# *Modern Approaches in Natural Language Processing*

# *Contents*

# *Preface*

In the last few years, there have been several breakthroughs concerning the methodologies used in Natural Language Processing (NLP). These breakthroughs originate from both new modeling frameworks as well as from improvements in the availability of computational and lexical resources.

In this seminar booklet, we are reviewing these frameworks starting with a methodology that can be seen as the beginning of modern NLP: *Word Embeddings*.

We will further discuss the integration of embeddings into end-to-end trainable approaches, namely convolutional and recurrent neural networks.

The second chapter of this booklet is going to cover the impact of Attention-based models, since they are the foundation of most of the recent state-of-the-art architectures. Consequently, we will also spend a large part of this chapter on the use of transfer learning approaches in modern NLP.

To cap it all of, the last chapter will be abour pre-training resources and benchmark tasks/data sets for evaluating state-of-the-art models followed by an illustrative use case on Natural Language Generation.

This book is the outcome of the seminar "Modern Approaches in Natural Language Processing" which took place in the summer term 2020 at the Department of Statistics, LMU Munich.


**FIGURE 1:** Creative Commons License



[1] http://creativecommons.org/licenses/by-nc-sa/4.0/

# *Foreword*

*Author: Matthias Aßenmacher*

This book is the result of an experiment in university teaching. We were inspired by a group of other PhD Students around Christoph Molnar, who conducted another seminar on Interpretable Machine Learning[2] in this format. Instead of letting every student work on a seminar paper, which more or less isolated from the other students, we wanted to foster collaboration between the students and enable them to produce a tangible outout (that isn't written to spend the rest of its time in (digital) drawers). In the summer term 2020, some Statistics & Data Science Master students signed up for our seminar entitled "Modern Approaches in Natural Language Processing" and had (before kick-off meeting) no idea what they had signed up for: Having written an entire book by the end of the semester.

We were bound by the examination rules for conducting the seminar, but otherwise we could deviate from the traditional format. We deviated in several ways:

1. Each student project is a chapter of this booklet, linked contentwise to other chapers since there's partly a large overlap between the topics.
2. We gave challenges to the students, instead of papers. The challenge was to investigate a specific impactful recent model or method from the field of NLP.
3. We designed the work to live beyond the seminar.
4. We emphasized collaboration. Students wrote the introduction to chapters in teams and reviewed each others individual texts.

## Technical Setup

The book chapters are written in the Markdown language. The simulations, data examples and visualizations were created with R (R Core Team, 2018). To

[2] https://compstat-lmu.github.io/iml_methods_limitations/

combine R-code and Markdown, we used rmarkdown. The book was compiled with the bookdown package. We collaborated using git and github. For details, head over to the book's repository[3].

[3] https://github.com/compstat-lmu/seminar_nlp_ss20

# 1

# *Introduction*

*Author: Xiao-Yin To*

*Supervisor: Daniel Schalk, Matthias Aßenmacher*

Over the course of the past decades the importance and utilization of artificial intelligence technology has continuously gained traction. In the present times, it is already inextricably linked with most of the surroundings that constitute the human shaped environment. Consequently, a myriad of sectors such as commerce, research and development, information services, engineering, social services, and medical science have already been irreversibly impacted by the capabilities of artificial intelligence. There are three major fields of artificial intelligence that comprise the technology: speech recognition, computer vision, and natural language processing (see Yeung (2020)). In this book we will take a closer look at the modern approaches in natural language processing (NLP).

## 1.1 History and development

The history of artificial intelligence and NLP dates back to the 18th century, where well-known philosophers such as Leibniz, Spinoza, Hobbes, Locke, Kant, and Hume as well as scientists such as La Mettrie and Hartley tried to formulate laws of thought (see Mccorduck and Cfe (2004)).

However, the first steps of development started in the 20th century. Alan Turing (1937) was the first to propose an abstract Universal Computing Machine and became one of the most defining scientists who shaped the path of the scientific development of artificial intelligence in the following years. He further developed these ideas in his work "Intelligent machinery" (see Turing (1948)) and "Computing Machinery And Intelligence" (see Turing (1950)). In 1949, Warren Weaver proposed that "given that all humans are the same (inspite of speaking a variety of languages), a document in one language could be viewed as having been written in code. Once this code was broken, it would be possible to output the document in another language. From this point of view, German was English in code" (see Weaver (1949)).

In the post World War II era, numerous German documents needed to be translated into the English language. Due to the sheer number of documents, an automatic decryption system was required in order to ensure time efficiency regarding the herculean task, which further accelerated the research concerning NLP. The first teams were comprised of numerous bilingual programmers. The idea behind this was that the knowledge of multiple languages might facilitate the process of creating programs, which could understand languages and their structures, and would subsequently be able to translate texts. While working on the first NLP programs the main difficulty that crystallized related to the complexity and irregularity of many languages (see Hancox (1996)). Beginning in the 1950s, linguists and machine learning teams congregated and introduced new ideas. During this time period, Georgetown University and International Business Machines Corporation (IBM) published the Georgetown Experiment, the first public demonstration of machine translation, which included the first fully automatic translation, being able to translate more than sixty Russian sentences into English (see Hutchins (2005)). Moreover, Noam Chomsky, one of the most important and influential scientists in linguistics, introduced the idea of Generative Grammar, which describes syntactic structures based on rules (see Chomsky (1957)). The most successful NLP systems that were developed back then were the first chatbot ELIZA (see Weizenbaum (1966)), STUDENT (see Bobrow (1964)), and SHRDLU, a language program that allowed user interaction with a block world (see Winograd (1972)). As the resources that were available for computing in the past were extremely undeveloped – access to computers was restricted, the machines were still really slow, storage was limited, and there were no suitable higher-level programming languages – the creation of such programs was considerably more difficult. The fact that any progress in this field was attained makes the achievements of those scientists all the more remarkable.

In addition to limited resources, researchers encountered the problem that research regarding the development of NLP software came with high costs: By the mid-1960s, machine translation research expenses amounted to 20 million USD, which were paid by U.S. government funding. Those two obstacles, resource limitations along with high costs, were the main reason for the slow advancement of research in this area. The history of NLP reached its lowest point, when in 1966 the Automatic Language Processing Advisory Committee evaluated the results that were attained through the funding and reported, that "there had been no machine translation of general scientific text, and none is in immediate prospect" (alp (1966)). This report caused U.S. funding to be discontinued, which is the reason why in the following decade the quantity of NLP in scientific literature decreased enormously. Nevertheless, compelling developments such as Augmented Transition Networks, which aid in the analysis of sentence structures, Case Grammar, which facilitates comprehension of linguistic structures by using the link between different components of sentences, and Semantic Representation, which signifies an abstract

language in which meanings can be represented, originated in that time (see Hancox (1996)).

In the 1980s, the so-called *Statistical Revolution* took place. Prior to that, NLP was a primarily "grammar-based approach", which denotes that systems were created by hand-coding rules and parameters. Via the statistical revolution, the empirical "statistical approach" was introduced (see Johnson (2009)) and consequently "NLP was characterized by the exploitation of data corpora and of (shallow) machine learning, statistical or otherwise, to make use of such data" (see Deng and Liu (2018)). This approach has dominated NLP ever since, as the amount of machine-readable data and computational power has continuously expanded. Since simple Machine Learning techniques are often not sufficient for creating NLP applications that can fulfil the requirements of real-life tasks, nowadays most of the methods are based on Deep Learning designs (see Deng and Liu (2018)).

## 1.2 Statistical Background

Ever since the Statistical Revolution, many challenging aspects could be tackled using statistical approaches and artificial intelligence. Statistics shaped a substantial part of the path of NLP, conjointly with fundamental knowledge of linguistics. The statistical approaches used in this booklet presuppose mathematical foundations such as elementary probability theory and essential information theory. In order facilitate the comprehension of the approaches explained in the later chapters, now some of the basic schemes that lie the foundations to those modern approaches in NLP will be introduced.

Human language courses usually consist of two elementary parts: vocabulary and grammar. The language skills are often measured by the number of words a person knows, while grammar allows using the words and form sentences correctly. Further, NLP systems basically consist of learning and understanding words as well as recognizing the patterns in which they occur.

The first step is characterized by the recognition and comprehension of words. One difficulty arising when trying to understand words is that many words possess multiple meanings. It might be challenging to ascertain which of the meanings is implied. For word sense, disambiguation methods such as bag of words models or Bayesian classification can be used, which inspect the words around the ambiguous word. Bag of words models consider the dependence of words in a so-called bag, a vector of words that appear in a sentence, so co-occurrences can be learned without understanding grammar. By applying the Bayesian decision rule, the meaning of the word will be decided by choosing the meaning with the highest conditional probability while minimizing the

probability of error. Another difficulty is that in many languages words exist, which do not (only) have a meaning themselves, but also possess combined meanings in a collocation, an expression consisting of two or more words. One way for NLP systems to implement this is by using basic statistics such as frequency, mean and variance, and hypothesis testing. If two or more words often occur together in a sentence, it may be concluded that these words together possess a special function in this sentence and cannot be explained by the combination of their respective meanings. Mean and variance can help finding the affiliation between words, which do not always appear in the same structure or with the same distance within a phrase, by calculating the mean distance between words in a sentence and the variance of this distance during training, in order to enable correct classification of these words into a collocation in later applications of the model.

The second step is understanding not only the words themselves, but also their meaning given their contexts. As an example, Markov Models can be used for the classification of texts depending on the surrounding context as well as grammatical structure finding. A Markov Model is a sequence classifier which assigns a sequence of classes to a sequence of observations, enabling the classification of texts depending on the class of the previous texts. Markov Models can be used for designing a part-of-speech tagger. Part-of-speech tagging allows assigning words to their part-of-speech in a sentence, allowing for the comprehension of a sentence without requiring complete understanding (see Manning and Schutze (2008)).

Combining the understanding of words and grammar, many NLP problems can be solved. In this booklet more advanced methods, of which some are based on the described basic methods, will be introduced.

## 1.3 Outline of the Booklet

This booklet circumstantiate modern approaches used for natural language processing, such as Deep Learning and Transfer Learning. Moreover, the resources that are available for the training of NLP tasks will be investigated and a use-case where NLP will be applied for generation of natural language will be shown.

For the analysis and comprehension of human language, NLP programs need to extract information from words and sentences. As neural networks and other machine learning algorithms require a numeric input for training, word embeddings, using dense vector representations for words, are applied. Those are usually learned by neural networks with multiple hidden layers, deep neural networks. In order to solve easy tasks, simple structured neural networks

can be applied. In order to overcome the limitations of those simple structures, recurrent and convolutional neural networks are utilized. Thereby, recurrent neural networks are used for models that can learn sequences without pre-defined optimal fixed dimensions while convolutional neural networks are applied for sentence classification. Chapter 2[1] of the booklet gives a short introduction to Deep Learning in NLP. The Foundations and Applications of Modern NLP will be described in chapter 3[2]. In chapter 4[3] and 5[4] recurrent neural networks and convolutional neural networks and their applications in NLP will be explained and discussed.

Transfer learning is an alternative to learning models for every task or domain. Here, existing labeled data of related tasks or domains can be used for training a model and applying it onto the task or domain of interest. The advantage of this approach is that there is no need for a long training in the target domain, and that time for training of the model can be saved, while still resulting in a (mostly) better performance. A concept used in transfer learning is Attention, which enables the decoder to attend to the entire input sequence, or Self-Attention, which allows a transformer model to process all input words at once and model the relationships between all words in a sentence, which renders fast modelling of long-range dependencies in a sentence possible. The concepts of transfer learning will be briefly introduced in chapter 6[5] of the booklet. Chapter 7[6] will describe transfer learning and LSTMs by presenting the models ELMo, ULMFiT, and GPT. Chapter 8[7] will illustrate the concepts of Attention and Self-Attention for NLP in detail. In chapter 9[8], transfer learning is combined with Self-Attention, introducing the models BERT, GTP2, and XLNet.

For NLP modelling, resources are needed. In order to find the best model for a task, benchmarks can be used. For comparing different models within a benchmark experiment, metrics such as exact match, F score, perplexity, or bilingual evaluation understudy, or accuracy, are required. Chapter 10[9] of the booklet provides a brief introduction to the resources for NLP and the manner in which they are used. Chapter 11[10] will explain the different metrics, give an insight into the benchmark datasets SQuAD, CoQa, GLUE and SuperGLUE, AQuA-Rat, SNLI, and LAMBADA as well as pre-trained models and databases where resources can be found, such as "Papers with Code" and "The Big Bad NLP Database".

[1] Chapter%20Introduction:%20Deep%20Learning%20for%20NLP
[2] Foundations/Applications%20of%20Modern%20NLP
[3] Recurrent%20neural%20networks%20and%20their%20applications%20in%20NLP
[4] Convolutional%20neural%20networks%20and%20their%20applications%20in%20NLP
[5] Introduction:%20Transfer%20Learning%20for%20NLP
[6] Transfer%20Learning%20for%20NLP%20I
[7] Attention%20and%20Self-Attention%20for%20NLP
[8] Transfer%20Learning%20for%20NLP%20II
[9] Introduction:%20Resources%20for%20NLP
[10] Resources%20and%20Benchmarks%20for%20NLP

In the last[11] chapter of the booklet, the generative NLP process Natural Language Generation, thus the generation of understandable text in a human language, is presented. Therefore, different algorithms will be described and chatbots as well as image captioning will be shown for illustrating the possibilities of application.

This introduction to the various methods in NLP functions as the foundation for the following deliberations. The individual chapters of the booklet will present modern methods in NLP and provide a more detailled discussion of the potential as well as the limitations along with various examples.

[11] Natural%20Language%20Generation

# 2

# *Introduction: Deep Learning for NLP*

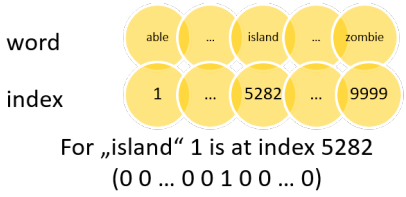
*Authors: Viktoria Szabo, Marianna Plesiak, Rui Yang*

*Supervisor: Prof. Dr. Christian Heumann*

## 2.1 Word Embeddings and Neural Network Language Models

In natural language processing computers try to analyze and understand human language for the purpose of performing useful tasks. Therefore, they extract relevant information from words and sentences. But how exactly are they doing this? After the first wave of rationalist approaches with handwritten rules didn't work out too well, neural networks were introduced to find those rules by themselves (see Bengio et al. (2003)). But neural networks and other machine learning algorithms cannot handle non-numeric input, so we have to find a way to convert the text we want to analyze into numbers. There are a lot of possibilities to do that. Two simple approaches would be labeling each word with a number (One-Hot Encoding, figure 2.1) or counting the frequency of words in different text fragments (Bag-of-Words, figure 2.1). Both methods result in high-dimensional, sparse (mostly zero) data. And there is another major drawback using such kind of data as input. It does not convey any similarities between words. The word "cat" would be as similar to the word "tiger" as to "car". That means the model cannot reuse information it already learned about cats for the much rarer word tiger. This will usually lead to poor model performance and is called a lack of generalization power.

The solution to this problem is word embedding. Word embeddings use dense vector representations for words. That means they map each word to a continuous vector with n dimensions. The distance in the vector space denotes semantic (dis)similarity. These word embeddings are usually learned by neural networks, either within the final model in an additional layer or in its own model. Once learned they can be reused across different tasks. Practically all NLP projects these days build upon word embeddings, since they have a lot of advantages compared to the aforementioned representations. The basic




Document 1
What time is it?

Document 2
The cat is black.

| Term | what | time | is | it | the | cat | black |
|---|---|---|---|---|---|---|---|
| Doc 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| Doc 2 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

**FIGURE 2.1:** One-Hot Encoding on the left and Bag-of-Words on the right. Source: Own figure.

idea behind learning word embeddings is the so called “distributional hypothesis” (see Harris (1954)). It states that words that occur in the same contexts tend to have similar meanings. The two best known approaches for calculating word embeddings are Word2vec from Mikolov et al. (2013c) and GloVE from Pennington et al. (2014). The Word2vec models (Continous Bag-Of-Words (CBOW) and Skip-gram) try to predict a target word given his context or context words given a target word using a simple feed-forward neural network. In contrast to these models GloVe not only uses the local context windows, but also incorporates global word co-occurrence counts. As mentioned, a lot of approaches use neural networks to learn word embeddings. A simple feed-forward network with fully connected layers for learning such embeddings while predicting the next word for a given context is shown in figure 2.2. In this example the word embeddings are first learnt in a projection layer and are then used in two hidden layers to model the probability distribution over all words in the vocabulary. With this distribution one can predict the target word. This simple structure can be good enough for some tasks but it also has a lot of limitations. Therefore, recurrent and convolutional networks are used to overcome the limitations of a simple neural network.

## 2.2 Recurrent Neural Networks

The main drawback of feedforward neural networks is that they assume a fixed length of input and output vectors which is known in advance. But for many natural language problems such as machine translation and speech recognition it is impossible to define optimal fixed dimensions a-priori. Other models that map a sequence of words to another sequence of words are needed (Sutskever et al., 2014). Recurrent neural networks or RNNs are a special family of neural networks which were explicitely developed for modeling sequential data like text. RNNs process a sequence of words or letters $x^{(1)}, ..., x^{(t)}$ by going through its elements one by one and capturing information based on the pre-

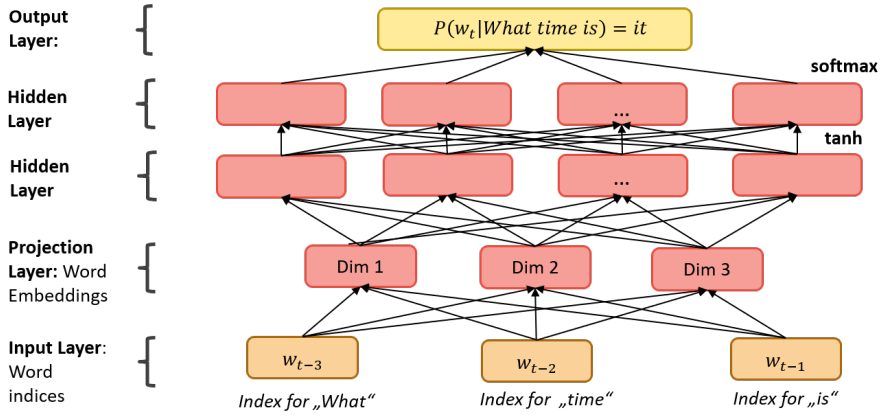



**FIGURE 2.2:** Feed-forward Neural Network. Source: Own figure based on Bengio et al. 2013.

vious elements. This information is stored in hidden states $h^{(t)}$ as the network memory. Core idea is rather simple: we start with a zero vector as a hidden state (because there is no memory yet), process the current state at time $t$ as well as the output from the previous hidden state, and give the result as an input to the next iteration (Goodfellow et al., 2016).

Basically, a simple RNN is a for-loop that reuses the values which are calculated in the previous iteration (Chollet, 2018). An unfolded computational graph (figure 2.3) can display the structure of a classical RNN. The gray square on the left represents a delay of one time step and the arrows on the right express the flow of information in time (Goodfellow et al., 2016).

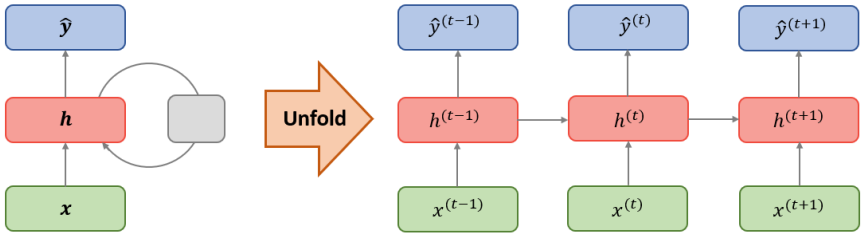



**FIGURE 2.3:** Right: Circuit diagram (left) and unfolded computational graph (right) of a simple RNN. Source: Own figure.

One particular reason why recurrent networks have become such a powerful technique in processing sequential data is parameter sharing. Weight matrices remain the same through the loop and they are used repeatedly, which makes

RNNs extremely convenient to work with sequential data because the model size does not grow for longer inputs. Parameter sharing allows application of models to inputs of different length and enables generalization across different positions in real time (Goodfellow et al., 2016).

As each part of the output is a function of the previous parts of the output, backpropagation for the RNNs requires recursive computations of the gradient. The so-called backpropagation through time or BPTT is rather simple in theory and allows for the RNNs to access information from many previous steps (Boden, 2002). In practice though, RNNs in their simple form are subject to two big problems: exploding and vanishing gradients. As the gradients are computed recursively, they may become either very small or very large, which leads to a complete loss of information about long-term dependencies. To avoid these problems, gated RNNs were developed and accumulation of information about specific features over a long duration became possible. The two most popular types of gated RNNs, which are widely used in modern NLP, are Long Short-Term Memory models (LSTMs, presented by Hochreiter and Schmidhuber (1997)) and Gated Recurrent Units (GRUs, presented by Cho et al. (2014)).

Over last couple of years, various extentions of RNNs were developed which resulted in their wide application in different fields of NLP. Encoder-Decoder architectures aim to map input sequences to output sequences of different length and therefore are often applied in machine translation and question answering (Sutskever et al., 2014). Bidirectional RNNs feed sequences in their original as well as reverse order because the prediction may depend on the future context, too (Schuster and Paliwal, 1997). Besides classical tasks as document classification and sentiment analysis, more complicated challenges such as machine translation, part-of-speech tagging or speech recognition can be solved nowadays with the help of advanced versions of RNNs.

## 2.3 Convolutional Neural Networks

Throughout machine learning or deep learning algorithms, no one algorithm is only applicable to a certain field. Most algorithms that have achieved significant results in a certain field can still achieve very good results in other fields after slight modification. We know that convolutional neural networks (CNN) are widely used in computer vision. For instance, a remarkable CNN model called AlexNet achieved a top-5 error of 15.3% in the ImageNet 2012 Challenge on 30 September 2012 (see Krizhevsky et al. (2012)). Subsequently, a majority of models submitted by ImageNet teams from around 2014 are based on CNN. After the convolutional neural network achieved great results in the

field of images, some researchers began to explore convolutional neural networks in the field of natural language processing (NLP). Early research was restricted to sentence classification tasks, CNN-based models have achieved very significant effects as well, which also shows that CNN is applicable to some problems in the field of NLP. Similarly, as mentioned before, one of the most common deep learning models in NLP is the recurrent neural network (RNN), which is a kind of sequence learning model and this model is also widely applied in the field of speech processing. In fact, some researchers have tried to implement RNN models in the field of image processing, such as (Visin et al. (2015)). It can be seen that the application of CNN or RNN is not restricted to a specific field.

As the Word2vec algorithm from Mikolov et al. (2013c) and the GloVe algorithm from Pennington et al. (2014) for calculating word embeddings became more and more popular, applying this technique as a model input has become one of the most common text processing methods. Simultaneously, significant effectiveness of CNN in the field of computer vision has been proven. As a result, utilizing CNN to word embedding matrices and automatically extract features to handle NLP tasks appeared inevitable.

The following figure 2.4 illustrates a basic structure of CNN, which is composed of multiple layers. Many of these layers are described and developed with some technical detail in later chapters of this paper.

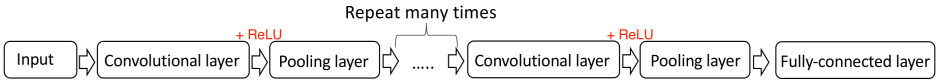



**FIGURE 2.4:** Basic structure of CNN. Source: Own figure.

It is obvious that neural networks consist of a group of multiple neurons (or perceptron) at each layer, which uses to simulate the structure and behavior of biological nervous systems, and each neuron can be considered as logistic regression.

The structure of CNN is different compared with traditional neural networks as illustrated in figure 2.5. In traditional neural networks structure, the connections between neurons are fully connected. To be more specific, all of the neurons in the layer $m-1$ are connected to each neuron in the layer $m$, but CNN sets up spatially-local correlation by performing a local connectivity pattern between neurons of neighboring layers, which means that the neurons in the layer $m-1$ are partially connected to the neurons in the layer $m$. In addition to this, the left picture presents a schematic diagram of fully-connected architecture. It can be seen from the figure that there are many edges between neurons in the previous layer to the next layer, and each edge has parameters. The right side is a local connection, which shows that there are relatively few

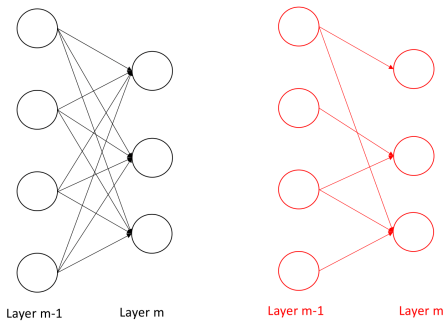



**FIGURE 2.5:** Comparison between the fully-connected and partial connected architecture. Source: Own figure.

edges compared with fully-connected architecture and the number of visible parameters has significantly decreased.

A detailed description of CNN will be presented in the later chapters and the basic architecture of if will be further explored. Subsection 5.1 gives an overview of CNN model depends upon (Kim (2014)). At its foundation, it is also necessary to explain various connected layers, including the convolutional layer, pooling layer, and so on. In 5.2 and later subsections, some practical applications of CNN in the field of NLP will be further explored, and these applications are based on different CNN architecture at diverse level, for example, exploring the model performance at character-level on text classification research (see Zhang et al. (2015)) and based on multiple data sets to detect the Very Deep Convolutional Networks (VD-CNN) for text classification (see Schwenk et al. (2017)).

# 3

# *Foundations/Applications of Modern NLP*

*Authors: Viktoria Szabo*

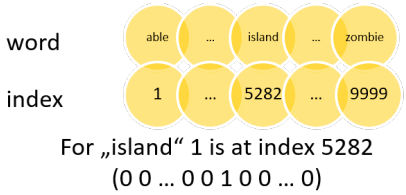
*Supervisor: Christian Heumann*

Word embeddings can be seen as the beginning of modern natural language processing. They are widely used in every kind of NLP task. One of the advantages is that one can download and use pretrained word embeddings. With this, it is possible to save a lot of time for training the final model. But if the task is not a standard one it is usually better to train own embeddings to get a better model performance for the specific task. In the following the evolution from sparse representations of words to dense word embeddings will be outlined in the first part. After that the calculation methods for word embeddings within a neural network language model and with word2vec and GloVe will be described. The third part shows how to improve the model performance regardless of the chosen model class based on hyperparameter tuning and system design choices and explains some model expansion to tackle problems of the aforementioned methods. The evaluation of word embeddings on different tasks and datasets is another topic which will be covered in the fourth part of this chapter. Finally some resources to download pretrained word embeddings will be presented.

## 3.1 The Evolution of Word Embeddings

Since computers work with numeric representations, converting the text and sentences to be analyzed into numbers is unavoidable. One-Hot Encoding and Bag-of-Words (BOW) are two simple approaches to how this could be accomplished. These methods are usually used as input for calculating more elaborate word representations called word embeddings.
The **One-Hot Encoding** labels each word in the vocabulary with an index. Let $n$ be size of the vocabulary, then each word is represented by a vector with dimension $n$. Every vector entry is zero except for the one corresponding to its index, which is set to 1. A sentence is represented as a matrix of shape $(n \times n)$ where $n$ is the number of unique words in the sentence or a document.

In figure 3.1 an example for a one-hot encoded word is shown on the left side. A more elaborate approach compared to the first one is called **Bag-of-Words (BOW)** and belongs to the count-based approaches. This approach counts the occurrences and co-occurrences of all distinct words in a document or a text chunk. Each text chunk is then represented by a row in a matrix, where the columns are the words. That means that, compared to the One-Hot Encoding, this approach already incorporates some context information in sentences and text chunks. An example for this kind of representation can be seen on the right side in figure 3.1.




Document 1: What time is it?

