Vector Semantics

High School of Digital Culture
ITMO University
dc@itmo.ru

Contents

1 Introduction

In this module, we are going to discuss vector semantics. Semantics as a subfield of linguistics is the study of word meanings, and vector semantics is an NLP subfield which studies the ways to represent word senses as vectors.

Representing word meanings as vectors is one of the core NLP problems mostly because many applied methods need to represent **tokens** (which are, roughly speaking, words contained in a text) as vectors so that these vectors convey important information about the words they represent.

That's why we need vector semantics, which aims to create relevant vector word representations (usually **word embeddings**) for algorithms that use these embeddings to extract the required information **from them directly or indirectly** and successfully solve the problem. When dealing with a classification problem, we can train a neural network using word vectors as an enriched input to it. When the number of datapoints is not enough to train a neural network, we can simply compare texts with the samples of each class.

For example, texts may be represented by averaged word vectors for such a comparison. Moreover, vector semantics allows studying the language itself. For

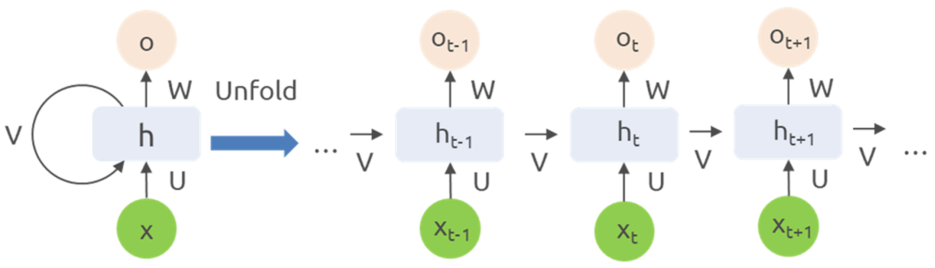


Figure 1

example, words represented by vectors allow us to visualize certain regularities that can be found in a language. The picture 2 of words **embedded into a space** shows **which** words have similar meanings and which are far **from each other**. We can track changes in word use over time. The figure shows how the use of the word 'mask' has changed in 2020 compared to 2019. We can select an odd-one-out word as in the quizzes and many more.

As in most natural **language processing problems**, the data-based approach dominates here. Word meanings are modeled based on the patterns of their use in a corpus. We are not trying to explain a machine explicitly that, for example, a rope is a long and flexible real-life object, but we rather assume that all details are included in the use of the word 'rope' in different contexts.

The assumption that words that occur in similar contexts have similar meanings is called the distributional hypothesis (and the corresponding NLP

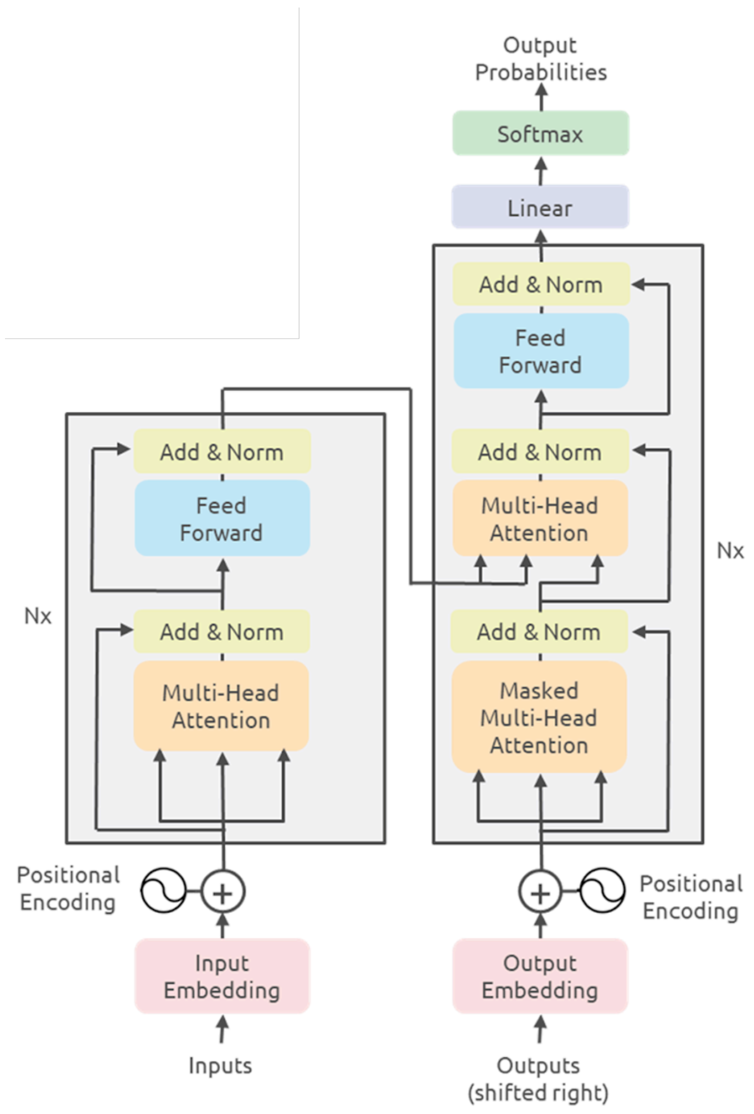


Figure 2: Words embedded into a space.

field is often called distributional semantics). English linguist John Firth famously summarized this principle as: "you shall know a word by the company it keeps". Here is another quote from him: "The complete meaning of a word is always contextual, and no study of meaning apart from context can be taken seriously".

The distributional hypothesis is essential for vector semantics because it allows us to use the variety of contexts of a word as an approximation of its meanings. If the words that occur in similar contexts have similar meanings, we can use vectors that represent contexts of a word as word vectors. The similarity or difference between words will be defined by the proximity of vectors that represent the contexts. For example, the words 'kitty' and 'cat' will share more contexts than the words 'kitty' and 'utilities'.

So, let us assume that the similarity of words is almost the same as the similarity of contexts. But how to find the similarity between contexts? The intuition is that it might be related to co-occurrences: words of similar contexts occur in the same or similar texts. Well, there are two different cases of context similarity.

Words that often appear close to each other are said to have first-order similarity. For example, the word 'ate' is a first-order neighbor of 'dinner' because both of them have many first-order co-occurrences. It is a syntagmatic association also called associative similarity.

Words that have similar neighbors are said to have second-order similarity. For example, the words 'Everest' and 'Karakoram' related to mountains occur in almost the same contexts. They have many second-order co-occurrences. This is a paradigmatic association.

And now that we have all the definitions, we can move on to vector semantics. First, we are going to learn how to evaluate the quality of our word vectors, **as well as which datasets** and tools are important for vector semantics. Then we will discuss the most straightforward approaches to construction of word vectors, which includes word vectors based on word context-related counts. After that, we will learn how to build other word embeddings using linear algebra methods. Next, we will dive into **the basics of neural networks** and their applications in NLP. In the next sections, we will need this knowledge to learn how to use shallow neural networks to train word vectors. After that we will find out how to make neural word vectors convey not only the general meaning of a word but also the subtleties of their use in a text in concern.

2 Evaluation, Datasets, and Tools

Before we turn to word vectorization, we need to be able to evaluate word embeddings. We need to find out not only how well **some** word vectors satisfy our requirements but also to what extent they convey word properties that we want to represent. Moreover, we need labeled data to evaluate vector semantics algorithms. Thus, we will review the available datasets that are often used in vector semantics. We will also discuss tools used to vectorize words.

We have already stated that word vectors of interest **mainly** serve two purposes. The first purpose was to preprocess (vectorize) text data before using it as an input to other algorithms and tasks, and the second purpose was to study the language from the inside. Word vectors evaluation methods can be divided into two groups: extrinsic and intrinsic.

The extrinsic vector evaluation uses word vectors to solve standard natural language processing end tasks such as text classification. For that, we need some standard text classification algorithm requiring texts vectorized in several ways.

Then, we can see how the algorithm quality changes depending on the selected vectorization method. If the algorithm utilizes some vectors much *better* than others, it means that these vectors convey the properties of words better, or at least that they are better suited for solving the problem.

Intrinsic evaluation compares the word vectors' properties with human perceptions of those. For example, since we assume that the word 'kitty' is more similar to 'cat' than to 'utilities', it's reasonable to expect that the vector of the word 'kitty' should be much **closer to** the vector of the word 'cat'.

