Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Comparing word vectors in favor of lexical semantics

Bachelor Thesis

2023
Barbora Vicianová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Comparing word vectors in favor of lexical semantics

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	Mgr. Endre Hamerlik

Bratislava, 2023
Barbora Vicianová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Barbora Vicianová
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Comparing word vectors in favor of lexical semantics
Porovnanie slovných vektorov z hľadiska lexikálnej sémantiky

Anotácia: Predmetná práca má pomôcť vyhodnotiť psychologické experimenty, kde autori skúmajú sémantickú pamäť a vybavovanie pojmov. [1]
V týchto experimentoch participanti zvyčajne generujú slovné odpovede na slovné podnety podľa určitých pravidiel. Výstup z týchto kognitívnych úloh je množina slov), ktoré sú buď vo forme slovných párov resp. "edgelistov" (Podnet->Odpoveď).

Cieľ: Cieľom predmetnej bakalárskej práce je vytvoriť softvérové riešenie (dielo), ktoré bude schopné vyhodnocovať sémantickú podobnosť slovných párov na základe podobnosti embeddingov (vektorových reprezentácii):
statických Word2vec [2]
a kontextuálnych Slovak BERT [3]

Literatúra: [1] Marko, M., Michalko, D., Dragašek, J., Vančová, Z., Jarčušková, D., & Riečanský, I. (2022). Assessment of automatic and controlled retrieval using verbal fluency tasks. Assessment, 10731911221117512.
[2] Mikolov, T., Grave, E., Bojanowski, P., Puhrsch, C., & Joulin, A. (2017). Advances in pre-training distributed word representations. arXiv preprint arXiv:1712.09405.
[3] Pikuliak, M., Grivalský, Š., Konôpka, M., Blšták, M., Tamajka, M., Bachratý, V., ... & Uhlárik, F. (2021). SlovakBERT: Slovak Masked Language Model. arXiv preprint arXiv:2109.15254.

Kľúčové slová: vektorová sémantika, lexikálna sémantika, statické embeddingy, kontextuálne embeddingy, verbálna plynulosť

Vedúci: Mgr. Endre Hamerlik
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 03.10.2022

Dátum schválenia: 04.11.2022 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

....

študent vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Barbora Vicianová
Study programme:	Applied Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Comparing word vectors in favor of lexical semantics

Annotation: Present work is intended to help evaluate psychological experiments where the authors investigate semantic memory and concept recall. [1]
In these experiments, participants usually generate verbal responses to verbal stimuli according to certain rules. The output of these cognitive tasks is a set of words, which are in the form of word pairs/edgelist (Stimulus->Response).

Aim: The goal of the subject bachelor's thesis is to create a software solution that will be able to evaluate the semantic similarity between pairs of words based on the similarity of vector representations of verbal responses using:
- static embeddings (Word2vec [2])
- and contextual embeddings as well (Slovak BERT [3])

Literature: [1] Marko, M., Michalko, D., Dragašek, J., Vančová, Z., Jarčušková, D., & Riečanský, I. (2022). Assessment of automatic and controlled retrieval using verbal fluency tasks. Assessment, 10731911221117512.
[2] Mikolov, T., Grave, E., Bojanowski, P., Puhrsch, C., & Joulin, A. (2017). Advances in pre-training distributed word representations. arXiv preprint arXiv:1712.09405.
[3] Pikuliak, M., Grivalský, Š., Konôpka, M., Blšták, M., Tamajka, M., Bachratý, V., ... & Uhlárik, F. (2021). SlovakBERT: Slovak Masked Language Model. arXiv preprint arXiv:2109.15254.

Keywords: vector semantics, lexical semantics, static embeddings, contextual embeddings, verbal fluency

Supervisor:	Mgr. Endre Hamerlik
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	03.10.2022
Approved:	04.11.2022

doc. RNDr. Damas Gruska, PhD.
Guarantor of Study Programme

Student

Supervisor

Abstrakt

Kľúčové slová: jedno, druhé, tretie (prípadne štvrté, piate)

Abstract

Keywords:

Contents

List of Figures

List of Tables

Chapter 1

Related work

1.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computing systems that are inspired by the structure and function of the human brain. ANNs have become increasingly popular in recent years due to their ability to learn complex patterns and make accurate predictions on a wide range of tasks, such as image classification, speech recognition, and natural language processing. In this chapter we write about the basic structure of ANN, training methods and optimization, based on [4] and [13].

