Deep Learning (2IMM10) - Word embedding (word2vec)

Vlado Menkovski

Spring 2020

3 Word embedding

The learning outcomes of this chapter:

- Able to develop representation of large (but countable) sets using neural network models

3.1 Introduction

In this chapter we consider tasks where the input data can only take discrete values — that is, our input variables x take values in some set V where this set has a fixed size $|V| = N$. There are many examples where the data has such discrete structure. For example, if we were to use a neural network to process DNA or RNA sequences, the input is a sequence that consists of only four different elements corresponding to the nucleotides. Similarly if we would process data that encodes the genes of proteins, we would have a fixed set of different components, the aminoacids.[1] Analogously, when dealing with natural language, the sentences we process are made up of a (large but) finite set of different words, and these words in turn are built by a finite set of different characters.

In each such case, we need to develop a way to represent or encode the data such that it can be processed by our model. For example, we can enumerate the discrete elements and encode them as integers or binary numbers.

However, in cases where the number of elements in the set, N, is very large, particularly compared to the number of datapoints in the dataset, it is very useful to have a representation that allows for better generalization. More specifically, we would like to have an encoding that reflects the relationships between the elements in the set, because in that case the model can learn to behave similarly for similar elements in the set. This in turn would allow us to be much more efficient with the amount of training data that we need.

In this chapter we discuss how neural network models can be used for learning representations of large sets of elements such as words in a natural language.

[1] A set of only 22 different "proteinogenic" amino acids form the building blocks of all proteins in all known lifeforms.

3.2 Example: Text prediction

Let us consider the task of predicting the next word in a sentence. Our model would need to take a sequence of words as its input and produce a suggestion for the most likely next word — see Figure 1.
Each of the words needs to be encoded and presented to our model, such that the model can produce the next word . For simplicity we can assume that the input is a fixed-length window of words rather than a variable-length sequence of words.
Suppose we would encode the words with sparse vectors. Specifically, with binary vectors with length $|V|$ as in Figure 2.
If we make the model do predictions for

- ”I want a glass of orange ...”

we can depict the process as in Figure 3. What happens if we instead want to do predictions for

- ”I want a glass of apple ...”

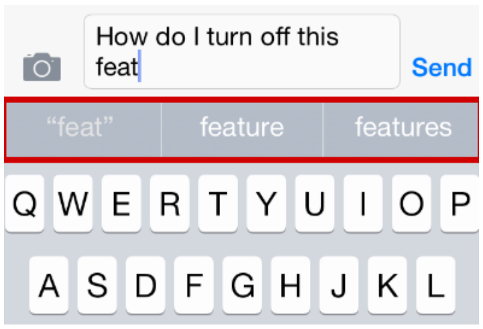


Figure 1: Predict the next word

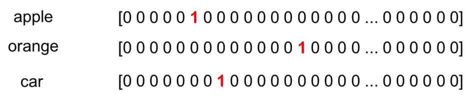
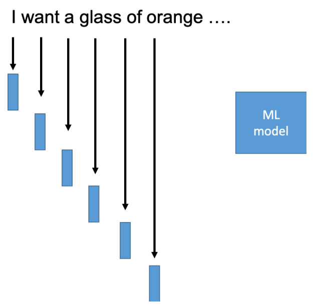

apple [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0]
orange [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 ... 0 0 0 0 0 0]
car [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0]

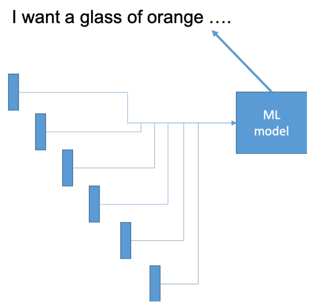
Figure 2: Sparse words

The current representation of the data as sparse encoding does not provide any possibilities for model to re-use what it has learned for the word ’orange’ on the input with the word ’apple’. This is not very efficient as we would need to have all sentences where both apply double in the training set — once with “apple” and once with “orange”.

In contrast if we would have a representation of ’apple’ and ’orange’ that allows for encoding the commonalities between the two words (e.g. both of them are fruits, but also types of juice) our model could benefit from this as it can learn to actually map the common properties to a target rather than the individual words. For example, all fruits that can be made into a tasty glass of juice could share a specific part of their encoding. So, if we have an example of a sentence with one, then the model would be able to generalize what it learns from that to the other fruits.



(a) The model takes the sentence as a sequence of one-hot vectors.