Document 2: The cat is black.

| Term | what | time | is | it | the | cat | black |
|---|---|---|---|---|---|---|---|
| Doc 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| Doc 2 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

**FIGURE 3.1:** One-Hot Encoding on the left and Bag-of-Words on the right. Source: Own figure.

These approaches definitely have some **positive points** about them. They are very simple to construct, robust to changes, and it was observed that simple models trained on large amounts of data outperform complex systems trained on less data. Bag-of-Words is especially useful if the number of distinct words is small and the sequence of the words doesn't play a key role, like in sentiment analysis. Without calculating word embeddings on top of them, these approaches should only be used if there is a small number of distinct words in the document, the words are not meaningfully correlated and there is a lot of data to learn from.
Nevertheless, the **problems** that arise from these approaches usually outweigh the positive points. The most obvious one is that these approaches lead to very sparse input vectors, that means large vectors with relatively few non-zero values. Many machine learning models won't work well with high dimensional and sparse features (Goldberg (2016)). Neural networks in particular struggle with this type of data. And with growing vocabulary the feature size vectors also increases by the same length. So, the dimensionality of these approaches is the same as the number of different words in your text. That means estimating more parameters and therefore using exponentially more data is required to build a reasonably generalizable model. This is known as the curse of dimensionality. But these problems can be solved with dimensionality reduction methods such as Principal Component Analysis or feature selection models where less informative context words, such as *the* and *a* are dropped.
The major drawback of these methods is that there is no notion of similarity between words. That means words like *cat* and *tiger* are represented as

similar as *cat* and *car*. If the words *cat* and *tiger* would be represented as similar words one could use the information won from the more frequent word "cat" for sentences in which the less frequent word *tiger* appears. If the word embedding for *tiger* is similar to that of *cat* the network model can take a similar path instead of having to learn how to handle it completely anew.

To overcome these problems **word embeddings** were introduced. Word embeddings use continuous vectors to represent each word in a vocabulary. These vectors have $n$ dimensions, usually between 100 and 500, which represent different aspects of the word. With this approach, semantic similarity can be maintained in the representation and generalization may be achieved. Through these vectors, the words are mapped to a continuous vector space, called a semantic space, where semantically similar words occur close to each other, while more dissimilar words are far from each other. Figure 3.2 shows a simple example to convey the idea behind this approach. In this fictional example the words are represented by a two-dimensional vector, which represents the cuteness and scariness of the creatures.

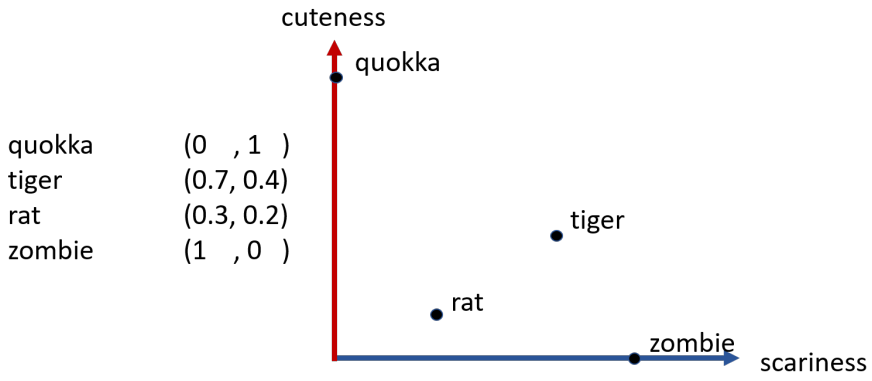



**FIGURE 3.2:** Example for word embeddings with two dimensions. Source: Own figure

If the goal is to represent higher dimensional word vectors one could use dimension reduction methods such as principal component analysis (PCA) to break down the number of dimensions into two or three and then plot the words. There is an example of this for selected country names and their capitals in figure 3.3.The country names all have negative values on the x-axis and the capitals all have positive values on the x-axis. Furthermore, the countries have y-axis values similar to their corresponding capitals.

With such word vectors even algebraic computations become possible as shown in Mikolov et al. (2013d). For example, $vector(King) - vector(Man) + vector(Woman)$ results in a vector that is closest to the vector representa-

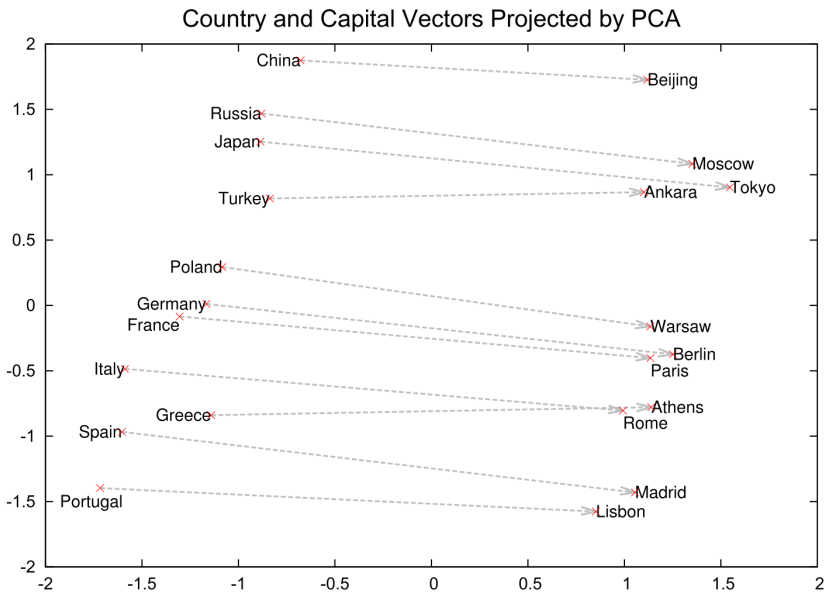



**FIGURE 3.3:** Two-dimensional PCA projection of 1000-dimensional word vectors of countries and their capital cities. Source: @Mikolov.2013c

tion of the word *Queen*. Another possibility to use word embeddings vectors is translation between languages. Mikolov et al. (2013b) showed that they can find word translations by comparing vectors generated from different languages. By searching for a translation one can use the word vector from the source language and search for the closest vector in the target language vector space, this word can then be used as a translation. The reason this works is that if a word vector from one language is similar to the word vector of the other language, this word is used in a similar context. This method can be used to infer missing dictionary entries. An example for this method depicted in figure 3.4. In figure 3.4 the vectors for numbers and animals are depicted on the left side and the same words are depicted on the right side. It can be seen that the vectors for the correct translation align in similar geometric spaces. Again, two-dimensional representation was achieved by using dimension reduction methods.

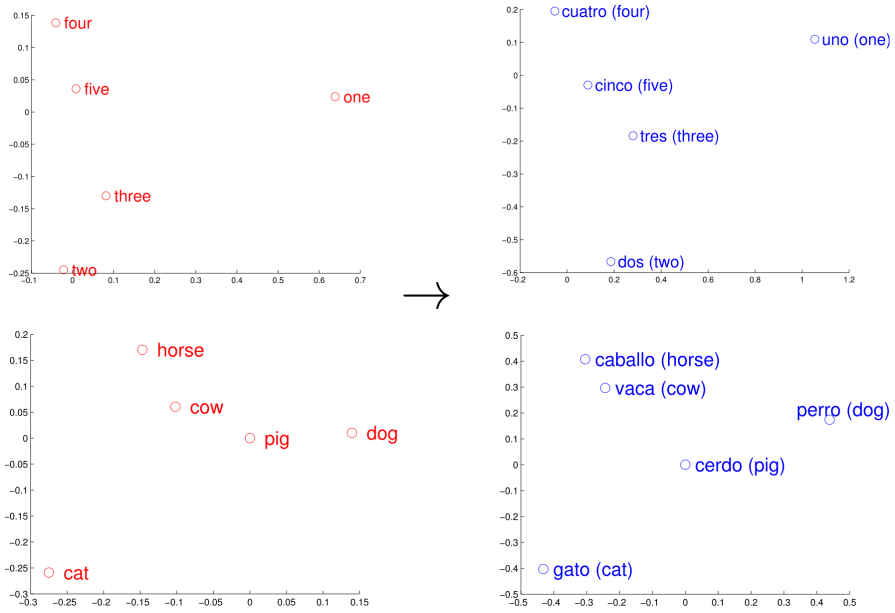



**FIGURE 3.4:** Distributed word vector representations of numbers and animals in English (left) and Spanish (right). Source: @mikolov2013exploiting

## 3.2 Methods to Obtain Word Embeddings

The basic idea behind learning word embeddings is the so called *distributional hypothesis* (Harris (1954)). It states that words that occur in the same contexts tend to have similar meanings. For instance, the words *car* and *truck* tend to have similar semantics as they appear in similar contexts, e.g., with words such as *road*, *traffic*, *transportation*, *engine*, and *wheel*. Hence machine learning and deep learning algorithms can find representations by themselves by evaluating the context in which a word occurs. Words that are used in similar contexts will be given similar representations. This is usually done as an unsupervised or self-supervised procedure, which is a big advantage. That means word embeddings can be thought of as unsupervised feature extractors for words. However, the methods to find such similarities in the context of words vary. Finding word representations started out with more traditional count-based techniques, which collected word statistics like occurrence and co-occurrence frequencies as seen above with BOW. But these representations often require some sort of dimensionality reduction. Later, when neural networks were introduced into NLP, the so-called predictive techniques, mainly popularized after 2013 with the introduction of word2vec, supplanted the tra-

ditional count-based word representations.These models learn what is called *dense representations* of words, since they directly learn low-dimensional word representations, without needing the additional dimensionality reduction step. In the following an introduction to the best-known predictive approaches to model word embeddings will be given. First, neural network language models, where word embeddings are learnt as a part of the final language model will be discussed. The description of the two popular algorithms word2vec and GloVe, which learn word embeddings in a pre-step before the actual statistical language model, follow afterwards.

### 3.2.1 Feedforward Neural Network Language Model (NNLM)

Bengio et al. (2003) were the first to propose learning word embeddings within a statistical **neural network language model (NNLM)**. The goal of the NNLM model of Bengio et al. (2003) is to predict the next word based on a sequence of preceding words. Using a simple feedforward neural network, the model first learns the word embeddings and in a second step the probability function for word sequences. This way, one obtains not only the model itself, but also the learned word representations, which can be used as input for other, potentially unrelated, tasks.
The proposed neural network architecture has an input layer with one-hot encoded word inputs, a linear projection layer for the word embeddings, and a hidden layer with a hyperbolic tangent function, where most of the computation is done, followed by a softmax classifier output layer. The output of the model is a vector of the probability distribution over all words given a specific context. That means a vector with probability scores for each word of the vocabulary. The $i$-th element of the output vector is the probability estimation $P(w_t = i|context)$. The softmax classifier is used to guarantee positive probabilities summing to one. It computes the following function:

$$\widehat{P}(w_t|w_{t-1}, ..., w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}$$

The $y_i$ are the unnormalized log-probabilities for each output word $i$, which were computed in the previous layer. The model architecture proposed in Bengio et al. (2003) is depicted in figure 3.5.
When training a neural network, one has to define a loss function $L(\hat{y}, y)$ stating the loss of predicting $\hat{y}$ when the true output is $y$. In the NNLM literature, a cross-entropy loss is very common (see Goldberg (2016)). The $\hat{y}$ is the network output vector, which was transformed by the softmax classifier and represents the conditional distribution. The $y$ is usually either a one-hot vector for the correct output word or a vector representing the true multinomial probability distribution over the vocabulary given the specific context. Then the parameter $\phi$ of the neural network (for example the weights for the embedding vectors) is iteratively changed in order to minimize the loss $L$ over the training examples.

This is usually done with the **stochastic gradient descent (SGD)** optimizer where the gradient is obtained via **backpropagation**. The gradient descent optimizer tries to find the direction of the strongest descent via partial derivatives and updates the parameter $\phi$ accordingly. The learning rate $\varepsilon$ defines the size of the step in this direction (see Goldberg (2016)).
In Bengio et al. (2003) a gradient ascent optimizer is used, which performs the following iterative update after presenting the t-th word of the training corpus:

$$\theta \leftarrow \theta + \varepsilon \frac{\partial log\widehat{P}(w_t|w_{t-1}, ..., w_{t-n+1})}{\partial \theta}$$

The method by which parameter adjustments are made during training so they can be optimized is called backpropagation. Backpropagation essentially consists of six steps:

1. Initialization of the parameter of the network
2. Calculation of $y(x_i)$ for the inputs $x_i$
3. Determining the cost of the inputs $x_i$
4. Calculation of the partial derivatives of the loss for each parameter
5. Update the parameter in the network using the partial derivatives calculated in step 4
6. Return to step 2 and continue the procedure until the partial derivatives of the loss approach zero

### 3.2.2 Word2Vec

In 2013 Mikolov et al. (2013a) proposed the two word2vec algorithms which led to a wave in NLP that popularized word embeddings. In contrast to the NNLM model above, the word2vec algorithms are not used for a statistical language modeling goal, but rather to learn the word embeddings themselves. The two word2vec algorithms named Continuous Bag-of-Words (CBOW) and Continuous Skip-Gram use shallow neural networks with an input layer, a projection layer, and an output layer. This means compared to the previously explained feedforward NNLM, the non-linear hidden layer is removed.
The general idea behind CBOW is to predict the focus word based on a window of context words. The order of context words does not influence the prediction, thus the name Bag-of-Words. In contrast, Skip-Gram tries to predict the context words given a source word. This is done while adjusting the initial weights during training so that a loss function is reduced.
In the **CBOW** architecture the $N$ input (context) words are each one-hot encoded vectors of size $V$, where $V$ is the size of the vocabulary. Compared to the NNLM model CBOW uses both previous and following words as context instead of only the previous words. The projection layer is a standard fully connected (dense) layer which has the dimensionality $1 \times D$, where $D$ is the size of the dimensions for the word embeddings. The projection layer is shared

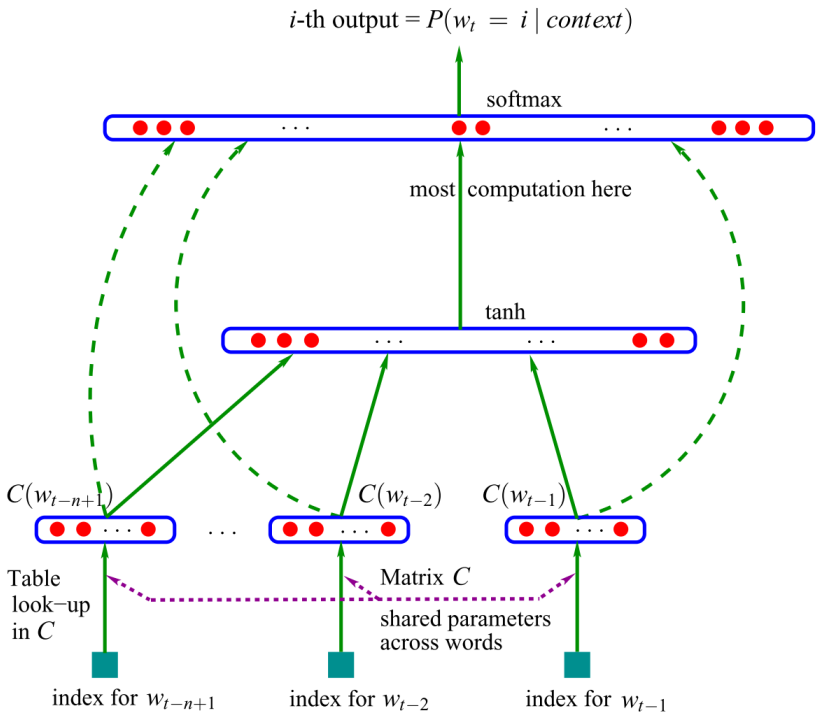



**FIGURE 3.5:** Architecture for NNLM proposed by @Bengio.2003 . Source: @Bengio.2003

for all words. That means all words get projected into the same position in a linear manner, where the vectors are averaged. The output layer outputs probabilities for the target words from the vocabulary and has a dimensionality of $V$. That means the output is a probability distribution over all words of the vocabulary as in the NNLM model, where the prediction is the word with the highest probability. But instead of using a standard softmax classifier as in the NNLM model the authors propose to use a log-linear hierarchical softmax classifier for the calculation of the probabilities. The model architecture is shown in figure 3.6. The **continuous Skip-gram** architecture also uses a log-linear hierarchical softmax classifier with a continuous projection layer, but the input is only one source word, and the output layer consists of as many probability vectors over all words as the chosen number of context words. Also, since the more distant words are usually less related to the source word, the skip-gram model weighs nearby context words more heavily than more distant context words by sampling less from those words in the training

examples. The model architecture for skip-gram can be found on the right side of figure 3.6.

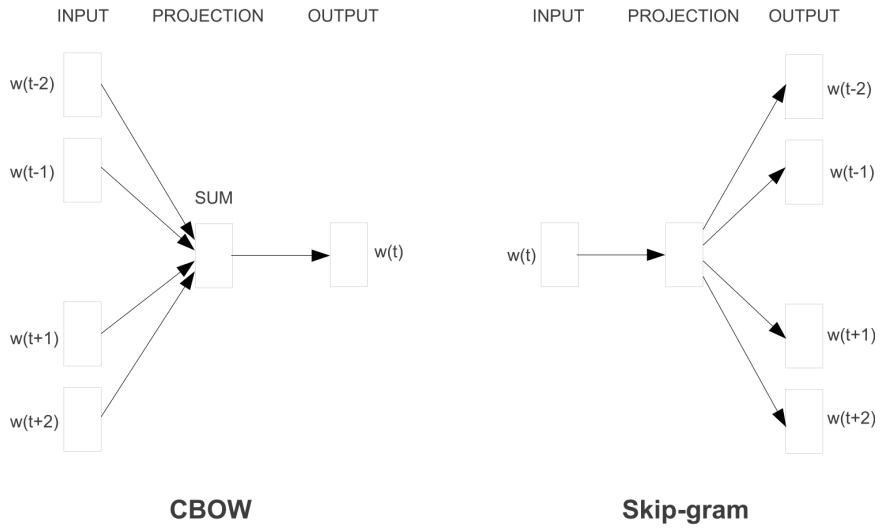



**FIGURE 3.6:** Learning word embeddings with the model architecture of CBOW and Skip-Gram. Source: @mikolov2013efficient

As said before the word2vec models use hierarchical softmax, where the vocabulary is represented as a Huffman binary tree, instead of the standard softmax classifier explained in the section before. With hierarchical softmax the size of the output vector can be reduced from the vocabulary size $V$ to the logarithm to base 2 of $V$, which is a dramatic change in computational complexity and number of operations needed for the algorithm. Further explanations for this method can be found in Morin and Bengio (2005). For both models Mikolov et al. (2013a) use gradient descent optimization and backpropagation as described in the previous chapter.
Mikolov et al. (2013a) show that their word2vec algorithms outperform a lot of other standard NNLM models. CBOW is faster while skip-gram does a better job for infrequent words. Skip-gram works well with small amounts of training data and has good representations for words that are considered rare, whereas CBOW trains several times faster and has slightly better accuracy for frequent words.

### 3.2.3 GloVe

GloVe stands for **Glo**bal **Ve**ctor word representation, which emphasizes the global character of this model. Unlike the previously described algorithms

like word2vec, GloVe not only relies on local context information but also incorporates global co-occurrence statistics. Instead of extracting the embeddings from a neural network that is designed to perform a task like predicting neighboring words (CBOW) or predicting the focus word (Skip-Gram), the embeddings are optimized directly, so that the dot product of two word vectors is equal to the log of the number of times the two words will occur near each other. The model builds on the possibility to derive semantic relationships between words from the co-occurrence matrix and that the ratio of co-occurrence probabilities of two words with a third word is more indicative of semantic association than a direct co-occurrence probability (see Pennington et al. (2014)).

Let $P_{ij} = P(j|i) = X_{ij}/X_i$ be the probability that word $j$ appears in the context of word $i$. Figure 3.7 shows an example with the words *ice* and *steam*. The relationship of these words can be examined by studying the ratio of their co-occurrence probabilities with other words $k$.

For words like *solid* which are related to *ice* but not *steam* the ratio is large (>1), while for words like *gas*, which is related to *steam* but not to *ice*, the ratio is small (<1). For words like *water* or *fashion*, which are either related to both of the words or to none the ratios are close to 1. It is evident that the comparison of co-occurrence probabilities with a third word ($P_{ik}/P_{jk}$) is more indicative of the semantic meanings of words $i$ and $j$ and is better at identifying words (*solid* and *gas*) that distinguish $i$ and $j$ and words that do not (*water* and *fashion*) than the raw probabilities.

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

**FIGURE 3.7:** Co-occurrence probabilities for target words ice and steam with selected context words. Source: @Pennington.2014

Pennington et al. (2014) tried to incorporate the ratio $P_{ik}/P_{jk}$ into computing word embeddings. They propose an optimization problem which aims at fulfilling the following objective:

$$w_i^T w_k + b_i + b_k = log(X_{ik})$$

Where $b_i$ and $b_k$ are bias terms for word $w_i$ and probe word $w_k$ and $X_{ik}$ is the number of times $w_i$ co-occurs with $w_k$. Fulfilling this objective minimizes the difference between the dot product of $w_i$ and $w_k$ and the logarithm of their number of co-occurrences. In other words, the optimization results in the construction of vectors $w_i$ and $w_k$ whose dot product gives a good estimate of their transformed co-occurrence counts. To solve this optimization problem, they reformulate the equation as a least squares problem and introduce a

weighting function, since rare co-occurrences add both noise to the model and less information than more frequent co-occurrences.

## 3.3 Hyperparameter Tuning and System Design Choices

Once adapted across methods, hyperparameter tuning significantly improves performance in every task. Levy et al. (2015) showed that in a lot of cases, changing the setting of a single hyperparameter could yield a greater increase in performance than switching to a better algorithm or training on a larger corpus. They conducted a series of experiments where they assessed the contributions of diverse hyperparameters. They also show that when all methods are allowed to tune a similar set of hyperparameters, their performance is largely comparable. However they also found that choosing the wrong hyperparameter settings can actually degrade performance of a model. That is why tuning the hyperparameter fitting the specific context is very important. Depending on the model one chooses there are a lot of hyperparameters available for tuning. These are parameters like the number of epochs, batch-size, learning rate, embedding size, window size, corpus size et cetera. In the following the focus will lie on hyperparameters which are frequently discussed. Furthermore, some system design and setup choices will be described which will tackle some of the problems posed by the algorithms mentioned above.

**Word embedding size**
The question of how many embedding dimensions should be used is mostly answered empirically. It depends on the task, computing capacity and vocabulary. The trade-off is between accuracy and computational concerns. More dimensions could potentially increase the accuracy of the representations; since the vectors can capture more aspects of the word. But more dimensions also mean higher computing time and effort. In practice word embedding vectors with dimensions around 50 to 300 are usually used as a rule of thumb (see Goldberg (2016)).
In contrast, Patel and Bhattacharyya (2017) found that the dimension size should be chosen based on some text corpus statistics. One can calculate a lower bound from the number of pairwise equidistant words of the corpus vocabulary. Choosing a dimension size below this bound results in a loss of quality of learned word embeddings. This result was tested empirically for the skip-gram algorithm (see Patel and Bhattacharyya (2017)).
Pennington et al. (2014) compare performance of the GloVe model for embedding sizes from 1 to 600 for different evaluation tasks (semantic, syntactic and overall). They found that after around 200 dimensions the performance increase begins to stagnate.

**Context Window**
In traditional approaches the context window around the focus word is constant-sized and unweighted. For example, if the symmetrical context size is 5 then the 5 words before the focus word and 5 words after the focus words are the context window. It is also possible to use an asymmetric context window, which for example only uses words that appear before the focus word. A window size of 5 is commonly used to capture broad topic/domain information like what other words are used in related discussions (i.e. *dog*, *bark* and *leash* will be grouped together, as well as *walked*, *run* and *walking*), whereas smaller windows contain more specific information about the focus word and produce more functional and syntactic similarities (i.e. *Poodle*, *Pitbull*, *Rottweiler*, or *walking*, *running*, *approaching*) (see Goldberg and Levy (2014), Goldberg (2016)).
Since words which appear closer to the focus word are usually more indicative of its meaning it is possible to give the context words weights according to their distance from the focus word. Both word2vec and GloVe use such weighting schemes. GloVe's implementation weights contexts using the harmonic function, e.g. a context word three tokens away will have 1/3 as a weight. Word2vec's uses weightings of the distance from the focus word divided by the window size. For example, a size-5 window will weigh its contexts by $\frac{5}{5}$, $\frac{4}{5}$, $\frac{3}{5}$, $\frac{2}{5}$, $\frac{1}{5}$ (Levy et al. 2015).
A performance comparison for GloVe using different window sizes for different evaluation tasks (semantic, syntactic and overall) can be found in Pennington et al. (2014). Which shows that, depending on the task, larger performance gains can be expected from a larger window size. But for all three tasks the performance gains decrease after a window size of 4.

**Document Context**
Instead of using a few words as the context window one could consider all the other words that appear with the focus word in the same sentence, paragraph, or document. One can either consider this as using very large window sizes or, as in the doc2vec algorithm from Le and Mikolov (2014), add another embedding vector for a whole paragraph to the other context word vectors. These approaches will result in word vectors that capture topical similarity (words from the same topic, i.e. words that one would expect to appear in the same document, are likely to receive similar vectors). (Goldberg (2016); Le and Mikolov (2014))

**Subsampling of Frequent Words**
Mikolov et al. (2013c) proposed for their word2vec algorithms to use a subsample of the most frequent words. Very frequent words are often so-called stop-words, like *the* or *a*, which do not provide much information. Using fewer of these frequent words leads to a significant speedup and improves accuracy of the representations of less frequent words (see Mikolov et al. (2013c)). The method randomly removes words $w$ with a probability $p$ that occur more often than a certain threshold $t$. In Mikolov et al. (2013c) the probability $p$ is

defined as follows:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

where $f$ marks the word's frequency in the text corpus. In Mikolov et al. (2013c) the threshold $t$ is set to $10^{-5}$, but generally this parameter is open for tuning. In Mikolov et al. (2013c) this subsampling is done before processing the text corpus. This leads to an artificial enlargement of the context window size. One could perform the subsampling step without affecting the context window size, but Levy et al. (2015) found that it does not affect performance too much.

**Negative Sampling**

In their first paper Mikolov et al. (2013a) proposed using hierarchical softmax instead of the standard softmax function to speed up the calculation in the neural network. But later they published a new method called negative sampling, which is even more efficient in the calculation of word embeddings. The negative sampling approach is based on the skip-gram algorithm, but it optimizes a different objective. It maximizes a function of the product of word and context pairs $(w, c)$ that occur in the training data, and minimizes it for negative examples of word and context pairs $(w, c_n)$ that do not occur in the training corpus. The negative examples are created by drawing $k$ negative examples for each observed $(w, c)$ pair. $k$ can be tuned as a hyperparameter. Mikolov et al. (2013c) found that $k$ in the range 5-20 is useful for small training datasets; while for large datasets $k$ can be as small as 2-5 (see Mikolov et al. (2013c); Goldberg and Levy (2014)).

**Subword Information**

An individual word can convey a lot of information besides the general meaning of the word. Ignoring the internal structure of words can lead to a large loss of information, especially in morphologically rich languages like Finnish or Turkish. Furthermore, coping with completely unseen words not included in the training data, so-called *out-of-vocabulary* (OOV) words, is not possible if every word gets assigned a distinct new vector representation like in the previously presented models. One way to deal with these problems is to train character-based models instead of word-based models. But as Goldberg (2016) states: "working on the character level is very challenging, as the relationship between form (characters) and function (syntax, semantics) in language is quite loose. Restricting oneself to stay on the character level may be an unnecessarily hard constraint." A more promising approach to solve the problem of OOV words is to use subword information. In this context **fastText** was introduced in two papers 2016 and 2017 (see Joulin et al. (2016) and Bojanowski et al. (2017)). fastText builds upon the previously described continuous skip-gram model. But instead of learning vectors for words directly as done by skip-gram, fastText represents each word as an n-gram of characters. So, for example, take the word, *planning* with n=3, the fastText representation of this word is <pl, pla, lan, ann, nni, nin, ing, ng>, where the angular brackets

indicate the beginning and end of the word. This helps capture the meaning of shorter words inside longer words and allows the embeddings to understand suffixes and prefixes. In addition to these n-grams fastText learns embeddings to the whole word. Bojanowski et al. (2017) extracted all the n-grams for n greater or equal to 3 and smaller or equal to 6. They also state that "the optimal choice of length ranges depends on the considered task and language and should be tuned appropriately". In the end one word will be represented by the sum of the vector representations of its n-grams and the word itself. In the case of an unseen word (OOV), the corresponding embedding is induced by averaging the vector representations of its constituent character n-grams. Hence fastText performs well when having data with a large number of rare words.