The structure of ANNs is based on interconnected nodes called neurons, which are organized into layers. These artificial neurons are conceptually derived from biological neurons in the human brain, which receive and transmit electrical signals. In ANNs, each neuron receives inputs from neurons in the previous layer, processes them using weights, and sends output signals to neurons in the next layer. The weights determine the impact of each neuron on the output of the next layer, and they are updated during the training process to optimize the network's performance.

1.1.1 Multi-Layer Perceptron (MLP)

Multi-Layer Perceptron (MLP) is the simplest deep network[13]. It consists of neurons, which are organized into multiple layers: an input layer, one or more hidden layer and an output layer.

Let see an example: the matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ denotes a batch of n input examples, each having d input features (d being 4 in the case of the example figure 1.1). In practice, we typically process data in minibatches, that refers to a subset of the entire dataset. Instead of processing the entire dataset at once, it is divided into smaller batches or subsets of a fixed size, which are processed sequentially. For the MLP with one hidden layer (as is our example 1.1) with h hidden units, we denote by $\mathbf{H} \in \mathbb{R}^{n \times h}$ the outputs of the hidden layer. By having both hidden and output layers fully connected, we are

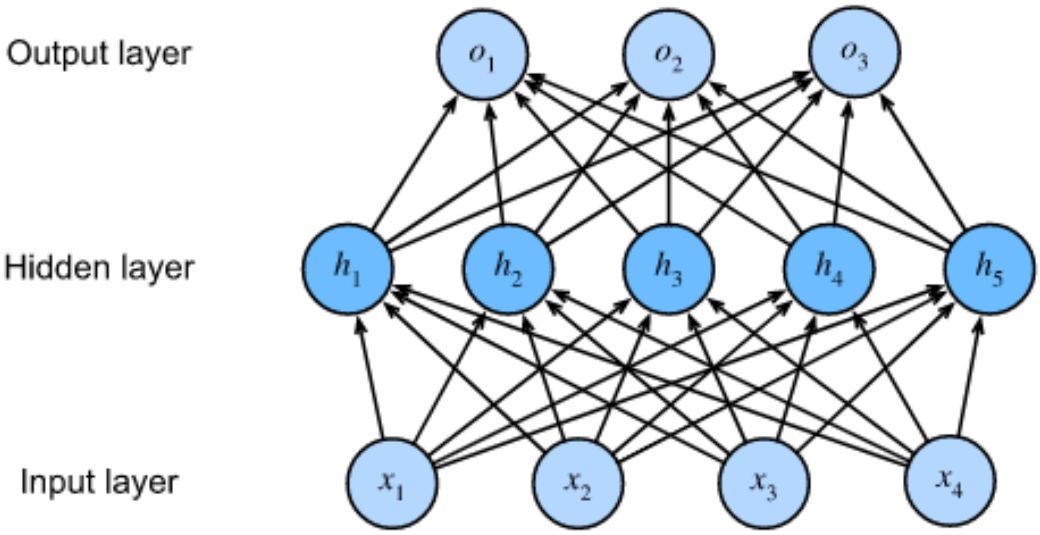

Figure 1.1: An MLP with a hidden layer of 5 hidden units. [13].

able to obtain the weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$ for the hidden layer and the weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and the biases $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$ for the output layer. This enables us to compute the outputs $\mathbf{O} \in \mathbb{R}^{n \times q}$ of the one-hidden-layer MLP as follows:

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)},$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

It can be proven that simply adding a hidden layer does not change the fact that the model can only approximate affine functions using linear operations (for a detailed explanation and proof, refer to [13]).