(b) The model processes these sparse vectors.

Figure 3: Predicting the next word using sparse encoding.

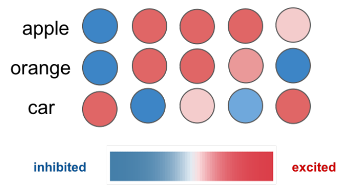


Figure 4: Distributed word representation

3.3 Distributed word representations

When developing a word embedding we aim at an encoding that captures different features, or aspects, of the words. This is referred to as *Distributed word representations.* Such a representation can describe a large number of words efficiently.

Let us consider the following example:

- king ← male
- queen ← female
- man ← male
- woman ← female

Here the representation of ‘king’ and ‘man’ would share the property of ‘man’ as would the representation of ‘queen’ and ‘woman’ share the property of ‘female’.

If we had such a representation then after learning about the word ‘aunt’, we would be able to learn ‘uncle’ by adjusting the ‘aunt’ representation as

$$\text{‘uncle’} = \text{‘aunt’} - \text{‘female’} + \text{‘male’}.$$

To be able to achieve such distributed representation we first represent the words in an **embedding space**, such that each word is represented by a d dimensional vector in $\mathbb{R}^d$ space. Here each of the axes in the space can take different roles, see Figure 4.

If we manage to create such a representation, the meaning of the individual words will be distributed over the different dimensions of the representation. Furthermore, the Euclidean distance in this space will capture the semantic distance of the words.
How can be develop such a representation?
One important property of natural text is that the meaning of a word is related to its context, so that words that appear next to similar words have similar, or related, meaning . [2]In the work by Benio et al. this property was used to develop the ”Neural probabilistic model”.

[1]Take the orange and apple example from earlier. Both words occur in similar contexts such as “I am going to eat a ...” and “she was drinking ... juice”. If another word systematically pops up near words such as “fresh”, “eat”, “juice”, “drink”, “grow”, and “sweat”, chances are it’s a word for some kind of fruit.

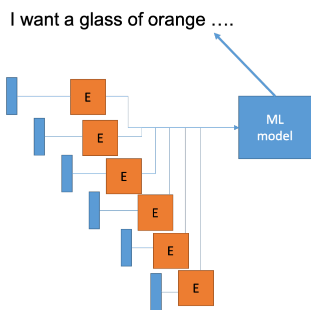


Figure 5: The model first uses an encoding map E to map the one-hot vectors to their dense representations, and then uses these dense representations to make predictions.

The approach is as follows: we try to learn a representation that captures semantics based on context by learning a probability distribution over sentences factorized as

$$\hat{P}(w_1^T) = \prod_{t=1}^{T} \hat{P}(w_t | w_1^{t-1}),$$

where w_t is the t-th word in the sequence, and $w_i^j = (w_i, \ldots, w_j)$. We simplify this by only looking at the last $n-1$ words:

$$\hat{P}(w_t | w_1^{t-1}) \approx \hat{P}(w_t | w_{t-n+1}^{t-1}).$$

The idea is then to write this as a function in the following way:

$$\begin{aligned} \hat{P}(w_t = i | w_{t-n+1}^{t-1}) &= f(i, w_{t-n+1}, \ldots, w_{t-1}) \\ &= g(i, E(w_{t-n+1}), \ldots, E(w_{t-1})), \end{aligned} \tag{1}$$

where $E(w)$ is the embedding of a word w into our embedding space — see Figure 5 for a schematic representation. In practice we use an embedding matrix that maps the one-hot encoding of the words to a dense representation of the data. For example, if our vocabulary V has size $|V| = 10^4$, and our embedding space is 300-dimensional, the map E is given by a 300×10^4 matrix — see Figure 6.

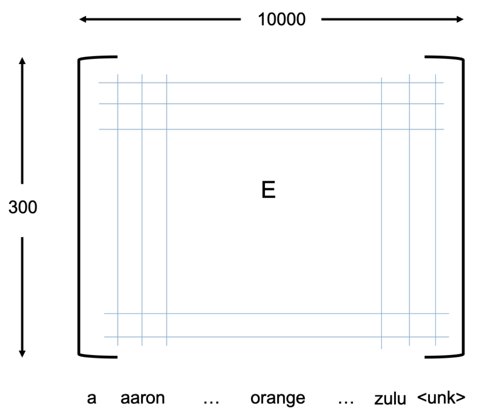
In the work of Bengio et al.[3] the function g was implemented as a feed-forward neural network with, since we want the output to be a probability distribution, a softmax function for its output layer. Since E is a differentiable function, we can train E and g simultaneously using back propagation and a gradient-based optimization algorithm such as SGD.