**Phrase representation**
The models described in the previous section all focus on individual words as input. But there are many words which will only be meaningful in combination with other words, or which change meaning completely when paired up with another word. Therefore, there are phrases with meanings that are more than a simple composition of the meanings of their individual words. Often these are names like "New York" for a city or "Toronto Raptors" for a basketball team. Since the meaning is changed completely when evaluating the combination of these words one embedding must be learned for the whole phrase instead of using the embeddings of the individual words.
Mikolov et al. (2013c) proposed an approach to deal with such phrases. First, they use the following score to find them in the text corpus:

$$score(w_i, w_j) = \frac{count(w_i w_j) - \delta}{count(w_i) \times count(w_j)}$$

With this score they try to find words that appear frequently together, and infrequently in other contexts. The score compares the appearance of two words together to their appearance alone in the text corpus. The $\delta$ is used as a discounting coefficient and prevents too many phrases consisting of very infrequent words to be formed. When the score of a word pair is above a chosen threshold value it will be used as a phrase. They repeat this calculation 2 to 4 times with decreasing threshold value, allowing longer phrases that consist of several words to be formed.

## 3.4 Evaluation Methods

The use of different algorithms, hyperparameter or system choices need to be evaluated and compared in their performance to make a reasonable choice. Therefore, there are several tasks and datasets which are used to evaluate

word embeddings. In this chapter the two most common tasks and five corresponding datasets for each task will be presented. The dataset collection was done by Bakarov (2018), also refer to this paper for a more thorough examination of the evaluation tasks.

**Word Similarity Task**
The word similarity task tries to evaluate the distances between embedding vectors, which should represent their similarity, by comparing them to similarity scores given by humans. The more similar the embedding distance to the human score the better the embedding. The word vectors are evaluated by ranking the pairs according to their cosine similarities, and measuring the correlation (Spearman's  ) with the human ratings. These are five datasets used to evaluate word similarity ranked by their size:

1. SimVerb-3500, 3 500 pairs of verbs assessed by semantic similarity with a scale from 0 to 4 (Gerz et al. (2016)).
2. MEN, 3 000 pairs assessed by semantic relatedness with a discrete scale from 0 to 50 (Bruni et al. (2014)).
3. RW (acronym for Rare Word), 2 034 pairs of words with low occurrences (rare words) assessed by semantic similarity with a scale from 0 to 10 (Luong et al. (2013)).
4. SimLex-999, 999 pairs assessed with a strong respect to semantic similarity with a scale from 0 to 10 (Hill et al. (2015)).

**Word analogy task**
In the word analogy task word relations are predicted in the form *"a is to a∗ as b is to b∗"*, where b∗ is hidden, and must be guessed from the entire vocabulary. A popular example of an analogy is that *king* relates to *queen* as *man* relates to *woman*. These are five datasets used to evaluate word analogy ranked by their size:

1. WordRep, 118 292 623 analogy questions (4-word tuples) divided into 26 semantic classes (Gao et al. (2014)).
2. BATS (acronym for Bigger Analogy Test Set), 99 200 questions divided into 4 classes (inflectional morphology, derivational morphology, lexicographic semantics and encyclopedic semantics) and 10 smaller subclasses (Gladkova and Drozd (2016)).
3. Google Analogy (also called Semantic-Syntactic Word Relationship Dataset), 19 544 questions divided into 2 classes (morphological relations and semantic relations) and 10 smaller subclasses (8 869 semantic questions and 10 675 morphological questions) (Mikolov et al. (2013a)).
4. SemEval-2012, 10 014 questions divided into 10 semantic classes and 79 subclasses prepared for the SemEval-2017 Task 2 (Measuring Degrees of Relational Similarity) (Jurgens et al. (2012)).

5. MSR (acronym for Microsoft Research Syntactic Analogies), 8 000 questions divided into 16 morphological classes (Mikolov et al. (2013d)).

## 3.5 Outlook and Resources

The use of word embeddings furthered much development in the field of natural language processing. Still, there are problems word embeddings are often not suited to resolve. When calculating word embeddings, the word order is not taken into account. For some NLP tasks like sentiment analysis, this does not pose a problem. But for other tasks like translation, word order can not be ignored. Recurrent neural networks, which will be presented in the following chapter, are one of the tools to face this difficulty.
Furthermore, there are a lot of words with two or more different meanings. *Mouse* for example can be understood as an animal or as an operator for a computer. Humans naturally take the context into account, in which the word was used, to infer the meaning. The above described word embeddings are not able to do this. But ELMO, which will be discussed in chapter Chapter 7[1], uses contextualized embeddings to solve this problem.
Still there are two additional problems, which will not be addressed in this book. First of the word embeddings are mostly learned from text corpora from the internet, therefore they learn a lot of stereotypes that reflect everyday human culture. Another problem is that there are some domains and languages for which only little training data exists on the internet. The algorithms described above all use large amounts of training data to learn exact word embeddings. Solutions to these problems are still work in progress.
Last but not least some resources for downloading pre-calculated word embeddings will be presented. As stated before, if the task is rather common and the words used are from a general vocabulary, one could use pre-calculated word embeddings for the training of the language model. The first two links lead to websites, where word embeddings learned with GloVe and fastText can be downloaded. These were trained on different training data sources like *Wikipedia*, *Twitter* or *Common Crawl* text. GloVe embeddings can only be downloaded for english words, whereas fastText also offers word embeddings for 157 different languages. The last link leads to a website, which is maintained by the Language Technology Group at the University of Oslo and offers word embeddings for many different languages and models.

**Glove**:
https://nlp.stanford.edu/projects/glove/

[1] ./transfer-learning-for-nlp-i.html

**fastText**:
https://fasttext.cc/docs/en/english-vectors.html

**Different Models and different languages**:
http://vectors.nlpl.eu/repository/

# 4

# *Recurrent neural networks and their applications in NLP*

*Author: Marianna Plesiak*

*Supervisor: Prof. Dr. Christian Heumann*

## 4.1 Structure and Training of Simple RNNs

**R**ecurrent **n**eural **n**etworks (**RNNs**) enable to relax the condition of non-cyclical connections in the classical feedforward neural networks which were described in the previous chapter. This means, while simple multilayer perceptrons can only map from input to output vectors, RNNs allow the entire history of previous inputs to influence the network output. (Graves, 2013)

The first part of this chapter provides the structure definition of RNNs, presents the principles of their training and explains problems with backpropagation. The second part covers gated units, an improved way to calculate hidden states. The third part gives an overview of some extended versions of RNNs and their applications in NLP.

### 4.1.1 Network Structure and Forwardpropagation

An **unfolded computational graph** can visualize the repetitive structure of RNNs:

Each node is associated with a network layer at a particular time instance. Inputs $x^{(t)}$ must be encoded as numeric vectors, for instance word embeddings or one-hot encoded vectors, see the previous chapter. Recurrently connected vectors $h$ are called **hidden states** and represent the outputs of the hidden layer. At time $t$, a hidden state $h^{(t)}$ combines information from the previous hidden state $h^{(t-1)}$ as well as the new input $x^{(t)}$ and transmits it to the next hidden state. Obviously, such an architecture requires the initialization of $h^{(0)}$ since there is no memory at the very beginning of the sequence processing.

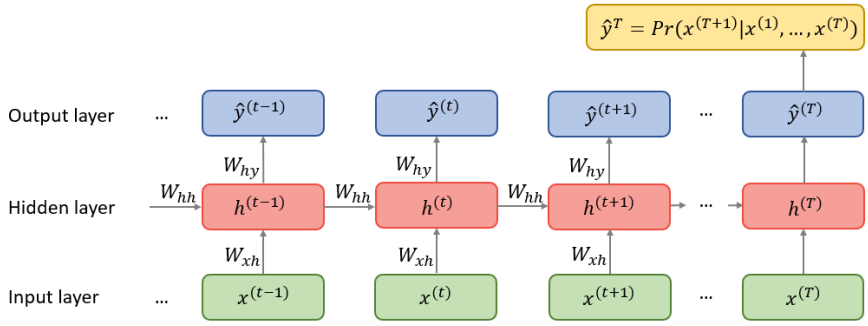



**FIGURE 4.1:** Unfolded computatinal graph of an RNN. Source: Own figure.

Given the hidden sequences, output vectors $\hat{y}^{(t)}$ are used to build the predictive distribution $Pr(x^{(t+1)}|y^{(t)})$ for the next input (Graves, 2013). Since the predictions are created at each time instance $t$, the total output has a shape $[\#\ time\_steps, \#\ output\_features]$. However, in some cases the whole history of output features is not necessary. For example, in sentiment analysis the last output of the loop is sufficient because it contains the entire information about the sequence. (Chollet, 2018)

The unfolded recurrence can be formalized as following:

$$h^{(t)} = g^{(t)}(x^{(t)}, x^{(t-1)}, ..., x^{(2)}, x^{(1)}) \tag{4.1}$$
$$= f(h^{(t-1)}, x^{(t)}|\theta) \tag{4.2}$$

After $t$ steps, the function $g^{(t)}$ takes into account the whole sequence $(x^{(t)}, x^{(t-1)}, ..., x^{(2)}, x^{(1)})$ and produces the hidden state $h^{(t)}$. Because of its cyclical structure, $g^{(t)}$ can be factorized into the repeated application of the same function $f$. This function can be considered to be a universal model with parameters $\theta$ which are shared across all time steps and generalized for all sequence lengths. This concept is called parameter sharing and is illustrated in the unfolded computational graph as a reuse of the same matrices $W_{xh}$, $W_{hh}$ and $W_{hy}$ through the entire network. (Goodfellow et al., 2016)

Considering a recurrent neural network with one hidden layer that is used to predict words or characters, the output should be discrete and the model maps input sequence to output sequence of the same length. Then, the forward propagation is computed by iterating the following equations:

$$h^{(t)} = f(a + W_{hh}h^{(t-1)} + W_{xh}x^{(t)}) \tag{4.3}$$
$$y^{(t)} = s(b + W_{hy}h^{(t)}) \tag{4.4}$$

where, according to Graves (2013), the parameters and functions denote the following:

- $x^{(t)}$, $h^{(t)}$ and $y^{(t)}$: input, hidden state and output at time step $t$ respectively;
- $f$: activation function of the hidden layer. Usually it is a saturating nonlinear function such as a sigmoid activation function ( Sutskever et al. (2014) and Mikolov et al. (2010));
- $W_{hh}$: weight matrix connecting recurrent connections between hidden states;
- $W_{xh}$: weight matrix connecting inputs to a hidden layer;
- $W_{hy}$: weight matrix connecting hidden states to outputs;
- $s$: output layer function. If the model is used to predict words, the softmax function is usually chosen as it returns valid probabilities over the possible outputs (Mikolov et al., 2010);
- $a$, $b$: input and output bias vectors.

Since inputs $x^{(t)}$ are usually encoded as one-hot-vectors, the dimension of a vector representing one word corresponds to the size of vocabulary. The size of a hidden layer must reflect the size of training data. The model training requires initialization of the initial state $h^{(0)}$ as well as the weight matrices, which are usually set to small random values (Mikolov et al., 2010). Since the model is used to compute the predictive distributions $Pr(x^{(t+1)}|y^{(t)})$ at each time instance $t$, the network distribution is denoted as following:

$$Pr(x) = \prod_t Pr(x^{(t+1)}|y^{(t)}) \tag{4.5}$$

and the total loss $\mathcal{L}(x)$, which must be minimized during the training, is simply the sum of losses over all time steps denoted as the negative log-likelihood of $Pr(x)$:

$$\mathcal{L}(x) = -\sum_t \log Pr(x^{(t+1)}|y^{(t)}) \tag{4.6}$$

(Graves, 2013)

### 4.1.2 Backpropagation

To train the model, one must calculate the gradients for the three weight matrices $W_{xh}$, $W_{hh}$ and $W_{hy}$. The algorithm differs from a regular backpropagation because a chain rule must be applied recursively, and the gradients

are summed up through the network (Boden, 2002). Using the notation $\mathcal{L}^{(t)}$ as the output at time $t$, one must first estimate single losses at each time step and then sum them up in order to obtain total loss $\mathcal{L}(x)$.

Chen (2016) provides a nice guide to backpropagation for a simple RNN in the meaning of equations (4.3) and (4.4). According to him, the gradient w.r.t. $W_{hy}$ for a single time step $t$ is calculated as follows:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{hy}} = \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial W_{hy}} \tag{4.7}$$

$$\tag{4.8}$$

Since $W_{hy}$ is shared across all time sequence, the total loss w.r.t. the weight matrix connecting hidden states to outputs is simply a sum of single losses:

$$\frac{\partial \mathcal{L}}{\partial W_{hy}} = \sum_t \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial W_{hy}} \tag{4.9}$$

$$\tag{4.10}$$

Similarly, derivation of the gradient w.r.t. $W_{hh}$ for a single time step is obtained as follows:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{hh}} = \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W_{hh}} \tag{4.11}$$

However, the last part $h^{(t)}$ also depends on $h^{(t-1)}$ and the gradient can be rewritten according to the **B**ack**p**ropagation **T**hrough **Time** algorithm (**BPTT**) starting from $t$ and going back to the initial hidden state at time step $k = 0$:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{hh}} = \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \frac{\partial h^{(t-1)}}{\partial W_{hh}} \tag{4.12}$$

$$= \sum_{k=0}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{hh}} \tag{4.13}$$

Single gradients are again aggregated to yield the overall loss w.r.t $W_{hh}$:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_t \sum_{k=0}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{hh}} \tag{4.14}$$

The derivation of gradients w.r.t. $W_{xh}$ is similar to those w.r.t $W_{hh}$ because both $h^{(t-1)}$ and $x^{(t)}$ contribute to $h^{(t)}$ at time step $t$. The derivative w.r.t. $W_{xh}$

for the whole sequence is then obtained by summing up all contributions from $t$ to 0 via backpropagation for all time steps:

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_{t} \sum_{k=0}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{xh}} \tag{4.15}$$

The derivatives w.r.t. the bias vectors $a$ and $b$ are calculated based on the same principles and thus are not shown here explicitly.

### 4.1.3 Vanishing and Exploding Gradients

In order to better understand the mathematical challenges of BPTT, one should consider equation (4.13), and in particular the factor $\frac{\partial h^{(t)}}{\partial h^{(k)}}$. Pascanu et al. (2013b) go into detail and show that it is a chain rule in itself and can be rewritten as a product $\frac{\partial h^{(t)}}{\partial h^{(t-1)}} \frac{\partial h^{(t-1)}}{\partial h^{(t-2)}} ... \frac{\partial h^{(2)}}{\partial h^{(1)}}$. Since one computes the derivative of a vector w.r.t another vector, the result will be a product of $t-k$ Jacobian matrices whose elements are the pointwise derivatives:

$$\frac{\partial h^{(t)}}{\partial h^{(k)}} = \prod_{t \geq\ i > k} \frac{\partial h^{(i)}}{\partial h^{(i-1)}} = \prod_{t \geq\ i > k} W_{hh}^{T} diag(f^{'}(h^{(i-1)})) \tag{4.16}$$

Thus, with small values in $W_{hh}$ and many matrix multiplications the norm of the gradient shrinks to zero exponentially fast with $t-k$ which results in a vanishing gradient problem and the loss of long term contributions. Exploding gradients refer to the opposite behaviour when the norm of the gradient increases largely and leads to a crash of the model. Pascanu et al. (2013b) give an overview of techniques for dealing with the exploding and vanishing gradients. Among other solutions they mention proper initialisation of weight matrices, sampling $W_{hh}$ and $W_{xh}$ instead of learning them, rescaling or clipping the gradient's components (putting a maximum limit on it) or using L1 or L2 penalties on the recurrent weights. Even more popular models are gated RNNs which explicitly address the vanishing gradients problem and will be explained in the next subchapter.

## 4.2 Gated RNNs

Main feature of gated RNNs is the ability to store long-term memory for a long time and at the same time to account for new inputs as effectively as possible. In modern NLP, two types of gated RNNs are used widely: Long Short-Term Memory networks and Gated Recurrent Units.

### 4.2.1 LSTM

**L**ong **S**hort-**T**erm **Memory** (**LSTM**) networks were introduced by Hochreiter and Schmidhuber (1997) with the purpose of dealing with problems of long term dependencies. Figure 4.2 illustrates the complex architecture of LSTM hidden state. Instead of a simple hidden unit that combines inputs and previous hidden states linearly and outputs their nonlinear transformation to the next step, hidden units are now extended by special input, forget and output gates that help to control the flow of information. Such more complex units are called memory cells, and the following equations show how a LSTM uses the gating mechanism to calculate the hidden state within a memory cell:

$$f^{(t)} = sigm(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f) \tag{4.17}$$
$$i^{(t)} = sigm(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i) \tag{4.18}$$
$$o^{(t)} = sigm(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o) \tag{4.19}$$
$$g^{(t)} = tanh(W_{xc}x^{(t)} + W_{hc}h^{(t-1)} + b_c) \tag{4.20}$$
$$c^{(t)} = f^{(t)}c^{(t-1)} + i^{(t)}g^{(t)} \tag{4.21}$$
$$h^{(t)} = o^{(t)}tanh(c^{(t)}) \tag{4.22}$$
$$\tag{4.23}$$

First, the forget gate $f^{(t)}$ decides which values of the previous output $h^{(t-1)}$ to forget. The next step involves deciding which information will be stored in the internal cell state $c^{(t)}$. This step consists of the following two parts: 1) the old cell state $c^{(t-1)}$ is multiplied by $f^{(t)}$ in order to forget information; 2) new candidates are calculated in $g^{(t)}$ and multiplied by the input gate $i^{(t)}$ in order to add new information. Finally, the output $h^{(t)}$ is produced with help of the output gate $o^{(t)}$ and applying a *tanh* function to the cell state in order to only output selected values. (Goodfellow et al. (2016), Graves (2013))

### 4.2.2 GRU

In 2014, Cho et al. (2014) introduced **G**ated **R**ecurrent **U**nits (**GRU**) whose structure is simpler than that of LSTM because they have only two gates, namely reset and update gate. The hidden state is thus calculated as follows:

$$r^{(t)} = sigm(W_{xr}x^{(t)} + W_{hr}h^{(t-1)}) \tag{4.24}$$
$$\tilde{h}^{(t)} = tanh(W_{xh}x^{(t)} + W_{hh}(r^{(t)} \odot h^{(t-1)})) \tag{4.25}$$
$$z^{(t)} = sigm(W_{xz}x^{(t)} + W_{hz}h^{(t-1)}) \tag{4.26}$$
$$h^{(t)} = z^{(t)} \odot h^{(t-1)} + (1 - z^{(t)}) \odot \tilde{h}^{(t)} \tag{4.27}$$
$$\tag{4.28}$$

First, the reset gate $r^{(t)}$ decides how much the previous hidden state $h^{(t-1)}$ is ignored, so if $r^{(t)}$ is close to 0, the hidden state is forced to drop past irrelevant information and reset with current input $x^{(t)}$. A candidate hidden state $\tilde{h}^{(t)}$ is then obtained in the following three steps: 1) weight matrix $W_{xh}$ is multiplied by current input $x^{(t)}$; 2) weight matrix $W_{hh}$ is multiplied by the element-wise product (denoted as $\odot$) of reset gate $r^{(t)}$ and previous hidden state $h^{(t)}$; 3) both products are added and a *tanh* function is applied in order to output the candidate values for a hidden state. On the other hand, the update gate $z^{(t)}$ decides how much of the previous information stored in $h^{(t-1)}$ will carry over to $h^{(t)}$ and how much the new hidden state must be updated with the candidate $\tilde{h}^{(t)}$. Thus, the update gate combines the functionality of LSTM forget and input gates and helps to capture long-term memory. Cho et al. (2014) used the GRU in the machine translation task and showed that the novel hidden units can achieve good results in term of BLEU score (see chapter 11[1] for definition) while being computationally cheaper than LSTM.

Figure 4.2 visualizes the architecture of hidden states of gated RNNs mentioned above and illustrates their differences:

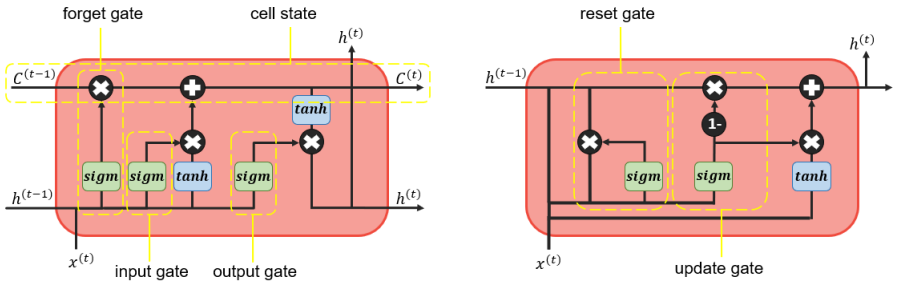



**FIGURE 4.2:** Structure of a hidden unit. LSTM on the right and GRU on the left. Source: Own figure inspired by http://colah.github.io/posts/2015-08-Understanding-LSTMs/

## 4.3 Extensions of Simple RNNs

In most challenging NLP tasks, such as machine translation or text generation, one-layer RNNs even with gated hidden states cannot often deliver successful results. Therefore more complicated architectures are necessary. Deep RNNs with several hidden layers may improve model performance significantly. On the other hand, completely new designs such as Encoder-Decoder architectures

[1] ./resources-and-benchmarks-for-nlp.html

are most effective when it comes to mapping sequences of an arbitrary length to a sequence of another arbitrary length.

### 4.3.1 Deep RNNs

The figure 4.1 shows that each layer is associated with one parameter matrix so that the model is considered shallow. This structure can be extended to a deep RNN, although the depth of an RNN is not a trivial concept since its units are already expressed as a nonlinear function of multiple previous units.

#### 4.3.1.1 Stacked RNNs

In their paper Graves et al. (2013) use a deep RNN for speech recognition where the depth is defined as stacking $N$ hidden layers on top of each other with $N > 1$. In this case the hidden vectors are computed iteratively for all hidden layers $n = 1$ to $N$ and for all time instances $t = 1$ to $T$:

$$h_n^{(t)} = f(a_n + W_{h_n h_n} h_n^{(t-1)} + W_{h_{n-1} h_n} h_{n-1}^{(t)}) \tag{4.29}$$

As a hidden layer function Graves et al. (2013) choose **bidirectional LSTM**. Compared to regular LSTM, BiLSTM can train on inputs in their original as well as reversed order. The idea is to stack two separate hidden layers one on another while one of the layers is responsible for the forward information flow and another one for the backward information flow. Especially in speech recognition one must take in consideration future context, too, because pronouncement depends both on previous and next phonemes. Thus, BiLSTMs are able to access long-time dependencies in both input directions. Finally, Graves et al. (2013) compare different deep BiLSTM models (from 1 to 5 hidden layers) with unidirectional LSTMs and a pre-trained RNN transducer (BiLSTM with 3 hidden layers pretrained to predict each phoneme given the previous ones) and show clear advantage of deep networks over shallow designs.

#### 4.3.1.2 Deep Transition RNNs

Pascanu et al. (2013a) proposed another way to make an RNN deeper by introducing **transitions**, one or more intermediate nonlinear layers between input to hidden, hidden to output or two consecutive hidden states. They argue that extending input-to-hidden functions helps to better capture temporal structure between successive inputs. A deeper hidden-to-output function, DO-RNN, can make hidden states more compact and therefore enables the model to summarize the previous inputs more efficiently. A deep hidden-to-hidden composition, DT-RNN, allows for the hidden states to effectively add

new information to the accumulated summaries from the previous steps. They note though, because of including deep transitions, the distances between two variables at $t$ and $t+1$ become longer and the problem of loosing long-time dependencies may occur. One can add shortcut connections to provide shorter paths for gradients, such networks are referred to as DT(S)-RNNs. If deep transitions with shortcuts are implemented both in hidden and output layers, the resulting model is called DOT(S)-RNNs. Pascanu et al. (2013a) evaluate these designs on the tasks of polyphonic music prediction and character- or word-level language modelling. Their results reveal that deep transition RNNs clearly outperform shallow RNNs in terms of perplexity (see chapter 11[2] for definition) and negative log-likelihood.

### 4.3.2 Encoder-Decoder Architecture

#### 4.3.2.1 Design and Training

The problem of mapping variable-length input sequences to variable-length output sequences is known as Sequence-to-Sequence or **seq2seq** learning in NLP. Although originally applied in machine translation tasks (Sutskever et al. (2014), Cho et al. (2014)), the seq2seq approach achieved state-of-the-art results also in speech recognition (Prabhavalkar et al., 2017) and video captioning (Venugopalan et al., 2015). According to Cho et al. (2014), the seq2seq model is composed of two parts as illustrated below:

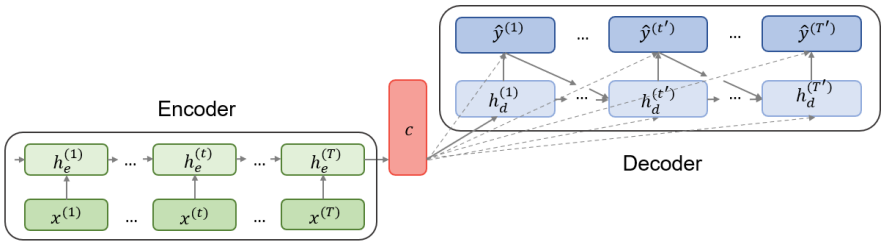



**FIGURE 4.3:** Encoder-Decoder architecture. Source: Own figure based on @cho2014learning.

The first part is an **encoder**, an RNN which is trained on input sequences in order to obtain a large summary vector $c$ with a fixed dimension. This vector is called context and is usually a simple function of the last hidden state. Sutskever et al. (2014) used the final encoder hidden state as context such that $c = h_e^{(T)}$. The second part of the model is a **decoder**, another RNN which generates predictions given the context $c$ and all the previous outputs. In

[2] ./resources-and-benchmarks-for-nlp.html

contrast to a simple RNN described at the beginning of this chapter, decoder hidden states $h_d^{(t)}$ are now conditioned on the previous outputs $y^{(t)}$, previous hidden states $h_d^{(t)}$ and the summary vector $c$ from the encoder part. Therefore, the conditional distribution of the one-step prediction is obtained by:

$$h_d^{(t)} = f(h_d^{(t-1)}, y^{(t-1)}, c) \tag{4.30}$$

$$P(y^{(t)}|y^{(t-1)}, ...y^{(1)}, c) = s(h_d^{(t)}, y^{(t-1)}, c) \tag{4.31}$$

$$\tag{4.32}$$

Both parts are trained simultaneously to maximize the conditional log-likelihood $\frac{1}{N}\sum_{n=1}^{N} \log p_\theta(y_n|x_n)$, where $\theta$ denotes the set of model parameters and $(y_n, x_n)$ is an (input sequence, output sequence) pair from the training set with size $N$ (Cho et al. (2014)).

#### 4.3.2.2 Multi-task seq2seq Learning

Luong et al. (2015a) extended the idea of encoder-decoder architecture even further by allowing **multi-task learning** (**MLT**) for seq2seq models. MLT aims to improve performance of one task using other related tasks such that one task complements another. In their paper, they investigate the following three settings: a) *one-to-many* - where the encoder is shared between different tasks such as translation and syntactic parsing, b) *many-to-one* - where the encoders learn different tasks such as translation and image captioning and the decoder is shared, c) *many-to-many* - where the model consists of multiple encoders and decoders which is the case of autoencoders, an unsupervised task used to learn a representation of monolingual data.