Therefore, we need to use a non-linear activation function, denoted by σ, which is applied to each hidden unit after the affine transformation. The outputs of activation functions are called activations. With activation functions in place, it is no longer possible to collapse our MLP into a linear model:

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}),$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

Because each row in $\mathbf{X}$ corresponds to an example in the minibatch, we define the activation function σ to apply to the inputs of each row individually, i.e., one example at a time, as defined in [13].

1.1.2 Activations

The activation function is another key component of the structure of ANNs. Activation functions play a key role in determining whether a neuron should be activated or not, based on the weighted sum of inputs plus a bias term. They are differentiable operators that transform input signals to outputs, with many of them adding non-linearity.

Given their importance to deep learning, it is useful to examine some commonly used activation functions.

Sigmoid Function

The sigmoid function is a commonly used activation function in artificial neural networks. It maps the input values to outputs between 0 and 1, hence it is often referred to as a squashing function [13].

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid activation functions are mostly applied output units for binary classification problems, where outputs are interpreted as probabilities.

Tanh Function

Similar to the sigmoid function, the hyperbolic tangent (tanh) function also squashes its input values, mapping them to the range between -1 and 1.

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

The function is similar in shape to the sigmoid function, but has point symmetry about the origin of the coordinate system. While the sigmoid function is commonly used as an activation function for output layers in binary classification problems, the *tanh* function is more frequently utilized in hidden layers due to its ability to model complex non-linear relationships between input and output.

ReLU Function

One of the most widely used activation functions in ANN is the Rectified Linear Unit (ReLU) because it is both easy to implement and performs well on many predictive tasks. ReLU is a simple non-linear transformation where the output is defined as the maximum of the input element and 0. It is a popular choice due to its simplicity and good performance.

$$\text{Relu}(x) = max(0, x)$$

Softmax function

The softmax function is commonly used in the output layer of machine learning models [13]. It takes a vector $\mathbf{x}$ of n real numbers and transforms it into another vector where all components are in the interval (0, 1) and their sum equals one, representing a probability distribution. This is done by normalizing the input vector to a probability

distribution of n probabilities, divided by the sum of exponentials of the input variables. The function is defined with the following equation:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{1.1}$$

where x_i are the components of vector $\mathbf{x}$ from x_1 to x_n and j going through all elements.

1.1.3 Forward Propagation

Forward propagation is the process of calculating the output of a neural network given an input[4] [13]. During forward propagation, the input values are multiplied by the weights of the first layer, then a non-linear activation function is applied, and this process is repeated for each layer until the final layer produces the output.

Let us look at this process step-by-step:

The input example is $\mathbf{x} \in \mathbb{R}^d$, then the intermediate variable is

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter from the hidden layer. Then we apply the activation function ϕ and we obtain the hidden activation vector of length h,

$$\mathbf{h} = \phi(\mathbf{z}).$$

The output $\mathbf{h}$ of the hidden layer is also an intermediate variable.

Given the example shown in Figure 1.1, if we assume that the parameters of the output layer correspond to a weight matrix $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, then we can obtain the output layer variable $\mathbf{o}$ with a vector of length q as follows:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$

The forward propagation step is performed for each input in the training dataset during the training. The output is compared to the expected output, and the difference between the two is used to adjust the weights and biases of the network in the backward propagation step.

1.1.4 Backpropagation

Backpropagation is a technique used to compute the gradient of neural network parameters. It involves traversing the network in a reverse order, starting from the output layer and moving towards the input layer using the chain rule from calculus. This algorithm stores any intermediate variables (partial derivatives) needed while computing the gradient with respect to specific parameters. This gradient is then used to

update the weights and biases in the opposite direction of the gradient, with the goal of reducing the loss function.[13]

1.1.5 Adam

Adam (adaptive moment estimation) is a first-order gradient-based optimization algorithm introduced by Kingma et al.[6]. This method combines the strengths of two commonly used optimization techniques: AdaGrad (Duchi et al., 2011[3]), which is effective for dealing with sparse gradients, and RMSProp (Tieleman & Hinton, 2012 [11]), which performs well in non-stationary and online settings. Let us look at the algorithm cited from [13].

