# Natural Language Processing: Part II (Paper 10)<br>Overview of Natural Language Processing (L90): Part III/ACS

2018, 12 Lectures, Michaelmas Term (these notes, lecture 9)
October 6, 2018

Lectured by Simone Teufel (`sht25@cl.cam.ac.uk`), Paula Buttery (`pjb48@cl.cam.ac.uk`), Ann Copestake (`aac@cl.cam.ac.uk`)
`http://www.cl.cam.ac.uk/users/{sht25,pjb48,aac10}/`

The notes from this lecture are partly based on slides written by Aurélie Herbelot. Additional material from other authors as specified below.

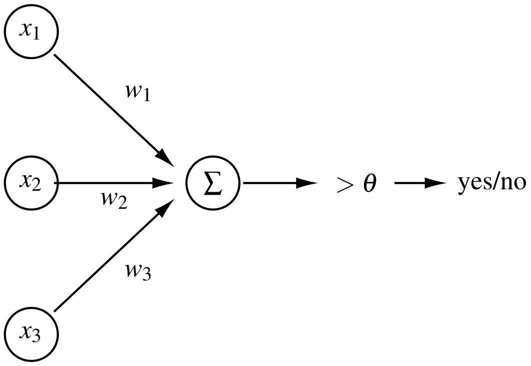
## 9 Lecture 9: Distributional semantics and deep learning

The aim of this lecture is to give an overview some aspects of the interaction of deep learning and NLP. It is not intended as an introduction to neural networks (NN) or deep learning. The main emphasis is on the notion of an *embedding*, which is discussed here as a form of distributional semantics, developed to interact well with neural architectures. In particular, I outline word2vec and doc2vec and introduce the topic of visualization.

### 9.1 A very brief overview of neural network architectures

In this section, I very briefly describe some neural architectures.

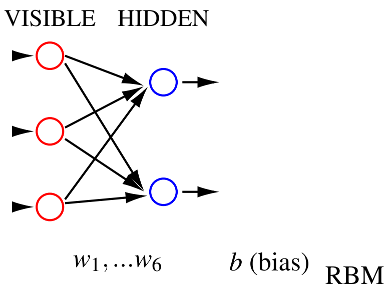
**Perceptron** This was a very early model (1962). It simply computes the dot product of an input vector $\vec{x}$ and a weight vector $\vec{w}$, compared to a threshold $\theta$. Learning (by a form of *gradient descent*) is very fast. There are no hidden layers, so this is just a linear classifier. The output is simply the summation: later systems use more complex *activation functions*: e.g. a sigmoid (which can output probabilities).

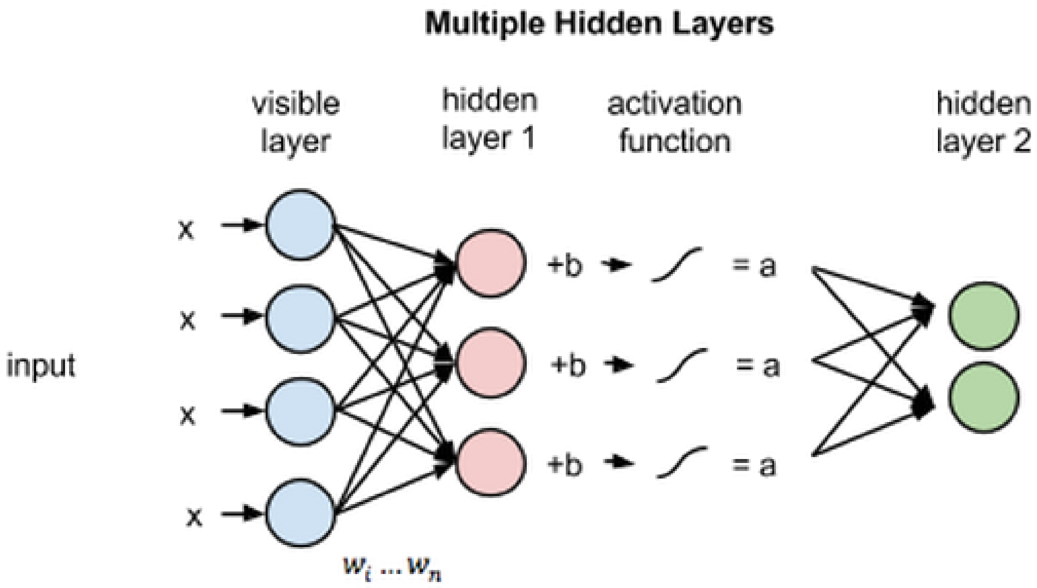


**Boltzmann Machines** NNs with one or more *hidden layer*s (i.e., layers between input and output) are theoretically capable of approximating any continuous function (mapping reals to reals) to an arbitrary degree of accuracy. However, there is no guarantee they can be effectively trained.