3.4 Further developments

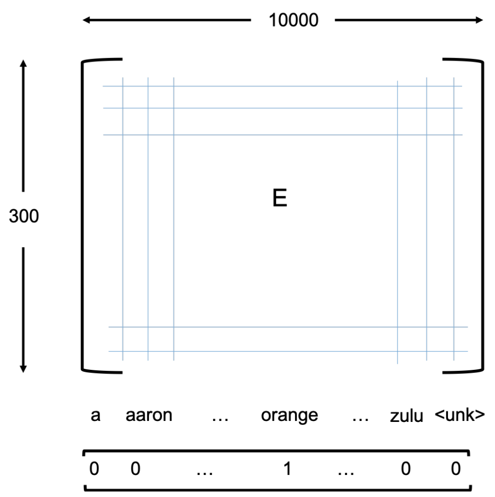
The model we discussed above can be used on its own, but far more importantly, the word vectors it has learned can be used for downstream tasks. I.e. we can use the (trained) embedding matrix as the first layer to another model[4]. If we keep in mind that our real goal is not to use the model we train directly, but just to learn a word embedding for a later task, what kind of model do we ideally want for g in eq. (1)?

[3] Bengio, Yoshua, et al. "A neural probabilistic language model." Journal of machine learning research 3.Feb (2003): 1137-1155.

[4] This is an example of *transfer learning*, a topic we will discuss more thoroughly in the next chapter.



(a) If our vocabulary has size 10 000, and our embedding space is 300 dimensional, our embedding matrix E is of size $300 \times 10\,000$.



(b) We can use the embedding matrix to map a one-hot encoded word to the corresponding dense representation.

Figure 6: Embedding matrix

The cheaper g is to train, the larger the vocabulary we can embed and the more data we can handle with a limited amount of resources. Let us therefore look at two techniques that simplified the model, allowing us to train word embeddings for much larger vocabularies, or on much more data: *Continuous Bag Of Words* (CBOW) and *Skipgram*, both introduced in "Efficient Estimation of Word Representations in Vector Space", Mikolov et al. (2013).

These two models consist of two matrices, a word embedding E and a context embedding, C. The context embedding, C is essentially the weights matrix of the output layer of g. In other words, these models remove all hidden layers from g. Graphical representations of both the original Neural Probabilistic Language Model and the two improvements we will discuss in this subsection can be found in Figure 7, Figure 8, and Figure 9 at the end of this chapter.

Continuous Bag of Words

The training objective with CBOW is similar to the one in Section 3.3: we try to predict words based on their context. However, instead of using only previous words, we try to predict based on a window around the word. That is, we try to learn

$$P(w_t \mid w_{t-L}, \ldots, w_{t-1}, w_{t+1}, \ldots w_{t+L}).$$

In order to do this, we embed all the context words using the same word embedding

$$u_i = E w_i$$

where E is a matrix as before. Now instead of putting $(u_{t-L}, \ldots, u_{t-1}, u_{t+1}, u_{t+L})$ through a hidden layer of a neural network, we simply compute their average

$$\hat{u} = \frac{u_{t-L} + \ldots + u_{t-1} + u_{t+1} + \ldots + u_{t+L}}{2L}$$

to get some kind of summary of the context.

This context vector is then transformed to a score vector

$$z = C\hat{u},$$

which is then put through a softmax function in order to obtain our estimated probability distribution

$$\hat{P}(w_t \mid w_{t-L}, \dots, w_{t-1}, w_{t+1}, \dots w_{t+L}) = \text{softmax}(z).$$

Question: what loss function should we use to train this?

Skipgram

With Skipgram, the training objective is in a way of reversed compared to CBOW: instead of guessing a word based on its context, we try to learn the context based on a word. That is, we try to learn the probability distributions of words close to a given word. The way we do this is by setting up our training data in a specific way.

To generate the training data, we specify a maximum window size L_{max} and then we go through our available sentences:

1. for a word w_t in the sentence we pick a window size $1 \leq L \leq L_{max}$
2. we add the the word pairs $(w_t, w_{t-L}), \dots, (w_t, w_{t-1}), (w_t, w_{t+1}), \dots, (w_t, w_{t+L})$ to the dataset.

