction to some such strategies is given in section 3.3.1. The system used here for word vector evaluation, however, does not make heavy use of manually crafted features, instead relying mostly on word vectors as the input. This system is described in section 3.3.2.1.

3.3.1 Traditional strategies for SRL, using manually crafted features

Gildea and Jurafsky [21] made an early attempt to solve the SRL problem with machine learning techniques. Their most successful method made use of several hand-crafted and generated features of the input text and used statistical methods to find probable labels for the input words.

The features that were used included parsing information. For each input sentence a parse tree was created using a standard parser. Such a parse tree gives a representation of the syntactical structure of a sentence, indicating hierarchically which words

form units, such as noun phrases, verb phrases and prepositional phrases. Then, for each input word, a feature that they used was the path in this tree from the input word to the predicate that was considered.

Other features included voice (active or passive), position in the sentence (before or after the predicate), the head words of phrases and syntactic function (subject or object). In order to predict SRL labels, various statistical methods were experimented with. The simplest of these was just to consider the cases in the training data with matching feature values, and then to find which role was the most common among those matching examples. However, not all combinations of values for the features were present in the training data, so when making predictions for labels for sentences not taken from the training data, some interpolations had to be made from the data that was available.

One example of a way to improve the accuracy for input word combinations that had not been seen in the training data was to use a large text corpus to find words that were commonly used in similar ways. The expectation maximization algorithm [22] was used to find clusters of words with similar usage patterns. Then, if one or a few of the words in a cluster were seen in the training data, the labeling information given in the training data could be generalized and used also for the other words in that same cluster.

The labeling accuracy achieved by Gildea and Jurafsky differed widely depending on which features were available. In particular, if parsing information was available, the classification performance increased by some 20 percentage points as compared to when such information was not available to the statistical classifier.

Other work in the same vein as Gildea and Jurafsky's has focused on experimenting with other kinds of classifiers and other kinds of features. For instance, Pradhan et al. [23] used a combination of support vector machines (another kind of machine learning tool for binary classification) and a sigmoid unit. Pradhan et al noted that a large proportion of the input words are supposed to get a null label, as they do not have a semantic role relationship to the predicate to be considered. So, they constructed a two-stage model where the first stage is a sigmoid unit that classifies input words as either null or non-null and the second stage is a set of support vector machines, one for each output class, only used when the first stage classifies the input as non-null.

The features used by Pradhan et al. included the ones used in [21], but also some new ones, like the part of speech of the head words of a phrase, whether a word denoted a person/organization/location or various kinds of numerical data, shortened parse tree paths (to mitigate the sparsity issues discussed above), and several others.

Another learning system using manually crafted features is the one of Johansson and Nugues [24] who used a different kind of parsing — dependency parsing. Here, a sentence is not split up into constituents but rather each word, w is assigned a head word h, based on what word syntactically governs w. The main verb of a sentence is considered as the root of the resulting tree of dependencies and it is thus not assigned a head.

Johansson and Nugues trained a system to jointly perform dependency parsing and SRL analysis. The dependency parsing was done with a variant of a support vector machine, and the SRL part was performed by several sigmoid units. The sigmoids were arranged to perform their task in three steps: 1) decide the sense of the predicate, 2) identify what the arguments are, which were to be labeled and 3) label the arguments with their roles. Different features were used as input to the various sigmoids, such as dependent words, part of speech tags for dependent and proximal words, the sense of the predicate (for disambiguation), voice and some tags used to disambiguate the parse information.

The system described above was used to generate several candidate solutions to the whole sentence labeling, and then this list of candidates was filtered to exclude such ones that were not linguistically feasible, and finally a choice was made among the 16 best candidates in the filtered list. This final selection was made with a softmax classifier, using the logs of the probabilities output from the previous steps as its input features.

3.3.2 SRL with neural networks

The methods in the prior discussion has some disadvantages that can be mitigated by the use of neural networks. First, they used manual feature engineering, which requires a great deal of work from the implementer. Second, they often need parse information as input or they rely on a parser to generate such information.

An early attempt to perform semantic role labeling with multi-layer neural networks was presented by Collobert and Weston [25]. They did not use any parsing information as input features, but only part-of-speech tags and the words of the input sentence. The words were each given encodings as vectors, and these vectors were learnt automatically by the neural network. Thus, the first stage of the network was a vector look-up, similar to the first layer described in the sections about the CBOW (2.6) and skip-gram methods (2.7). The rest of the network consisted of a layer where the input word was combined with a window of context words, a tanh layer, a projection layer and a final softmax layer with one output for each possible classification for the input word. In their work, only per-word performance figures were given, and they were close to state-of-the art at the time. However, with performance measured on the phrase level, a similar system constructed in [10] performed poorly.

3.3.2.1 Sequence labeling with long short term memory

Better performance on the SRL task with neural networks can be achieved if one makes use of artificial neurons with memory. One example of such neurons is the long short-term memory (LSTM) neurons introduced by Hochreiter and Schmidhuber [26] and improved by Gers et al. in [27] with forget gates and in [28] with peephole connections. Zhou and Xu [10] used a network comprising such neurons to solve

the SRL task, together with a CRF sequence model and traditional neural network units. Below is a description of LSTM units, CRF sequence models and their use in Zhou and Xu's learning system. For a pictorial representation of this whole system, see figure 3.2.

The neural units that Hochreiter and Schmidhuber proposed do not just feed forward a value, but they can also use their input to update a stored value that can be used when the next word is input for the next time step. The LSTM unit contains three "gates" that control what value to store and what value to output. The LSTM unit also contains a traditional non-linear unit like the ones described in section 2.5, which combines and squashes the cell's inputs into a single value that can be remembered or added to the value already remembered. In figure 3.1, showing an LSTM artificial neuron, the gates are labeled In, Out and Forget and the unit that calculates what value to add to the remembered value is labeled Value. In the center is the stored value, i.e. the remembered state of the neuron, and on top is the output of the whole neuron. As can be seen in the picture, not only the state and the output of the neuron are passed on to the next time step, but the output from the gates from time t is also used as inputs at time $t + 1$, for all neurons of the same layer.