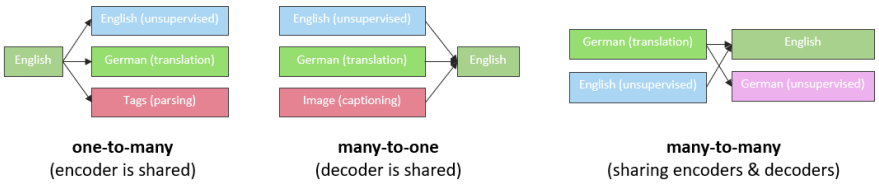



**FIGURE 4.4:** Multi-task settings. Source: @luong2015multi.

Luong et al. (2015a) consider German-to-English and English-to-German translations as the primary task and try to determine whether other tasks can improve their performance and vice versa. After training deep LSTM models with four layers for different task combinations, they conclude that MLT can improve the performance of seq2seq models substantially. For instance, the translation quality improves after adding a small number of parsing minibatches (one-to-many setting) or after the model have been trained to

generate image captions (many-to-one setting). In turn, translation task helps to parse large data corpus much better (one-to-many setting). In contrast to these achievements, autoencoder task does not show significant improvements in translation after two unsupervised learning tasks on English and German language data.

## 4.4 Conclusion

RNNs are powerful models for sequential data. Even in their simple form they can show valid results in different NLP tasks but their shortcomings led to introduction of more flexible networks such as gated units and encoder-decoder designs described in this chapter. As NLP has become a highly discussed topic in the recent years, even more advanced concepts such as Transfer Learning and Attention have been introduced, which still base on an RNN or its extension. However, RNNs are not the only type of neural networks used in NLP. Convolutional neural networks also find their application in modern NLP, and the next chapter will describe them.

# 5

# *Convolutional neural networks and their applications in NLP*

*Authors: Rui Yang*

*Supervisor: Prof. Dr. Christian Heumann*

## 5.1 Introduction to Basic Architecture of CNN

This section presents a brief introduction of the Convolutional neural network (CNN) and its main elements, based on which it would be more effective for further exploration of the applications of a Convolutional neural network in the field of Natural language processing (NLP).

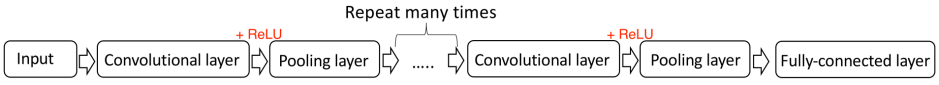



**FIGURE 5.1:** Basic structure of CNN

As illustrated in Figure 5.1, a convolutional neural network includes successively an input layer, multiple hidden layers, and an output layer, the input layer will be dissimilar according to various applications. The hidden layers, which are the core block of a CNN architecture, consist of a series of **convolutional layers**, **pooling layers**, and finally export the output through the **fully-connected layer**. In the following sub-chapters, descriptions of the critical layers of CNN and their corresponding intuitive examples will be provided in detail.

### 5.1.1 Convolutional Layer

The convolutional layer is the core building block of a CNN. In short, the input with a specific shape will be abstracted to a **feature map** after passing the

convolutional layer. a set of learnable **filters (or kernels)** plays an important role throughout this process. The following Figure 5.2 provides a more intuitive explanation of the convolutional layer.

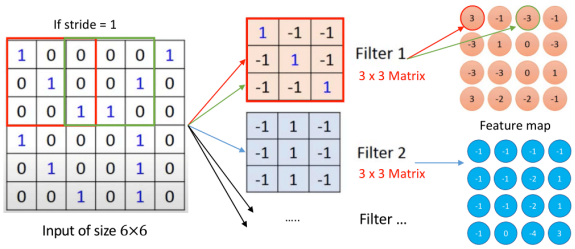



**FIGURE 5.2:** Basic operational structure of the convolutional layer

The input of the Neural Networks might be assumed as a $6 \times 6$ matrix and each element of which can be presented as the integer '0', '1'. As mentioned before, there is a set of learnable filters in the convolutional layer and each filter can be considered as a matrix, which is similar to a neuron in a fully-connected layer. In this instance, filters of size $3 \times 3$ slide over with a specific stride across the entire input image, and each element of the matrix or filter serves as a parameter (weight and bias) of Neural Networks. Traditionally, these parameters are not based on the initial setting but are trained through the training data.

An activated filter of size $3 \times 3$ has an ability to detect a pattern of the same size at some spatial position in the input. The algebraic operation explicates the transformation process from the input to the feature map.

$$X := \begin{pmatrix} X_{11} & X_{12} & \cdots & X_{16} \\ X_{21} & X_{22} & \cdots & X_{26} \\ \vdots & \vdots & \ddots & \vdots \\ X_{61} & X_{62} & \cdots & X_{66} \end{pmatrix}$$

where $X$ is the input matrix of size $6 \times 6$ as mentioned before;

$$F := \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}$$

where $F$ denotes one filter of size $3 \times 3$;

$$\beta := (w_{11} \quad w_{12} \quad \cdots \quad w_{16})$$

where $\beta$ is a unrolled matrix or filter;
$A_{11} = (F \times X)_{11} := \beta \cdot (w_{11} \quad w_{12} \quad w_{13} \quad w_{21} \quad w_{22} \quad w_{23} \quad w_{31} \quad w_{32} \quad w_{33})^T$

Therefore, the first element of the feature map $A_{11}$ can be calculated through the dot product operation shown above. Sequentially, the second element of the feature map is determined by the sliding dot product of filter and the succeeding input matrix with the same size after setting a specific value of stride, which can be considered as a moving distance. After the whole process, a feature map of size $4 \times 4$ has been generated. Generally, there is more than one filter in the convolutional layer and each filter generates a feature map with the same size. The result of this convolutional layer is multiple feature maps (also referred to as activation map) and these feature maps corresponding to different filters are stacked together along the depth dimension.

Another improved convolutional layer was proposed by (Kalchbrenner et al. (2016a)) and this kind of convolution is named Dilated convolution, in order to solve the problem that the pooling operation in the pooling layer will lose a lot of information. The critical contribution of this convolution is that the receptive field of the network will not be reduced by removing the pooling operation. In other words, the units of feature maps in the deeper hidden layer can still map a larger region of the original input. As illustrated in Figure 5.3 provided by (van den Oord et al. (2016)), although there is no pooling layer, the original input information is still increased as the layers are deeper.

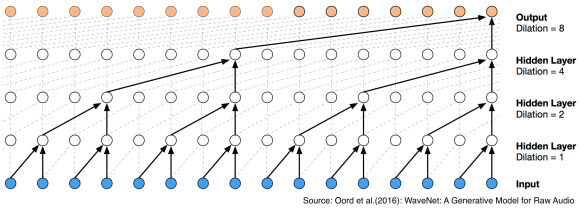



**FIGURE 5.3:** Visualization of dilated causal convolutional layers

### 5.1.2 ReLU layer

A non-linear layer (or activation layer) will be the subsequent process after each convolutional layer and the purpose of which is to introduce non-linearity to the neural networks because the operations during the convolutional layer are still linear (element-wise multiplications and summations). Generally, the major reason for introducing non-linearity is that there is a certain non-linear relationship between separate neurons. However, a convolutional layer is to perform basically a linear operation, and therefore, consecutive convolution layers are essentially equivalent to a single convolution layer, which is only used to reduce the representational power of the networks. As a result, the

property of non-linearity between neurons has not been reflected and it is necessary to establish an activation function between the convolutional layer to avoid such an issue.

**Activation function**, which performs a non-linear transformation, plays a critical role in CNN to decide whether a neuron should be activated or ignored. Several activation functions are available after the convolutional layer, such as hyperbolic function and sigmoid function, etc., among of which ReLU is the most commonly used activation function in neural networks, especially in CNNs(Krizhevsky et al., 2012) because of its two properties:

- Non-linearity: ReLU is the abbreviation of Rectified Linear Unit and defined mathematically as below:
$$R(z) = z^+ = max(0, z)$$

Where z denotes the output element of the previous convolutional layer. All negative values of feature maps from the previous will be replaced by setting them to zero.

- Non-Saturation: Saturation arithmetic is a kind of arithmetic in which all operations are limited to a fixed range between a minimum and maximum value.
  - $f$ is non-saturating iff $(|\lim_{z\to-\infty} f(z)| = +\infty) \cup (|\lim_{z\to+\infty} f(z)| = +\infty)$
  - $f$ is saturating iff $f$ is not non-saturating

As illustrated in Figure 5.4, compared with saturating activation sigmoid function that saturate at large values of the input, ReLU activation function does not saturate(Krizhevsky et al., 2012) and the gradient of it is 0 on the negative x-axis and 1 on the positive side, which is a benefit of using this activation function because the updates to the weights of the neural networks at each iteration are consistent with the gradient of the activation function. To be more specific, neuron‘s weights will stop updating if its gradient is close to zero. It is obviously problematic if such a scenario appears too early in the training process.

The following Figure 5.5 based on Figure 5.2 indicates a simplified version of the ReLU layer. Every single element of multiple feature maps, which is determined from the previous convolutional layer, will be further calculated by the ReLU activation function in this layer. Specifically, all positive values remain the same, and negative values are replaced by setting them to zero. The output after the ReLU layer, which has an identical network structure with the feature map from the previous convolutional layer, will be used as an input for the subsequent convolutional layer.

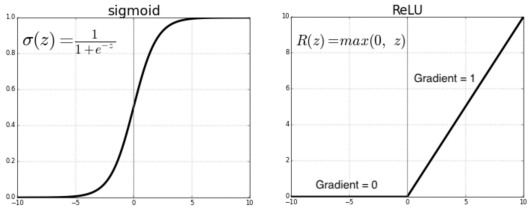



**FIGURE 5.4:** Comparison between saturating and non-saturating activation function

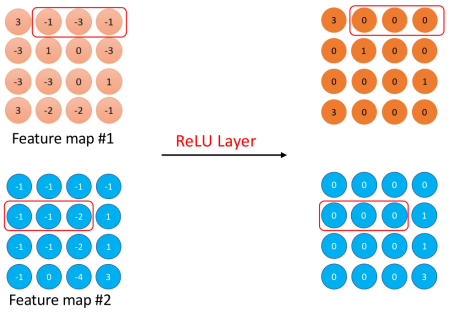



**FIGURE 5.5:** Basic operational structure of the ReLU layer

### 5.1.3 Pooling layer

The pooling layer is a concept that can be intuitively understood. The purpose of the pooling layer is to reduce progressively the spatial size of the feature map, which is generated from the previous convolutional layer, and identify important features.

There are multiple pooling operations, such as average pooling, $l_2 - norm$ pooling, and **max pooling**, Among which max pooling is the most commonly used function(Scherer et al. (2010)), and the idea of max pooling is that the exact location of a feature is less important than its rough location relative to other features (Yamaguchi et al. (1990)). Simultaneously, this process helps to control overfitting to a certain extent. The following Figure 5.6 illustrates an example that constructs a basic operational structure of the max pooling.

The example mentioned above shows that two feature maps are generated according to two different filters. In this case, these feature maps of size $4 \times 4$ are separated into four non-overlapping sub-regions of size $2 \times 2$, and every single sub-region is named as depth slice. The Maximum value of each sub-region will be stored in the output of the pooling layer. As a result, the input dimensions are further reduced from $4 \times 4$ to $2 \times 2$.

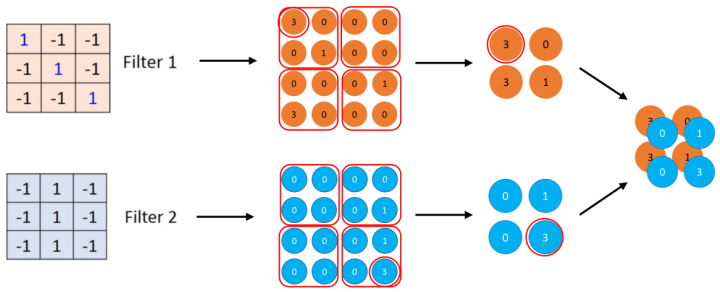



**FIGURE 5.6:** Basic operational structure of the max pooling layer

Some of the most critical reasons why adding max pooling layer to neural networks include the following:

- **Reducing computation complexity**: Since max pooling is reducing the dimension of the given output of a convolutional layer, the networks will be able to detect larger areas of the output. This process reduces the number of parameters in the neural networks and consequently reduces computational load.
- **Controlling overfitting**: Overfitting appears when the model is too complex or fits the training data too well. It may lose the true structure and then becomes difficult to generalize to new cases that are in the test data. With max-pooling operation, not all features but the primary features from each sub-region are extracted. Therefore, max-pooling operation reduces the probability of overfitting to a great extent.

Except for this most commonly applied operation in NLP, several pooling operations for different intention include the following:

- **Average pooling** is usually used for topic models. If a sentence has different topics and the researchers assume that max pooling extracts insufficient information, average pooling can be considered as an alternative.
- **Dynamic pooling** proposed by (Kalchbrenner et al. (2014)) has an ability to dynamically adjust the number of features according to the network structure. More specifically, by combining the adjacent word information at the bottom and passing it gradually, new semantic information is recombined at the upper layer, so that words far away in the sentence also have interactive behavior (or some kind of semantic connection). Eventually, the most important semantic information in the sentence is extracted through the pooling layer.

### 5.1.4 Fully-connected layer

As we mentioned in the previous section, one or more fully-connected layers are connected after multiple convolutional layers and pooling layers, each neuron in the fully connected layer is fully connected with all the neurons from the penultimate layer. The fully-connected layer, shown in Figure 5.7, can integrate local information with class distinction in the convolutional layer or pooling layer. In order to improve the CNN network performance, the excitation function of each neuron in the fully connected layer generally uses the ReLU function.

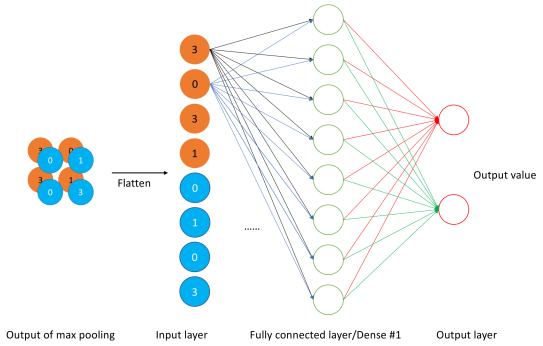



**FIGURE 5.7:** Basic operational structure of the fully connected layer

## 5.2 CNN for sentence classification

The explanation of CNN's basic architecture provided in the first sub-chapters is based on a general example. Many researchers constructed their own specific CNN models based on this basic architecture in recent years and achieved outstanding results in the field of NLP. Therefore, this section explores four superior CNN architecture with some technical detail and their performance comparison will be provided in later sub-chapters of this report.

### 5.2.1 CNN-rand/CNN-static/CNN-non-static/CNN-multichannel

The first model to explore is published by (Kim (2014)), one of the highlights of this model is that the architecture is conceptually simple and efficient when dealing with the tasks of sentiment analysis and question classification. As

illustrated in Figure 5.8 provided by (Kim (2014)), a simple CNN architecture of (Collobert et al. (2011)) with a single convolutional layer is utilized and the general architecture includes the following sub-structure:

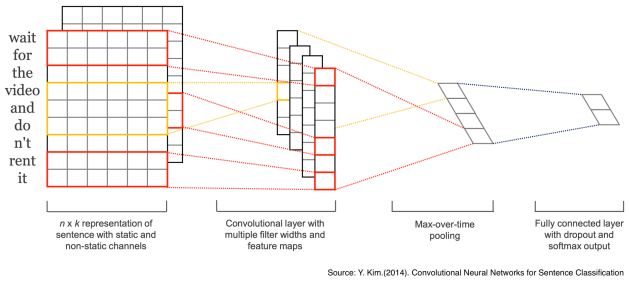



**FIGURE 5.8:** Model architecture of CNN for sentence classification

1. **Representation of sentence**: Assume that there are $n$ words in a sentence, and each word is denoted as $x_i; \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$ and $x_i \in \mathbb{R}^k$ to the $k$-dimensional word vector. Therefore, a sentence can be represented as:
$$X_{1:n} = X_1 \oplus X_2 \oplus ... \oplus X_n$$
Where $\oplus$ is the concatenation operator.
2. **Convolutional layer**: Let a filter denote as $w \in \mathbb{R}^{hk}$, which is used to a window of $h$ words. A feature map $c = [c_1, c_2, ..., c_{n-h+1}]$ can be generated by:
$$c_i = f(w \times x_{i:i+h-1} + b)$$
where $b \in \mathbb{R}$ is a bias term.
3. **Max-over-time pooling**: Pooling operation has been applied for the respective filter to select the most important feature from each feature map $\hat{c} = max(c)$, notice that one feature $\hat{c}$ is generated by one filter, and these features will be passed to the last layer.
4. **Fully connected layer**: The selected features $Z = [\hat{c}_1, \hat{c}_2, ..., \hat{c}_j]$ from the previous layer have been flattened into a single vector, in order to aggregate each of them and therefore a specific class can be assigned to it based on the entire input.

CNN is a feed-forward model without cyclic connection. To be more specific, the direction of information flow in a forward model is in one direction (i.e from inputs to outputs). However, the models are trained or learned by the use of backward propagation (i.e from outputs to inputs), where the gradients are recalculated at each epoch to avoid co-adaptation.

In a forward propagation of this CNN, the output unit y based on the selected features $Z$ is determined by using
$$y = w \cdot z + b$$

In a backward propagation, a dropout mechanism is applied as follows.

$$y = w \cdot (z \circ r) + b$$

Where $\circ$ is the element-wise multiplication operator and $r \in \mathbb{R}^m$ denotes a 'masking' vector of Bernoulli random variables with probability $p$ of being 1. As a result, gradients are backpropagated with probability $p$ and weights $\hat{w}$ are trained by using

$$\hat{w} = pw$$

Based on the model architecture described above, four derivative CNN models are introduced by (Kim (2014)) and the major difference among them are listed below:

- **CNN-rand**: All words are randomly initialized and then modified during training.
- **CNN-static**: A model with pre-trained word vectors by using word2vec and keep them static.
- **CNN-non-static**: A model with pre-trained word vectors by using word2vec and these word vectors are fine-tuned for each task.
- **CNN-Multichannel**: A model with two channels generated by two sets of words vectors and each filter is employed to both channels.

### 5.2.2 Character-level ConvNets

The second model is published by (Zhang et al. (2015)), the two major differences of which compared with the previous model from (Kim (2014)) include the following:

1. The model architecture with 6 convolutional layers and 3 fully-connected layers (9 layers deep) is relatively more complex.
2. Different from the previous word-based Convolutional neural networks (ConvNets), this model is at character-level by using character quantization.

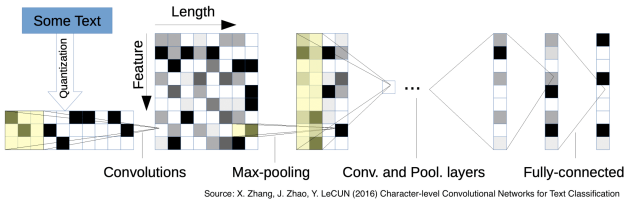



**FIGURE 5.9:** Model architecture of character-level CNN

The Figure 5.9 above provided by (Zhang et al. (2015)) shows the basic architecture of Character-level ConvNets, the corresponding explanation of the main components will be provided below:

1. **Character quantization**: The input characters will be transformed into an encoding matrix of size $m \times l_0$ by using 1-of-$m$ encoding (or "one-hot" encoding). To be noticed that, The length of the character exceeds the determined value $l_0$ and will be ignored and the black characters as well as the characters that are not in the alphabet will be quantized as zero vectors.
2. **Temporal convolutional module**: A sequence of encoded characters followed by a temporal convolutional module, which is a variation over Convolutional Neural Networks works for sequence modelling tasks. To be more specific, when sentiment analysis is performed by using ConvNets, a fixed-size input will be a precondition, we can adjust the initial input length by truncating or padding the actual input to satisfy this criterion without affecting the sentiment and sequentially generate fixed-size outputs. Conceptually, this kind of 1-D convolution is called temporal convolution and the convolutional function defined as follow:
$$h(y) = \sum_{x=1}^{k} f(x) \cdot g(y \cdot d - x + c)$$
where $h_j(y)$ denotes outputs of the convolutional layer; a discrete kernel functions $f_{i,j} \in [1, k] \to \mathbb{R}$ $((i = 1, ..., m \text{ and } j = 1, ..., n))$ is also called weights; $d$ is denoted as stride; $c = k - d + 1$ is used as an offset constant.
3. **Temporal max-pooling**: Based on the research of (Boureau et al. (2010)), a 1-D version of the max-pooling $h(y)$ is employed in this ConvNets, which is defined as
$$h(y) = max(g(y \cdot d - x + c))$$

With the help of this pooling function, it is possible to train ConvNets deeper than 6 layers.

4. **ReLU layer**: the activation function used in this model is similar to ReLU $h(x) = max\{0, x\}$. More specifically, the algorithm is stochastic gradient descent (SGD). However, SGD is influenced by the strong curvature of the optimization function and moves slowly towards the minimum. Therefore, based on the research of Sutskever et al. (2013), a momentum of 0.9 and an initial step size 0.01 are established to reach the minimum more quickly.

### 5.2.3 Very Deep CNN

The third character-level model to explore is from Schwenk et al. (2017). Inspired by (Simonyan and Zisserman (2015)), a CNN model with deep architectures of many convolutional layers is developed, the significant difference of which from the previous model architecture is that this model applies much deeper architectures (i.e. using up to 29 convolutional layers), in order to learn hierarchical representations of whole sentences. The overall architecture will be explored based on Figure 5.10 and 5.11 also provided by (Schwenk et al. (2017)).

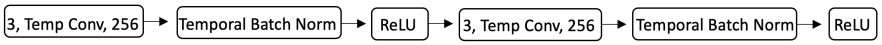



**FIGURE 5.10:** Architecture of convolutional block

The above Figure 5.10 shows an architecture of a convolutional block with 256 feature maps and the kernel size of all the convolutions is 3. The convolutional block is composed of two consecutive convolutional layers, and each one followed by a temporal BatchNorm layer (Ioffe and Szegedy (2015)) and a ReLU activation.

**Batch normalization**, as the name suggests, is a normalized operation commonly used to improve the speed and stability of neural networks, especially for the deep neural network. The following shows the algorithm of Batch Normalizing Transform.

For a layer with $d$-dimensional input $x = (x^{(1)}, ..., x^{(d)})$

1. $\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$ //mini-batch mean
2. $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ // mini-batch variance
3. $\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}}$ // normalize

where $k \in [1, d]$ and $i \in [1, m]$; $\mu_B^{(k)}$and $\sigma_B^{(k)^2}$ are the per-dimension mean and variance; $\epsilon$ is an arbitrarily small constant.

4. $y_i^{(k)} = \gamma^{(k)}\hat{x}_i^{(k)} + \beta^{(k)}$ // scale and shift

where the parameters $\gamma^{(k)}$and $\beta^{(k)}$ are subsequently learned in the optimization process.

To be more specific, in the SGD training process, use mini-batch to normalize the corresponding activation so that the mean value of the results (all dimensions of the output signal) is 0 and the variance is 1. Subsequently,

two independent learnable parameters $\beta$ and $\gamma$ will be introduced in the final operation "scale and shift".

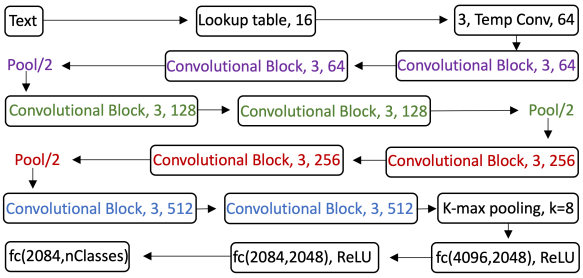



**FIGURE 5.11:** Very Deep Convolutional Networks architecture

The overall architecture of VDCNN is shown in Figure 5.11 above, The first layer contains 64 convolutions of size 3 after importing the initial text, a series of convolutional blocks are connected after the convolutional layer, and the number of feature maps is determined by two rules:

1. If the temporal resolution of output remains the same the layers have the same number of feature maps.
2. If the temporal resolution is halved, the number of feature maps is doubled.

Based on the above rules, the size of resolution will be halved after each pooling operation, so the number of feature maps will be corresponding to double from 128 to 512.

**k-max pooling**: The result of k-max pooling is not to return a single maximum value, but to return k sets of maximum values, which are a subsequence of the original input. The parameter $k$ in the pooling can be a dynamic function, and this specific value depends on the input or other parameters of the network. the specific dynamic function is as follows.

$$k_l = max(k_{top}, \lceil \frac{L-l}{L} s \rceil)$$

In this formula, $s$ denotes the length of the sentence, $L$ represents the total number of convolutional layers, and $l$ represents the number of convolutional layers that are currently in. so it can be seen that $k$ varies with the length of the sentence and the depth of the network change. The advantage of K-max pooling is that it not only extracts more than one important information in the sentence but also retains their order.

The output of k-max pooling is transformed into a single vector which will be used as input to a three fully-connected layer with ReLU activation function and softmax outputs.

### 5.2.4 Deep Pyramid CNN

The last model we will discuss in this chapter is Deep Pyramid Convolutional Neural Networks (DPCNN), which is published by (Johnson and Zhang (2017)). The major difference from the previous deep CNN is that this model is constructed of a low-complexity word-level deep CNN architecture. The motivation for using such an architecture based on the point of view of (Johnson and Zhang (2017)) includes the following:

**low-complexity**: As the research of (Johnson and Zhang (2016)) shows that very shallow 1-layer word-level CNNs such as (Kim (2014)) performs more accurate and much faster than the deep character-level CNNs of (Schwenk et al. (2017)).

**word-level**: word-level CNNs are more accurate and much faster than the state-of-the-art very deep networks such as character-level CNNs even in the setting of large training data.

The description of key features of DPCNN will be provided based on the Figure 5.12 from (Johnson and Zhang (2017)) as follows.

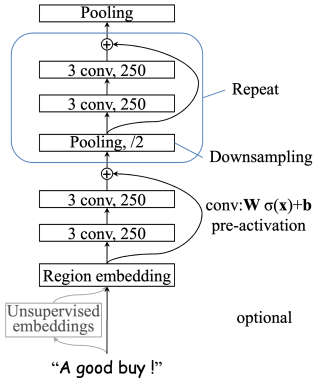



**FIGURE 5.12:** Architecture of Deep Pyramid CNN

**Text region embedding with unsupervised embeddings**: This technique is based on the basic region embedding. More specifically, the embedding generated after performing a set of convolution operations on different sizes of text area/segment (such as bi-grams or tri-grams) and these convolution operations are achieved by using multi-size learnable convolutional filters. There are two options when performing convolution operation on a text area, For example, by using tri-grams, one is to preserve the word order, which means that setting a set of 2D convolution kernels with size= $3 \times D$ to convolve the 3 words (where D is the word embedding dimension), another is not to preserve the word order (i.e. using the Bag-of-words model). Preliminary experiments indicated that considering word order does not significantly

help to improve accuracy and increase computational complexity. Therefore, DPCNN adopts a method similar to the Bag-of-words model. In addition to this, to avoid the overfitting caused by high representation power of n-gram, a technique called tv-embedding (two-views embedding) will be introduced in DPCNN, which means that a region of text as view-1 and its adjacent regions as view-2 will be pre-defined, and view-1 will be trained to predict view-2 in one hidden layer and it will be considered as input, this process is called unsupervised embedding.

**Shortcut connections with pre-activation** : From experience, the depth of the network is critical to the performance of the model Because by giving the model more parameters, the model can fit the train data at least as well as before and detect more complex feature patterns. However, the experimental results show that the training accuracy will decrease as the depth of the network increases, the reason is Degradation problem or so-called vanishing/exploding gradients. More specifically, Assume we consider a shallow architecture and add more layers linked by identity mapping (i.e. $f(z) = z$). The value of the output after multiple nonlinear layers should also be z, but the result will be biased due to the existence of this Degradation problem. Therefore, the shortcut connections were proposed by (He et al. (2016a)). Formally, denoting the desired underlying mapping as $H(z)$, $f(z) := H(z) - z$ is denoted as a residual function, so the original mapping is recast into $f(z)+z$. This technique is used to solve the problem of depth model accuracy.

**Downsampling with the number of feature maps fixed**: Similar to the VDCNN described in the previous sub-chapter is that multiple convolutional blocks are also introduced to the architecture of DPCNN. However, the difference is that the number of feature maps is fixed instead of increasing like VDCNN, because (Johnson and Zhang (2017)) thinks increasing the number of feature maps will lead to increased computation time substantially without accuracy improvement, Therefore, the computation time and sample size for each convolution layer are halved (Downsampling).