Our training set then consists of a (large) number of word couples (w_i, w_j) and our training objective is to predict w_j based on w_i. *Question: how does this force the model to learn the probability distribution of the context of w_i?*

The network itself is now even further simplified compared to CBOW: we only get one word as the input so the averaging is no longer necessary. This means we do prediction in the following way:

1. We embed the input word using our embedding matrix
$$u = E\, w_i$$
2. Based on the embedded word, we calculate a score vector for the context, i.e.
$$z = C\, u$$
3. We apply the softmax function to the score vector to get our estimated probability distribution
$$\hat{P}(w_j \mid w_i) = \text{softmax}(z).$$

This setup forces our word embedding E to give similar output for words that appear in the same contexts since such words will need to result in similar context distributions. For example, “orange” and “pear” both occur frequently near words like “fresh”, “juice”, “eat”, “drink”, “tasty”, “buy”, and “healty” but much less frequently near words like “headphone”, “war”, “guitar”, or “chair”. Since our way of training forces the model to give high probability to frequently occurring combinations, and low probability to rarer combinations, the output distribution for “orange” and “pear” will need to be similar, forcing the word embeddings for these words to be similar too.

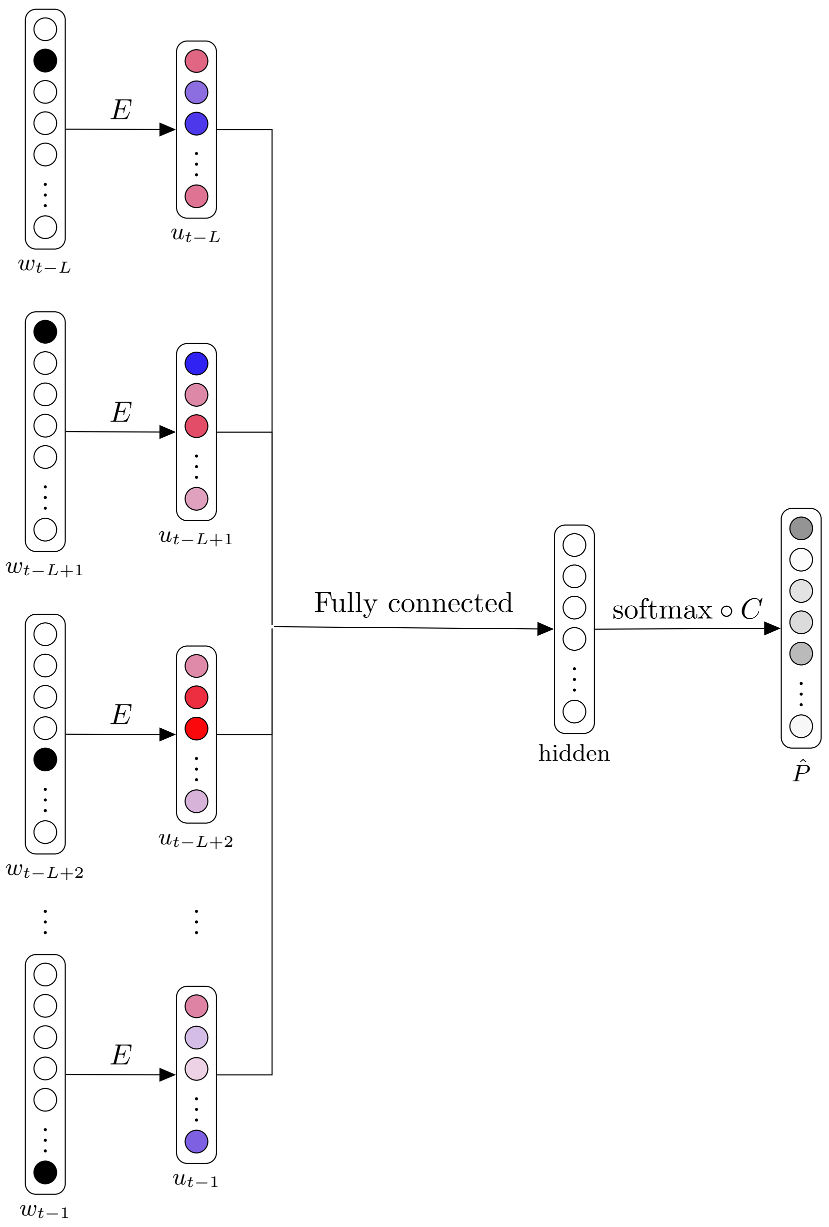


Figure 7: Neural Probabilistic Language Model. *Question: what probability measure is being estimated?*