All gates function like the neurons described by equation 2.3, and different non-linearities can be used for different gates. If "peephole connections" [28] are used, the set of inputs to the gates also includes the current value of the state of the neuron. The input gate decides whether the value stored in the cell should be updated, and the forget gate decides whether the previous value should be discarded, as seen in equation (3.1) where s_t is the value stored in the neuron, f_t is the value of the forget gate, i_t is the value of the input gate and v_t is the value from the value sigmoid, all at time step t. The output gate decides if the value stored should be output to the next layer at this time step and the current layer at the next time step, as seen in equation 3.2, where y_t is the output from the neuron, s_t is the value of the stored state and o_t is the value of the output gate, all at time step t.

$$s_t = s_{t-1} f_t + i_t v_t \tag{3.1}$$

$$y_t = s_t o_t \tag{3.2}$$

For each LSTM neuron in each layer of the network, there are separate weights for each gate as well as for the inputs to each neuron. However, the addition of more time steps does not increase the number of weights, but rather they are reused for each time step. Such reuse is what signifies a recurrent neural network.

3.3.2.2 Using LSTM for language tasks — a bidirectional network

In an LSTM network as described above, information only flows in the forward time direction in forward propagation. This constitutes a problem when the objective for the network is a language task that deals with semantics, since the meaning of a

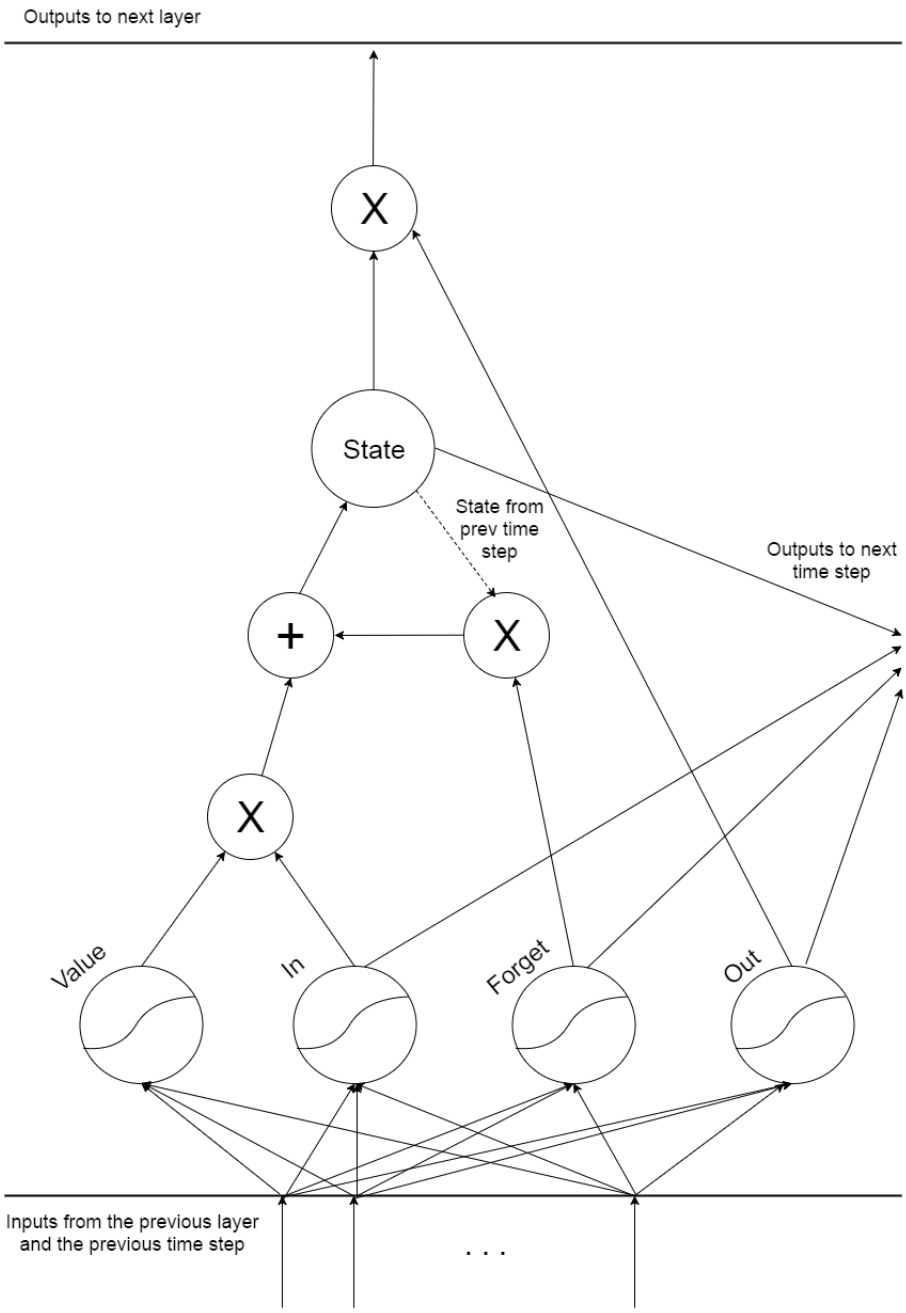

Figure 3.1: A long short-term memory (LSTM) unit.