Adam uses exponential weighted moving averages, also called leaky averaging, to estimate the momentum and the second moment of the gradient, by using these variables:

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\mathbf{g}_t,$$
$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2.$$

The β_1 and β_2 are non-negative weighting parameters, which are often set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$.If $\mathbf{v}_0 = \mathbf{s}_0$ are initialized as 0, initially we experience a significant bias towards smaller values. This can be managed by using $\sum_{i=0}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$ to re-normalize terms. Accordingly, the normalized state variables are given by

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}.$$

Now that we have the correct estimates, we can write down the update equations. First, we scale the gradient in a similar way to RMSProp.

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$$

Now we have everything to calculate updates, which can be computed using this formula:

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t'.$$

1.1.6 Generalization

Generalization in machine learning refers to the ability of a model to make accurate predictions on new, unseen data, beyond the data it was trained on. It is the ultimate goal of machine learning, where the purpose is not only to fit the training data but also to discover underlying patterns that enable the model to make accurate predictions on new, unseen data. In deep learning, generalization is a crucial challenge due to the complexity of the models and the high dimensionality of the data. While the

theory of deep learning is still evolving and far from a comprehensive account of both optimization and generalization, practitioners have developed a range of techniques and heuristics that can help to produce models that generalize well in practice.

1.1.7 Overfitting

Overfitting occurs when the model learns to fit the training data too closely and fails to generalize well to new, unseen data. This can happen when the model is too complex or when it is trained for too many epochs, resulting in a model that is too specialized to the training data and unable to capture the underlying patterns in the data. As a result, the model may perform very well on the training data but poorly on the test data, leading to a large generalization gap. To prevent overfitting, techniques such as early stopping, regularization, and dropout can be applied during training.

There are several methods that can be used to address overfitting. These methods include weight decay, dropout and early stopping.

Weight decay Weight decay adds a penalty term to the loss function that the model is trying to minimize. This penalty term is proportional to the square of the weights in the model, and it encourages the model to use smaller weights. The mathematical equation for weight decay is:

$$\mathcal{L}(w, b) = \mathcal{L} + \frac{\lambda}{2}||w||^2 \tag{1.2}$$

In this equation, $\mathcal{L}$ is the original loss function, λ is the weight decay coefficient and $||w||^2$ is the norm of the weight vector.

Dropout Dropout works by randomly dropping out (setting to zero) a proportion of the neurons in a layer during training. The dropped out neurons are selected at random, with a given probability, typically around 0.5. With dropout probability p, each intermediate activation h is replaced by a random variable h' as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases} \tag{1.3}$$

Early stopping Early stopping constrains the number of epochs of training. During the training, the performance on validation data is monitoring, usually by checking it once after each epoch. The training stops, when the validation error has not decreased by more than some small amount ϵ for some number of epochs.

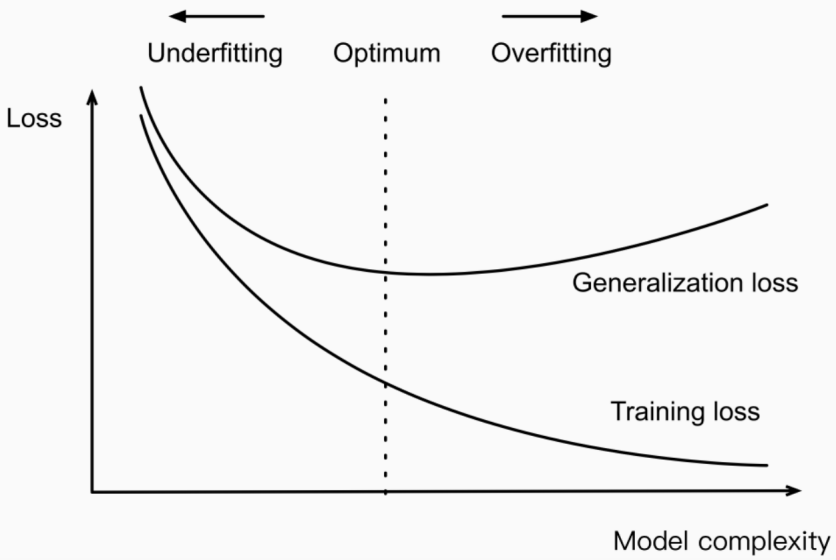

Figure 1.2: Influence of model complexity on underfitting and overfitting [13].