Intrinsic evaluation includes many different approaches. In particular, one of the standard evaluation methods is comparing the proximities of pairs of word vectors with word similarity ratings assigned by experts. (This requires a **dataset** of word pairs, each of which is assigned a similarity score, for example, from zero to ten. An example of such a dataset is shown on the slide). For comparison, we need to calculate the Spearman's rank correlation coefficient between human ratings of word similarity and the vector similarity of representations of the same words.

Let's pause for a while to remind ourselves what the Spearman's coefficient is. You have probably heard of correlation coefficients. They are used to find out if there's a relationship between two given variables.

$$\rho_{XY} = \frac{\sum(X-\bar{X})(X-\bar{Y})}{\sqrt{\sum(X-\bar{X})^2\Sigma(Y-\bar{Y})^2}}, \quad \textit{where} \quad \bar{X} = \frac{1}{n}\sum_{t=1}^{n} X_t$$

For example, the Pearson coefficient (its formula is shown on the slide) shows the presence or absence of a linear relationship or association. A high correlation coefficient indicates the presence of the relationship between two variables (whereas a negative coefficient with a high absolute value points to an inverse relationship).

In this case, the relationship between human and vector scores shows that different ways of comparison produce almost the same result, and, therefore, word embeddings convey word similarity reasonably well.

However, the Pearson coefficient and similar indicators are not the best choice for comparing the scores of word similarity. The relationship between the ratings doesn't have to be linear. What matters is that the same word pairs are closer or more distant (for example, that the word 'kitty' is closer to 'cat' than 'utilities'). That's why we apply the Spearman's coefficient. Instead of a linear relationship, it shows a so-called rank correlation.

Look at the example. Here we have four word pairs, each of which is assigned a similarity score. This similarity score is a variable of interest. Moreover, a score of each word pair has its rank. A rank is the position of a value in the list that it would get if all values had been sorted in descending order (from largest to smallest). For example, the rank of the pair stock-market is 1 because the score of this pair is higher than that of others, and the rank of the pair stock-jaguar is 4 because these two words have almost nothing in common. Rather than evaluating the relationship between the values of two variables, the rank correlation compares the ranks of these values.

(It follows from the Spearman's correlation formula, which is shown on the slide, that the value of the Spearman's coefficient is the same as the Pearson coefficient between the ranks of variables instead of the variables themselves. Therefore, that allows us to use the simplified Spearman's correlation formula, which is also shown on the slide.

$$r_{\mathrm{s}} = \rho_{rgx,rgy}, \text{where rgx, rgy are the ranks of X and Y}$$
$$r_s = 1 - \frac{6\Sigma d_i^2}{n\left(n^2-1\right)}, \text{where } d_i = rg\left(X_i\right) - rg\left(Y_i\right)$$

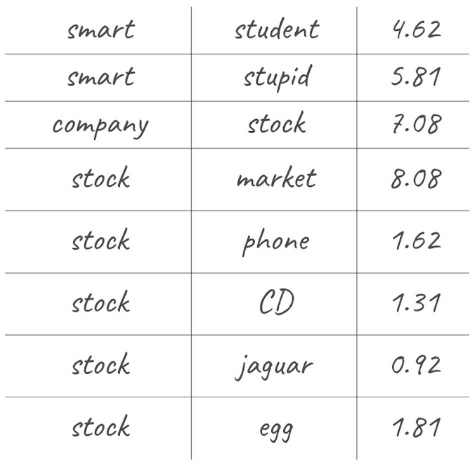
If the rank correlation is high, then human and vector word similarity ratings are, **so to say**, in the same order. Even if the values slightly differ, the word 'kitty' will always be closer to 'cat' than 'utilities'.

A high value of the rank correlation with human ratings shows that the studied vector semantics algorithm reflects the **expert's opinion** on words properties.

Another way to perform the intrinsic evaluation is analogies. With certain word vectors, if we subtract the vector of the word Russia from the vector of the word Moscow and add the vector of the word France, the obtained vector should be close to the vector of the word Paris. It's like solving a word equation: Moscow relates to Russia just like Paris to X. It turns out that certain word vectors can be useful for solving such equations.

These two approaches to the intrinsic evaluation are the most common. But there are many more. For example, word vector clustering-based approaches. Many datasets developed for different languages and tasks of distributional semantics can be employed for intrinsic evaluation. WordSim353 is **one of the oldest and well-known datasets**. WordSim353 consists of 353 English word pairs, each assigned a score from 0 (for completely different words) to 10 (if the word is the same). For example, the similarity between the words stock and market is estimated to be equal to 8.08, while for stock and jaguar, it is 0.92. Excerpts from WordSim353 are shown on the slide. Moreover, there's SimLex999 for which synonymy is more significant than any kind of context similarity. For example, the words wardrobe and clothes are deemed very similar in the first dataset but completely different in the second because wardrobe and clothes are not the same. The Stanford Rare Word Similarity dataset and the Human Judgment dataset created for Russian are also worth mentioning.

A **standard dataset** used to evaluate word analogies is the Google analogy test set (it is also shown on the slide). This dataset is divided into two parts:



smart	student	4.62
smart	stupid	5.81
company	stock	7.08
stock	market	8.08
stock	phone	1.62
stock	CD	1.31
stock	jaguar	0.92
stock	egg	1.81

Figure 3

semantic and syntactic ones. Analogies of the semantic part analyze the word meaning (for example, to find antonyms), while analogies of the syntactic part consider the grammar (for example, derivation of the word acting from the word act). The **dataset** of Deep Semantic Analogies that includes English and German contains datasets for several tasks such as looking for synonyms, antonyms, and hypernyms. **Hypernym of a word is a word that has a more general meaning. For example, a hypernym of the word poodle is a dog.**

The RUSSE workshop presented such Russian datasets as **datasets** of similar words, synonyms, and hypernyms, as well as a word associations dataset. Before applying distributional semantics to a particular problem, it's necessary to review the existing tools. One of such tools is called **Gensim**. It is a Python library compatible with many **common distributional semantics** models. Gensim allows performing all basic operations, for example, to convert a word into a vector, find nearest neighbors of a word, solve an analogy problem, and be useful in several other common scenarios. Additionally, many models have libraries of their own. For example, Word2Vec, FastText, and GloVe are distributional semantics models and Python libraries at the same time (which imposes no restrictions on their application through Gensim).

Moreover, trained vector models are also often published. In particular, such models trained on Russian corpora are for example available on `https://rusvectores.org/`.

Tools used to process contextualized vectors are also worthy of mention. This kind of vectors is based on deep neural networks. Contextualized vectors cannot be used through Gensim-like libraries. You'll need specific libraries for deep learning, for example, TensorFlow or Torch. The Transformer library is also a good way to obtain contextualized vectors. It is compatible with many contemporary language models that help to extract contextualized word vectors. *We will discuss contextualized vectors a little bit later, and a separate module will cover language models.*

3 Sparse Vectors

The first group of **methods** that we will use to model word meanings as vectors includes approaches based on direct word counting in each word's context. **But what does it mean?** This approach assumes that each element of a word vector conveys information about the so-called contexts of this word, for example, about the number of co-occurrences not far from another word or about the total frequency of the word in a document. Such vectors are called sparse vectors because they usually have a large number of zeros, since the word usually has no shared contexts with most of the other words, and most of the time it is not present in all the documents. That's the difference between sparse vectors and dense vectors which mostly contain non-zero values. Besides, sparse vectors are easier to interpret since each element conveys specific information about the word occurrence. At the same time, dense vectors either contain more fine-grained features or are better utilized when interpreted geometrically as points in a vector space, each of which may be closer to or further away from another point, but a particular element of a vector contains no information in an explicit form.

Earlier, we discussed the difference between syntagmatic and paradigmatic associations. Syntagmatic associations are words that frequently occur together, for example, 'ate' and 'dinner'. Paradigmatic associations are words with similar neighbors, for example, 'Everest' and 'Karakoram'.