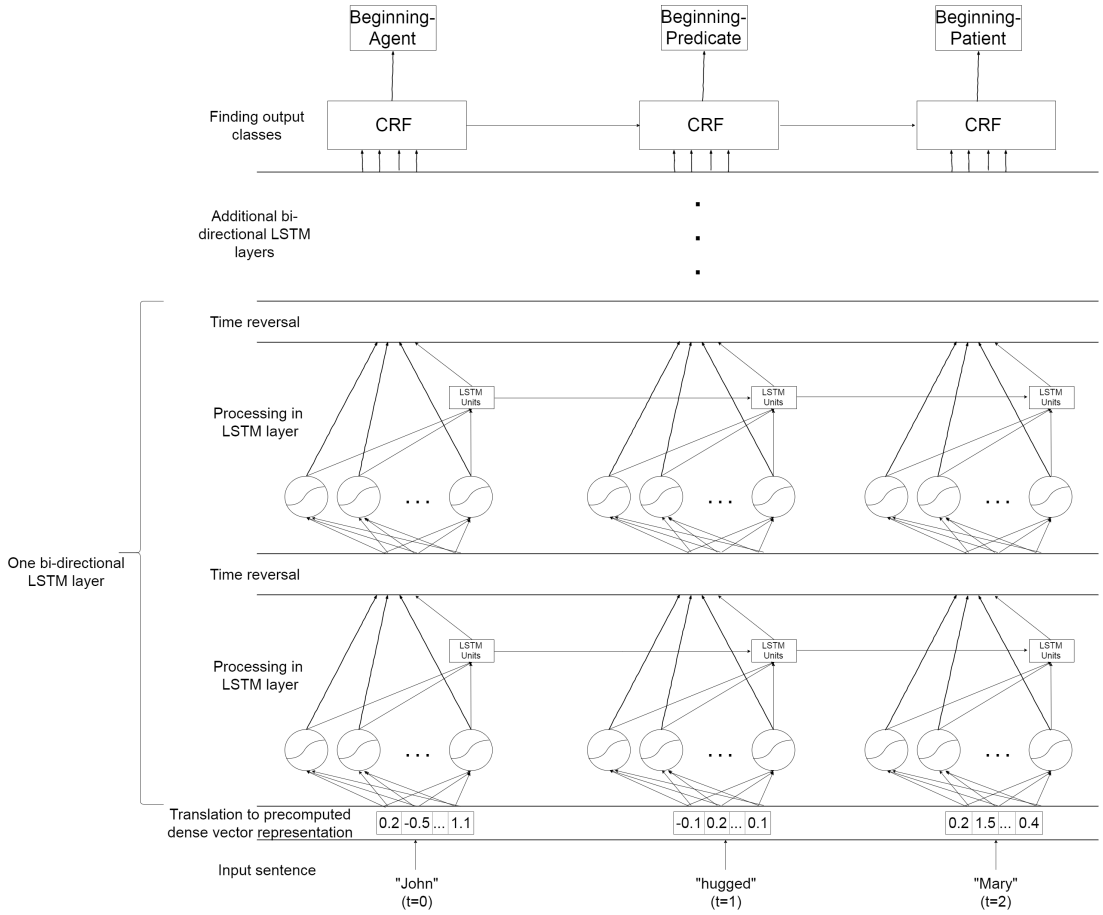

Figure 3.2: An overview of the neural network used in this thesis for SRL. Note that the boxes marked LSTM represent several LSTM units. Peephole connections (from the LSTM states to the gate inputs) are not shown. Beginning-X in the output means that the tag X begins at that word.

word in a sentence or phrase does not depend only on the words prior to that word, but also on the words that follow. For instance, consider the sentences "He arrived in Gothenburg" and "He arrived in December." To be able to label the word "in" in the phrases "in Gothenburg" and "in December", one needs to know the meaning of the following word. In the first case it is a place, and so the label "Location" would be correct, but in the second case it is a time expression, and so the label "Time (Temporal)" would be correct.

In order to mitigate this problem, Zhou and Xu [10] used a bi-directional LSTM network. In such a network, between each layer, the order of the inputs is reversed so that every other layer of the network sees the input in the backward time direction. (See the middle of figure 3.2 for a pictorial representation of this structure.) In their testing, Zhou and Xu achieved the best results with a network utilizing 8 layers (this is, 4 bidirectional layers) with 128 LSTM units in each layer.

In their bidirectional LSTM network Zhou and Xu included a layer of traditional neural network units between each LSTM layer in the configuration shown in figure 3.2. In their paper there is little information about this layer. It is not explicitly stated which kind of units are used and it is not entirely clear what the dimensionality of this layer is.

3.3.2.3 Input features for the neural network

The inputs to Zhou and Xu's [10] network contained for each time step 1) a vector representation of the word for that time step, 2) a flag equal to 1 if the word was within the predicate's context, 3) a vector representation for the predicate as well as 4) a vector representation for the predicate's context. Thus, if the vector size was v values, and the context length was c then the total input vector for each time step would contain $v(c+2)+1$ values. Note that this context length is not (necessarily) the same as the context lengths described in the sections about vector generation.

Field	Current word	Flag	Predicate	Context of predicate
Size (number of 32-bit floats)	n	1	n	nc

Table 3.2: Inputs to the network for each time step. Here n is the size of each word vector and c is the length of the context in words.

3.3.2.4 Conditional random fields as classifiers for sequences

Just as an LSTM cell can be better than a traditional neural network unit for dealing with sequences, the final stage of the network, which outputs the labels, can also be adapted to better suit sequences. If for instance the IOB (Inside, Outside, Beginning) tagging scheme is used (see section 4.3), there is a very strong correlation between adjacent labels, and some labels can never appear next to each other. For instance, if one word is marked as the first word (beginning) of an agent phrase,

then the next word cannot be marked as inside any other label than an agent. A sequence model that makes use of such background information about dependencies among adjacent labels is a conditional random field (CRF) [29]. Below is presented a first-order linear-chain CRF, which is a simple kind of CRF that is often used for language tasks.

A first-order linear chain CRF uses two matrices of values: the first matrix, O, contains weights associated with predicting a label, given an input. For the machine learning system used in this thesis, the inputs to the CRF are the outputs from the bidirectional LSTM network. The other matrix, A, contains unnormalized state transition scores, that is, values that can be used to predict a label given the previous label in the output sequence. The predicted, unnormalized, probability score of a sequence of outputs, or states, y, given a series of inputs, x, over T time steps in the sequence would then be proportional to the expression in equation 3.3. Note that $O_{y_t} x_t$ is the dot product of a vector in O with the input vector x_t.