1.2 Word2vec

In this section, we will discuss pre-trained word representations, such as Word2vec, according to a papers [9] and [8]. Word2vec representations are commonly used in NLP pipelines to improve their performance. These pre-trained representations provide distributional information about words that can improve the generalization of models learned on limited amount of data. The typical method for obtaining word representations is through log-bilinear models trained using either the skip-gram or continuous bag-of-words (CBOW) architectures, which are commonly implemented in word2vec. These models are trained on large unlabeled corpora of text data, and capture statistical information from vast sources of data.

Several modifications to the standard word2vec training pipeline significantly improves the quality of the resulting word vectors. These include position-dependent features, phrase representations. These modifications have been evaluated on various benchmarks such as syntactic, semantic, and phrase-based analogies, rare words dataset, and as features in a question-answering pipeline.

The $CBOW$ model learns to predict a target word based on its surrounding context. The context is defined as a symmetric window that includes all the adjacent words. In other words, given a sequence of T words $w_1, w_2, ..., w_T$, the $CBOW$ model aims to maximize the log-likelihood of the probability of the words, given their surrounding context, i.e.:

$$\sum_{t=1}^{T} \log p = (w_t|C_t) \tag{1.4}$$

where C_t is the context of the t-th word.

The Continuous Skip-gram Model is similar to *CBOW* model, but it maximizes the classification of a word using another word within the same sentence, instead of predicting the current word based on the context. Specifically, the model uses a log-linear classifier with a continuous projection layer to predict words within a certain range before and after the current word. Increasing the range improves the quality of the resulting word vectors, but it also increases the computational complexity. To reduce the impact of less relevant distant words, less weight is given to them by sampling less from those words in the training examples.

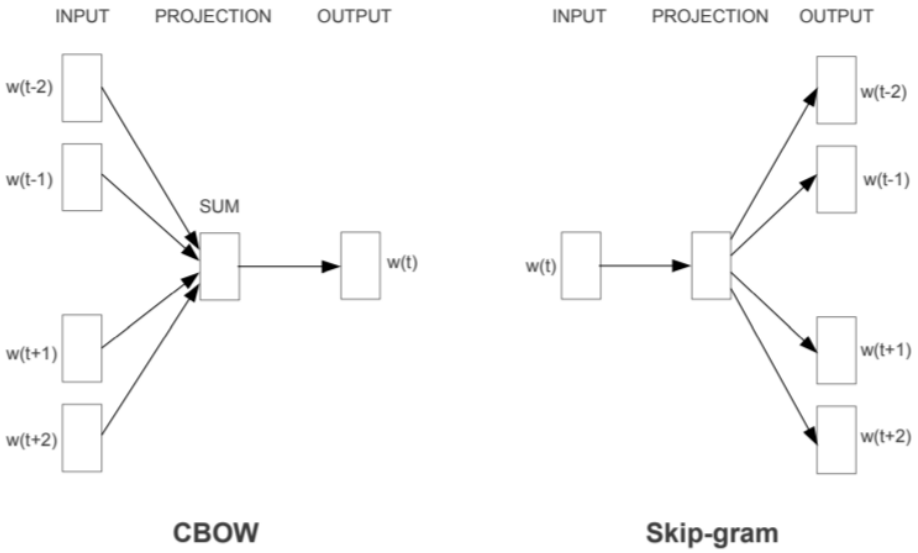

Figure 1.3: CBOW and Skip-gram models architecture [8].

1.3 Transformer

Recurrent neural networks, including long short-term memory [5] and gated recurrent neural networks [1], are commonly used for sequence modeling and transduction tasks such as language modeling and machine translation. Researchers have continued to improve these models and explore new architectures. However, these models are sequential, which limits parallelization within training examples and becomes critical at longer sequence lengths.