To encode the **associations** of the first type, a term-document embedding is used. This approach assumes that word vectors contained in the same documents **are close to each other**. For this, we can use a term-document matrix. It is an $n \times m$ matrix, where n is the size of a vocabulary, and m is the number of documents in the collection. The number of occurrences of the nth word in the m^{th} document is located at the intersection of the n^{th} row and m^{th} column. Earlier, we were interested in columns of this matrix, each of them was a **bag of words**, a vector that showed the number of occurrences of different words from the vocabulary in a certain text of interest. Now, we focus on rows of this matrix. The i^{th} element in a row is the number of occurrences in the i^{th} document in the collection. So, two vectors are similar if two corresponding words appear in **almost the same** documents.

Note that this is a syntagmatic association. Two words are deemed similar because they occur in the same documents, which means they often appear in the same context. However, this approach has its disadvantages. The quality of such a representation heavily depends on a collection of documents. If the number of documents is insufficient or their topics are unevenly distributed, word vectors will not represent the words' properties adequately.

If a collection of documents is quite large, the dimensionality of word vectors will increase proportionally because it's equal to the number of documents in the collection. Vectors of words contained in many documents will not be very sparse, and this will hamper the calculations.

To solve this, we need to go from the document level to the level of short sequences of words, in other words, contexts. To do so, we need a so-called **word-context matrix**. An example of such a matrix is shown on the slide. The number of occurrences of the j^{th} word in the vocabulary in the contexts of the i^{th} word lies at the intersection of the i^{th} row and j^{th} column. Contexts are defined differently. For example, we can establish a two-word window for two words to the left and two words to the right of a given word.

Important: *a word-context matrix is usually a square one. It means that we usually calculate how many times a word occurred in the context of another word. Rows correspond to target words. They are the vectors we want to obtain. Columns correspond to context words. They are often called contexts. So beware the confusion between the two meanings of the words 'context'!*

The impact of the content of the document collection on vectors decreases, and we consider only word occurrences nearby the given word rather than an entire document. The dimensionality of a vector doesn't depend on the collection size anymore since it equals the size of a vocabulary. In this case, almost all elements will be zeros.

By the way, note that we have just switched from syntagmatic to paradigmatic associations. Now words are considered similar if they appear in similar contexts (so to say, they have the same neighbors) rather than those that are frequently found together in documents (they are neighbors).

The disadvantage of this approach is related to the common words that are found almost everywhere. For example, the words it, this, if are used to structure the text and are found in the context of almost any word. They don't add much information to a word vector (such neighbor as the word this doesn't allow to distinguish the meaning of one word from another). At the same time, the elements of the vector that correspond to these words will have very high values. The quality of the algorithms based on word vectors will suffer due to this. They will wrongly assume that the neighborhood with these words is an important feature of a word and that these neighboring words deserve special attention.

To handle this, we use weighting. Weighting assigns each word a special coefficient (weight) that has to be smaller for popular words that are useless for vector semantics and higher for rare and specific words that help to describe the use of other words. For example, such a coefficient of the word this should be small. This word is used with different words, while the word idempotency should have a high coefficient because the words it is used with are most likely related to math or computer science.

The first weighting method that we are to review (well, actually remind ourselves about it) is called TF-IDF. TF-IDF is a short name for term frequency, inverse document frequency. The formula on the slide shows that TF-IDF is a product of TF and IDF.

$$\mathrm{tf(t,d) = n_t}$$

$$\mathrm{idf(t,D) = \ln \frac{|D|}{|\{d_i \in D \mid t \in d_i\}|} = -\ln \frac{|\{d_i \in D \mid t \in d_i\}|}{|D|}}$$

$$\mathrm{tf \cdot idf(t,d,D) = tf(t,d) \cdot idf(t,D)},$$

where t is a token, d is a document, D is a collection, nt is the number of occurrences of token t in the document.

TF is a function of a token and a document. It **returns** the number of token occurrences in the document. We used this number in word-document matrices. IDF is a weight or a coefficient to be multiplied by the number of occurrences. IDF is a function that receives a token and a document collection as input. This function calculates the logarithm of the fraction where the numerator is a collection size, and the denominator is the number of documents that contain the token at least once. To put it differently, this fraction is an inverse share of documents that contain this token.

For example, if we have 1000 documents, and the token is in 700 of them, the document frequency is 70%, and the fraction from the formula of the inverse document frequency is $\frac{1}{0.7}$, which is about 1.43. The function outputs the logarithm of this fraction. (A logarithm of a number multiplied by negative one equals the logarithm of an inverse number. So, IDF is a negative logarithm of the share of the documents that contain this word). For words that appear in many documents, the fraction related to the inverse document frequency is a little more than one, and the logarithm will be positive but close to zero. For rare words, IDF will be a positive number with a higher absolute value. (The logarithm grows slower than the inverse document frequency-related fraction.)

Here's an example of calculating TF-IDF for two words in a collection. The frequent word this receives a smaller value than the informative word idempotency even though the number of instances is 6 times higher. TF-IDF is good for term-document-like matrices and cases when a document needs to be represented via the words it contains. For example, if one chooses to obtain a document representation by averaging the vectors of all the words it contains, it often makes sense to multiply the vector of each word by the IDF value of this word.

However, TF-IDF is not suitable for term-term-like matrices, for example, **word-context matrices** because it heavily relies on the text in the document and document collection. That's why you cannot calculate TF-IDF based on a set of contexts of several words. We can weigh a term-term matrix using PMI, for example.

PMI stands for pointwise mutual information. Take a look at the formula on the slide. PMI equals the logarithm of a fraction where the numerator is a probability to encounter the two words together, and the denominator is a product of probabilities to encounter each of them. The point is to find out how many shared contexts the words would have if they had been independently distributed and then compare this number with the actual number of shared contexts. **The larger this value is, the less random the presence of the words in the same context is.**

For example, the probability to encounter one word in contexts is 10% and that for another word is 20%. Then, given the independence of distributions, the probability to encounter both words is 2%. In practice, if they appear in 10% of contexts, which is 5 times higher, their PMI will be equal to the logarithm of 3.

$$\operatorname{pmi}(x;y) \equiv \log\frac{p(x,y)}{p(x)p(y)} = \log\frac{p(x \mid y)}{p(x)} = \log\frac{p(y \mid x)}{p(y)}$$

Negative PMI values are usually not informative, and the recommendation is to replace them with zeros. It is called positive PMI.

4 Dense Vectors

The approaches we've discussed so far boil down to the explicit representations of words using simple features, such as the number of **occurrences** in contexts containing certain words from a vocabulary. To obtain embeddings of higher quality, we need to learn how to prepare more relevant word features. They should convey as much meaningful information about word properties using as few resources as possible. For this, we can use the so-called **dense word vectors** or **dense embeddings**. Such vectors are usually based on information about the joint use of words as well. However, their dimensionality is lower as compared to sparse vectors, and vectors' elements are usually not zeros.

We can say that the information represented by dense vectors is a result of original data compression. It means that vector dimensionality is much lower than the size of a vocabulary or a document collection, and, therefore, hopefully dense vectors encode only the most important word characteristics. It is not possible to describe the relationship of this word with all the words in a vocabulary or document collection. This allows dense vectors to represent word meanings more accurately.

Such vectors don't have a straightforward interpretation because it's not easy to understand why a particular value is at a certain position in a vector. We can think of such vectors as geometric points in a **multidimensional** space rather than encoded values of previously designed word features. We can interpret a point in vector space by comparing it to other points rather than on its own. Such an interpretation may include, for example, the search for nearest neighbors of this point or analysis of patterns that the points form in a vector space. We will consider two ways of creating dense vectors: via matrix factorization and via word context prediction. Let's begin with matrix factor-

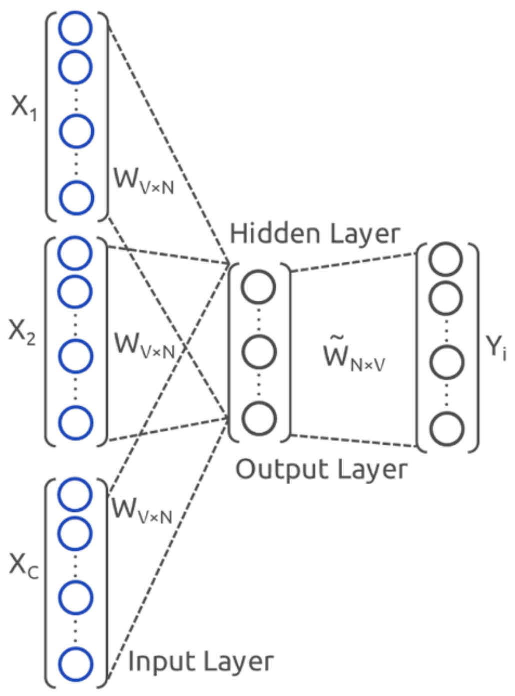


Figure 4

ization. The most common choice is singular value decomposition. Imagine a **rectangular** matrix M. In the case of distributional semantics, it can be a term-document matrix (in this case, the number of rows equals the size of a vocabulary, and the number of columns equals the number of documents in the collection), or it can be a word-context matrix (usually a square one; m equals n equals the size of a vocabulary). One can prove that any such rectangular matrix may be represented by a product of 3 matrices: an orthogonal matrix

y, a diagonal matrix S, and an orthogonal matrix V^T (V transposed). What do these matrices mean, exactly?