$$F(x, y) = exp(\sum_{t=1}^{T} O_{y_t} x_t + A_{y_t, y_{t-1}}) \tag{3.3}$$

3.3.2.5 Output labelings using a CRF

If one wants to use a CRF to predict the most probable sequence of labels, one could imagine a naive approach where one calculates the probability for all possible output sequences. To get the probability for this sequence, one would have to normalize F over all possible output sequences y. It is usually too computationally expensive to find this normalizing sum, usually called Z or the partition function, as well as F for all the output sequences naively. (see equations 3.4 and 3.5, where y ranges over all possible output sequences.) However, to make a prediction for the most probable sequence, one does not need to calculate these values. Instead, a dynamic programming technique, the Viterbi algorithm is commonly utilized to make label predictions. For training, there is also a dynamic programming algorithm called forward/backward (not the same as forward and backward propagation described previously) that can be used. Both these algorithms simplify the calculations by splitting them up so that only one time step is considered at once. The time complexity thereby drops from exponential in the number of time steps to polynomial in the number of time steps and number of output classes. For details see [29].

$$P(y|x) = \frac{F(x, y)}{Z(x)} \tag{3.4}$$

$$Z(x) = \sum_{y} F(x, y) \tag{3.5}$$

4

Implementation

This section covers implementation details for the word-vector generation algorithms, the neural network learning system, and the methods for testing the performance of the vectors.

4.1 Implementation of word-vector generation algorithms

An implementation of NCE was written in Python 3.5 and Numpy. It followed Mnih and Kavukcouglu's [1] description, except for some details. Mnih and Kavukcouglu [1] made use of Adagrad, but it was simplified such that only one learning rate per word in the dictionary was maintained. This was useful for them, since in their testing they used word vectors of sizes ranging up to 600 dimensions. However, for the system used in this thesis, the sizes for the word vectors are much smaller, and so maintaining a separate learning rate for each model parameter was viable. Also, instead of Adagrad, Adadelta was used in order to reduce the possibly inordinate impact of the first few training examples for a word.

The function of the NCE implementation was tested informally by considering the closest neighbors, in the sense of smallest cosine distance, for some common words. In table 4.1, ten common nouns and the words whose vectors are most similar to the vectors corresponding to those nouns are shown. For these examples, 100-element word vectors created from 6-word position-dependent contexts were used. The closest neighbors that were found are mostly close in meaning, which is what would be expected if the implementation was correct.

Standard implementations of the GloVe [3], CBOW and skip-gram methods [30] are available and these were used. The implementation of NCE is available upon request from the author of this thesis. All vectors were normalized so that all dimensions had a mean of 0 and standard deviation of 1 before they were used as input in the neural network. The training time for the vectors was not a research question for this thesis, but as a general comment, training GloVe vectors took approximately a few hours, for CBOW and Skip-gram the time taken was less than a day and for NCE the time was a few days.

Target word	Most Similar Words		
Time	Distance	Moment	Period
Year	Month	Day	Week
People	Citizens	Residents	Men
Way	Terms	Method	Direction
Day	Week	Night	Month
Man	Woman	Girl	Gentleman
Thing	Things	Reason	Sight
Woman	Girl	Man	Person
Life	Vocations	Lodgings	Pleasures
Child	Grandchild	Baby	Pet

Table 4.1: Closest neighbors for some common words in the output of the NCE implementation.

4.2 Implementation of the SRL-system

The whole learning system, including neural network and CRF classifier, was implemented in Python using the Tensorflow [31] framework. Both the Windows CPU (version r0.12) and Linux GPU (version r0.11) versions of Tensorflow were used. For the LSTM units, Tensorflow's nn.rnn_cell.LSTMCell class was used, with peepholes activated. For the traditional neural network units, Tensorflow's nn.relu class was used. For the final CRF layer (including loss function) Tensorflow's contrib.crf module was used.

As mentioned in section 3.3.2.2, the number of traditional network units in each layer was not specified by Zhou and Xu. In the tests, this number was set to $128+n$ where n is the number of elements in the input for one time step and 128 corresponds to the number of LSTM neurons in each layer. This meant that the number of values output by the sigmoid layer was the same as the number of values in the input to the sigmoid layer for all layers except the first one.

4.3 Training and evaluation datasets

The corpus that was used for training all word vectors was a dump of all Wikipedia articles from 2006. The corpus contained approximately 600 million tokens, including punctuation and clitics. The dictionary for the dump contained approximately 1.5 million different word types initially, but many of these had a very low occurrence count. Therefore, those that occurred less than 10 times in the whole corpus were consolidated into one "low-frequency" word type. Furthermore, numbers were also consolidated into five categories: negative integers, small positive integers (less than 1500), medium positive integers (1500 to 2100) large positive integers (larger than 2100) and numbers with a decimal point. This consolidation reduced the size of the dictionary to just under 500 000.

Input	The	angered	man	hit	the	dog	.
Labels	B-Agent	I-Agent	I-Agent	B-Predicate	B-Patient	I-Patient	O

Table 4.2: An example of the Inside Outside Beginning (IOB) tagging scheme.

4.3.1 Data sets for the SRL task

The CoNLL-2005 shared task dataset [11] was used for the supervised training of the neural network and CRF sequence model. This is a corpus of texts from The Wall Street Journal for which manual SRL tagging has been performed, and these tags were used as correct training labels. There are three subdivisions in this dataset, called test, development (dev) and train. The train set, containing some 90000 predicates, was used for training the system. The use of development and test is described in the next section. Sentences in the datasets can contain many predicates, and each sentence-predicate pair is considered as a training example.

Some pre-processing was performed before it was used to train the learning system: the correct outputs from the dataset were converted to an IOB (Inside Outside Beginning) tagging scheme [32], where each token was either marked as Outside, meaning no tag, Beginning-X, meaning that the tag X started at that token or Inside-X, meaning that the token was marked as the continuation of the tag X. Inside-X could thus only occur after Beginning-X or another Inside-X. See table 4.2 for an example of the IOB tagging scheme. Furthermore, a few (less than one percent) of the sentences in the CoNLL-2005 dataset were not tagged with a predicate and these sentences were neither used for training nor testing.

The dev, test and train sets contained some tags that only occurred a few times (less than 40). These tags were changed to Outside in the train set, and the labels output by the network were limited to the ones that occurred in this modified train set. For the testing, the classification for the removed tags was always considered as errors.