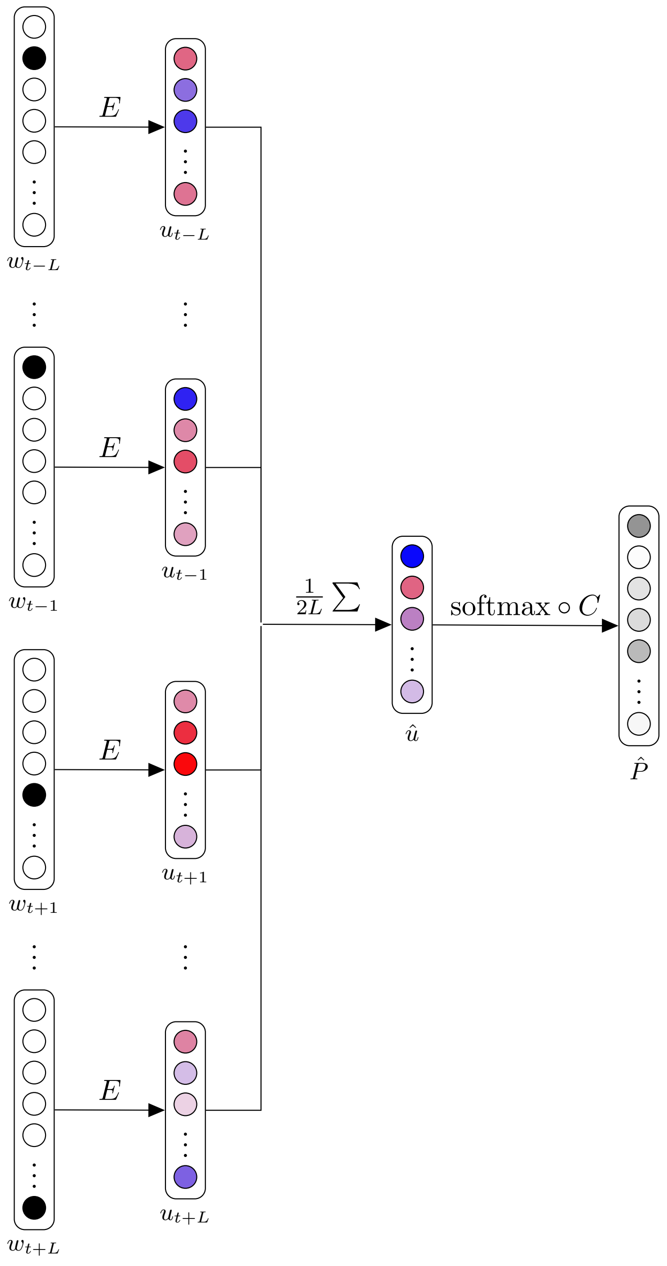
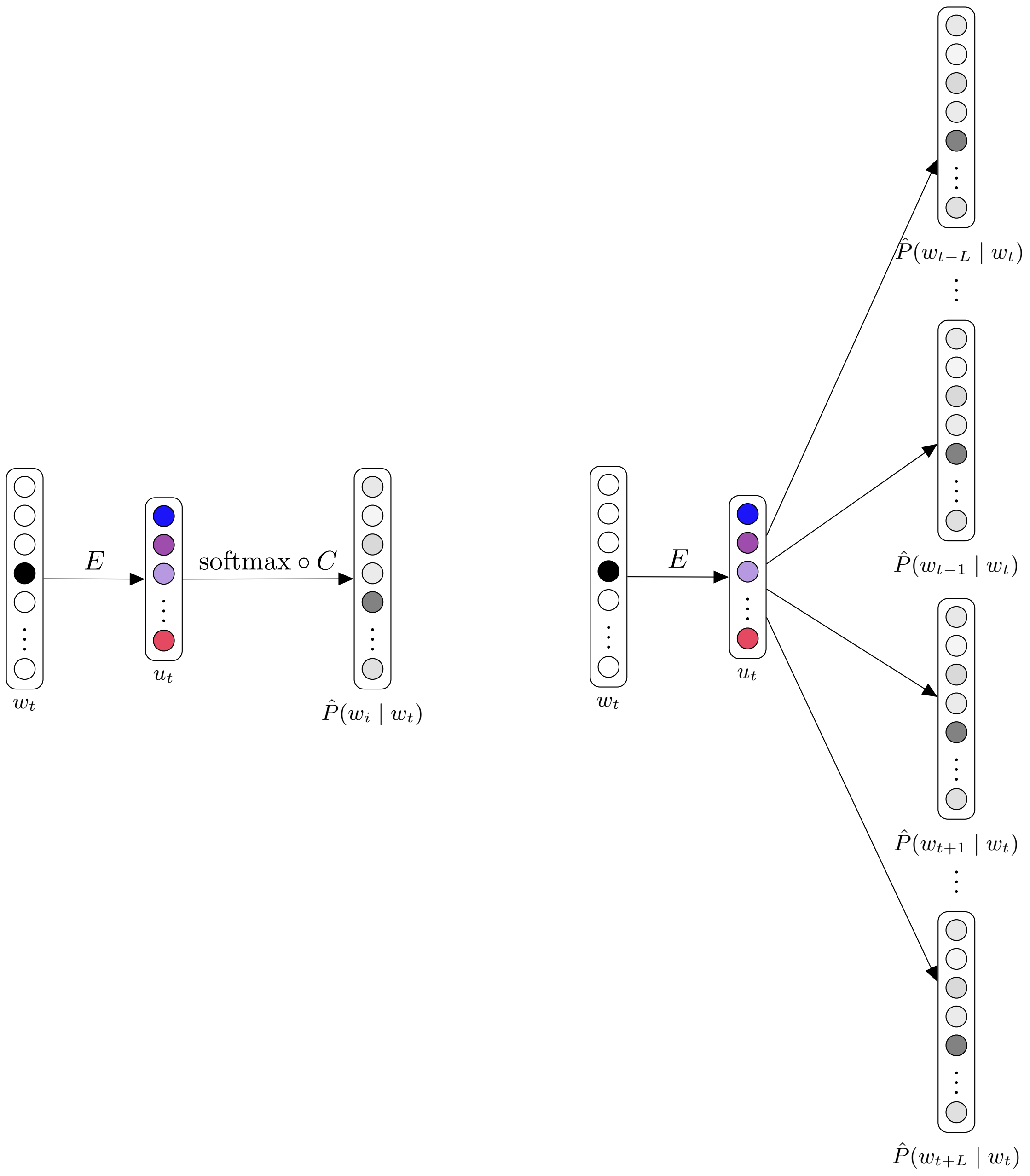