To address this issue, the Transformer [12] model was proposed. It relies entirely on an self-attention mechanism to draw global dependencies between input and output sequences, instead of recurrence. This allows for significantly more parallelization and achieves state-of-the-art results in translation quality with a shorter training time.

Architecture of the transformer

Most competitive neural models for sequence translation have an encoder-decoder structure. The encoder maps an input sequence of symbols to continuous representations, while the decoder generates an output sequence one symbol at a time, using the previously generated symbols as additional input. The Transformer follows this structure, using self-attention and fully connected layers for both the encoder and decoder. This is shown in Figure 1.4, where the left half represents the encoder and the right half represents the decoder.

The encoder in the Transformer model consists of six identical layers. Each layer has two parts: a multi-head self-attention mechanism and a simple, fully connected network. Both parts are surrounded by a residual connection and layer normalization. The output of each sub-layer is added to the input to facilitate residual connections. All sub-layers in the model and embedding layers produce outputs of dimension 512 to enable these connections.

The decoder also has six identical layers, with two sub-layers in each layer as in the encoder. Additionally, a third sub-layer performs multi-head attention over the output of the encoder stack. To prevent attending to subsequent positions, the self-attention sub-layer in the decoder stack is modified with masking, and the output embeddings are offset by one position. This ensures that the predictions for position i only depend on known outputs at positions less than i.

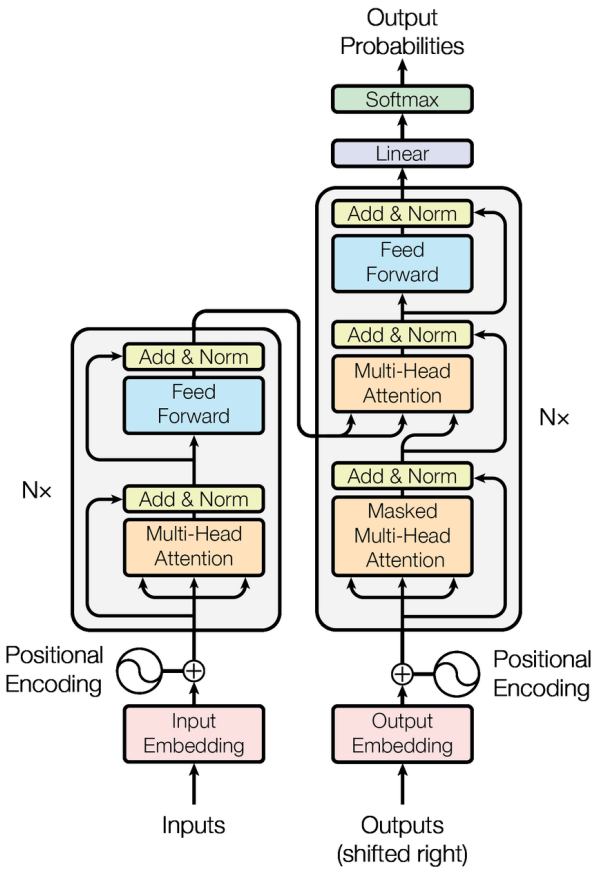

Figure 1.4: The architecture of the Transformer - model from [12]

1.4 BERT

The paper titled "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" [2], introduces a language representation model called BERT (Bidirectional Encoder Representations from Transformers). Unlike previous language models, BERT is designed to pretrain deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. This approach enables BERT to be fine-tuned for a wide range of tasks, such as question answering and language inference, with just one additional output layer, without substantial task-specific architecture modifications.

The model is first pre-trained on unlabeled data over different pre-training tasks. One of the two tasks was masked language modeling, which involved randomly masking a certain percentage of the input tokens and then predicting the masked tokens. The second task was Next Sentence Prediction (NSP), which involve predicting whether a given sentence was the actual next sentence in a pair of sentences or a random sentence from the corpus, and helped the BERT model understand the relationship between two sentences.