4.4 Evaluation benchmark

The evaluation of the different vector-generation techniques was also performed with the standard CoNLL-2005 shared task [11] benchmark. The dev set was used for the stopping criterion — when the classification scores for the system did not increase for the dev data set, training was stopped to reduce overfitting. The dev set results were also used for comparisons of parameter setting performance.

The test set was used for the comparison of the methods against each other. The test set includes two parts: one with text and labels from the Wall Street Journal and one with texts and labels from the Brown corpus, which is a more general collection of texts published in America. Since the latter texts are generally from other genres than those found in the Wall Street Journal, it is often the case that the performance

on the Brown part of the test is lower than on the Wall Street Journal part. The test and dev sets are circa one twentieth the size of the train set.

For both the stopping criterion and the evaluation criterion, the F1 score was used. The F1 score is the harmonic mean of the recall and the precision, where the recall (see equation 4.1) is defined as the percentage of the training labels that were correctly identified by the learning system and the precision (see equation 4.2) is defined as the percentage of the labels output by the learning system that were correct. The F1 score is calculated on the phrase level, rather than on the word level. Thus, in the example sentence in table 4.2, for the first tag be considered correct, "The" would have to be labeled "B-Agent" and also "angered" and "man" would have to be labeled "I-Agent".

$$\text{recall} = \frac{\#\text{correct output labels}}{\#\text{correct labels in data set}} \tag{4.1}$$

$$\text{precision} = \frac{\#\text{correct output labels}}{\#\text{ output labels}} \tag{4.2}$$

$$\text{F1} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{4.3}$$

4.5 Overview of investigated settings for the vector generation algorithms

The word-vector-generation methods were compared against each other, and different parameters for the methods were also tested. For all the models, context length and vector size were varied. Context length varied between 6 and 14 words, and one-sided contexts as well as two-sided ones were used where the option was available. The vector size ranged from 32 (which is the vector size used in [10]) to 150 values (the maximum possible size such that the training processes did not run out of memory on computer systems with 8 GiB of memory.) As a midpoint, 100 element-vectors were also used. For NCE, both position-dependent contexts and position-independent contexts were tested. CBOW and Skip-gram can be trained using either negative sampling or a hierarchical softmax and both these methods were used. Negative sampling was used with both five and ten negative samples. GloVe and NCE can be used with one-sided or two-sided contexts, and for both methods both types of contexts were tested. For a summary of these parameters, see table 4.3.

Method	Parameter	Settings
All Methods	Vector length	32
		100
		150
	Context size	6
		10
		14
NCE	Position-dependence	Independent
		Depdendent
CBOW and Skip-gram	Training method	Hierarchical softmax
		Five-sample negative sampling
		Ten-sample negative sampling
GloVe and NCE	Context shape	One-sided
		Two-sided

Table 4.3: Investigated vector generation parameters.

5

Results

This chapter presents the results from the SRL task. First, a baseline is established by the use of random word vectors. In order to investigate which method is better, first the best parameter settings for each method is found by comparison of results on the development (dev) set, and then for the best performing model for each method, their results on the Wall Street Journal and Brown test sets are compared. In order to answer whether the setting of some parameters have a statistically significant effect, Friedman's p-values are calculated for the dev set results.

5.1 Preliminary testing

Some preliminary testing showed that the implemented learning system was not able to reach anywhere close to the performance levels presented in [10], which is an F1 score of circa 82 for the test set and 79 for the development set. Therefore, some adjustments were made, including setting the intermediate non-LSTM layers to use rectified linear units, which helps propagate gradients through a deep network more efficiently. Also, the Adam training method was used [15] rather than the Adagrad method used in [10]. Finally, a context size of one word on either side of the predicate was used for the input into the network, as described in section 3.3.2.3. This increased performance, but still not quite to the levels presented by Zhou and Xu.

Since the training method was changed from the one used in [10], suitable values for training parameters had to be found in preliminary experiments. These showed that regularization did not have a positive impact on results, something that was confirmed in [10], where it was also found that the model did not show signs of overfitting to the training data. Thus, regularization was not used in the following tests, in slight contrast to the system used in [10] where L2 regularization with a small weighting was used. The learning rate of 0.0003 was found to work well as did a batch size of 200 training examples. Training the SRL system took between 20000 and 30000 iterations until F1 scores stopped improving for the development data set. The number of iterations until convergence was not recorded for the experiments presented below, but there was no obvious impact on training speed from the training method that had been used to produce the word vectors.

Vector size:	32			100			150		
Vectors	Dev	Wsj	Brn	Dev	Wsj	Brn	Dev	Wsj	Brn
Constant	68.33	70.33	54.84	73.54	75.85	60.67	**58.17**	59.98	50.70
Variable	**70.23**	72.23	57.34	**74.33**	76.35	61.99	55.41	57.68	50.11

Table 5.1: F1 scores with randomly initialized word vectors. In the constant vectors case, vectors were randomized at the start of the training and never changed. In the variable vectors case, vectors were randomized at the start of the training and then trained using backpropagation together with the rest of the neural network. Dev, Wsj and Brn refer to the development set, Wall Street Journal test set and Brown test set of the CoNLL-2005 shared task benchmark, respectively.

5.2 Results using random vectors

As a baseline, the performance of the neural network was recorded using randomly initialized word vectors. There were two cases: either the vectors were kept constant throughout the training or they were updated using backpropagation. As can be seen in table 5.1, the results were best for vectors of size 100, and updating the vectors with backpropagation further improved performance in this case, with development set F1 scores of 73.54 and 74.33, respectively. However, for larger vectors, development set F1 scores dropped drastically to below 60 and in the large vector case, allowing backpropagation updates to the vectors even worsened performance. For the rest of the results presented in this chapter, the word vectors were held constant during training.

5.3 Results using vectors from NCE

For the NCE vector training method, the parameters whose impact were investigated were the vector length, the context length, whether the context was one-sided or two-sided around the target word and whether the position of the word in the context mattered or not (i.e., whether vectors c_i in equation 2.15 were kept at the constant value of 1). The results from these experiments are presented in table 5.2.