A *Boltzmann machine* has a hidden layer and arbitrary interconnections between units. This is not effectively trainable in general. A *Restricted Boltzmann Machine* (RBM) has one input layer and one hidden layer and no intra-layer links. The RBM is trainable: the structure allows for efficient implementation since the weights can be described by a matrix.

One popular *deep learning* architecture can be described as a combination of RBMs, so the output from one RBM is the input to the next. In principle, the RBMs can be trained separately and then fine-tuned in combination. The intuition is that the layers allow for successive approximations to concepts.





https://deeplearning4j.org/restrictedboltzmannmachine

**Architectures for sequences** Although effective for many tasks, combined RBMs and similar architectures cannot handle sequence information well when the sequences are of arbitrary length (we can pass them sequences encoded as vectors, but the input vectors are fixed length). So a different architecture needed for sequences and most language and speech problems. *Recurrent neural network* describes a class of architecture that can handle sequences. This includes the *Long short term memory (LSTM)* models which are a development of basic RNNs, which have been found to be more effective for at least some language applications.

The claim is that LSTMs are better at capturing *long-term* dependencies: i.e., any relationship between elements in the sequence that are separated by more than a very small number of other elements.

(1) She shook her head.

(2) She decided she did not want any more tea, so shook her head when the waiter reappeared.

The point of this example is that, in the use here, the object of the verb *shake* is a possessive NP which agrees with the subject. We cannot say *she shook the head* with this meaning. Hence *her* can be predicted by looking at the subject, which may be textually quite distant. Note that this is not the same as *long distance dependency* in linguistics. The example above can be analysed syntactically as coordination of VPs without any form of ‘gap’. Note that, if we were using a dependency parser, at the point where we might shift *shook*, the stack could contain ‘she decided so’, hence the ‘long-term’ nature refers to the surface text, not the syntactically/semantically structured information.

LSTMs are now standard for speech recognition (after decades where no approach could beat HMMs in any practical situation), but there is currently lots of experimentation for other language applications.

However, our focus here is on the embeddings, which can be described in terms of a language model using Recurrent Neural Networks. This is illustrated below:

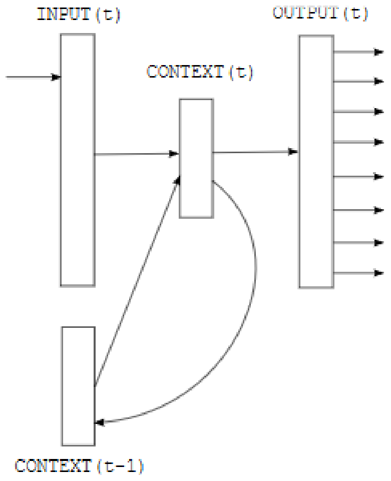


**Figure 1:** ***Simple recurrent neural network.***

RNN language model: Mikolov et al, 2010

The model is trained on a very large corpus to predict the next word. The vector for word at $t$ is concatenated to the vector which is output from context layer at $t-1$. Words could be encoded using a *one-hot* vector: one dimension per word (i.e., a simple index). This would be analogous to HMMs, which use the words themselves for prediction. However better performance can be obtained using input *embeddings*. These are, in effect, a distributional model with dimensionality reduction created on-the-fly, via prediction. The embeddings may be externally created (from another corpus) or learned as part of the task. If an embedding is to be learned as part of prediction, the input is a one-hot vector, and the embedding is created in the first layer of the network (this is described in J+M third edition chapter about neural networks, draft online at `web.stanford.edu/~jurafsky/slp3`).

Note that the input to a NN is just a vector and we can combine vectors from different sources. In multimodal architectures, the features from a CNN for visual recognition (for example) can be concatenated with word embeddings (although better performance may involve something more complex than simple concatenation). Such systems can be used for task such as captioning, visual question answering (VQA) (to be discussed in lecture 12).

## 9.2 word2vec

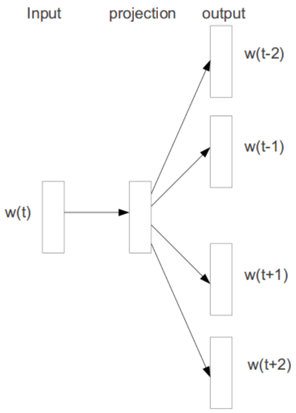
Although embeddings are often described as involving deep learning, this is misleading. Embeddings are created using a neural model, trained via prediction (as introduced above). The best known example of such an approach is word2vec (Mikolov et al 2013), which is the topic of this section. The neural models used to create word2vec embeddings are only two layer — hence not ‘deep’. word2vec is sometimes called a *predict* model, in contrast to the earlier distributional models, which are sometimes called *count* models.