## 5.3 Datasets and Experimental Evaluation

A brief description of the eight data sets will be provided in this sub-chapter, based on which the performance comparison of all four models mentioned above will also be listed.

### 5.3.1 Datasets

The comparison results of all CNN models explored in this chapter are based on the eight datasets complied by (Zhang et al. (2015)), summarized in the following table.

| Data | AG | Sogou | Dbpedia | Yelp.p | Yelp.f | Yahoo | Ama.f | Ama.p |
|---|---|---|---|---|---|---|---|---|
| # of training documents | 120K | 450K | 560K | 560K | 650K | 1.4M | 3M | 3.6M |
| # of test documents | 7.6K | 60K | 70K | 38K | 50K | 60K | 650K | 400K |
| # of classes | 4 | 5 | 14 | 2 | 5 | 10 | 5 | 2 |
| Average #words | 45 | 578 | 55 | 153 | 155 | 112 | 93 | 91 |

Based on the description of the datasets of (Johnson and Zhang (2017)), **AG** and **Sogou** are news. **Dbpedia** is an ontology. **Yahoo** consists of questions and answers from the 'Yahoo! Answers' website. **Yelp** and **Amazon ('Ama')** are reviews where `.p` (polarity) in the names indicates that labels are binary (positive/negative), and `.f` (full) indicates that labels are the number of stars. **Sogou** is in Romanized Chinese, and the others are in English. Classes are balanced on all the datasets. Furthermore, These eight datasets can be subdivided into two groups, the first four datasets are relatively smaller datasets and the rest of them can be classified as relatively larger data sets.

### 5.3.2 Experimental Evaluation

The following table provided by (Johnson and Zhang (2017)) shows the error rates in percentage on datasets described in the previous sub-chapter in comparison with all models. To be more specific, a lower error rate represents better model performance and the corresponding best results are marked in bold. `tv` stands for tv-embeddings. `w2v` stands for word2vec. `(w2v)` indicates that the best results among those with and without word2vec pretraining are shown. Note that `best` next to the model name is denoted as the model by choosing the best test error rate among several variations presented in the respective papers.

| Models | Deep | Unsup. embed. | AG | Sogou | Dbp |
|---|---|---|---|---|---|
| DPCNN + unsupervised embed.(Johnson and Zhang (2017)) | √ | tv | 6.87 | **1.84** | |
| ShallowCNN + unsup. embed.(Kim (2014)) | | tv | **6.57** | 1.89 | |
| Very Deep char-level CNN: best (Schwenk et al. (2017)) | √ | | 8.67 | 3.18 | |
| fastText bigrams (Joulin et al. (2017)) | | | 7.5 | 3.2 | |
| [ZZL15]'s char-level CNN: best(Zhang et al. (2015)) | √ | | 9.51 | 4.88 | |

| Models | Deep | Unsup. embed. | AG | Sogou | Dbp |
|---|---|---|---|---|---|
| [ZZL15]'s word-level CNN : best(Zhang et al. (2015)) | √ | (w2v) | 8.55 | 4.39 | |
| [ZZL15]'s linear model: best(Zhang et al. (2015)) | | | 7.64 | 2.81 | |

Based on the comparison results shown above, some of the significant results include the following:

1. DPCNN achieves an outstanding performance, which shows the lowest error rate in six of the eight datasets.
2. Shallow CNN, which possesses the simplest model architecture with only one convolutional layer, performs even better than other deep models.
3. By mere comparison of character-level CNN (Zhang et al. (2015)) and word-level CNN (Zhang et al. (2015)), the result will be that words-level CNN performs better in the smaller datasets and character-level CNN performs better in the bigger datasets.

## 5.4 Conclusion and Discussion

In summary, this chapter introduces the architecture of multiple CNN models and compares the performance of these models in the application of NLP especially for text categorization in different scale datasets. Subsequently, we will discuss the results obtained from the previous table in two ways including a comparison between the character-level approach and the word-level approach and depth of the model.

1. **Character-level and word-level** Extending from the experimental studies and the corresponding comparison results shown in the previous sub-chapter, word-level CNN possesses higher accuracy presented in lower error rate in comparison of character-level CNNs in general, although character-level CNN holds an advantage in not having to deal with millions of distinct words, word-level approach displays higher accuracy than character-level because the advantage of word-level is that Word can represent the meaning.
2. **Deep and shallow model architecture** DPCNN, which shows the best test error rate among all CNN models discussed in this chapter, could be considered as a kind of deeper Shallow CNN. Consequently, increasing the depth of the model can lead to an

increase in parameters and increased capacity to handle complex feature patterns. Better performance of DPCNN compared with Shallow CNN evidence that increasing depth can improve accuracy to a certain extent. However, it comes to the opposite conclusion by comparing with Very Deep CNN since the probability of potential issues such as vanishing/exploding gradients will increase as the number of layers increases. Therefore, it is essential to improve the accuracy of complex and deep models by introducing appropriate components (e.g. shortcut connections).

# 6

# *Introduction: Transfer Learning for NLP*

*Authors: Carolin Becker, Joshua Wagner, Bailan He*

*Supervisor: Matthias Aßenmacher*

As discussed in the previous chapters, natural language processing (NLP) is a very powerful tool in the field of processing human language. In recent years, there have been many proceedings and improvements in NLP to the state-of-art models like BERT. A decisive further development in the past was the way to transfer learning, but also self-attention.

In the next three chapters, various NLP models will be presented, which will be taken to a new level with the help of transfer learning in a first and a second step with self-attention and transformer-based model architectures. To understand the models in the next chapters, the idea and advantages of transfer learning are introduced. Additionally, the concept of self-attention and an overview over the most important models will be established

## 6.1 What is Transfer Learning?

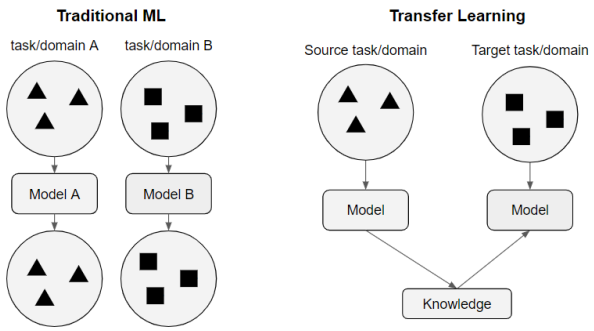



**FIGURE 6.1:** Classic Machine Learning and Transfer Learning

In figure 6.1 the difference between classical machine learning and transfer learning is shown.

For classical machine learning a model is trained for every special task or domain. Transfer learning allows us to deal with the learning of a task by using the existing labeled data of some related tasks or domains. Tasks are the objective of the model. e.g. the sentiment of a sentence, whereas the domain is where data comes from. e.g. all sentences are selected from Reddit. In the example above, knowledge gained in task A for source domain A is stored and applied to the problem of interest (domain B).

Generally, transfer learning has several advantages over classical machine learning: saving time for model training, mostly better performance, and not a need for a lot of training data in the target domain.

It is an especially important topic in NLP problems, as there is a lot of knowledge about many texts, but normally the training data only contains a small piece of it. A classical NLP model captures and learns a variety of linguistic phenomena, such as long-term dependencies and negation, from a large-scale corpus. This knowledge can be transferred to initialize another model to perform well on a specific NLP task, such as sentiment analysis. (Malte and Ratadiya, 2019)

## 6.2 (Self-)attention

The most common models for language modeling and machine translation were, and still are to some extent, recurrent neural networks with long short-term memory (Hochreiter and Schmidhuber, 1997) or gated recurrent units (Chung et al., 2014). These models commonly use an encoder and a decoder archictecture. Advanced models use attention, either based on Bahdanau's attention (Bahdanau et al., 2014) or Loung's attention (Luong et al., 2015b).

Vaswani et al. (2017) introduced a new form of attention, self-attention, and with it a new class of models, the *Transformers*. A Transformer still consists of the typical encoder-decoder setup but uses a novel new architecture for both. The encoder consists of 6 Layers with 2 sublayers each. The newly developed self-attention in the first sublayer allows a transformer model to process all input words at once and model the relationships between all words in a sentence. This allows transformers to model long-range dependencies in a sentence faster than RNN and CNN based models. The speed improvement and the fact that "individual attention heads clearly learn to perform different tasks" Vaswani et al. (2017) lead to the eventual development of **B**idirectional **E**ncoder **R**epresentations from **T**ransformers by Devlin et al. (2018). **BERT**

and its successors are, at the time of writing, the state-of-the-art models used for transfer learning in NLP. The concepts attention and self-attention will be further discussed in the **"Chapter 9: Attention and Self-Attention for NLP"**.

## 6.3 Overview over important NLP models

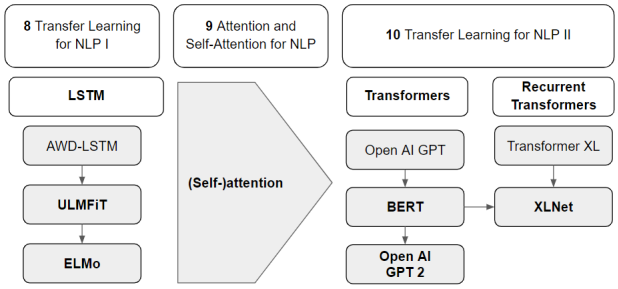



**FIGURE 6.2:** Overview of the most important models for transfer learning

The models in figure 6.2 will be presented in the next three chapters.

First, the two model architectures ELMo and ULMFit will be presented, which are mainly based on transfer learning and LSTMs, in **Chapter 8: "Transfer Learning for NLP I"**:

- **ELMo** (Embeddings from Language Models) first published in Peters et al. (2018) uses a deep, bi-directional LSTM model to create word representations. This method goes beyond traditional embedding methods, as it analyses the words within the context
- **ULMFiT** (Universal Language Model Fine-tuning for Text Classification) consists of three steps: first, there is a general pre-training of the LM on a general domain (like WikiText-103 dataset), second, the LM is finetuned on the target task and the last step is the multilabel classifier fine tuning where the model provides a status for every input sentence.

In the **"Chapter 10: Transfer Learning for NLP II"** models like BERT, GTP2 and XLNet will be introduced as they include transfer learning in combination with self-attention:

- **BERT** (Bidirectional Encoder Representations from Transformers Devlin

et al. (2018)) is published by researchers at Google AI Language group. It is regarded as a milestone in the NLP community by proposing a bidirectional Language model based on Transformer. BERT uses the Transformer Encoder as the structure of the pre-train model and addresses the unidirectional constraints by proposing new pre-training objectives: the "masked language model"(MLM) and a "next sentence prediction"(NSP) task. BERT advances state-of-the-art performance for eleven NLP tasks and its improved variants **Albert** Lan et al. (2019) and **Roberta** Liu et al. (2019) also reach great success.

- **GPT2** (Generative Pre-Training-2, Radford et al. (2019)) is proposed by researchers at OpenAI. GPT-2 is a tremendous multilayer Transformer Decoder and the largest version includes 1.543 billion parameters. Researchers create a new dataset "WebText" to train GPT-2 and it achieves state-of-the-art results on 7 out of 8 tested datasets in a zero-shot setting but still underfits "WebText".

- **XLNet** is proposed by researchers at Google Brain and CMU(Yang et al., 2019). It borrows ideas from autoregressive language modeling (e.g., Transformer-XL Dai et al. (2019)) and autoencoding (e.g., BERT) while avoiding their limitations. By using a permutation operation during training, bidirectional contexts can be captured and make it a generalized order-aware autoregressive language model. Empirically, XLNet outperforms BERT on 20 tasks and achieves state-of-the-art results on 18 tasks.

# 7

# *Transfer Learning for NLP I*

*Author: Carolin Becker*

*Supervisor: Matthias Aßenmacher*

Natural language processing (NLP) has seen rapid advancements in recent years, mainly due to the growing transfer learning usage. One significant advantage of transfer learning is that not every model needs to be trained from scratch.

In the first "wave" of transfer learning, **ELMo** (Embeddings from Language Models), **ULMFiT** (Universal Language Model Fine-tuning for Text Classification), and **GPT** (Generative Pretraining) were published.

These models enhanced the performance and introduced ideas like contextual embeddings, fine-tuning, and transformers in natural language processing.

## 7.1 Outline

First, the **taxonomy of transfer learning** and **inductive sequential transfer learning** are introduced. Inductive sequential transfer learning will give a framework for the following models and the difference between embeddings and fine-tuning will be elucidated.

Second, the models are introduced in detail within the framework of the two steps of inductive sequential transfer learning:

- The Allen Institute introduced with **ELMo** a state-of-the-art model that can distinguish between the same word's varying meaning in different contexts (**contextual word embedding**).
- Howard and Ruder introduced **ULMFiT**, an adaption of **fine-tuning** in NLP.
- Open AI published **GPT**, the first **transformer model**, which is an urge for the upcoming chapters. This model architecture is mainly used in the most recent developments.

Lastly, essential facts are **summarized**, and the **following chapters are motivated**.

## 7.2 Sequential inductive transfer learning

To better understand the models introduced in this chapter, the taxonomy of transfer learning is introduced. Primarily, sequential inductive transfer learning will be explained further.

### 7.2.1 Types of transfer learning

Pan and Yang (2010) divided **transfer learning** into different types (figure 7.1).

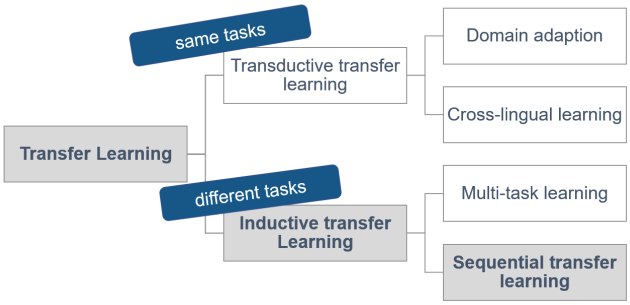



**FIGURE 7.1:** Four types of transfer learning. Source: based on (Ruder, 2019)).

On the one hand, in **transductive transfer learning**, the source and target task are the same. There is another distinction between **domain adoption** (data from different domains) and **cross-lingual-learning** (data from different languages) (Ruder, 2019).

On the other hand, there is **inductive transfer learning**, where the source and the target task is different. Inductive transfer learning can be divided into multi-task transfer learning and sequential transfer learning. In **multi-task** transfer learning, several tasks are learned simultaneously, and common knowledge is shared between the tasks. In **sequential transfer learning**, the source data's general knowledge is transferred to only one task (Ruder, 2019).

Inductive sequential transfer learning has led to the most significant improvements in the past. The following models in this chapter can be explained in the framework of inductive sequential transfer learning.

### 7.2.2 Feature Extraction vs. Fine-tuning

Sequential inductive transfer learning mainly consists of two steps: pretraining and adoption.

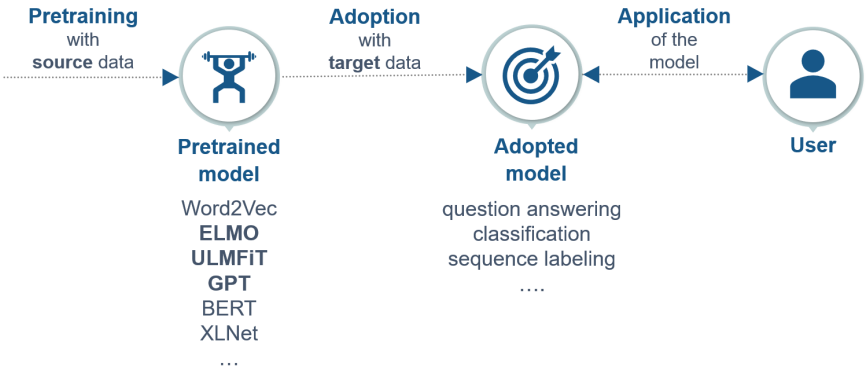



**FIGURE 7.2:** Steps in sequential inductive transfer learning. Source: based on Ruder et al. (2019).

As in figure 7.2, the model is first pretrained with the source task, and second, the model is adapted to the target task. The user can use this adopted model for the designated task (Ruder et al., 2019).

In the first step, all models are **pretrained** on an extensive source data set, which is, in the best case, very close to the target task (Peters et al., 2019). This chapter's pretrained language models are uni-directional models that (only) predict the next word during pretraining (figure 7.3).

In second step follows the **adoption** on the target task. Here, the main distinction is if the pretrained model weights are kept (**embedding** or **feature extraction**) or adjusted to the target task (**fine-tuning**) (Peters et al., 2019).

In **feature extraction**, single parts (sentences or characters) are extracted to a fixed-length matrix with the dimensions $\mathbb{R}^n \times k$ where $k$ is the fixed-length. This matrix represents the context of every word given of every other word. In the adoption phase, the LM's weights do not change, and just the top layer of the model is used. The adopted model learns a linear combination of the top layer (Peters et al., 2019).

On the other hand, **fine-tuning** adjusts the pretrained model's weights on

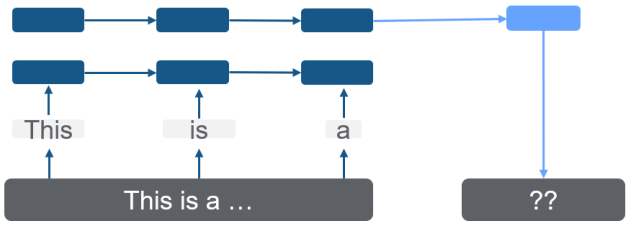



**FIGURE 7.3:** Unidirectional language model

a specific task. This is much more flexible, as no particular adjustments are needed. The method's disadvantage is that the general knowledge and relationship between words can get lost in the adjustment phase. This is called "catastrophic forgetting" (McCloskey and Cohen, 1989; French, 1999). Techniques for preventing catastrophic forgetting are freezing learning rates and regularization, mainly explained in the ULMFiT section.

Fine-tuning and feature extraction are the most extreme versions of adoption.

## 7.3 Models

In the following sections, the models **ELMo** (Peters et al., 2018), **ULMFiT** (Howard and Ruder, 2018), and **GPT** (Radford et al., 2018) are presented, which have shaped the "first wave" of transfer learning before bidirectional transformers like BERT have been developed and have become popular. All models have been presented in 2018 and are inductive sequential transfer learning models.

### 7.3.1 ELMo - The "new age" of embeddings

In 2018, Peters et al. (2018) from AllenNLP introduced **Embeddings from Language Models** (ELMo). Its most significant advance compared to previous models like word2vec and Glove (**chapter 3**), is that ELMo can handle the different meanings of a word in different contexts (**polysemy**).

One good example of polysemy is the word "mouse." In a technical context, it means the computer mouse, and in other contexts, it could mean a rodent. Previous models can just capture the semantical and syntactical characteris-

tics of the word. ELMo can also capture the varying meaning of the word in different contexts (here: computer mouse or rodent).

ELMo can be divided into the steps pretraining and adoption:

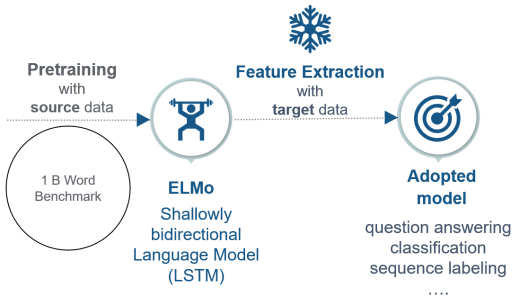



**FIGURE 7.4:** Feature extraction in ELMo.

It is pretrained on the 1 Billion Word Benchmark, a standard dataset for language modeling from Google. ELMo's model architecture is a shallowly bidirectional language model. The meaning of "shallowly bidirectional" will be explained in the following section. Afterward, the pretrained model is frozen. That means that all learning rates are set to 0. In the adoption phase, a linear combination of the internal states is learned in an additional layer. This representation is different for every task.

#### 7.3.1.1 Pretraining: Bidirectional language model (biLM)

ELMo is based on a two-layer bidirectional LSTM, the shallow concatenation of independently trained left-to-right and right-to-left multi-layer LSTMs. Bidirectional is, in this case, misleading, as the two steps happen independently from each other:

As shown in figure 7.5, $N$ tokens are taken into a one-layer bidirectional LSTM. Every token $t_k$ has a context-independent character-based representation called $x_k^{LM}$. Following, the first token would be $x_1^{LM}$, and the last one would be $x_N^{LM}$. By either token embedding or a CNN over characters, these representations are computed. These are the first representations of the words in the model.

Additionally, there are per layer and token two additional representations; one from the forward language model and one from the backward language model.

A forward language model calculates the probability of a sequential token $t_k$ at the position $k$ with the provided history (previous tokens) $t_1, \dots, t_{k-1}$ with:

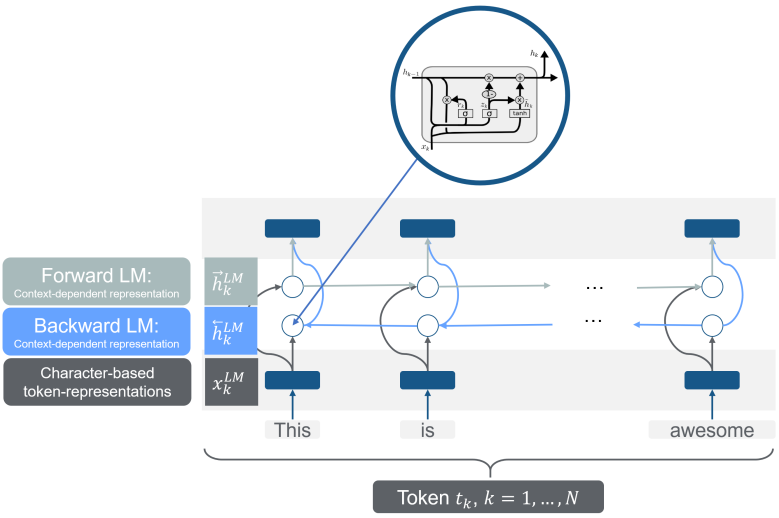



**FIGURE 7.5:** Bidirectional LSTM as a pretrained model. Source: based on Peters et al. (2018) and lst (2015).

$$p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k | t_1, t_2, \ldots, t_{k-1})$$

The backward language can be defined accordingly to the forward language model. It does not take the previous tokens, but the upcoming tokens, into account to calculate the joint distributions of the tokens.

$$p(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} p(t_k | t_{k+1}, t_{k+2}, \ldots, t_N)$$

These context-dependent representation are notated as $\overrightarrow{\mathbf{h}}_{k,j}^{LM}$ in the forward direction and as $\overleftarrow{\mathbf{h}}_{k,j}^{LM}$ in the backward direction.

Peters et al. (2018) have chosen a bidirectional LSTM with two layers, shown in figure 7.6. As with every layer, two additional context-dependent representations are added. This two-layer bidirectional model has five representations per token ($2L + 1 = 5$).

In the forward direction, a next token $t_{k+1}$ can predict the top layer $\overrightarrow{\mathbf{h}}_{k,L}^{LM}$ with a Softmax layer. In the biLM the directions are combined and optimized with a log likelihood:

$$\sum_{k=1}^{N} \left( \log p\left(t_k | t_1, \ldots, t_{k-1}; \Theta_x, \vec{\Theta}_{LSTM}, \Theta_s\right) + \log p\left(t_k | t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s\right) \right)$$

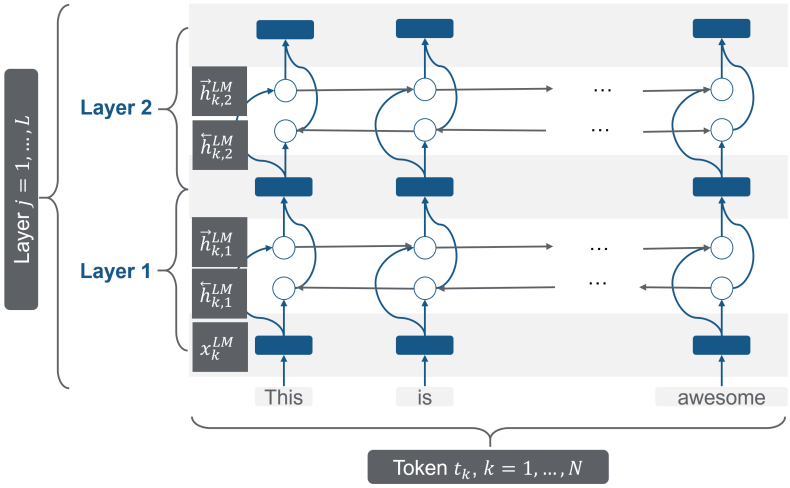



**FIGURE 7.6:** Bidirectional LSTM as a pretrained model. Source: based on Peters et al. (2018).

where $\Theta_s$ are the parameters for the token representations and $\Theta_x$ are the parameters of the Softmax-layer.

This model architecture is trained with the extensive 1B Word data set to calculate representations. For every single token $R_k$ is calculated, which is a set of representations $R_k$ for every token $t_k$: all forward and backward context-dependent representations and also the character-based token representation $x_k$:

$$\begin{aligned} R_k &= \left\{\mathbf{x}^{LM}_k, \overrightarrow{\mathbf{h}}^{LM}_{k,j}, \overleftarrow{\mathbf{h}}^{LM}_{k,j} | j = 1, \dots, L\right\} \\ &= \left\{\mathbf{h}^{LM}_{k,j} | j = 0, \dots, L\right\} \end{aligned}.$$

This pretrained model, including the representation for every token, is here available for download: `https://allennlp.org/elmo`.

#### 7.3.1.2 Adoption: Extracting ELMo representation

After the biLM is pretrained, the model can be adapted to the target task. ELMo adopts the target by **feature extraction**; the pretrained model is frozen (all learning rates are set to 0), and the task-specific ELMo representations are calculated.

With the target data, $s^{task}_j$ is trained, which are soft-max-normalized weights for every layer. $\gamma_{task}$ is a task-specific parameter that scales the entire ELMo vector. After training these parameters, ELMo has a task-specific representation for every token $t_k$, a linear combination of the internal representations.

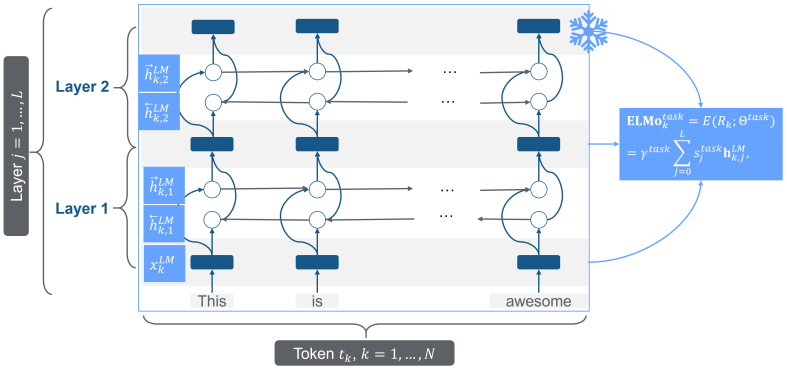



**FIGURE 7.7:** Bidirectional LSTM as a pretrained model. Source: based on Peters et al. (2018) and lst (2015).

The ELMo specific task is formulated by

$$\mathbf{ELMo}^{task}_k = E\left(R_k; \Theta^{task}\right) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}^{LM}_{k,j},$$

where $\gamma$ is the optimization parameter which allows scaling the model, $s_j^{task}$ are "softmax-normalized weights," and $R_k$ is the representation of the tokens $t_k$. For every task (e.g., question answering, sentiment analysis), the ELMo representation needs a task-specific calculation.

#### 7.3.1.3 ELMo: Summary and further links

To summarize, ELMo is a shallowly bidirectional language model, which is a two-layer LSTM. It is pretrained on the 1 Billion word data set and then adopted with feature extraction, a linear combination of the internal states.

In this way, ELMo can create unique, character-based, and deep word representations. The unique representations are adopted to every task. This is a considerable enhancement compared to the previous embedding models, which did not take the context of the words into account (**chapter 3**).

According to the ELMo paper authors, ELMo saves semantic - context-dependent features - information more in the higher layer and syntactic information more in the lower layers. That is also why the weighting in every task should be different.