The F1 scores hover mostly between 76 and 78 for the dev set, with the best result at 77.97 for one-sided 14-word contexts and 100 element vectors. For the position-independent test cases, the scores have a wider range, and slightly lower values in general. The best position-independent result is seen for one-sided contexts and 100 element vectors, at 76.99. There are also some rather extreme outliers. For instance, for the one-sided, position-independent, ten-word context, 150-element vector case, the result is a very low 61.04, close to, but still beating, the results achieved with 150-element random vectors.

The best results for 32, 100 and 150 element vectors are very similar, but the range of scores is larger for 32 element vectors and 150 element vectors. Also when it

		Vector size:	32	100	150
		Context length			
Position-dependent	One-sided	6	76.63	77.85	77.77
		10	77.70	77.63	77.73
		14	77.90	**77.97**	77.79
	Two-sided	6	75.17	75.62	76.63
		10	76.30	77.14	76.83
		14	76.05	76.33	77.25
Position-independent	One-sided	6	75.76	76.99	74.29
		10	75.02	76.95	*61.04*
		14	64.37	69.31	68.08
	Two-sided	6	74.55	76.00	75.87
		10	67.53	76.66	76.43
		14	74.49	76.94	74.52

Table 5.2: F1 scores for NCE on the dev set.

comes to context length, the best results for the different parameter settings are similar.

5.4 Results using vectors from CBOW and Skip-gram

Next, for the CBOW and Skip-gram methods, the investigated parameters were vector size, context length and training method (five-sample negative sampling, ten-sample negative sampling or hierarchical softmax). First, the effect of vector size and context length was investigated, and the results are shown in table 5.3. The F1 scores on the dev set are consistently close to 78, with the best result at 78.42 for 14-word contexts and 150 element vectors. For Skip-gram, the best result is similar, at 78.15 for ten-word contexts and 150-element vectors. However, the Skip-gram results have a wider range, with the most extreme outlier at 71.73 for 14-word contexts and 32-element vectors. For all results in this table, five-sample negative sampling was used.

The effect of the training method was also investigated. For these experiments, the context length was kept constant at ten words. See table 5.4 for the results from these tests. The differences between the training methods are small for CBOW. For Skip-gram, using ten-sample negative sampling or hierarchical softmax yields worse results than does five-sample negative sampling.

	Vector size:	32	100	150
	Context length			
CBOW	6	77.39	78.15	77.59
	10	77.12	77.63	78.29
	14	76.90	77.78	**78.42**
Skip-gram	6	75.68	77.44	77.50
	10	74.95	77.73	**78.15**
	14	*71.73*	75.60	76.53

Table 5.3: F1 scores for CBOW and Skip-gram, using five-sample negative sampling.

	Vector size:	32	100	150
	Training method			
CBOW	Five-sample negative sampling	77.12	77.63	**78.29**
	Ten-sample negative sampling	77.01	75.29	77.03
	Hierarchical softmax	77.02	77.72	78.14
Skip-gram	Five-sample negative sampling	74.95	77.73	**78.15**
	Ten-sample negative sampling	72.29	68.02	74.4
	Hierarchical softmax	71.19	73.18	67.56

Table 5.4: F1 scores for CBOW and Skip-gram for different training methods. All tests used ten-word contexts.

5.5 Results using vectors from GloVe

Finally, for the GloVe method, one-sided and two-sided contexts were tried with varying lengths. As for the other methods, the vector length was also a parameter whose impact was investigated. The results for the GloVe method are shown in tables 5.5. The best F1 score is 77.99, from one-sided 10-word contexts and 100-word vectors. There is no consistent trend in the GloVe results favoring any of the parameter settings over any others, other than 100- and 150-element vectors outperforming 32-element vectors slightly. Due to a mistake, the results marked with an asterisk, actually make use of contexts one word larger than is stated in the table.

5.6 Confirmation of outlier results

Some results from the previous sections were clearly much worse than others, but seemingly not as a result of a clear trend, but appearing apparently randomly in the tables. To see whether this was an anomaly due to random chance or a significant result, three tests which resulted in outliers were re-run. The results can be seen in table 5.6. The word vectors were generated again, and the SRL network was retrained with the new vectors. Two of the results in the re-run were clearly much

	Vector size:	32	100	150
	Context length			
One-sided	6	76.01	77.74*	76.79*
	10	76.51	**77.99***	77.05*
	14	75.66	76.62*	76.27*
Two-sided	6	74.34	76.81	76.51
	10	75.44	77.57	76.58
	14	74.86	*72.90*	76.11

Table 5.5: F1 scores for GloVe. The asterisks indicate a context length of +1 compared to the labels in the table.

Method and parameters	F1 score
Skip-gram, 100 elements, 10-word context, 10-word negative sampling	75.99
NCE, 150 elements, 10-word position-independent context	59.48
GloVe, 100 elements, two-sided 14-word context	75.96

Table 5.6: F1 scores for re-run tests.

better than in the first attempt, reaching levels that would not be clear outliers compared to the other results. In one case, the result was still low, however.

5.7 Statistical comparison of some parameter settings

To investigate whether the choice of vector size or context length had a significant impact on the results, the Friedman p-value was used. For the comparison of vector sizes, the dev set results were put in a matrix A where the columns corresponded to vector sizes and rows corresponded to the valid combinations of method and context length. Next, the values of A were ranked by row, and these rankings were subjected to the Friedman test. For the comparison of context sizes, a similar method was followed to create a matrix B, but now the columns corresponded to context lengths and the rows to combinations of method and vector size. For vector size, the calculated p-value indicated a highly significant effect at $1.4 \cdot 10^{-9}$. Looking back at the data, the 32-element case for each row of A was most often ranked worst and never ranked best, probably causing the low p-value. The effect of context length, however, was not significant, with a p-value of 0.43.

5.8 Comparison of methods

Finally, a comparison of the best results from the four different methods was carried out, using the Wall Street Journal and Brown test sets from the CoNLL-2005 bench-

Method	Dev	Wsj	Brn
NCE	77.97	79.09	66.12
CBOW	**78.42**	79.2	**68.63**
Skip-gram	78.15	79.37	66.14
GloVe	77.99	**79.41**	67.54

Table 5.7: F1 scores for the best vectors from the previous tests. Dev, Wsj and Brn refer to the development set, Wall Street Journal test set and Brown test set of the CoNll-2005 shared task benchmark.