Omer Levy and colleagues investigated word2vec very carefully, and found that many of its hyperparameters could usefully be applied to count models. Once trained, word2vec is more-or-less equivalent to a count model with dimensionality reduction and the essential reason for its better performance on some tasks was the improved hyperparameters. word2vec gives dense vectors which are very effective for many tasks: its hyperparameters have been tuned to give good performance of some of the standard similarity datasets. However, Levy et al found some tasks for which count vectors without dimensionality reduction were more effective. word2vec models which have been previously trained on very large corpora are available to download and give good performance out-of-the-box. It is very efficient computationally compared to most count approaches and therefore easy to incorporate in experiments.

There are two main word2vec architectures:

- CBOW: given some context words, predict the target
- Skip-gram: given a target word, predict the contexts

I concentrate on skip-gram here, since it is used more frequently.



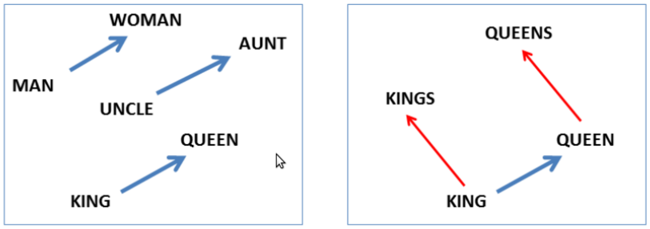
The Skip-gram model

The dimensionality of the vectors produced is a parameter of the system: usually the size is a few hundred. The intuition is that the dimensionality reduction captures meaningful generalizations, but the dimensions are not directly interpretable: it is impossible to look into ‘characteristic contexts’ as we can with the count models. Of course, the advantage of the smaller vectors is greater efficiency. As outlined below, there are visualization techniques which allow one to examine the closeness of different words.

word2vec can be tested on similarity datasets (although note that the hyperparameters have been tuned for high performance on the standard similarity datasets). It can also be used for clustering, as can any model giving a notion of similarity. Mikolov et al introduced a new task with word2vec: a form of analogy. The idea is that one solve puzzles such as:

man is to woman as king is to ?

where the correct answer is supposed to be *queen*. The idea is that one can derive the vector between the pair of words *man* and *woman* and combine it with *king*, and that the nearest word to the region of vector space that results will be the answer to the analogy. It should be pointed out that the space is very sparse and that there are many word pairs for which this does not work (also see Levy et al and Levy and Goldberg for discussion of the appropriate computation for the task).



One interesting aspect of word2vec training is the use of *negative sampling* instead of *softmax* (which is computationally very expensive). word2vec is trained using logistic regression to discriminate between real and fake words. In outline, whenever considering a word-context pair, also the network is also given some contexts which are not the actual observed word. The negative contexts are sampled from the vocabulary (in a manner so that the probability of sampling something more frequent in the corpus is higher). The number of negative samples used affects the results.

Another interesting aspect is the use of *subsampling*. Instead of considering all words in the sentence, it is transformed by randomly removing words from it. For example, the previous sentence might become: *considering all sentence* *transform randomly words*. The subsampling function makes it more likely to remove a frequent word. word2vec does not use a stop list. Subsampling affects the window size around the target (so the word2vec window size is not fixed). This approach can be emulated in count models, of course.

An additional complication is that the weights of elements in context window vary, so that closer words are given higher weights (assuming they haven't been removed by subsampling). Again, this is something that can be replicated in count based systems. Although word2vec is usually used with unparsed data, it can be modified for use with dependencies, as described by Levy and Goldberg.

## 9.3 doc2vec

Le and Mikolov (2014) describe doc2vec, which is a modification of word2vec so a vector is learned to represent a 'document' (i.e., any collection of words, including a sentence, paragraph or short document). In an analogous fashion to the way skip-gram is trained by predicting context word vectors given an input word, *distributed bag of words (dbow)* is trained by predicting context words given a document vector. In dbow, the order of the document words is ignored, but there is also *dmpv*, which is analogous to *cbow* and is sensitive to document word order. The learned document vector is effective for various tasks, including sentiment analysis (note that there is a large space of hyperparameters to investigate).