ELMo is suitable for every NLP task, and by adding ELMo due to Peters et al. (2018)'s paper, a relative error reduction of 6-20% can be achieved.

**Interesting Links**

- Exploring contextual meanings with ELMo https://towardsdatascience.com/elmo-contextual-language-embedding-335de2268604
- How to implement text classification with ELMo https://www.analyticsvidhya.com/blog/2019/03/learn-to-use-elmo-to-extract-features-from-text/
- Illustrations of ELMo http://jalammar.github.io/illustrated-bert/

### 7.3.2 ULMFiT - cutting-edge model using LSTMs

In the same year, Howard and Ruder (2018) proposed **Universal language model fine-tuning (ULMFiT)**, which exceeded many of the cutting-edge models in text classification. It decreased the error by 18-24% on most of the datasets.

Also, ULMFiT can be divided into the steps of pretraining and adoption:

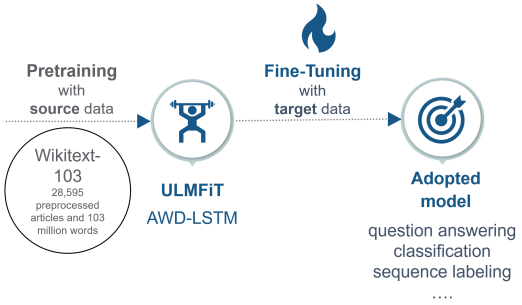



**FIGURE 7.8:** Fine-Tuning in ULMFiT.

ULMFiT is based on an AWD-LSTM (ASGD Weight-Dropped LSTM) combined with fine-tuning in the adoption where novel techniques like "discriminative fine-tuning," "slanted triangular learning rates," and "gradual unfreezing of layers." were introduced. Hence, it can fine-tune a generalized language model to a specific language model for a target task.

ULMFiT follows three steps to achieve its notable transfer learning results:

1. Language Model pretraining
2. Language Model fine-tuning (Adoption I)
3. Classifier fine-tuning (Adoption II)

#### 7.3.2.1 Pretraining: AWD-LSTM

As language models with many parameters tend to overfit, Merity et al. (2017) introduced the **AWD-LSTM**, a highly effective version of the Long Short Term Memory (LSTM, **chapter 4**). The Dropconnect Algorithm and the Non-monotonically Triggered ASGD (NT-ASGD) are two main improvements of this model architecture.

As in figure 7.9, the **Dropconnect Algorithm** (Wan et al., 2013) regularizes the LSTM and prevents overfitting by setting the activation of units randomly to zero with a predetermined probability of $p$. By this, only a subset of the units from the previous layer is passed to every unit. However, by using this method also long-term dependencies can go lost. That is why the algorithm drops weights and not the activations in the end with the probability of $1 - p$. As the weights are set to zero, the drop connect algorithm reduces the information loss while reducing overfitting.

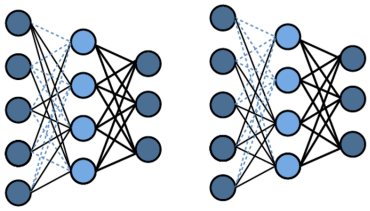


**FIGURE 7.9:** Dropconnect Algorithm.

To improve the optimization in the AWD-LSTM further, Merity et al. (2017) introduced the **Non-monotonically Triggered Average SGD** (or NT-ASGD), which is a new variant of Average Stochastic Gradient Descent (ASDG).

The Average Stochastic gradient descent takes a gradient descent step like a standard gradient descent algorithm. However, it also takes the weight of the previous iterations into account and returns the average. On the contrary, the NT-ASGD only takes the averaged previous iterations into account if the validation metric does not improve for a fixed amount of steps. Subsequently, the SGD turns into an ASGD if there is no improvement for $n$ steps.

Additionally, several other regularization and data efficiency methods are proposed in the paper:

- Variable Length Backpropaation Sequences (BPTT),
- Variational Dropout,
- Embedding Dropout,
- Reduction in Embedding Size,
- Activation Regularization, and

- Temporal Activation Regularization.

For further information, the paper of Merity et al. (2017) is a great way to start.

The AWD-LSTM (language model) is trained on general-domain data like the Wikipedia data set. This trained model can be downloaded here: `https://docs.fast.ai/text.html`.

#### 7.3.2.2 Adoption I: LM fine-tuning

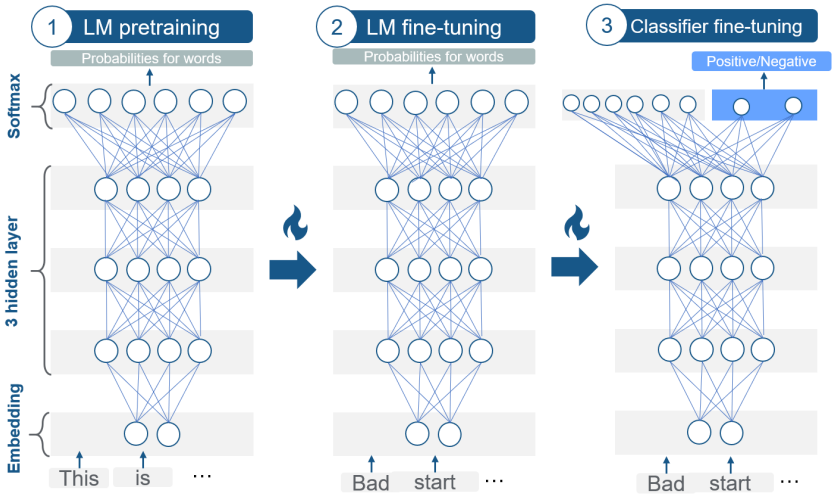
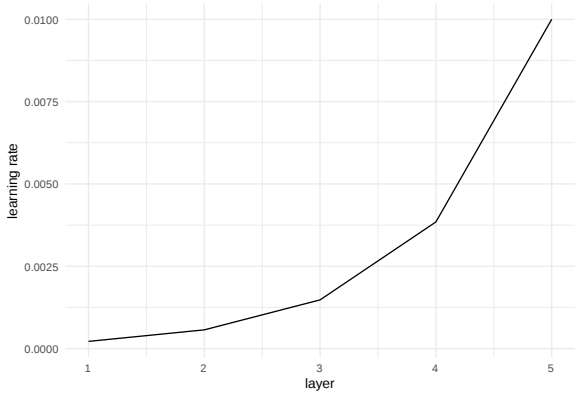


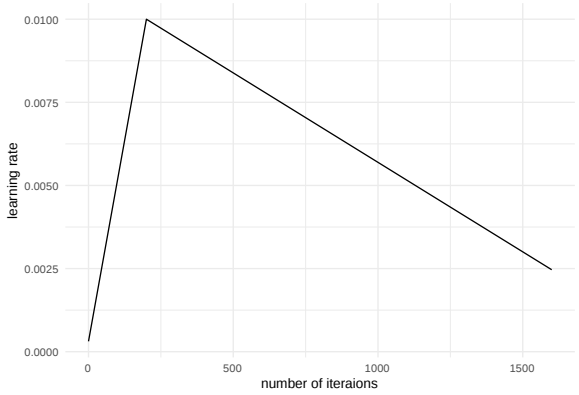

**FIGURE 7.10:** Three steps of ULMFiT. Source: based on Howard and Ruder (2018).

The model is fine-tuned on the tasks' dataset. For this purposed Howard and Ruder (2018) proposed two training techniques to stabilize the fine-tuning process:

**Discriminative fine-tuning** uses that distinctive layers of the LM capture distinct types of information. It is proposed to tune each layer with different learning rates. The higher the layer, the higher the learning rate:




On the other side, **slanted triangular learning rates** (STLR) are particular learning rate scheduling that first linearly increases the learning rate, and then gradually declines after a cut. That leads to an abrupt increase and a more extensive decay:




The learning rates $\eta_t$ are calculated with the number of iterations $T$:

$$
\begin{aligned}
cut &= \lfloor T \cdot cut\_frac \rfloor \\
p &= \begin{cases} t/cut, & \text{if } t < cut \\ 1 - \frac{t-cut}{cut \cdot (1/cut\_frac - 1)}, & \text{otherwise} \end{cases} \\
\eta_t &= \eta_{\max} \cdot \frac{1 + p \cdot (ratio - 1)}{\text{ratio}},
\end{aligned}
$$

where $cut\_frac$ is the increasing learning rate factor, $cut$ the iteration where the decreasing is started, $p$ the fraction of the number of iterations that are

increased or decreased, *ratio* is the ratio the difference between the lowest and highest learning rate

By these two changes in the learning rates depending on the iteration and the layer, information in the higher layers can learn more specific knowledge and lower layers to keep general language knowledge.

#### 7.3.2.3 Adoption II: Classifier fine-tuning

In the last step, the language model is expanded with two common feed-forward layers and a softmax normalization at the end to predict a target label distribution. Again, two new techniques are added:

- **Concat pooling**: As in text classification, single words are essential, concat pooling save the mean and the max representations. By that, theses signal words do not get lost. For example, good or bad are perfect indicators of sentiment. If the fine-tuning loses them, they are still saved in the GPU with concat pooling.
- **Gradual unfreezing**: A common problem of retraining the model is losing information about the general data (the Wikipedia dataset): **"catastrophic forgetting."** Hence, with gradual unfreezing, the model will be trained step by step, starting from the last layer. So first, all layers are "frozen" except the last layer. In every step, one additional layer is "unfrozen." By that, the specific knowledge which is in the higher layers is adapted more to the target task than the first layer, which contains the most general knowledge according to the authors of the ULMFiT paper.

With these improvements, the model is trained to sentiment texts on a particular target task.

#### 7.3.2.4 ULMFiT: Summary and further links

To sum up, ULMFiT is pretrained with the Wikitext-103 dataset on a highly efficient version of an LSTM. Then it is fine-tuned in two steps on the target task with the target data. ULMFiT achieved, in this way, a well-working form of fine-tuning in NLP. Here the last layer contains the most specific knowledge, and the first layer the most general knowledge. To contain this, the regularization methods explained are used. Although the paper is called ULMFIT for text classification, the pretrained model can also be used for any other task. Maybe the fine-tuning methods must be adapted. Due to the paper, ULMFiT could decrease the error by incredible 18-24% in text classification.

**Interesting Links**

- Tutorial on how to use ULMFiT for Text Classification https://www.analyticsvidhya.com/blog/2018/11/tutorial-text-classification-ulmfit-fastai-library/ or https://medium.com/technonerds/using-fastais-ulmfit-to-make-a-state-of-the-art-multi-label-text-classifier-bf54e2943e83
- Another students' seminar website about ULMFiT https://humboldt-wi.github.io/blog/research/information_systems_1819/group4_ulmfit/

### 7.3.3 GPT - First step towards transformers

Radford et al. (2018) from Open AI published **Generative Pretraining** (GPT).

In the context of inductive sequential transfer learning, the training steps of GPT can be divided in:

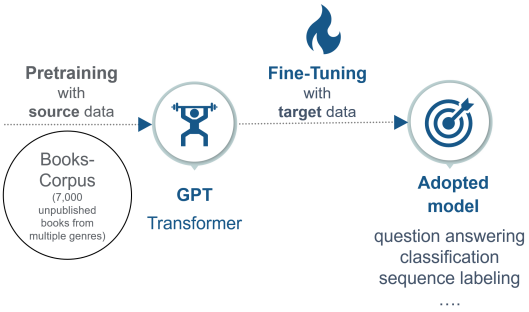



**FIGURE 7.11:** Fine-tuning in GPT.

GPT is trained on the Book-corpus data set, which is a dataset of 7,000 unpublished books. The underlying model architecture is a so-called transformer. This model is then fine-tuned on the target data.

Despite some similarities, GPT has significant differences to ELMo and ULMFiT:

- ELMo is based on word embeddings, whereas GPT is based on fine-tuning like ULMFiT.
- GPT uses a different model architecture. Instead of the multi-layer LSTM, GPT is a multi-layer transformer decoder. This model architecture will be explained in the upcoming chapters, as it is a significant step towards the state-of-the-art NLP models.

- In contrast to ELMo, that works character-wise, GPT uses tokens (sub-words) from the words.
- GPT is trained on a large number of data (especially the subsequent models like GPT-2 or GPT-3)

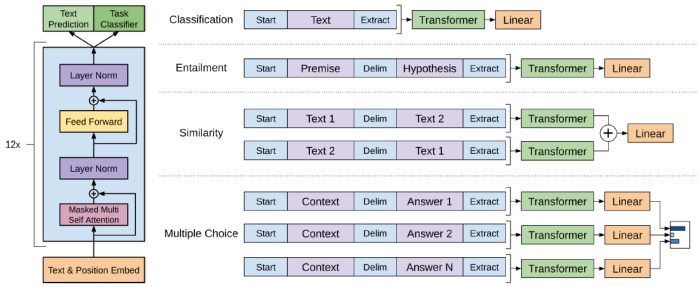



**FIGURE 7.12:** Basic model of GPT. Source: based on Radford et al. (2018).

As shown in figure 7.12, GPT is a uni-directional transformer-decoder-only with 12 layers with masked self-attention.

As the basic idea of transformers is discussed in the following chapters, further explanations of the functionality of the transformer model architectures will follow (**chapter 8** and **chapter 9**)

#### 7.3.3.1 GPT: Summary

To sum up, GPT is a multi-layer transformer-decoder with task-aware input transformations GPT is trained on the Book-Corpus data set, where the input is tokenized as a sub-word. Like the other models, GPT is unidirectional and suitable for any NLP task. This new architecture leads to improvements in many fields like 8.9% in commonsense reading, 5.7% in question answering, and 1.5% in textual entailment.

## 7.4 Summary

In 2018, a new generation of NLP models had been published, as transfer learning mainly pushed further enhancements from computer vision. The **main advances** of these models are

- due to the use of **transfer learning** the training for the target task needs less time and less target specific data,
- **ELMo** adds the contextualization to word embeddings,
- **ULMFiT** introduces many ideas like fine-tuning, which lowered the error rate notable, and
- **GPT** first uses the transformer model architecture, which cutting-edge NLP models use.

Besides, many features of these models show high **potential for improvements**:

- All models are **not genuinely bi-directional**, as ULMFiT and GPT are uni-directional, and ELMo is a concatenation of a right-to-left and left-to-right LSTM. As the human language understanding is bidirectional, bidirectional models represent the language more precisely.
- ELMo uses a character-based **model input**, and **ULMFit** uses a word-based **model input**. **GPT** and following transformer-based models use **tokenized** words (subwords), taking advantage of both other model inputs.
- ULMFiT and ELMo are **based on LSTMs**, whereas the transformer-based model architecture of GPT has many advantages like parallelization and subsequent performance improvements.
- Starting with 12 layers in GPT, the models get deeper and use more parameters and data for the pretraining.

In the next chapter, the main idea behind transformers, self-attention, is explained. More popular state-of-art models are presented in **chapter 9**.

# 8

# *Attention and Self-Attention for NLP*

*Authors: Joshua Wagner*

*Supervisor: Matthias Aßenmacher*

Attention and Self-Attention models were some of the most influential developments in NLP. The first part of this chapter is an overview of attention and different attention mechanisms. The second part focuses on self-attention which enabled the commonly used models for transfer learning that are used today. The final part of the chapter discusses the developments made with Self-Attention and the most common transfer learning architecture today, the Transformer.

## 8.1 Attention

In this part of the chapter, we revisit the Encoder-Decoder architecture that was introduced in chapter 3[1]. We focus on the improvements that were made with the development of attention mechanisms on the example of neural machine translation (NMT).

As seen in chapter 3, vanilla Encoder-Decoder architecture passes only the last hidden state from the encoder to the decoder. This leads to the problem that information has to be compressed into a fixed length vector and information can be lost in this compression. Especially information found early in the sequence tends to be "forgotten" after the entire sequence is processed. The addition of bi-directional layers remedies this by processing the input in reversed order. While this helps for shorter sequences, the problem still persists for long input sequences. The development of attention enables the decoder to attend to the whole sequence and thus use the context of the entire sequence during the decoding step.

[1] ./recurrent-neural-networks-and-their-applications-in-nlp.html

### 8.1.1 Bahdanau-Attention

In 2014, Bahdanau et al. (2014) proposed attention to fix the information problem that the early encoder-decoder architecture faced. Decoders without attention are trained to predict $y_t$ given a fixed length context vector $c$ and all earlier predicted words $\{y_t, \dots, y_{t-1}\}$. The fixed length context vector is computed with

$$c = q(\{h_1, \dots, h_T\})$$

where $h_1, \dots, h_T$ are the the hidden states of the encoder for the input sequence $x_1, \dots, x_T$ and $q$ is a non-linear function. Sutskever et al. (2014) for example used $c = q(\{h_1, \dots, h_T\}) = h_T$ as their non-linear transformation which remains a popular choice for architectures without attention. It is also commonly used for the initialisation of the decoder hidden states.

Attention changes the context vector $c$ that a decoder uses for translation from a fixed length vector $c$ of a sequence of hidden states $h_1, \dots, h_T$ to a sequence of context vectors $c_i$. The hidden state $h_i$ has a strong focus on the $i$-th word in the input sequence and its surroundings. If a bi-directional encoder is used, each $h_i$ is computed by a concatenation of the forward $\overrightarrow{h_i}$ and backward $\overleftarrow{h_i}$ hidden states:

$$h_i = [\overrightarrow{h_i}; \overleftarrow{h_i}], i = 1, \dots, n$$

These new variable context vectors $c_i$ are used for the computation of the decoder hidden state $s_t$. At time-point $t$ it is computed as $s_t = f(s_{t-1}, y_{t-1}, c_t)$ where $f$ is the function resulting from the use of a LSTM- or GRU-cell. The context vector $c_t$ is computed as a weighted sum of the hidden states $h_1, \dots, h_{T_x}$:

$$c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i.$$

Each hidden state $h_i$ is weighted by a $\alpha_{t,i}$. The weight $\alpha_{t,i}$ for each hidden state $h_i$ is also called the alignment score. These alignment scores are computed as:

$$\alpha_{t,i} = align(y_t, x_i) = \frac{exp(score(s_{t-1}, h_i))}{\sum_{i'=1}^{n} exp(score(s_{t-1}, h_{i'}))}$$

with $s_{t-1}$ being the hidden state of the decoder at time-step $t-1$. The alignment score $\alpha_{t,i}$ models how well input $x_i$ and output $y_t$ match and assigns the weight to $h_i$. Bahdanau et al. (2014) parametrize their alignment score with a single-hidden-layer feed-forward neural network which is jointly trained with the other parts of the architecture. The score function used by Bahdanau et al. is given as

$$score(s_t, h_i) = v_\alpha^T tanh(\mathbf{W}_\alpha [s_t; h_i])$$

were *tanh* is used as a non-linear activation function and $v_\alpha$ and $W_\alpha$ are the weight matrices to be learned by the alignment model. The alignment score function is called "concat" in Luong et al. (2015b) and "additive attention" in Vaswani et al. (2017) because $s_t$ and $h_i$ are concatenated just like the forward and backward hidden states seen above. The attention model proposed by Bahdanau et al. is also called a global attention model as it attends to every input in the sequence. Another name for Bahdanaus attention model is soft attention because the attention is spread thinly/weakly/softly over the input and does not have an inherent hard focus on specific inputs. A nice by-product of attention mechanisms is the matrix of alignment scores which can be visualised to show the correlation between source and target words as seen in 8.1.

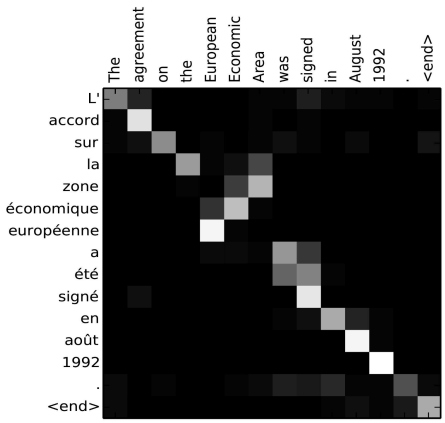



**FIGURE 8.1:** Alignment Matrix visualised for a French to English translation. White squares indicate high aligment weights between input and output. Image source: Fig 3 in (Bahdanau et al., 2014)

### 8.1.2 Luong-Attention

While Bahdanau et al. (2014) were the first to use attention in neural machine translation, Luong et al. (2015b) were the first to explore different attention mechanisms and their impact on NMT. Luong et al. also generalise the attention mechanism for the decoder which enables a quick switch between different attention functions. Bahdanau et al. only consider

$$score(s_t, h_i) = v_\alpha^T tanh(\mathbf{W}_\alpha[s_t; h_i]),$$

while Luong et al. additionally introduce a general, a location-based and a dot-product score function for the global attention mechanism as described in table 8.1.2.

| Score-function | Name |
|---|---|
| $\text{score}(s_t, h_i) = s_t^\top \mathbf{W}_a h_i$ | General |
| $\alpha_{t,i} = \text{softmax}(\mathbf{W}_a s_t)$ | Location-based |
| $\text{score}(s_t, h_i) = s_t^\top h_i$ | Dot-product |

Table 8.1.2: Different score function proposed by Luong et al.

As Luong et al. (2015b) don't use a bidirectional encoder, they simplify the hidden state of the encoder from a concatenation of both forward and backward hidden states to only the hidden state at the top layer of both encoder and decoder.

The attention mechanisms seen above attend to the entire input sequence. While this fixes the problem of forgetful sequential models discussed in the beginning of the chapter, it also has the drawback that it is expensive and can potentially be impractical for long sequences e.g. the translation of entire paragraphs or documents. These problems encountered with global or soft attention mechanisms can be mitigated with a local or hard attention approach. While it was used by Xu et al. (2015) for caption generation of images with a CNN and by Gregor et al. (2015) for the generation of images, the first application and differentiable version for NMT is from Luong et al. (2015b).

Different from the global attention mechanism, the local attention mechanism at timestep $t$ first generates an aligned position $p_t$. The context vector is then computed as a weighted average over only the set of hidden states in a window $[p_t - D, p_t + D]$ with $D$ being an empirically selected parameter. This constrains the above introduced computation for the context vector to:

$$c_t = \sum_{i=p_t-D}^{p_t+D} \alpha_{t,i} h_i.$$

The parts outside of a sentence are ignored if the window crosses sentence boundaries. The computation of the context vector changes compared to the global model which can be seen in Figure 8.2.

Luong et al. introduce two different concepts for the computation of the alignment position $p_t$.

The first is the *monotonic* alignment(**local-m**). This approach sets $p_t = t$ with the assumption that both input and output sequences are roughly monotonically aligned.

The other approach, *predictive* alignment (**local-p**), predicts the aligned position with:

$$p_t = S \cdot sigmoid(v_p^\top tanh(W_p h_t))$$

where $W_p$ and $v_p$ are the parameters that are trained to predict the position.

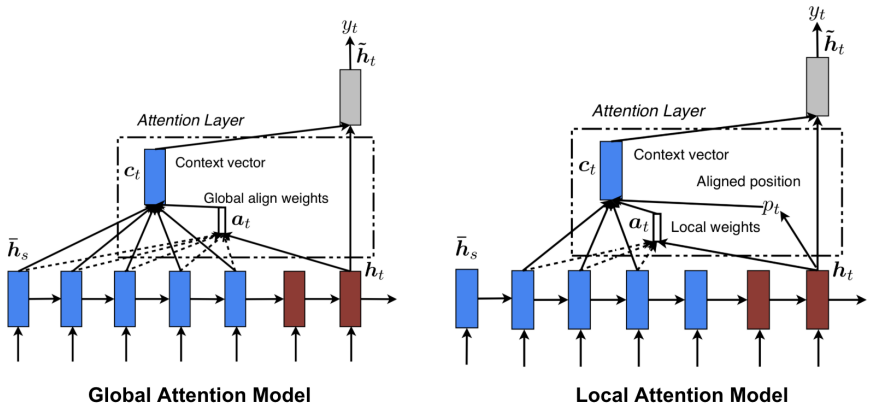



**FIGURE 8.2:** Global and local attention illustrated. Encoder in blue, Decoder in red. $\bar{h}_s$ and $h_t$ in the image correspond to $h_t$ and $s_t$ in the previous text. Additional computations and differences to previous described architecture is found in the next sub-chapter. Image source: Fig. 2 and 3 in (Luong et al., 2015b)

$S$ is the length of the input sentence which leads, with the additional sigmoid function, to $p_t \in [0, S]$. A Gaussian distribution centred around $p_t$ is placed by Luong et al. (2015b) to favour alignment points closer to $p_t$. This changes the alignment weights to:

$$\alpha_{t,i} = align(y_t, x_i) exp(-\frac{(i - p_t)^2}{2\sigma^2})$$

where the standard deviation is empirically set to $\sigma = \frac{D}{2}$. This utilization of $p_t$ to compute $\alpha_{t,i}$ allows the computation of backpropagation gradients for $W_p$ and $v_p$ and is thus "differentiable almost everywhere" Luong et al. (2015b) while being less computationally expensive than global attention.

### 8.1.3 Computational Difference between Luong- and Bahdanau-Attention

As previously mentioned, Luong et al. not only introduced different score functions in addition to Bahdanau et al.'s concatenation/additive score function, they also generalized the computation for the context vector $c_t$. In Bahdanau's version the attention mechanism computes the variable length context vector first which is then used as input for the decoder. This necessitates the use of the last decoder hidden state $s_{t-1}$ as input for the computation of the context vector $c_t$:

$$\alpha_{t,i} = align(y_t, x_i) = \frac{exp(score(s_{t-1}, h_i))}{\sum_{i'=1}^{n} exp(score(s_{t-1}, h_{i'}))}$$

Luong et al. compute their context vector with the current decoder hidden state $s_t$ and modify the decoder output with the context vector before it is processed by the last softmax layer. This allows for easier implementation of different score functions for the same attention mechanism. Implementations of both vary e.g. this version[2] of Bahdanau attention in Pytorch concatenates the context back in after the GRU while this version[3] for an NMT model with Bahdanau attention does not. Readers that are trying to avoid a headache can build upon this version[4] from Tensorflow which uses the **AttentionWrapper** function which handles the specifics of the implementation.

An illustration of a reduced version of the two different attention concepts can be found in Figure 8.3.

### 8.1.4 Attention Models

Attention was in the beginning not directly developed for RNNs or even NLP. Attention was first introduced in Graves et al. (2014) with a content-based attention mechanism (score$(s_t, h_i) =$ cosine$[s_t, h_i]$) for Neural Turing Machines. Their application for NLP related tasks were later developed by Luong et al. (2015b), Bahdanau et al. (2014) and Xu et al. (2015). Xu et al. (2015) were the first publication to differentiate between soft/global and hard/local attention mechanisms and did this in the context of Neural Image Caption with both mechanisms being close to what was used by Luong et al. in the previous section. Cheng et al. (2016) were the first to introduce the concept of self-attention, the third big category of attention mechanisms.