$$M_{m\times n} = U_{m\times r} \times S_{r\times r} \times V_{r\times n}$$

For example, we are trying to decompose a term-document matrix with w rows and d columns (by the way, if we re-weight the terms in documents, it will be called latent semantic analysis). As a result, we obtain 3 matrices: a matrix U of size w by f, a diagonal matrix S of size f by f, and a matrix V^T (V transpose) of size f by d. The first matrix U contains as many rows as there

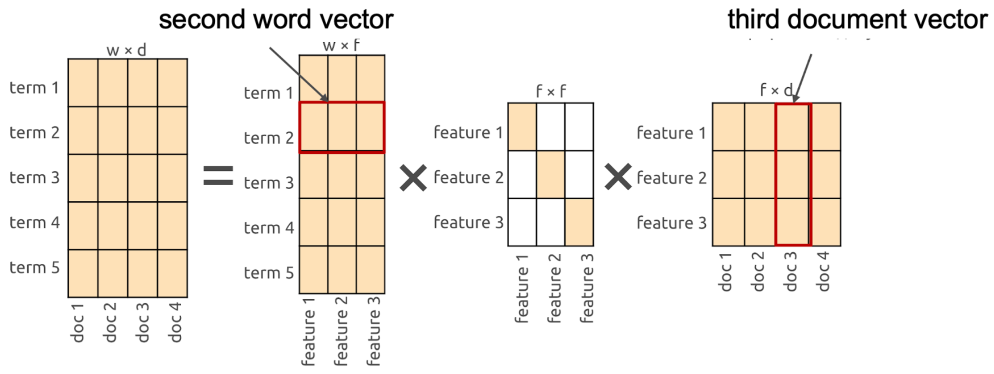


Figure 5

are words in the vocabulary, and each row is an embedding of a specific word.

The third matrix V^T contains as many columns as there are documents in the collection. It falls under the same logic. Each column is a vector of a separate document.

Both word embeddings and document embeddings contain f elements, and each element of a vector is an individual feature in the embedding. where w is the word number, d is the number of documents and f is the number of features, as well as terms-documents matrix rank

The second matrix (the diagonal matrix S) is necessary to scale these features. The values on the diagonal correspond to each feature. It is believed that the more important a feature is, the larger the value corresponding to it in a diagonal matrix is. When multiplying the matrix by matrix S, the i^{th} entry in the rows (for multiplication on the right) or columns (for multiplication on the left) is multiplied by the i^{th} value of the diagonal matrix. Thus, important and relevant features in word vectors and documents are increased while other features decrease.

With the decomposition of the like, we establish the relationship between words and texts. To reduce the dimensionality even more (for example, if R is large), we can choose some number k (for example, one hundred) of the most important features in the vector and disregard the remaining ones. To do so, we can use our diagonal matrix, sort its non-zero values and choose the k largest ones. This method is called truncated singular value decomposition (SVD).

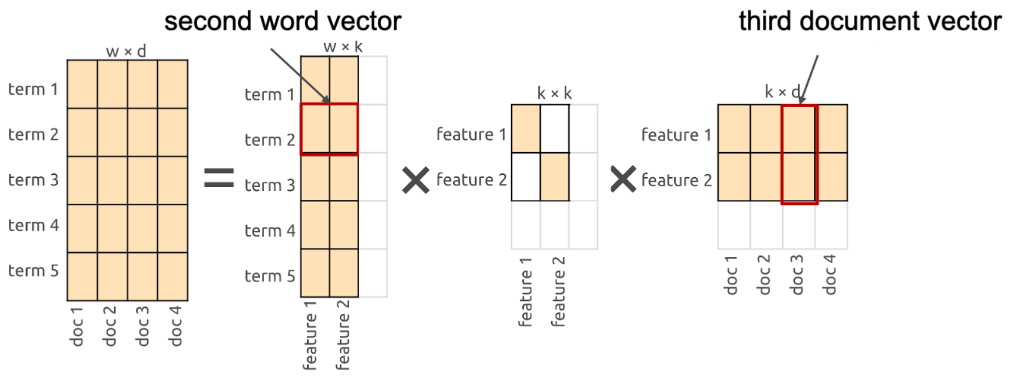


Figure 6

How good is this strategy? Please note that, when these 3 matrices are multiplied, we obtain a matrix of the same size as before but of a smaller rank. Its rank will be equal to the number of **supposedly important features that we have selected**. The Eckart–Young theorem states that this matrix is the best approximation of the original matrix among all matrices of this rank, k. So, this method effectively reduces the dimensionality of the word vectors and effectively preserves information at the same time.

Singular value decomposition is used in natural language processing to obtain embeddings of words and documents. One of the main SVD applications related to documents is latent semantic analysis. It is the decomposition of the term-document matrix as it has just been shown in the example. We can use word-context matrix factorization to obtain word vectors, for example, by applying the truncated SVD to the PPMI matrix. This approach allows to build effective vector representations of high quality, and the use of the top-k most relevant features helps to reduce noise in **embeddings**. The **disadvantage** of singular value decomposition is its high **computational complexity**. Since the sizes of a vocabulary and document collections (and, thus, the dimensionality of matrices) is often large in natural language processing, computing singular value decomposition may take a while.

Another approach to matrix factorization is called NNMF, which stands for **non-negative matrix factorization**. A **non-negative** matrix V is factorized into two matrices W and H. Interestingly, all three matrices are non-negative. For example, term-document factorization in text analysis is performed to obtain a term-feature matrix and a feature-document matrix. The logic is similar to what we have discussed in the case of singular value decomposition. The rows of the term-feature matrix are used as word embeddings, and the columns of the feature-document matrix serve as document representations. To find such a matrix, we usually turn to numerical approximation methods. **The values in matrices** W and H are initialized randomly and then iteratively updated until the product of W and H is close enough to the original matrix V.

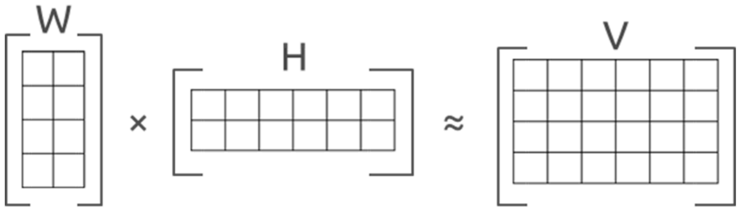


Figure 7

5 Neural Networks

The next class of distributional semantics methods is called predictive vector representation methods. As well as the vectors based on matrix factorization, vector representations obtained via predictive methods also belong to the class of dense word vectors. But unlike vectors based on matrix factorization, they employ predictive algorithms focused on the recovery of word contexts. This group of algorithms includes neural networks.

Before we discuss the distributional semantic methods based on predictions, let's take a brief look at neural networks. Neural networks are a collective name of multiple machine learning algorithms.

Just like the other machine learning algorithms, neural networks are often designed to tackle the problems that don't have an easy or straightforward or a direct analytical solution found algorithmically. Instead, neural networks use a large amount of data (for example, texts in English and their translations into German) to find hidden or implicit patterns and learn how to reproduce them (for example, to translate texts from English into German). Neural networks are not the only member of the machine learning algorithms family, yet lately they are used more often than before and are employed in many research papers.

An entire subfield of machine learning is devoted to multilayer neural networks. It is often referred to as deep learning.

Neural networks have become so important in the last decade because they can learn and reproduce various patterns in data **surprisingly well**.