Figure 8: Continuous Bag of Words. *Question: what are the differences with Figure 7? And what are the similarities?*


w_t
E
u_t
softmax $\circ$ C
$\hat{P}(w_i \mid w_t)$
w_t
E
u_t
$\hat{P}(w_{t-L} \mid w_t)$
$\hat{P}(w_{t-1} \mid w_t)$
$\hat{P}(w_{t+1} \mid w_t)$
$\hat{P}(w_{t+L} \mid w_t)$

(a) Skipgram (b) Skipgram

Figure 9: Two representations of Skipgram. *Question: why are these both accurate representations of the same model?*

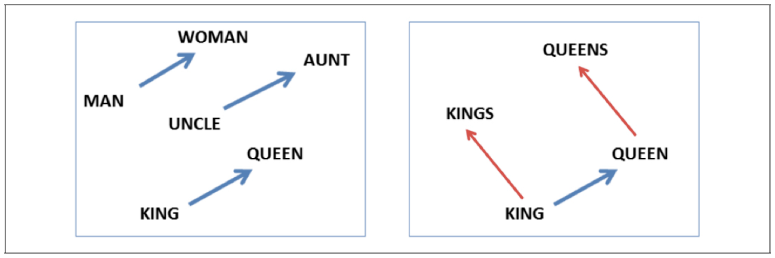


Figure 10: Properties of word embedding

3.5 Properties of the embedding

Assuming that our model has succeeded to develop a distributed representation of the words in the vocabulary we can expect certain properties of this representation. One such property is that we can compute analogies such as: ”man to a woman” is as a ”king to a queen” fig. 10.

In principle we can use these analogies to search for a word that should have the desired properties. Again in the example: ”man to a woman” is as a ”king to a ?”, using the embedding space we would like to find the word queen. Using the embedding vectors for ”man”, ”woman”, and ”king” (e_{man}, e_{woman}, e_{king}). We can search for a word that is closest to the algebraic expression: ”king” - ”man” + ”woman” as in eq. (2).

$$w = \arg\min L(e_w, e_{king} - e_{man} + e_{woman}) \tag{2}$$

where L is a distance metric in the embedding space.