## 8.2 Self-Attention

Cheng et al. (2016) implement self-attention with a modified LSTM unit, the Long Short-Term Memory-Network (LSTMN). The LSTMN replaces the memory cell with a memory network to enable the storage of "contextual representation of each input token with a unique memory slot and the size

[2] https://github.com/spro/practical-pytorch/blob/master/seq2seq-translation/seq2seq-translation.ipynb

[3] https://www.tensorflow.org/tutorials/text/nmt_with_attention

[4] https://www.tensorflow.org/addons/tutorials/networks_seq2seq_nmt#defining_nmt_model

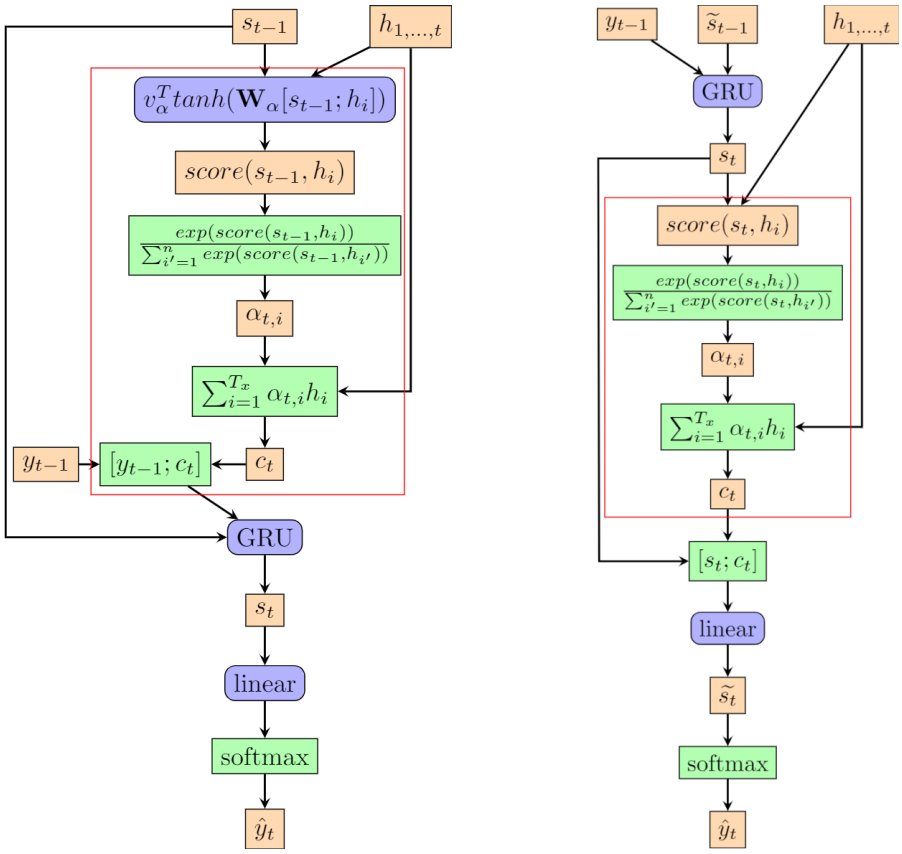



**FIGURE 8.3:** Bahdanau and Luong attention mechanisms illustrated. The attention layer is bounded in the red box. Orange denotes inputs, outputs and weights. Blue boxes are layers. Green boxes denote operations e.g. softmax or concatenation([x;y]).

of the memory grows with time until an upper bound of the memory span is reached" Cheng et al. (2016). Self-Attention, as the name implies, allows an encoder to attend to other parts of the input during processing as seen in Figure 8.4.

While the LSTMN introduced self-attention, it retains the drawbacks that come from the use of a RNN which are discussed at the end of the Transformer section. Vaswani et al. (2017) propose the Transformer architecture which uses self-attention extensively to circumvent these drawbacks.

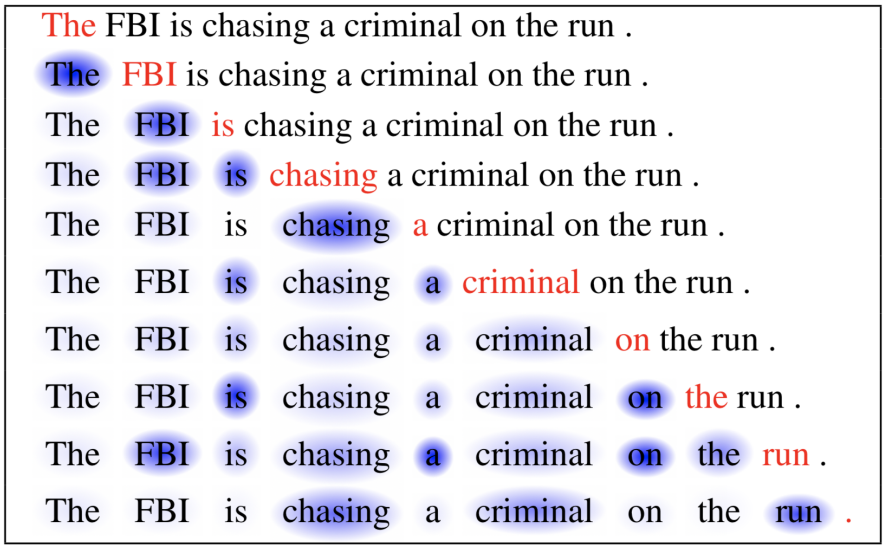



**FIGURE 8.4:** Illustration of the self-attention mechanism. Red indicates the currently fixated word, Blue represents the memories of previous words. Shading indicates the degree of memory activation. Image source: Fig. 1 in (Cheng et al., 2016).

### 8.2.1 The Transformer

RNNs were, prior to Transformers, the state-of-the-art model for machine translation, language modelling and other NLP tasks. But the sequential nature of a RNN precludes parallelization within training examples. This becomes critical at longer sequence lengths as memory constraints limit batching across examples. While much has been done to minimize these problems, they are inherent in the architecture and thus still remain. An attention mechanism allows the modelling of dependencies without regard for the distance in either input or output sequences. Most attention mechanisms, as seen in the previous sections of this chapter, use recurrent neural networks. This limits their usefulness for transfer learning because of the previously mentioned constraints that recurrent networks have. Models like ByteNet from Kalchbrenner et al. (2016b) and ConvS2S from Gehring et al. (2017) alleviate the problem with sequential models by using convolutional neural networks as basic building blocks. ConvS2S has a linear increase in number of operations to relate signals from two arbitrary input or output positions with growing distance. ByteNet has a logarithmical increase in number of operations needed. The Transformer architecture from Vaswani et al. (2017) achieves the relation of two signals with arbitrary positions in input or output with a constant number of operations. It was also the first model that relied entirely on self-

attention for the computation of representations of input or output without using sequence-aligned recurrent networks or convolutions.

While the Transformer architecture doesn't use recurrent or convolutional networks, it retains the popular encoder-decoder architecture.

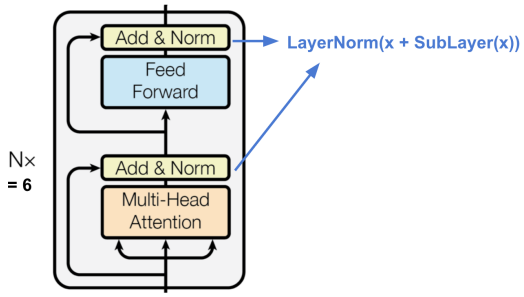



**FIGURE 8.5:** Single layer of the Encoder of a Transformer with two distinct sub-layers each with residual connections and a LayerNorm. Original image: (Vaswani et al., 2017), Additions and cropping: (Weng, 2018)

The encoder is composed of a stack of N = 6 identical layers. Each of these layers has two sub-layers: A multi-head self-attention mechanism and a position-wise fully connected feed-forward network. The sub-layers have a residual connection around the main components which is followed by a layer normalization. The output of each sub-layer is LayerNorm$(x + \text{Sublayer}(x))$ where Sublayer$(x)$ is the output of the function of the sublayer itself. All sub-layers and the embedding layer before the encoder/decoder produce outputs of $dim = d_{model} = 512$ to allow these residual connections to work. The position-wise feed-forward network used in the sublayer is applied to each position separately and identically. This network consists of two linear transformations with a ReLU activation function in between:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

The decoder is, as seen in Figure 8.6, composed of a stack of $N = 6$ identical layers.

It inserts, in addition to the two already known sub-layers from the encoder, a third sub-layer which also performs multi-head attention. This third sub-layer uses the encoder output as two of its three input values, which will be described in the next part of the chapter, for the multi-head attention. This sub-layer is in its function very close to the before seen attention mechanisms between encoders and decoders. It uses, same as the encoder, residual connections around each of the sub-layers. The decoder also uses a modified, masked self-attention sub-layer "to prevent positions from attending to subsequent positions" Vaswani et al. (2017). This, coupled with the fact that the

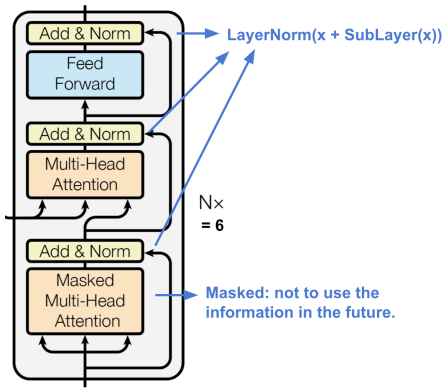



**FIGURE 8.6:** Decoder of a Transformer, Original image: (Vaswani et al., 2017), Additions and cropping: (Weng, 2018)

output embeddings are shifted by one position to the right ensures that the predictions for position $i$ only depend on previous known outputs.

As seen in 8.7, the Transformer uses positional encodings added to the embeddings so the model can make use of the order of the sequence. Vaswani et al. (2017) use the sine and cosine function of different frequencies:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

For further reasoning why these functions were chosen see Vaswani et al. (2017).

#### 8.2.1.1 The self-attention mechanism(s)

Vaswani et al. (2017) describe attention functions as "mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key". The *Q*uery and the *K*ey-*V*alue pairs are used in the newly proposed attention mechanism that is used in Transformers. These inputs for the attention mechanisms are obtained through multiplication of the general input to the encoder/decoder with different weight matrices as can be seen in figure 8.8.

These different transformations of the input are what enables the input to attend to itself. This in turn allows the model to learn about context. E.g. **bank** can mean very different things depending on the context which can be recognized by the self-attention mechanisms.

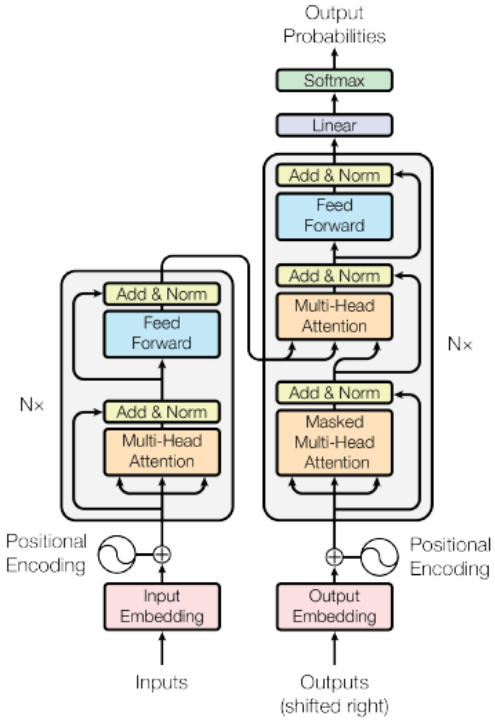



**FIGURE 8.7:** The Transformer-model architecture, Image Source: Fig. 1 in (Vaswani et al., 2017)

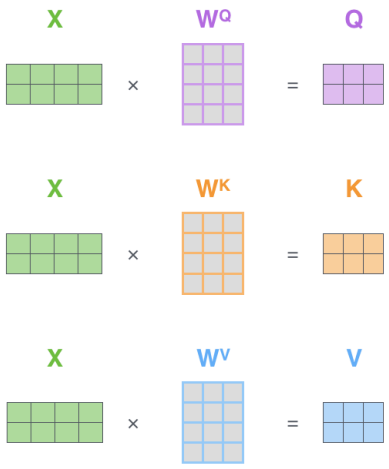



**FIGURE 8.8:** Input transformation used in a Transformer, $Q$ = Query, $K$ = Key, $V$ = Value. Image Source: (Alammar, 2018)

As seen in Figures 8.5, 8.6 and 8.7, the Transformer uses an attention mechanism called "Multi-Head Attention".

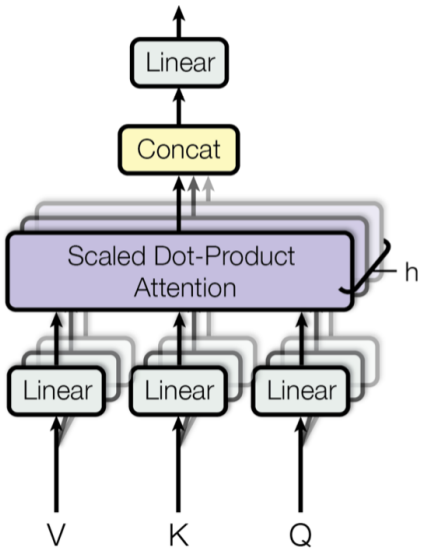



**FIGURE 8.9:** Multi-Head Attention, Image Source: Fig. 2 in (Vaswani et al., 2017)

The multi-head attention projects the queries, keys and values $h$ times instead of performing a single attention on $d_{model}$-dim. queries and key-value pairs. The projections are learned, linear and project to $d_k$, $d_k$ and $d_v$ dimensions. Next the new **scaled dot-product attention** is used on each of these to yield a $d_v$-dim. output. These values are then concatenated and projected to yield the final values as can be seen in 8.9. This multi-dimensionality allows the attention mechanism to jointly attend to different information from different representation at different positions. The multi-head attention can be written as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

and projections are the parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$. Vaswani et al. (2017) use $h = 8$ and $d_k = d_v = d_{model}/h = 64$. This reduced dimensionality leads to a reduction in computational cost that is similar to that of a single scaled-dot-product attention head with the full initial dimensionality of 512.

The *scaled dot-product attention* is, as the name suggests, just a scaled version of the dot-product attention seen previously in this chapter.

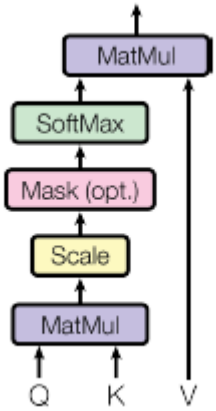



**FIGURE 8.10:** Scaled Dot-Product Attention, Image Source: Fig. 2 in (Vaswani et al., 2017)

The optional *Mask*-function seen in Fig. 8.10 is only used in the masked-multi-head attention of the decoder. The querys and keys are of dim. $d_k$ and the values are of dim. $d_v$. The attention is for practical reasons computed for a set of queries, $Q$. The keys and values are thus also used in matrix format, $K$ and $V$. The matrix of outputs is then computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V$$

where $\text{Attention}(Q, K, V)$ corresponds to an non-projected head of multi-head attention. These attention mechanisms allow Transformers to learn different and distant dependencies in language and are thus a good candidate for transfer learning.

#### 8.2.1.2 Complexity for different attention models

The architecture of a Transformer allows for parallel encoding of every part of the input at the same time. This also enables the modelling of long-range dependencies regardless of the distance. The Transformer architecture, as a minimum, only needs a constant number of sequential operations regardless of input length $n$ due to extensive parallelization as can be seen in Table 8.2.1.2. A RNN based model in comparison needs a linearly scaling number of sequential operations due to its architecture. The Maximum Path Length between long-range dependencies for a transformer is $O(1)$ while the RNN again has $O(n)$ due to its sequential input reading.

The fast modelling of long-range dependencies and the multiple attention heads which learn different dependencies makes Transformers a favourable choice for Transfer Learning. The transfer learning models that were developed from the Transformer architecture enabled models which were trained on more data to gain a deeper understanding about language and are state-of-the-art today (June, 2020).

They are also a popular research topic as they are quadratically scaling, with regard to input length, complexity per layer inhibits their use for very long sequences and makes them time consuming to train. This quadratic complexity comes from the self-attention mechanism $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V$. The used softmax needs to calculate the attention score between the currently processed input $x_i$ and every other input $x_j$ for each $j \in 1, \dots, n$ and $i \in 1, \dots, n$. This limits the length of the context that Transformers can process and increases the time they need for training in practical uses. The computational costs are especially severe in very long tasks.

In the last few years new variations of the vanilla transformer, as described previously in the chapter, were published which lower the computational cost (e.g. Shen et al. (2018) and Choromanski et al. (2020)). Some of these variations of the Transformer architecture managed to decrease the complexity from quadratic to $O(N\sqrt{N})$ (Child et al. (2019)), $O(N \log N)$ (Kitaev et al. (2020)) or $O(n)$ (Wang et al. (2020)) as seen in Table 8.2.1.2.

| Layer-type | Complexity per Layer | Sequential Operations |
|---|---|---|
| Recurrent | $O(n \cdot d^2)$ | $O(n)$ |
| Convolutional | $O(k \cdot n \cdot d^2)$ | $O(1)$ |
| Transfomer | $O(n^2 \cdot d)$ | $O(1)$ |
| Sparse Transfomer | $O(n\sqrt{n})$ | $O(1)$ |
| Reformer | $O(n \log(n))$ | $O(\log(n))$ |
| Linformer | $O(n)$ | $O(1)$ |
| Linear Transformer | $O(n)$ | $O(1)$ |

Table 8.2.1.2 : Complexity per layer and the number of sequential operations. Sparse Transformers are from Child et al. (2019), Reformer from Kitaev et al. (2020), Linformer from Wang et al. (2020) and Linear Transformers from Katharopoulos et al. (2020). $n$ is the sequence length, $k$ the kernel size, $d$ the representation dimension and $r$ the size of the neighbourhood in restricted self-attention. Source: Vaswani et al. (2017) (Table 1) and Wang et al. (2020) (Table 1)

### 8.2.2 Transformers as RNNs

A version from Katharopoulos et al. (2020) uses a linear attention mechanism for autoregressive tasks. The vanilla Transformer uses, as described above, the softmax to calculate the attention between values. This can also be seen as a similarity function. $V_i' = \frac{\sum_{j=1}^{N} \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^{N} \text{sim}(Q_i, K_j)}$ is equal to the row-wise calculation of $\text{Attention}(Q, K, V) = V' = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V$ if $\text{sim}(q, k) = \exp(\frac{q^\top k}{\sqrt{d_k}})$. While we can use the exponentiated dot-product to generate the vanilla scaled-dot-product attention, we can use any non-negative function, including kernels, as similarity function to generate a new attention mechanism Katharopoulos et al. (2020). Given the feature representation $\phi(x)$ of such a kernel, the previous row-wise calculation can be rewritten as $V_i' = \frac{\phi(Q_i)^\top \sum_{j=1}^{N} \phi(K_j) V_j^\top}{\phi(Q_i)^\top \sum_{j=1}^{N} \phi(K_j)}$. Or in vectorized form: $\phi(Q)(\phi(K)^\top V)$ where the feature map $\phi(\cdot)$ is applied row-wise to the matrices. This new formulation shows that the computation with a feature map allows for linear time and memory scaling $O(n)$ because $\sum_{j=1}^{N} \phi(K_j) V_j^\top$ and $\sum_{j=1}^{N} \phi(K_j)$ can be both computed once and reused for every row of the query (Katharopoulos et al. (2020)). No finite dimensional feature map of the exponential function exists, which makes a linearisation of the softmax attention impossible. This forces the use of a polynomial kernel which has been shown to work equally well with the exponential kernel Tsai et al. (2019). This results in a computational cost of $O(ND^2M)$ which is favourable if $N > D^2$. Katharopoulos et al. (2020) suggest $\phi(x) = \text{elu}(x) + 1$ for $N < D^2$.

For autoregressive tasks we want an attention mechanism that can't look ahead of its position. Vaswani et al. (2017) introduced masked-self-attention for their decoder, which is adapted for linearised attention. The addition of masking changes the previous formulation to $V_i' = \frac{\sum_{j=1}^{i} \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^{i} \text{sim}(Q_i, K_j)}$. With the linearisation through kernels we get $V_i' = \frac{\phi(Q_i)^\top \sum_{j=1}^{i} \phi(K_j) V_j^\top}{\phi(Q_i)^\top \sum_{j=1}^{i} \phi(K_j)}$.

$S_i = \sum_{j=1}^{i} \phi(K_j) V_j^\top$ and $Z_i = \sum_{j=1}^{i} \phi(K_j)$ can be used to simplify the formula to:

$$V_i' = \frac{\phi(Q_i)^\top Si}{\phi(Q_i)^\top Z_i}$$

where $S_i$ and $Z_i$ can be computed from $S_{i-1}$ and $Z_{i-1}$ which allows the linearised attention with masking to scale linearly with respect to the sequence length. The derivation of the numerator as cumulative sums allows for the computation in linear time and constant memory, which leads to computational complexity of $O(NCM)$ and memory $O(N \max(C, d_k))$ where $C$ is the dimensionality of the feature map Katharopoulos et al. (2020).

Given the previous formalization of feature maps to replace the softmax we can rewrite the layers of a Transformers as a RNN. This RNN has two hidden states, the attention state $s$ and the normalizer state $z$ with subscripts denoting the timestep in recurrence. With $s_0 = 0$ and $z_0 = 0$ we can define $s_i$ as $s_i s_{i-1} + \phi(x_i W_K)(x_i W_V)^\top$ and $z_i$ as $z_i = z_{i-1} + \phi(x_i W_K)$ with $x_i$ as the $i$-th input for the layer. The $i$-th output $y_i$ can then be written as $y_i = f_l(\frac{\phi(x_i W_Q)^\top s_i}{\phi(x_i W_Q)^\top z_i} + x_i)$ where $f_l$ is the function given by the feed-forward network of a Transformer layer.

This shows that the Transformer layers can be rewritten into RNN layers, for all similarity functions that can be represented with $\phi$ (Katharopoulos et al. (2020)), which are the first models that used attention for NLP tasks.

# 9

# *Transfer Learning for NLP II*

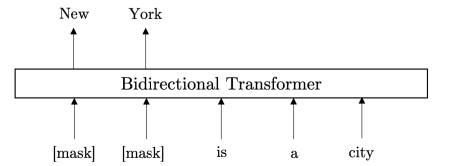
*Authors: Bailan He*

*Supervisor: M. Aßenmacher*

Unsupervised representation learning has been highly successful in NLP. Typically, these methods first pre-train neural networks on large-scale unlabeled text corpora and then fine-tune the models on downstream tasks. Here we introduce the three remarkable models, BERT, GPT-2, and XLNet. Transformers[1] is an excellent Github repository, where readers can find their implementations.

## 9.1 Bidirectional Encoder Representations from Transformers (BERT)

### 9.1.1 Autoencoding




$$\log p(\bar{\mathbf{x}}|\hat{\mathbf{x}}) = \sum_{t=1}^{T} \text{mask}_t \log p(x_t|\hat{\mathbf{x}})$$

**FIGURE 9.1:** Autoencoding

Autoencoding(AE) have been most successful pre-training objectives and figure 9.1 shows the modeling of it. AE based model does not perform explicit

[1] %22https://github.com/huggingface/transformers%22

density estimation but instead aims to reconstruct the original data from corrupted input. Specifically, given a text sequence $X = (x_1, ..., x_T)$, AE factorizes the log-likelihood into a partial sum

$$logp(\bar{X}|\hat{X}) = \sum_{t=1}^{T} mask_t p(x_t|\hat{X})$$

, where $mask_t$ is an indicator to show if a token is masked,Yang et al. (2019).

The training objective is to reconstruct the masked tokens $\bar{X}$ given the sequence $\hat{X}$. AE tries to find the best model $P$ to predict the masked tokens. The BERT introduced next is one of the most important AE models.

### 9.1.2 Introduction of BERT

BERT is published by researchers at Google AI in 2018. It is regarded as a milestone in the NLP community by proposing a bidirectional Language model based on Transformer.

BERT is a notable example of AE and it uses the structure of the AE model just mentioned, which means that some tokens in the training data will be masked. In BERT 15% of tokens are replaced by a special symbol [MASK], and the model is trained to reconstruct the original sequence from the masked tokens. By contrast with previous efforts that looked at a text sequence either from left to right(RNN **chapter 5**) or combined left-to-right and right-to-left training (ELMO **chapter 8**), the Transformer Encoder (**chapter 9**) utilizes bidirectional contexts simultaneously. Therefore BERT uses the Transformer Encoder as the structure of the pre-train model and addresses the unidirectional constraints by proposing new pre-training objectives: the Masked Language Model(MLM) and Next-sentence Prediction(NSP).

The special pre-training structure of BERT enables it to pre-train deep bidirectional representations and after fine-tuning based the representations, BERT advances state-of-the-art performance for eleven NLP tasks. Its unique structural ideas and excellent model performance make BERT the most important NLP model at present.

### 9.1.3 Input Representation of BERT

For NLP models, the input representation of the sequence is the basis of excellent model performance, many scholars have conducted in-depth research on methods to obtain word embeddings for a long time **chapter 4**. As for BERT, due to the model structure, the input representations need to be able to unambiguously represent both a single text sentence or a pair of text sentences in one token sequence. For a given token, its input representation is

constructed by summing the corresponding token, segment, and position embeddings, Devlin et al. (2018).

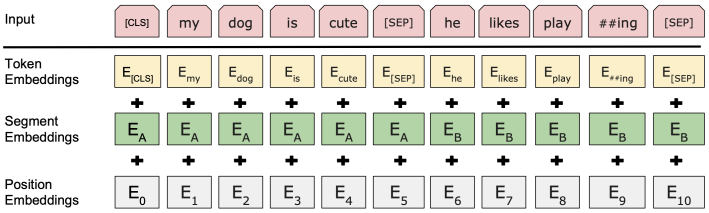



**FIGURE 9.2:** BERT input representation

Figure 9.2 is the visual representation of input representations of BERT tokens. The specifics are:

- As for Token Embeddings (yellow block): Use WordPiece embeddings Wu et al. (2016) with a 30,000 token vocabulary and split word pieces denoted with ##. e.g.[ playing = play and ##ing ] and the first token of every sequence is always the special classification embedding [CLS]. For non-classification tasks, this vector is ignored. Sentence pairs are packed together into a single sequence and are separated with a special token [SEP].
- As for Segment Embeddings (green block): For the input is a sequence with two sentences, different learned sentence embedding[e.g., A and B] will be added to every token of each sentence. For single-sentence inputs, we only use the sentence A embeddings.
- As for Position Embeddings (grey block): For languages, the order of each word in a sentence is quite important, so the position of each token will be marked as Position embeddings.
- BERT limits the length of the entire sequence to no more than 512 tokens. Whether it is a one-sentence sequence or a sentence-pairs sequence, sequences exceeding 512 will be divided at intervals of 512 tokens. In practice, considering computational efficiency, BERT mostly divides the sequence with a length of 128 tokens.

Finally, BERT adds these three types of embeddings to get the final input representation. And BERT will use the input representation obtained above to pre-train the model, below we respectively introduce how BERT uses MLM and NSP for pre-training.

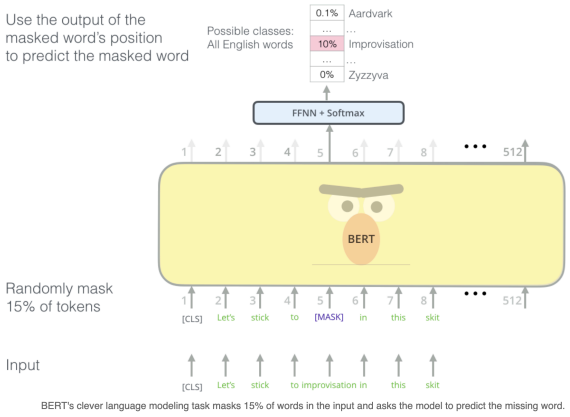



**FIGURE 9.3:** BERT Masked Language Model Alammar, Jay (2018). The Illustrated BERT, ELMo, and co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-bert/

### 9.1.4 Masked Language Model

Masked Language Model is the most important model structure of BERT and it mainly combines the ideas of Transformer Encoder and masked tokens.

Figure 9.3 is the visual representation of the Masked Language Model. As shown in the figure, when word embeddings are fed into BERT, 15% of the tokens in each sequence will be replaced by a special token [MASK]. Here, token [improvisation] is replaced with [MASK]. The output of MLM is the word embeddings of corresponding tokens, then feed the word embedding of [MASK] token into a simple softmax classifier and get the final prediction of [MASK] token. The task of MLM is to predict the original value of masked tokens, we hope that the result obtained by the softmax classifier is close to the true value. And the most important part is the "yellow" block, it's basically a multi-layer bidirectional Transformer Encoder based on implementation described in Kaiser et al. (2017).

Here I summarize the main points of MLM in the form of question and answer as follows:

- What is the Masked Language Model?
  - 15% of all WordPiece tokens in each sequence will be randomly masked.
  - Input: token embedding(one sequence, begin with [CLS])
  - Output: BERT token embedding.
  - Using softmax classifier to predict the masked token. (words match each other may have the same BERT embedding)

- How to mask?
  - 80% of the time: Replace the word with the [MASK] token, e.g., my dog is hairy ⇒ my dog is [MASK]
  - 10% of the time: Replace the word with a random word,e.g., my dog is hairy ⇒ my dog is apple.
  - 10% of the time: Keep the word unchanged, e.g., my dog is hairy ⇒ my dog is hairy.
- Why there are two other methods to replace the word?
  - Why keep 10% of masked tokens unchanged?
    In some downstream tasks like POSTagging, all the tokens are known, if BERT only trained the Masked sequences, then the model only uses the information of context, exclude the information of the masked words. It will lose a part of the information, then weakens the performance of the model.
  - Why replace 10% of the masked tokens with random words?
    Since we keep 10% of the masked token unchanged, if we do not add random noise, the model will be "lazy" in our training, the model will plagiarize current tokens, rather than learning.