For example, neural networks can add captions in natural languages to the pictures, and these captions do correspond to the content of these pictures, or neural networks can generate realistic images of things or people that never existed, create texts similar to those written by a person, or play chess. All these tasks require a good grasp of the data structure and the domain of the data

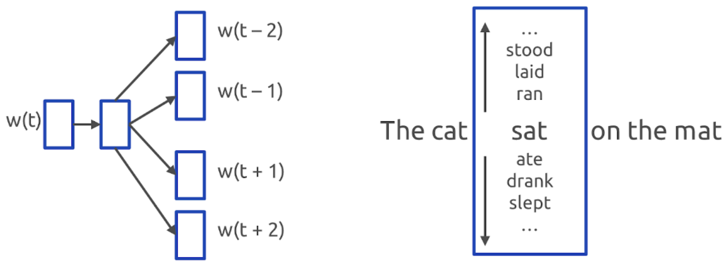


Figure 8

that is fed to the neural network since the data may consist of texts, images, or chess games, for example. Neural networks are called so because the smallest element of their architecture is called a neuron (by analogy with the nerve cells of humans and animals). We can think of a neuron as a machine with several inputs and one output (these are called input and output connections).

Each input connection is assigned a number, a so-called weight. Each input of the neuron receives a number, which is then multiplied by the weight corresponding to that exact input.

After that, the products are summed up, and the sum is fed to the nonlinear function called the activation function. Nonlinearity is very important. If the neurons were linear functions, then the entire neural network (a composition of linear functions, in essence) would be linear, and it wouldn't be necessary to describe it that way. Neurons usually form layers, where each neuron receives the outputs of the neurons of the previous layer, and the neuron output is transferred to the next layer.

An example is a fully connected network, also known as a perceptron. In a perceptron, every neuron of the previous layer can be connected to every neuron of the next layer. Look at the perceptron consisting of two layers, excluding

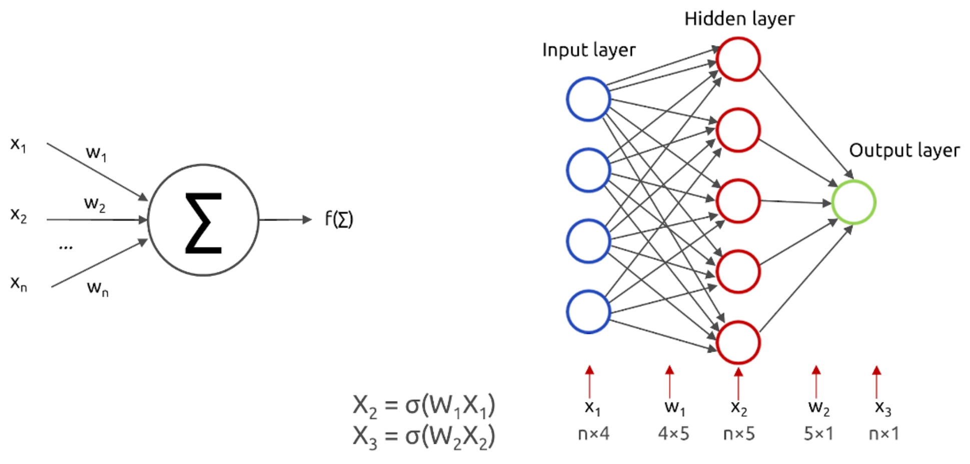


Figure 9

the input. Its performance is described by the formulas on the slide.

As input, it takes vectors consisting of four components (the n-by-4 matrix X1, where n is the number of data points). This corresponds to the fact that there are 4 inputs in the input layer. Then the matrix X_1 is multiplied by the weight matrix of the hidden layer W_1. It is a 4-by-5 matrix because there are 5 neurons in the next, so-called hidden layer. The obtained product passes through the activation function, which is denoted by the Greek letter sigma in the formula. Thus, we get the latent state of the neural network, the n-by-5 matrix X_2. Then, in the same way, X_2 is multiplied by the weights of the output layer W_2, then the result passes through the nonlinear function, and one number remains for each example. This number is the value of the activation of the output neuron. The prediction of the neural network for a given object is the n-by-1 matrix X_3, essentially a vector.

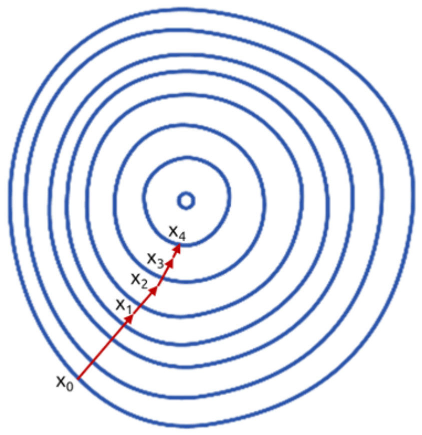


Figure 10

But how do we know what the weights between neurons should be? And here lies the problem! When we say something like a three-layer neural network with ten neurons on the first and second layers and twenty neurons on the third layer, we don't describe a specific function but rather an enormous function space defined by its parameters (weights). The task is to find a single function in this space that can solve the problem (for example, text classification, machine translation, and so on) or, equivalently, we need to figure out what the connections between neurons should be like. This search for the best weights is called training. When the weights are found, the training is complete, and the neural network is ready to be used.

Skipping the details, we can describe training like this. First, the neural network is initialized randomly. Thus one specific point is selected from the enormous function space. Usually, an untrained neural network is not good for the task. For example, if you ask the untrained neural network to classify texts, it will simply assign labels at random. **The most important trick** is to find small adjustments to the neural network's weights in order to **slightly correct** its parameters, so that it copes with the task somehow better than before. This process repeats until the stopping criterion is met (for example, the overall quality of the neural network's predictions has stopped growing, or the maximum number of steps or iterations is exceeded). If you imagine a function space, this is like a journey that we start at a random point and then take small steps to the point corresponding to the best (or at least acceptable) function.

Technically, it is done like this. The quality (or rather, poor quality) of the neural network performance on some data (training samples) is expressed via a function of our choice. Regardless of the exact chosen function, it is called a loss function.

Technically, its parameters are all the parameters of the neural network, and it returns a single number, loss. The higher the loss, the more errors (or the worse errors!) the neural network makes.

For example, cross-entropy is often used for classification. Cross-entropy equals zero when a response of the neural network is perfectly correct, and it grows larger when the difference between the response of the neural network and the correct answer grows. The cross-entropy formula is shown on the slide.

The loss function should be differentiable. *We will not discuss the standard scheme for neural network training in detail but rather briefly review it.* To improve the neural network's quality, we compute the partial derivative of the loss function for each parameter in the network and then subtract a number proportional to the value of the corresponding partial derivative (say, a one tenth or one hundredth of it) from each weight.

$$\text{Cross Entropy} = -\frac{1}{\text{m}} \sum_{i=1}^{m} y_i \cdot \log\left(\hat{y}_i\right)$$

$$w_i := w_i - \eta \frac{\partial L\left(w_1, \ldots, w_i, \ldots, w_n\right)}{\partial w_i}$$

The slide shows an example of the case when the loss takes n parameters as input. Then, to calculate an update for the i^{th} parameter, we compute the partial derivative of the loss by this parameter and subtract the number proportional to it from the previous value of the parameter.