Method	
NCE	100-element vectors, trained with 14-word two-sided contexts, using the context-position-dependent method
CBOW	150-element vectors, trained with 14-word contexts, using five-sample negative sampling
Skip-gram	100-element vectors, trained with 10-word contexts, using five-sample negative sampling
GloVe	100-element vectors, trained with one-sided 10-word contexts

Table 5.8: Parameter settings corresponding to the performance figures presented in table 5.7.

mark. The results are shown in table 5.7. The settings that were used are shown in table 5.8. As is seen in table 5.8, the settings for the best results are not consistent across the methods: both one-sided and two-sided contexts are represented, as are different context lengths and different vector sizes.

As can be seen, the results are all within .35 of each other for the Wall Street Journal test and also very close for the Brown test (remember that the Brown test is much smaller, so larger variations due to random chance are more probable).

6

Conclusion

This work is a comparison of the performance of four different word-vector generation algorithms: GloVe, CBOW, skip-gram and NCE. The vectors generated were used in the input for a bi-directional LSTM artificial neural network and the performance was evaluated with the CoNLL-2005 shared task data sets. Random vectors were used as a baseline performance level.

In the results presented in the previous chapter, there were some weak trends in the data, but the large variations between two runs with identical parameters is a reason to take the results as tentative or inconclusive. In the data there were some clear outliers, so some of the parameter settings were retested. The new results were in some cases no longer outliers, indicating that the random effects of noise in the results are large. For this reason, the results presented below discuss trends that appear in several runs, rather than a comparison of single results, when this is possible.

6.1 Comparison to previous results

In this work, the most consistently good results were achieved with the CBOW method, but the best results for CBOW were matched by some performance results from the other categories as well. The largest proportion of low results were achieved with the skip-gram method. All methods clearly outperformed the random word vectors, showing that there is an advantage to pretraining the vectors.

It is useful to compare this result to results from the articles that originally presented the methods. For instance in [3], where GloVe was originally presented, CBOW, skip-gram and several other methods were tested against each other on a word similarity task as well as on a named entity recognition task. In these benchmarks, there were significant differences in the performance between the methods, such that the GloVe method outperformed the others by a significant margin on the word-similarity task and by a small margin on the named-entity recognition task. In [2], where skip-gram and CBOW were introduced, they were compared against several other methods on semantic and syntactic analogy tests, and there too there were significant differences in the performance between the methods. Here, the skip-gram was far better than the CBOW method on the semantic test, but CBOW and

skip-gram were similar in performance on the syntactic test. Also remember from section 3 that CBOW outperformed several other methods in many intrinsic tests but that these differences were not necessarily reflected in extrinsic performance tests.

As to the impact of the vector size, good results were most consistently achieved for the 100-element case for all algorithms, but equally good results could be seen also for vectors with 32 or 150 elements, however less consistently. This peak performance for the 100 element case also shows up, even more markedly, when randomly generated vectors are used. This is true both if they are updated in backpropagation and if they are not updated. This result is inconsistent with most other results from extrinsic and intrinsic evaluations, where it is often found that using larger vectors leads to better performance. However, in prior evaluations, the evaluation methods have been comparatively less complex than in this thesis.

A probable reason for the reduction in consistent performance for 150-element vectors is that the increase in complexity or dimensionality of the learning problem causes problems for the applied training method. In the article by Zhou and Xu [10] on which the artificial neural network used in this thesis is based, the number of traditional neural network units between the LSTM units in the bi-directional LSTM network was not explicitly stated. For this study, this number was set to be equal to the number of LSTM neurons plus the size of the initial input vector, so that the size of the input vector would equal the size of the output vector for each layer after the first. However, this also means that the number of parameters to be trained in the neural network increases when the vector size goes up. Since it is harder to train a network with more parameters, this could be a reason that the results were less consistent for the larger vector case. Something that supports this conjecture is that the test case with the updatable 150-element random vectors had worse performance than the non-updatable 150-element random vector case, since allowing these updates is another way to increase the number of learnt parameters in the model.

The performance impact of context length was also inconsistent, but 14-token contexts were mostly slightly worse than the other sizes. Since words that are close in the context can be expected to have a closer relationship to a target word than those that are further away, it is reasonable to assume that there should a limit to the context size, increases beyond which should add more noise than signal to the training examples, especially when it comes to syntactic information. This performance impact does not exist for the NCE context dependent case, however, which is possibly explained by the fact that the method learns to give less weight to the distant words in the context. For skip-gram, context words further from the target are also given less weight, but in skip-gram this is done instead by more rarely sampling more distant words in the negative sampling. The distribution for the chosen distances to the context words is not learnt, but set by the researcher. Furthermore, for skip-gram, this weighting is done on a per-context-position basis, but for NCE, it is done on a per-context-position and per-vector-element basis, leading to a possibly more expressive model for the position weights in the NCE case.

For the CBOW and the skip-gram methods, the effect of the training method was investigated. For CBOW, the results showed small differences between the training methods. For skip-gram, however, using negative sampling with 5 samples seemed to give better results than 10-sample negative sampling or using the hierarchical softmax. This result, too, is inconsistent with prior results from intrinsic testing. Comparing the present results to those from the article that introduced negative sampling with neural network models [16], the results go in different directions. In [16], larger sample sizes lead to slightly better performance in a semantic intrinsic benchmark and similar performance in a syntactic intrinsic benchmark.

A conclusion from this study thus seems to be similar to that in Schnabel et al's evaluation [17], that the performance in intrinsic tests may not always predict the results in extrinsic tests. When there were tendencies for differences in this work, they sometimes ran in the opposite direction as predicted from intrinsic testing. Again, though, a reminder that the results here were not very consistent across testing runs is in order due to seemingly random effects. Intrinsic testing is often less complex, and so such random effects may be less impactful there.