After pretraining, the model is fine-tuned using end-task labeled data. This process

is simple and efficient due to the self-attention mechanism in the Transformer that enables the model to handle various downstream tasks. The input and output layers are plugged into BERT for each task, and all parameters are fine-tuned end-to-end. Fine-tuning is less expensive compared to pre-training.

The architecture of the model is a bidirectional Transformer encoder with multiple layers, which is based on the original implementation explained in the paper [12] BERT comes in two sizes: $BERT_{base}$ and $BERT_{large}$. $BERT_{base}$ has 12 Transformer blocks, 12 self-attention heads, and 110 million parameters, while $BERT_{large}$ has 24 Transformer blocks, 16 self-attention heads, and 340 million parameters.

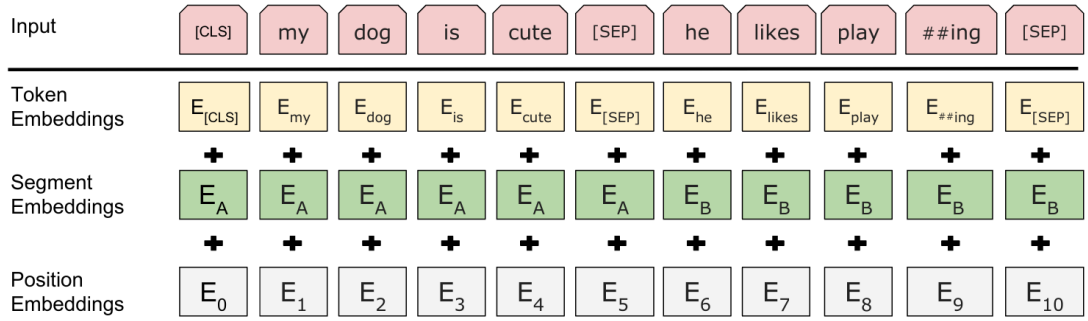

Figure 1.5: The representation of the BERT input. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.[2]

Input/Output Representations

The input representation of BERT is able to handle both single sentences and pairs of sentences in one token sequence. Each sequence starts with a special classification token ([CLS]) and is separated by a special token ([SEP]). A learned embedding is added to every token indicating whether it belongs to sentence A or sentence B. The input embedding is constructed by summing the corresponding token, segment, and position embeddings as seen on Figure 1.5.

The output representation of BERT is a set of contextualized word embeddings, where each word is associated with a vector representation that captures its contextual meaning in the sentence.

1.5 RoBERTa

RoBERTa (Robustly Optimized BERT Approach) is a transformer-based language model that was developed by Liu et al. [7]. This model is based on BERT released by Jacob Devlin et al.[2]. In their study, Liu et al. conducted a replication of the BERT

pre-training approach with a focus on the effects of hyperparameter tuning and training set size. They found that, the original BERT model was undertrained, therefore they proposed an improved recipe for training BERT models that they called RoBERTa. Their modifications to the original approach includes training the model longer with larger batches and over more data, removing the next sentence prediction objective, training on longer sequences and dynamically changing the masking pattern applied to the training data.

1.6 SlovakBERT

In this section, we describe Slovak masked language model called SlovakBERT [10]. It has RoBERTa architecture with 12 layers, 12 self-attention heads and 768 hidden size. It was trained using Web-crawled corpus. The available corpora they used were: Wikipedia (326MB of text), Open Subtitles (415MB) and OSCAR 2019 corpus (4.6GB). They crawled .sk top-level domain webpages, they extracted the title and the main content of each page as clean text without HTML tags (17.4GB).

Bibliography

[1] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[3] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. In *Journal of machine learning research*, 2011.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[7] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[8] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[9] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations, 2017.

[10] Matúš Pikuliak, Štefan Grivalský, Martin Konôpka, Miroslav Blšták, Martin Tamajka, Viktor Bachratý, Marián Šimko, Pavol Balážik, Michal Trnka, and Filip Uhlárik. Slovakbert: Slovak masked language model, 2022.

[11] T. Tieleman and G. Hinton. Lecture 6.5 - rmsprop, coursera: Neural networks for machine learning. technical report. 2012.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[13] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning, 2023.