Returning to the function space journey analogy, we can describe it as a walk up and down the hills (points with high loss value) and in the valleys (points with low loss value), and the goal of the walk is to reach the lowest or deepest place. This idea is called the loss landscape, and the entire training method is called gradient descent because the partial derivatives of a function in each of its variables are called gradients and because this method of training can be imagined as a descent in the loss landscape. To efficiently calculate the

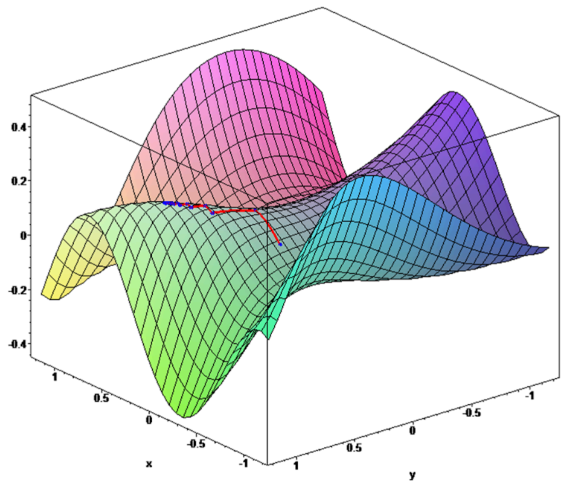


Figure 11

gradient of the loss function for multilayer neural networks, a technique known as **backpropagation** is used. The backpropagation is based on the so-called chain rule from calculus. The chain rule allows decomposing the derivative of a composite function into the product of derivatives of individual functions. In

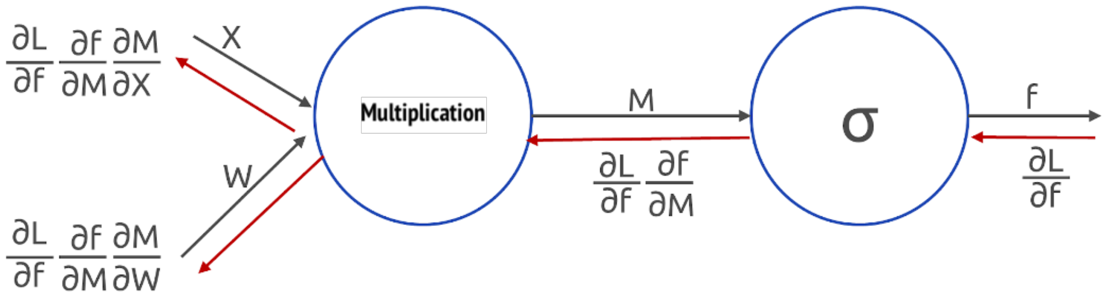

$\frac{\partial L}{\partial f}\frac{\partial f}{\partial M}\frac{\partial M}{\partial X}$
X
W
$\frac{\partial L}{\partial f}\frac{\partial f}{\partial M}\frac{\partial M}{\partial W}$
Multiplication
M
$\frac{\partial L}{\partial f}\frac{\partial f}{\partial M}$
σ
f
$\frac{\partial L}{\partial f}$

Figure 12

neural networks, each layer is applied to the output of the previous layer, so they can be represented by compositions of several functions. The chain rule makes it possible to calculate the partial derivatives of the loss function for each parameter of the neural network using the derivatives given the weights of each layer when calculating the weights of the previous layer. For this, the output of the last layer of the neural network is first calculated. Then, it is compared with the expected correct response (it is known in this case, because the output of the last layer is the output of the entire neural network). Then, the partial derivatives of the loss for the weights of the last layer are calculated. They can be used to adjust the weights of the layer so that the partial derivative values can be considered the error value for the weights. The error in the weights of the last layer is used to calculate the error value in the weights of the layer before the last one. The derivatives for the entire neural network are calculated in the same fashion. In this case, the algorithm works by computing from the last layer to the first, iterating backward from the last layer. That's why this method is called backpropagation. The slide shows an example of backpropagation for two inputs, which are multiplied, and the result of the multiplication is passed to the activation function.

Many neural networks are tailored for specific tasks, such as image or sequence processing, and so on. Neural networks that we use in NLP include, for example, recurrent neural networks and transformers. Recurrent neural networks were designed to predict sequences of arbitrary length, which is especially useful in text processing.

Such neural networks process sequences, one token at a time. They also have access to their previous states, so they can perceive any word based on words encountered in the text before. It is possible because recurrent networks have an internal state, which is, in essence, a subset of activation function values that the model had when processing the previous item in a sequence. Unlike ordinary **feedforward networks**, which we briefly reviewed, in each step they have access to the internal state of the previous step and can update it before using it in the next step. One could treat this as an analog of memory. Recurrent networks remember what they received in the previous step and can keep the information about the **current state for subsequent ones**.

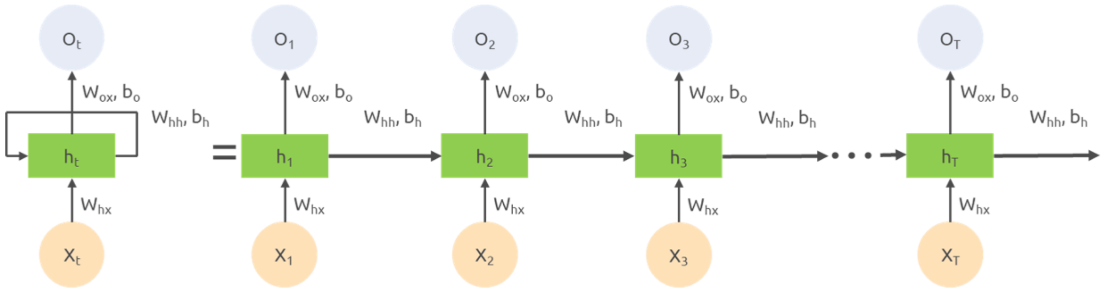


Figure 13

This basic principle has many technical details. For example, how to decide which information from the text should be remembered and which should be forgotten? Three main architectures of recurrent neural networks are the so-called **vanilla recurrent networks** (the simplest form, they are also known as **Elman** networks), gated recurrent units, or **GRU** (one so-called gate is a special layer with activation, which decides what information from the previous state is passed to the neural network as input at the next step), and networks of the long short-term memory kind, or **LSTM** (they have two gates: one for keeping information, the other for throwing it away).

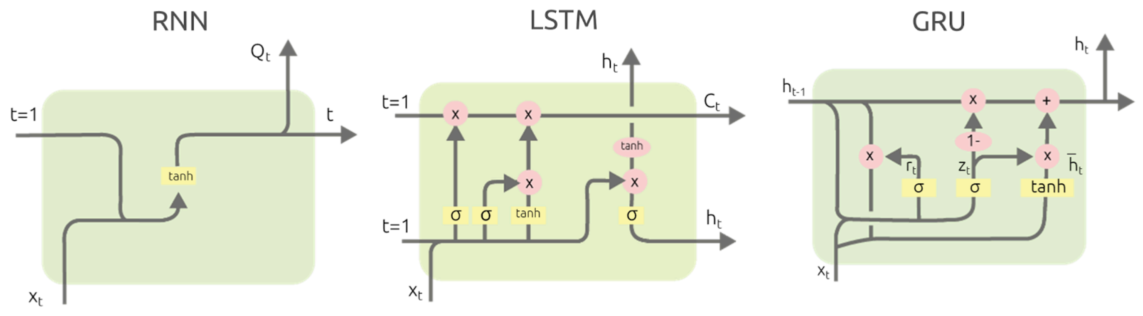


Figure 14

6 Predictive Methods

How to construct word vectors with predictive methods? The idea is to use the latent state of neural networks to describe the word properties. As you know, data in a neural network sequentially passes one layer after another. This way, the output of the previous layer becomes the input for the next layer.

These intermediate results -- hidden layers activations -- are called hiddenstates. It turns out that we can use a hidden state to create embeddings of a wide variety of objects, not only in NLP but also in image processing, for example. Here's why. A neural network that undergoes training tends to extract information from the input data that helps to solve the problem. (We say “tends to” even though the neural network is not alive and has no intentions. Anyway, during training, obtained weights of the neural network usually have this property).

Thus, to produce a good **embedding** of a word, we need to select such a machine learning problem that will require the most important features of its meaning to be used. This is where the distributional hypothesis comes into play. It states that the meaning of a word is defined by the contexts in which it is used. If so, to present the word meaning as a vector, we need to make the neural network guess either the context in which the given word is used or the word in its context. Then the state of the layer which is, for example, close to the output of the neural network could be easy to use to to guess the words nearby. It means that this hidden state will be a good vector representation.

To begin with, let's look at a method (or rather, a group of methods) called word2vec. It was published in 2013 by a group of researchers led by Tomas Mikolov. To train a word2vec model, we will need a neural network of two layers (hidden and output layer), and the hidden layer will have no activation function.