6.2 Future work

A question that consequently arises is why clear performance differences do not show up in the tests performed for this thesis, but all vector generation methods yield similar results. Semantic and syntactic information about words is, after all, obviously important to solve the SRL task. A possible answer to this question is that the first few layers of the network on their own are able to generate word representations that lead to good performance on the SRL task, as long as there is some basic semantic and syntactic information captured in the input vectors. As long as this base level of information is reached, the network may be able to implicitly generate whatever other information about the words is needed.

A suggestion for further study is whether these representations in first layers of the network work well in intrinsic syntactic and semantic tests. In [10], Zhou and Xu comment that their network seems to be able to implicitly encode syntactic information about the input. However, they only tested this by considering four example sentences, comparing the values of the output from the forget gates over time with a syntactic analysis of the sentences. Thus, a more thorough investigation using more standard tests could be more enlightening in answering what syntactic information is captured by the network. Similarly, tests for semantic content could be run to examine to what degree semantic information is captured by the first layers of the network.

The neural network for the SRL task used in this thesis is based on the descriptions of a network described in Zhou and Xu's article [10]. However, it was not possible in the present tests to reproduce the performance levels of Zhou and Xu using their exact network. The reasons for this are not known, but some possibilities include that different frameworks for machine learning may have been used as Zhou and Xu

do not state which framework was used in their paper or that there were differences in the layouts of the networks due to incomplete information given by Zhou and Xu. For instance, there is no information about the number of traditional network units in each bidirectional LSTM layer is in the paper. So suggestions for further research are to investigate whether such performance differences between the frameworks exist and why and to further investigate the performance impact of the unspecified parameters.

Bibliography

[1] A. Mnih and K. Kavukcuoglu, "Learning word embeddings efficiently with noise-contrastive estimation", in *Advances in Neural Information Processing Systems*, 2013, pp. 2265–2273.

[2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space", *ArXiv preprint arXiv:1301.3781*, 2013.

[3] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation.", in *EMNLP*, vol. 14, 2014, pp. 1532–43.

[4] Y. Goldberg, "A primer on neural network models for natural language processing", *ArXiv preprint arXiv:1510.00726*, 2015.

[5] M. Palmer, D. Gildea, and N. Xue, "Semantic role labeling", *Synthesis Lectures on Human Language Technologies*, vol. 3, no. 1, pp. 1–103, 2010.

[6] J. Persson, R. Johansson, and P. Nugues, "Text categorization using predicate–argument structures", in *Proceedings of NODALIDA*, 2009, pp. 142–149.

[7] D. Shen and M. Lapata, "Using semantic roles to improve question answering.", in *EMNLP-CoNLL*, 2007, pp. 12–21.

[8] B. Liu, "Sentiment analysis and opinion mining", *Synthesis lectures on human language technologies*, vol. 5, no. 1, pp. 1–167, 2012.

[9] D. Wu and P. Fung, "Can semantic role labeling improve smt", in *Proceedings of the 13th Annual Conference of the EAMT*, 2009, pp. 218–225.

[10] J. Zhou and W. Xu, "End-to-end learning of semantic role labeling using recurrent neural networks", in *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2015.

[11] X. Carreras and L. Màrquez, "Introduction to the conll-2005 shared task: Semantic role labeling", in *Proceedings of the Ninth Conference on Computational Natural Language Learning*, Association for Computational Linguistics, 2005, pp. 152–164.

[12] S. Clark, "Vector space models of lexical meaning", *Handbook of Contemporary Semanticssecond edition*, pp. 1–42, 2012.

[13] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[14] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[15] D. Kingma and J. Ba, "Adam: A method for stochastic optimization", *ArXiv preprint arXiv:1412.6980*, 2014.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality", in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[17] T. Schnabel, I. Labutov, D. Mimno, and T. Joachims, "Evaluation methods for unsupervised word embeddings", in *Proc. of EMNLP*, 2015.

[18] M. Baroni, G. Dinu, and G. Kruszewski, "Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors.", in *ACL (1)*, 2014, pp. 238–247.

[19] E. F. Tjong Kim Sang and S. Buchholz, "Introduction to the conll-2000 shared task: Chunking", in *Proceedings of the 2nd workshop on Learning language in logic and the 4th conference on Computational natural language learning-Volume 7*, Association for Computational Linguistics, 2000, pp. 127–132.

[20] N. Nayak, G. Angeli, and C. D. Manning, "Evaluating word embeddings using a representative suite of practical tasks", *ACL 2016*, p. 19, 2016.

[21] D. Gildea and D. Jurafsky, "Automatic labeling of semantic roles", *Computational linguistics*, vol. 28, no. 3, pp. 245–288, 2002.

[22] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm", *Journal of the royal statistical society. Series B (methodological)*, pp. 1–38, 1977.

[23] S. S. Pradhan, W. H. Ward, K. Hacioglu, J. H. Martin, and D. Jurafsky, "Shallow semantic parsing using support vector machines.", in *HLT-NAACL*, 2004, pp. 233–240.

[24] R. Johansson and P. Nugues, "Dependency-based semantic role labeling of propbank", in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 2008, pp. 69–78.

[25] R. Collobert and J. Weston, "Fast semantic extraction using a novel neural network architecture", in *Annual meeting-association for computational linguistics*, vol. 45, 2007, p. 560.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[27] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm", *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[28] F. A. Gers, N. N. Schraudolph, and J. Schmidhuber, "Learning precise timing with lstm recurrent networks", *Journal of machine learning research*, vol. 3, no. Aug, pp. 115–143, 2002.

[29] J. Lafferty, A. McCallum, and F. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data", in *Proceedings of the eighteenth international conference on machine learning, ICML*, vol. 1, 2001, pp. 282–289.

[30] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora", in *Proceedings of Language Resources and Evaluation Conference 2010 workshop New Challenges for NLP Frameworks*, Valletta, Malta, 2010, pp. 46–50.

[31] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems", *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: `http://arxiv.org/abs/1603.04467`.

[32] E. Tjong Kim Sang and J. Veenstra, "Representing text chunks", in *Proceedings of the Ninth Conference of the European Chapter of the Association for Computational Linguistics*, Bergen, Norway, 1999, pp. 173–179.