Lau and Baldwin (2016) undertook a careful empirical investigation which addressed some of the initial difficulties in replication of doc2vec results. One important point they emphasize is the status of the word vectors. In principle, one could start with random word vector initialization. Lau and Baldwin found empirically that it was preferable to do one initial run of skip-gram first, and that starting with word embeddings pretrained on some large corpus was even better.

## 9.4 Visualization techniques

Visualization is very important in experimentation with embeddings (and in other NN contexts). You are strongly encouraged to develop appropriate visualization techniques as part of any project as a way of investigated performance in more depth. This section just illustrates two examples. The first demonstrates the use of t-SNE (van der Maarten `https://lvdmaaten.github.io/tsne/`), which is a very important approach for visualizing high dimensionality datasets. The example below, from Lau and Baldwin (2016), illustrates that the document vector produced by doc2vec is closer to what (intuitively) are the most important words in the sentence rather than the closed class and less significant content words.

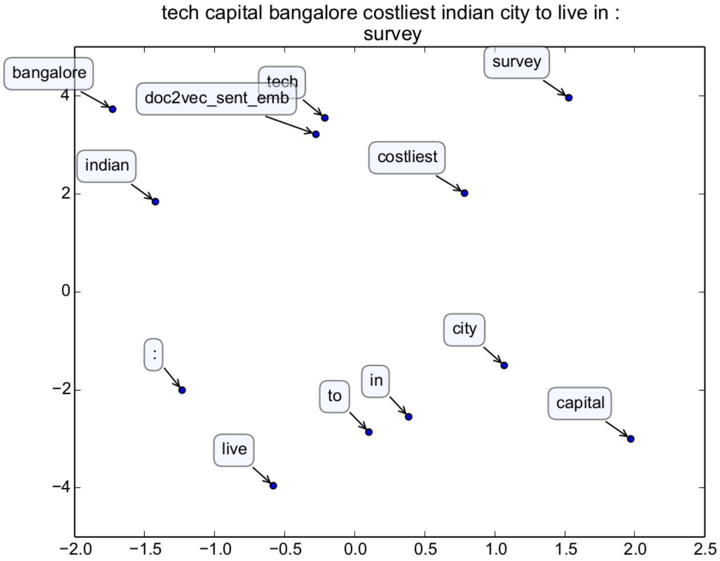


Figure from `arxiv.org/abs/1607.05368`

The second example is a heatmap from Li et al (2015), demonstrating the effect of intensifiers.

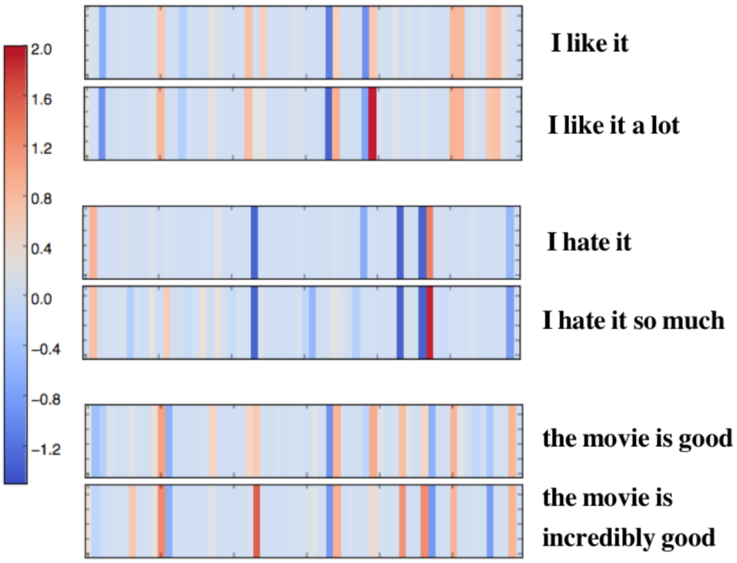


Figure from arxiv.org/abs/1506.01066

## 9.5 Further Reading

Omer Levy's papers are essential reading for anyone seriously interested in word2vec.

There is a Bayesian version of word2vec due to Barkan (2016) https://arxiv.org/ftp/arxiv/papers/1603/1603.06571.pdf

The Tensorflow tutorial on word2vec gives a more detailed description than I have attempted here, as well as explaining how to build a word2vec model:
https://www.tensorflow.org/versions/r0.11/tutorials/word2vec/index.html

Many researchers use the gensim implementation https://radimrehurek.com/gensim/.

There has been a problem with different distributed implementations of word2vec giving different results, sometimes significantly different. If comparing one's own results against results reported for a word2vec based system, it is therefore advisable to rerun the word2vec experiments, if at all possible. As with any software, when reporting word2vec results, the implementation (and the implementation version) should be given.