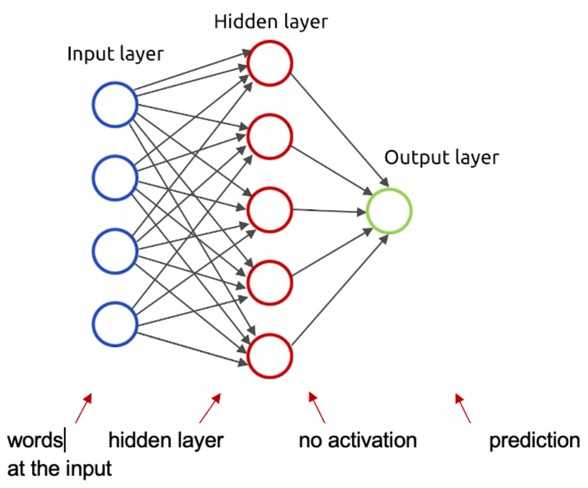


Figure 15

(as we've said earlier this is not how it is done in the neural networks world, because a composition of linear functions is a linear function, which means that a layer without an activation function can simply be joined with the next layer. Here it is necessary because our main focus is to train the state of the hidden layer. Moreover, since the dimensionality of the hidden layer is much smaller compared to the input layer, we can consider it as a dimensionality reduction task, just like with matrix factorization.)

To train our neural network on a corpus, we first need a naive representation of texts in a vector form so we can feed them to the neural network. Let's first write down all the words for which we want to create an embedding using the word2vec method (for example, it can be all words present in the corpus, or all words that have more than a certain number of occurrences in the corpus, and so on). Thus, we get a vocabulary of size k. Then we assign a unique number to each word. The easiest way is to sort them alphabetically and enumerate them accordingly. For example, if we have 50000 words, then the number 1 goes for the word *aardvark* and the number 50000 to the word *zythum*.

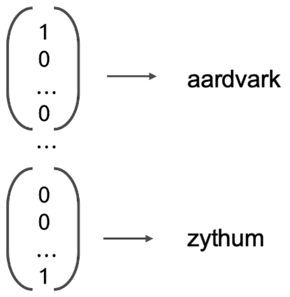


Figure 16

The naive representation of a word that can be fed to the neural network during training will be the so-called one-hot vector representation. For the i^{th} word in a vocabulary of size k, it is a k-dimensional vector in which the i^{th} element equals one, and all others are zeros. Such a vector does not reveal anything interesting about the properties of a word. Its ordinal number is the only thing we get. However, it's better than feeding the word number into the neural network. When we pass a one-hot vector to the neural network, the i^{th} neuron of the input layer is associated with the i^{th} word because it receives a non-zero signal. After that, we can train the neural network. Training word2vec could be approached by solving the problems of two types.

Here's the first type. We have a window context of a particular word, but the word itself is removed. The task is to recover the omitted word. This method is called a **continuous bag of words**, or **CBoW** for short. It resembles the bag of words that we've discussed earlier. Thus the word is represented by all the words in a span of text where it is used. The second type is opposite

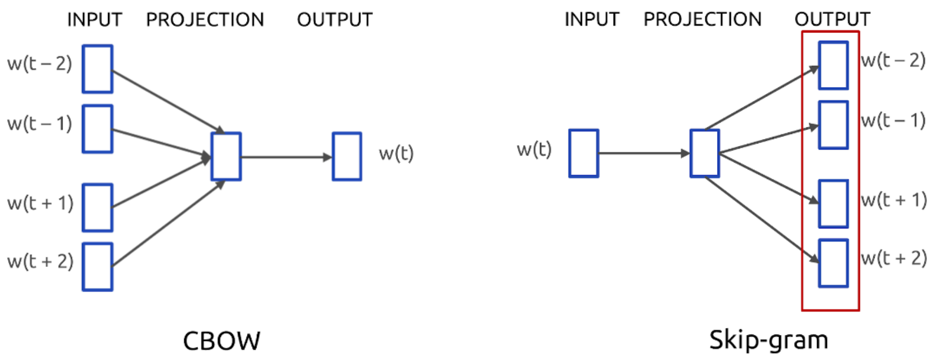


Figure 17

in a sense. We have a central word in a window context and we need to guess 2 words to the left and 2 words to the right of the given word. This method is called skip-gram, that is, an n-gram (a sequence of n words) in which one word is missing.

In the first case, all one-hot vectors are summed up, and a bag of words is fed to the neural network.

By the way, it's like extracting the vector corresponding to each row from the weight matrix and then summing them up. It is the same thing because multiplying a matrix by a one-hot vector is like taking a row out of a matrix, and matrix multiplication is associative with respect to addition.

The neural network should output one vector, ideally a one-hot vector of the word of interest. In practice, it is a vector, and the greater its i^{th} component is, the higher the probability to meet the original word in a similar context is.

In the second case, the neural network receives a one-hot vector and outputs several vectors with estimated probabilities to encounter a particular word in each position in the context of the word. These vectors are similar to the probability vector at the output of the neural network in the first case but there are several of them, not one. **This way, a model is trained for each word pair, in which one word is central and the second is contextual.** For training, we need a perceptron, a fully connected neural network. It will have two layers, namely a hidden one and an output one. The output layer should have as many neurons as there are words in the vocabulary, which is k. Therefore, we just need to select the size of the hidden layer. This size will be the same as the size of the vector representation of a word that we will get as a result. This number must be significantly smaller than the size of the vocabulary for dimensionality reduction purposes but not too small for the embedding to encode meaningful information. The standard choice is **from tens to a few hundreds**, for example, $n = 300$.

Since no activation function is present in the hidden layer, the functioning of such a multilayer perceptron can be expressed using the formula shown on the slide. The input is multiplied by two weight matrices, and then it is passed to the softmax activation function (the softmax formula is also shown on the slide). The input data matrix is multiplied by the weight matrix of the first layer, then by the weight matrix of the second layer. After that, it is passed to the activation function that normalizes the output of the neural network making it look like a probability distribution.

What will the weights of this neural network be after training? The last layer should be a function that solves the CBoW or skip-gram problem. Its input is the latent state obtained in the previous layer. If the output layer received the original one-hot vector or bag of words, it could have simply learnt the correct answer for each word separately. Having encountered a one-hot vector with the i^{th} word, the output layer would simply predict the same words that were encountered in the training sample. However, there are only 300 elements in the latent representation. For the output layer to successfully predict the solution to the problem, the latent state must put all the important information about the contexts of the given word into the three hundred elements of the vector. Now, to obtain the embedding of the i^{th} word in the vocabulary, we simply need to multiply the i^{th} one-hot vector

by the weight matrix W_1. The output will be the latent state of the neural network that received the given word as input.

We can use it as a word embedding. By the way, multiplying the i^{th} one-hot vector by the matrix is equivalent to extracting the i^{th} row from the matrix, so we can assume that the matrix W_1 is what we need. It contains vector representations of each word in the vocabulary, and the i^{th} row is a representation of the i^{th} word.

Why didn't we use the activation function in the hidden layer? Note that without the activation function, we can substitute the two matrices W_1 and W_2 with a $k \times k$ matrix W. It is similar to the co-occurrence matrices we have discussed earlier. However, the described method allows us to decompose the co-occurrence matrix into two matrices of lower dimensionality. The paper of Levy and Goldberg (the link is shown on the slide) shows that skip-gram model (with a certain set of parameters) is equivalent to matrix factorization of the PMI word-context matrix. Thus, predictive methods of distributional semantics and vector representations with a matrix factorization are similar indeed. Clearly, such a neural network has a lot of parameters. To speed up training, researchers apply various techniques. **Perhaps the most well-known** are hierarchical softmax and negative sampling.

$$\text{output } = \operatorname{softmax}\left(X_1 \cdot W_1 \cdot W_2\right)$$
$$\operatorname{softmax}\left(X_1\right) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}},$$

where k is the dictionary size, n is the hidden layer size.

Here's what hierarchical softmax does. Rather than keeping a separate parameter for each word in the output layer, all words in the vocabulary are represented by a binary tree. This way we create a tree, each leaf of which is one word in the vocabulary, and all internal nodes (all nodes except leaves) have exactly two outgoing branches (children). An example of such a tree is shown on the slide. Each node has two child nodes that have their own two child nodes each. The process is repeated until the tree ends with terminal nodes (leaves corresponding to words). To point to a particular word in the vocabulary, we don't need its number. Instead, we can describe the path to get from the root node of the tree to the leaf of interest. Imagine that you are helping a friend get somewhere in the city. You explain to your friend how to get there by saying on the phone: "Turn left, now right, now move forward." The same thing here, but the path description consists only of the words left or right.

$$J_t(\theta) = \log \sigma\left(u_0^{\top} v_c\right) + \sum_{j \sim P(w)} \left[\log \sigma\left(-u_j^{\top} v_c\right)\right]$$

For example, the word w1 on the slide corresponds to the following path: "left, left, left", and the word w2 the following: "left, left, right". When a neural network learns to predict words based on contexts or contexts based on words, instead of one classification into a large number of classes, we make several binary classifications corresponding to the left or right instructions. In practice, this is done when we match each output neuron with one of the internal nodes of the tree. During training, we are trying to ensure that the neuron corresponding to an internal node produces a value either as high as possible or as low as possible. It depends on whether we need to turn left or

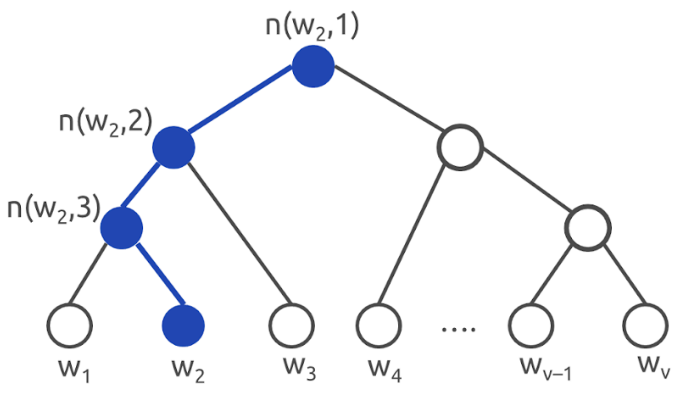


Figure 18

right after this internal node. To predict a word, first, we are trying to predict whether we need to turn left or right from the root node, then left or right from the node we arrived at, and so on until we reach a leaf of the tree. The number of internal nodes of the tree (as well as the number of output neurons and the number of parameters) is still large. However, when we make the neural network produce a higher or lower probability that a word occurs in a particular context, we are dealing with fewer internal nodes, therefore, a smaller number of parameters that need to be changed during training.

Another method is called negative sampling. It assumes that the number of neural networks' parameters does not change, and **each output neuron still corresponds to a word in the vocabulary**, but some of the parameters are frozen during training. Each iteration, we select all the output neurons that should output 1, and we also randomly choose the number of neurons that should output 0. (They are called negative samples, that is, samples of words that do not appear near the given word). We calculate the gradient for them and update their parameters (weights) during training. All other neurons remain the same. The set of neurons the parameters of which we update is different in each iteration. This way, all parameters in the neural network are eventually trained. Moreover, this reduces the overall training time.

The slide shows the formula of the function that we are trying to maximize to train word2vec using negative sampling. It consists of two terms.

The first term is a predicted probability that the original word and the context word appear together.

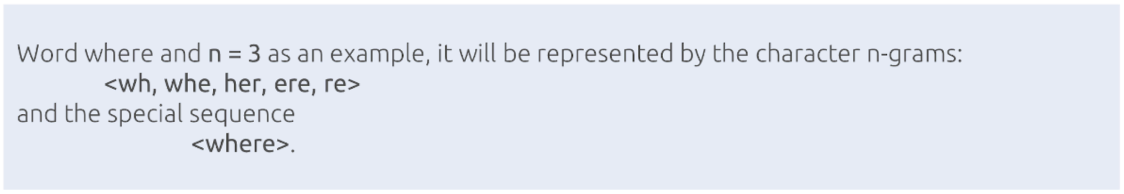
The second term is a predicted probability that the original word will appear in the text in the context of a random word of the vocabulary. This probability should be as low as possible; therefore, the score is negative.

The activation function of the output layer has also changed. It uses a sigmoid instead of softmax, the formula of which we have already seen. It's because softmax needs the outputs of all neurons of the output layer. **Note**

that their number equals the size of the vocabulary. The sigmoid works independently for each neuron, **which is much faster in terms of computational complexity**.

We can say that we have replaced one k-class classification problem with less than k binary-classification problems. One of the disadvantages of the classic word2vec algorithm is that it models the meanings of only those tokens that appeared in the training sample at least once. With word2vec, it's harder to model the meaning of rare words, while the human brain does it. For example, if you come across the word pachycephalosaurus, you will immediately guess that it is a dinosaur, even if you have never heard this word before. This is also a problem for languages in which words take many different forms. Without lemmatization, which reduces different forms of the word to one vector, word2vec will have to learn the meaning of the tokens walk, walked, walks, and so on, one by one. Besides, if one of these forms is not found in the corpus, its meaning will not be modeled, although a native speaker will immediately grasp it.

$$s(w, c) = \sum_{g \in G_w} z_g^T v_c$$

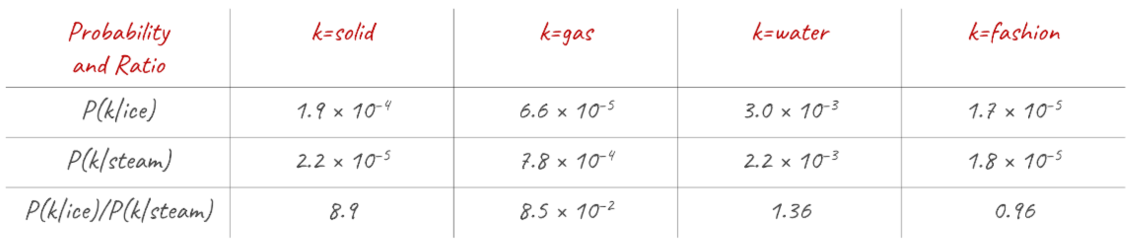


Word where and **n = 3** as an example, it will be represented by the character n-grams:
<wh, whe, her, ere, re>
and the special sequence
<where>.

Figure 19

One approach we can use to solve this problem is called fasttext. fasttext is also a vector semantics model. It is based on the skip-gram model with negative sampling, but the vector of the target word is represented by the sum of the character n-gram embeddings that constitute the word. It solves the problem of rare words, the meaning of which can be guessed by their form. For example, the word pachycephalosaurus contains saurus, which is a clue. It also solves the problem of word forms because they usually have the same or similar stems and different endings.

GloVe is another common distributional semantics model. GloVe was designed to combine the advantages of local and global statistics in distributional semantics. GloVe differs from word2vec since GloVe uses a word-context counts matrix **explicitly** during training. This matrix employs first-order co-occurrence. It stores the counts of the contexts in which two words occur together. *When discussing sparse vectors, we considered a way to move on to second-order similarity. For this, we compared row vectors in a matrix, and the similarity of*



Probability and Ratio	k=solid	k=gas	k=water	k=fashion
$P(k/ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k/steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k/ice)/P(k/steam)$	8.9	8.5×10^{-2}	1.36	0.96

Figure 20

row vectors was an indicator of the second-order similarity in the corresponding words.

GloVe takes a different approach. To calculate the second-order similarity, GloVe uses the ratio of co-occurrences of different words. For example, to measure the similarity between the words ice and steam, we can count their co-occurrences with relevant words like water and irrelevant words like fashion. Since the words water and steam have almost the same set of relevant co-occuring words, the number of co-occurrences will be approximately the same for all contexts. Hence,

the co-occurrence ratio for most contexts will be approximately equal to one. (An example of a calculation is shown on the slide). Correspondingly, GloVe embeddings are based on an attempt to predict the co-occurrence ratio of two given words to the words in a vocabulary. The disadvantage of all these approaches is that the meaning of a word may vary in different texts. A word can have multiple meanings (the word marriage can mean the ceremony or a combination of two things), and usually, the context clarifies which meaning is implied in a sentence. Also, a word may relate to other words in a sentence differently. It may refer to a person who performs an action or experiences it, or the word can be used literally or figuratively. There are many subtleties of a kind, but they are not considered in models, in which a word is represented by a single stable vector that does not change depending on a text.

So-called **contextualized vectors** are used to grasp all these subtleties. These vectors are not trying to encode the meaning of a word in general, out of context, but to represent its meaning in a particular text and the relationships between this word and other words in the context.

Contextualized vectors are produced by neural language models that deserve a separate discussion, maybe even several lectures. Their idea is also based on reusing the values at the hidden layer of the neural network when it is fed with a word for the purpose of the word representation.

Unlike the approaches we have discussed, a neural language model processes the entire text that the model is fed (this time as a sequence of tokens), and the latent state on the word of interest is used as a contextualized vector.