### 9.1.5 Next-sentence Prediction

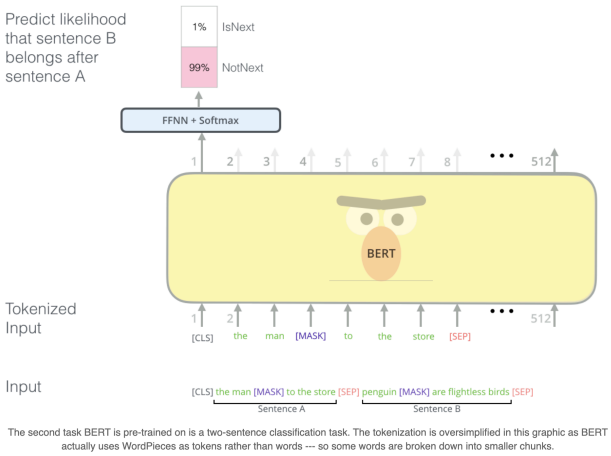



**FIGURE 9.4:** BERT Next-sentence Tasks Alammar, Jay (2018). The Illustrated BERT, ELMo, and co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-bert/

Figure 9.4 is the visual representation of Next-sentence Prediction(NSP). Many important downstream tasks such as Question Answering (QA) and

Natural Language Inference (NLI) are based on understanding the relationship between two text sentences, which is not directly captured by language modeling. So BERT proposes the NSP by using the special token [CLS] as the first token of every sequence. The pre-training structure of NSP and MLM are the same, and NSP and MLM are trained together. NSP using token [CLS] to get the result. For example, token [CLS] will get a binary classification prediction through the softmax classifier, which represents whether the model believes that the sentiment of the two sentences in the input sequence is the same.

Here I also use the question and answer format to summarize the main points of NSP:

- What is Next-sentence Prediction?
  - Input: token embedding (two sentences, begin with [CLS], each sentence ends with [SEP])
  - Output: BERT token embedding.
  - Using a softmax classifier to explain the relationship between two sentences.
  - Using [CLS] token to pre-train classification tasks.
  - Sentences can be trivially generated from a monolingual corpus.
  - Choose sentences A and B for each example, 50% of B is the actual next sentence that follows A, and 50% of B is a random sentence from the corpus.
- For example:
  - Input = [CLS] the man went to [MASK] store [SEP] he bought a gallon [MASK] milk[SEP]
    Label = IsNext.
  - Input = [CLS] the man [MASK] to the store [SEP] penguin [MASK] are flight ##less birds [SEP].
    Label = NotNext.

### 9.1.6 Pre-training Procedure of BERT

For the pre-training corpus, BERT uses the concatenation of BooksCorpus (800M words) (Zhu et al., 2015) and English Wikipedia (2,500M words) to create two versions of BERT (L stands for the number of layers, H stands for the hidden size, A stands for the number of self-attention heads):

- BERT-Base: L = 12, H = 768, A = 12, Total parameters = 110M
- BERT-Large: L = 24, H = 1024, A = 16, Total parameters = 340M

### 9.1.7 Fine-tuning Procedure of BERT

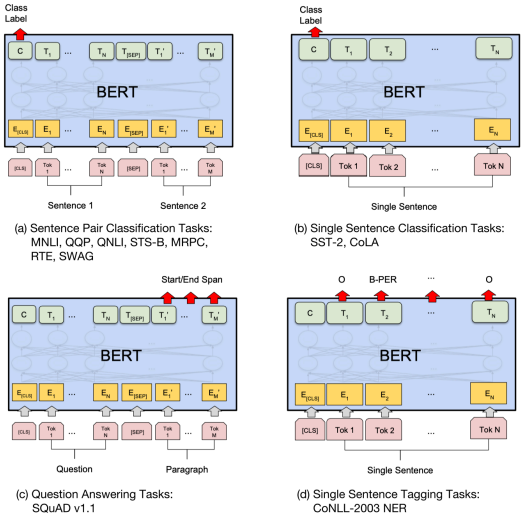



**FIGURE 9.5:** BERT Task Specific Models

For the BERT model obtained by pre-training, different types of tasks require different modifications to the model, and the modification of the model before fine-tuning is quite simple. As shown in Figure 9.5:

As for the sequence-level classification problem, such as sentiment analysis, task (a) and (b) in the figure, BERT takes the output representation of the first token[CLS] then feeds it to a softmax classifier to get the classification prediction and uses this prediction as model output for fine-tuning.

As for the Question Answering Task (e.g.Reading comprehension, task(c)), BERT needs to find the correct answer in the latter sentence to answer the question raised by the previous sentence. For each token in the second sentence, BERT will use the output embeddings of the token to make two predictions, representing whether the token is the beginning or the end of the answer. For example, if the third token of the second sentence is considered to be the beginning of the answer, and the fifth token is considered to be the end of the answer, then BERT will use the third token to the fifth token as the result to answer the questions raised by the first sentence. If the fifth token is considered to be the beginning of the answer and the third token is considered to be the end of the answer, that is, the end appears before the beginning, then the resulting output by BERT will be: No Answer. The true answer in reading comprehension may happen to be No Answer. BERT performs fine-tuning by comparing the loss between the prediction and the true value. It is worth noting that in reading comprehension or summarization tasks, BERT

is not doing real summarization. It cannot generate new vocabulary by itself, but can only choose from the vocabulary of the latter sentence. This may be the reason why BERT does not perform so well in corresponding tasks.

As for token-level classification (e.g. NER, Task (d) in the figure), BERT takes the output of the last layer transformer of all tokens then feeds it to the softmax layer for classification and uses the prediction of each token of the model to compare with the real answer and fine-tune the parameters.

### 9.1.8 Feature Extraction

Like many other Language models, the pre-trained BERT can create contextualized word embeddings. Then the word embeddings can be used as features in other models. Readers can try out BERT through BERT FineTuning with Cloud TPUs[2].

### 9.1.9 BERT-like models

The state-of-the-art performance of BERT reveals the deep bidirectional language model can significantly improve the model performance in NLP tasks, and BERT chart a new course that how a real bidirectional model should be.

However, BERT has also the following weaknesses:

- First of all, the input to BERT contains artificial symbols like [MASK] that never occur in downstream tasks, which creates a pre-train-fine-tuning discrepancy problem.
- Secondly, BERT assumes the predicted tokens are independent of others given the unmasked tokens, which is oversimplified for natural language.

Several models are inspired by BERT to solve these problems:

Roberta Liu et al. (2019) shows hyperparameter choices have a significant impact on the final results. It improves BERT pre-training in the following aspects to get better performance:

- Changing the input embedding to Byte Pair Encoding (BPE) Sennrich et al. (2015).
- Using dynamic masking: each train has different training data.
- Using full sentence without NSP.
- More Data, larger batch size (8k), and longer training (100k to 300k steps).

[2] https://colab.research.google.com/github/tensorflow/tpu/blob/master/tools/colab/bert_finetuning_with_cloud_tpus.ipynb

ALBERT Lan et al. (2019) mainly makes three improvements to BERT, which reduces the overall parameter amount, accelerates the training speed, and improves the model performance under the same training time.

- Using factorized embedding parameterization.
- Cross-layer parameter sharing, which significantly reduces the number of parameters.
- Replacing NSP with Sentence-order prediction loss (SOP).

There are also BERT-like models pre-trained on domain-specific corpora, for example, SciBERT Beltagy et al. (2019) on scientific publications, ERNIE Zhang et al. (2019) on a large corpus incorporating knowledge graph in the input. In comparison to fine-tune original BERT, training on the domain-specific corpora then fine-tuning them on downstream NLP tasks has shown to yield better performance.

## 9.2 Generative Pre-Training(GPT-2)

The GDP-2 mentioned next is also the most important NLP model in recent years. It and BERT are often mentioned at the same time because the model structure it uses is also based on the transformer, but it is the transformer decoder. As mentioned in **chapter 9**, the transformer decoder can be regarded as a unidirectional language model, so GDP-2 represents a different way of thinking from BERT and has the surprising ability in writing tasks.

### 9.2.1 Auto-regressive Language Model(AR)

GPT-2 is a unidirectional language model, such model structure is also called Auto-regressive language model. Figure 9.6 shows the modelling of Auto-regressive language model, it tries to estimate the probability distribution of a sequence with a auto-regressive pattern. Specifically, given a text sequence $X = (x_1, ..., x_T)$, AR language model factorizes the log-likelihood into a forward sum $logp(x) = \sum_{t=1}^{T} p(x_t|x < t)$ or a backward one $logp(x) = \sum_{T}^{t=1} p(x_t|x > t)$ Yang et al. (2019). Since an AR language model is only trained to encode a uni-directional context (either forward or backward), it is not effective at modeling deep bidirectional contexts. On the contrary, downstream language understanding tasks often require bidirectional context information.

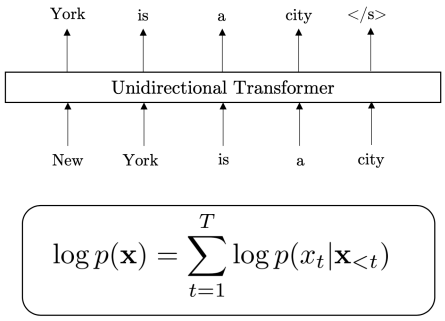



**FIGURE 9.6:** Autoregressive

### 9.2.2 Introduction of GPT-2

GPT-2 is proposed by researchers at OpenAI in 2019. It captures the attention of the NLP community for the following characters:

- Instead of the fine-tuning model with specific tasks, GPT-2 demonstrates language models can perform down-stream tasks in a zero-shot setting, which means without any parameter or architecture modification.
- To perform better under the zero-shot setting, GPT-2 becomes extremely large. The result is that training GPT-2 needs enormous data, so researchers also create a new dataset WebText[3], which contains millions of webpages.

With the characters above, GPT-2 achieves state-of-the-art results on 7 out of 8 tested language modeling datasets in a zero-shot setting but still underfits WebText.

Readers can experiment with GPT-2 by using AllenAI GPT-2[4]. You can input a sentence and see the prediction of the next words.

### 9.2.3 Input Representation of GPT-2

GPT-2 uses a human-curated dataset called WebText[5], that contains text scraped from 45 million web-links. All results presented in paper use a preliminary version of WebText, which contains slightly over 8 million documents for a total of 40GB of text after de-duplication and some heuristic-based cleaning Radford et al. (2019).

[3] https://skylion007.github.io/OpenWebTextCorpus/
[4] https://demo.allennlp.org/next-token-lm?text=AllenNLP%20is
[5] https://skylion007.github.io/OpenWebTextCorpus/


**FIGURE 9.7:** GPT-2 Input Representation Alammar, Jay (2018). The Illustrated GPT-2 co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-gpt2/

Figure 9.7 shows the input representation of GPT-2. The input sequence has a start token [S], and input embedding of each token is the corresponding specially designed Byte Pair Encoding (BPE) Sennrich et al. (2015), adding up the positional encoding vector.

### 9.2.4 The Decoder-Only Block

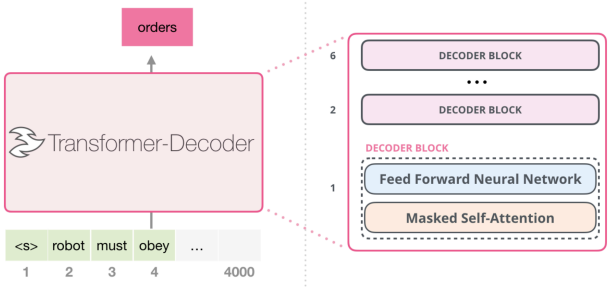



**FIGURE 9.8:** GPT-2 Model Alammar, Jay (2018). The Illustrated GPT-2 co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-gpt2/

Figure 9.8 shows the model of GPT-2. This model essentially is the Transformer decoder, except they threw away the second self-attention layer. In each decoder, Layer normalization Ba et al. (2016) is moved to the input of each sub-block, similar to a pre-activation residual network He et al. (2016b), and an additional layer normalization is added after the final self-attention block. A modified initialization which accounts for the accumulation on the residual path with model depth is used Radford et al. (2019).

GPT-2 reads from the start token [s] till the last predicted token to predict the next token. For example, in the first round, model uses [s] to predict [robot],

**TABLE 9.1:** GPT-2 models size

| Parameters | Layers | Dimensionality |
|---|---|---|
| 117M | 12 | 768 |
| 345M | 24 | 1024 |
| 762M | 36 | 1280 |
| 1542M | 48 | 1600 |

in the next round the input is updated as {[s],[robot]} since [robot] has been predicted. This is how Masked self-Attention in Decoder block works.

Otherwise, since a general system should be able to perform many different tasks, even for the same input, it should condition not only on the input but also on the task to be performed.$_n$The model of GPT-2 should be :

$$logp(x) = log \sum_{i=1} p(s_n|s_1, ..., s_{n-1}; task_i)$$

For example, a translation training example can be represented as the sequence (Translate to the french, English text, French text). Likewise, a reading comprehension training example can be written as (Answer the question, Document, Question, Answer).

By using the specially designed WebText, GPT-2 can be used by following patterns for different tasks.

- Reading Comprehension: data sequence, “Q:”, question sequence, “A:”
- Summarization: data sequence, “TL;DR:”
- Translation: English sentence 1 = French sentence 1, English sentence 2 = French sentence 2, “English sentence 3 =”

### 9.2.5 GPT-2 Models

Four versions models are trained, the architectures are summarized in 9.1. The smallest model is equivalent to the original GPT, and the second smallest equivalent to the largest model from BERT Devlin et al. (2018). The largest model is called GPT-2, which has 1.5 billion parameters.

### 9.2.6 Conclusion

The framework of GPT-2 is the combination of pre-training based on Transformer Decoder and fine-tuning based on unsupervised downstream tasks.

After the great success of bidirectional models like BERT, GPT-2 insists on using unidirectional models and still achieves state-of-the-art performance. It proves that the performance of language models can be significantly improved by simply increasing the size of training datasets and models, which is exactly what GPT-2 did and even GPT-2, which has 1.5 billion parameters, still underfits WebText. This result also suggests that datasets are as important as models.

## 9.3 XLNet

### 9.3.1 Introduction of XLNet

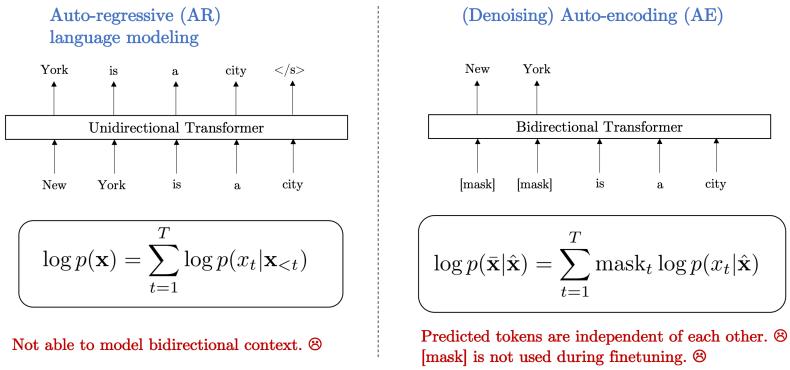



**FIGURE 9.9:** AR Language Modeling and AE

XLNet is proposed by researchers at Google in 2019. Since the autoregressive language model (e.g.GPT-2) is only trained to encode a unidirectional context and not effective at modeling deep bidirectional contexts and autoencoding (e.g.BERT) suffers from the pre-train-fine-tune discrepancy, XLNet borrows ideas from the two types of objectives while avoiding their limitations.

It is a new objective called Permutation Language Modeling. By using a permutation operation during training time, bidirectional context information can be captured and makes it a generalized order-aware autoregressive language model. Besides, XLNet introduces a two-stream self-attention to solve the problem that standard parameterization will reduce the model to bag-of-words.

Two XLNet are released, i.e. XLNet-Base and XLNet-Large, and include

the similar settings of corresponding BERT. Empirically, XLNet outperforms BERT on 20 tasks and achieves state-of-the-art results on 18 tasks.

### 9.3.2 Permutation Language Modeling(PLM)

Now Figure 9.10 illustrates the permutation language modeling objective.

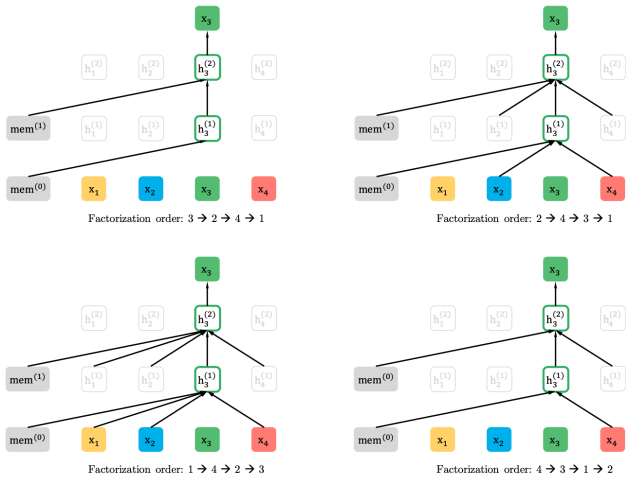



**FIGURE 9.10:** Illustration of the permutation language modeling objective for predicting x3 given the same input sequence x but with different factorization orders.

Specifically, for a sequence $X$ of length $T$, there are $T!$ different orders to perform a valid autoregressive factorization. Intuitively, if model parameters are shared across all factorization orders, in expectation, the model will learn to gather information from all positions on both sides. Let $P_T$ be the set of all possible permutations of a sequence [1,2,..., T] and use $z_t$ and $z_{<t}$ to denote the t-th element and the first t−1 elements of a permutation $p \in P_T$. Then the permutation language modeling objective can be expressed as follows:

$$\max_{\theta} \mathbb{E}_{p \sim P_T} \left[ \sum_{t=1}^{T} log p_\theta(x_{z_t | x_{z_{<t}}}) \right] \tag{9.1}$$

For instance, assume we have a input sequence {I love my dog}.

In the upper left plot of the above Figure 9.10, when we have a factorization order: {3, 2, 4, 1}, the probability of sequence can be expressed as follows:

$$P(I, love, my, dog) = P(my) * P(love|my) * P(dog|love, my) * P(I|love, my, dog) \tag{9.2}$$

as for the third token: {my}, it cannot use the information of all other tokens, so only one arrow from the starting token points to the third token in the plot.

In the upper right plot of the above Figure 9.10, when we have a factorization order: {2, 4, 3, 1}, the probability of sequence can be expressed as follows:

$$P(I, love, my, dog) = P(love) * P(dog|love) * P(my|love, dog) * P(I|love, my, dog) \tag{9.3}$$

as for the third token: {my}, it can use the information of the second and fourth tokens because it places after these two tokens in the factorization order. Correspondingly, it cannot use the information of the first token. So in the plot, in addition to the arrow from the starting token, there are arrows from the second and fourth tokens pointing to the third token. The rest two plots in the figure have the same interpretation.

During training, for a fixed factorization order, XL-Net is a unidirectional language model based on the transformer decoder, which performs normal model training. But different factorization order makes the model see different order of words when traversing sentences. In this way, although the model is unidirectional, it can also learn the bidirectional information of the sentence.

It is noteworthy that the sequence order is not actually shuffled but only attention masks are changed to reflect factorization order. With PLM, XLNet can model bidirectional context and the dependency within each token of the sequence.

### 9.3.3 The problem of Standard Parameterization

As just mentioned, XL-Net is a language model when the factorization order is fixed, which means that we want the model to be able to predict the t-th word under the condition of knowing the word before t.

The Standard Parameterization can be expressed as follows:

$$p_\theta(X_{p_t} = x|x_{p<t}) = \frac{e(x)^T h_\theta(x_{p<t})}{\sum_{x'} e(x')^T h_\theta(x_{p<t})}$$

where $e(x)$ denotes the embedding of input token and $h_\theta(x_{p<t})$ denotes the hidden representation of $x_{p<t}$.

While the permutation language modeling objective has desired properties, naive implementation with standard Transformer parameterization may not work. Specifically, let's consider two different permutations $p^1$:I love my dog and $p^2$ : I love dog my. The probability of my in $p^1$: P(my|I, love) and the probability of dog in $p^2$: P(dog|I, love)are identical. The model will be reduced

to predicting a bag-of-words, because $h_\theta(x_{z<t})$ does not contain the position of the target.

XLNet resolves the problem by reparameterizing with positions:

$$p_\theta(X_{p_t} = x | x_{p<t}) = \frac{e(x)^T g_\theta(x_{p<t}, p_t)}{\sum_{x'} e(x')^T g_\theta(x_{p<t}, p_t)}$$

where $e(x)$ denotes the embedding of input token and $g_\theta(x_{p<t}, p_t)$ denotes the hidden representation of $x_{p<t}$ and position $p_t$. But reparameterization with positions brings another contradiction Yang et al. (2019):
(1) To predict the token $x_{p_t}$ , $g_\theta(x_{p<t}, p_t)$ should only use the position $p_t$ and not the content $x_{p_t}$, otherwise the objective becomes trivial.
(2) To predict the other tokens $x_{p_j}$ with j > t, $g_\theta(x_{p<t}, p_t)$ should also encode the content $x_{p_t}$ to provide full contextual information.

XLNet proposes the Two-Stream Self-Attention to resolve the contradiction.

### 9.3.4 Two-Stream Self-Attention

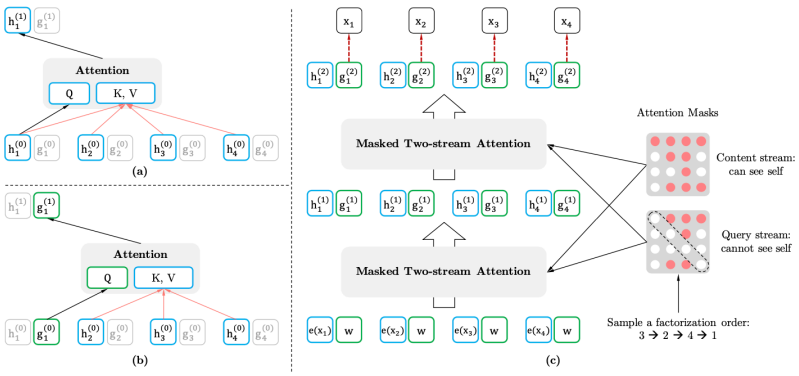



**FIGURE 9.11:** Two-Stream Self-Attention

Instead of one, two sets of hidden representation are proposed:

- The content representation $h_\theta(x_{p \le t})$, this representation encodes both the context and $x_{p_t}$ itself.
- The query representation $g_\theta(x_{p<t}, p_t)$, which only has information $x_{p<t}$ and the position $p_t$ but not the content $x_{p_t}$.

Figure 9.11 is an example with the Factorization order: $3, 2, 4, 1$:

- $h_i^{(t)}$ denotes the content representation of the i-th token in the t-th layer

of self-attention. It is the same as the standard self-attention. For instance, $h_1^{(1)}$ can see all the $h_i^{(0)}$ since the 1-st token is after token $3, 2, 4$.

- $g_i^{(t)}$ denote the query representation of the i-th token on the t-th layer of self-attention. It does not have access information about the content $x_{p_t}$, other trivial.
- Computationally, $h_i^{(0)}$ is the word embeddings, and $g_i^{(0)}$ is a trainable parameter initialized with a trainable vector. Only $h_i^{(t)}$ is used during fine-tuning. The last $g_i^{(t)}$ is used for optimizing the LM loss. A self-attention layer $m = 1, ..., M$ are schematically updated with a shared set of parameters as follows:

$h_{p_t}^m \leftarrow Attention(Q = h_{p_t}^{m-1}, KV = h_{p \leq t}^{(m-1)}; \theta)$ (content stream: use both $p_t$ and $x_{p_t}$)$g_{p_t}^m \leftarrow Attention(Q = g_p^{(}$

where Q, K, V denote the query, key, and value in an attention operation Vaswani et al. (2017). More details are included in Appendix A.2 for reference Yang et al. (2019).

### 9.3.5 Partial Prediction

While the PLM has several benefits, optimization is challenging due to the permutation operator. To reduce the optimization difficulty, XLNet only predicts the last tokens in a factorization order. It sets a cutting point $c$ and split the permutation $p$ into a non-target subsequence $p_{\leq c}$ and a target subsequence $p_{>c}$. The objective is to maximize the log-likelihood of the target subsequence conditioned on the non-target subsequence, i.e.,

$$\max_{\theta} \mathbb{E}_{p \sim P_T} \left[ log p_\theta(x_{z_{>c}} | x_{z_{\leq c}}) \right] = \mathbb{E}_{p \sim P_T} \left[ \sum_{t=c+1}^{|z|} log p_\theta(x_{z_t} | x_{z_{\leq t}}) \right] \tag{9.4}$$

For unselected tokens, their query representations need not be computed, which saves speed and memory. XLNet incorporates ideas from Transformer-XL and inherits two important characters of it, i.e. Segment-Level Recurrence and Relative Position Encoding, to enable the learning of long-term dependency and resolve the context fragmentation Dai et al. (2019). There is also a good Blog[6] to introduce Transformer-XL, Readers can read if interested.

### 9.3.6 XLNet Pre-training Model

After tokenization with SentencePiece Kudo and Richardson (2018), Researchers obtain 2.78B, 1.09B, 4.75B, 4.30B, and 19.97B subword pieces for

[6] https://medium.com/@shoray.goel/transformer-xl-9fc13473e0a4

Wikipedia, BooksCorpus, Giga5, ClueWeb, and Common Crawl respectively, which are 32.89B in total, to pre-train the XLNet.

Analogous to BERT, two versions of XLNet have been trained:

- XLNet-Base: L = 12, H = 768, A = 12, Total parameters = 110M (on BooksCorpus and Wikipedia only)
- XLNet-Large: L = 24, H = 1024, A = 16, Total parameters = 340M (on total datasets)

### 9.3.7 Conclusion

Language modeling has been a rapidly developing research area. However, most language modelings are unidirectional and BERT Devlin et al. (2018) shows that bidirectional modeling can significantly improve model performance. Unidirectional modelings without specific structure are hard to capture the bidirectional context. Now with the permutation operator, the unidirectional language modelings can become bidirectional modeling. XLNet has built a bridge between language modeling and bidirectional models. Overall, XLNet is a generalized AR pre-training method that uses a permutation language modeling objective to combine the advantages of AR and AE methods.

## 9.4 Latest NLP models

Nowadays NLP has become a competition between big companies. When BERT first came, people talked about it may cost thousands of dollars to train it. Then came GPT-2, which has 1.5 billion parameters and is trained on 40GB data. As I mentioned above, GPT-2 of Open-AI shows that increasing the size of models and datasets is at least as important as proposing a new model architecture.

After GPT-2, researchers at Google did the same thing, they proposed a general language model called T5 Raffel et al. (2019), which is trained on 750GB corpus - "C4 (Colossal Clean Crawled Corpus)"[7]. If you read the paper, you will find that the last page of it is a table of several experience results, which may cost millions of dollars to reproduce it.

On 28th May 2020, the "Arms race" goes into another level, GPT-3 Brown et al. (2020) emerged, the new paper takes GPT to the next level by making it even bigger - GPT-3 has 175 billion parameters and is trained on a dataset that

[7] https://www.tensorflow.org/datasets/catalog/c4

has 450 billion of tokens "GPT-3 Dataset"[8]. GPT-3 experiments with the three different settings: zero-shot, one-shot, and Few-shot to show that scaling up language models can greatly improve performance, sometimes even reaching competitiveness with prior SOTA approaches. However, it is conservatively estimated that training GPT-3 will cost one hundred million dollars.

Models like T5 and GPT-3 are very impressive, but the biggest problem at the moment is to find a way to make the current model put into use in the industry. If it can't bring benefits, the AI industry can't be sustained by burning money. As for researchers, the truth is, with the resources it is also possible to fail, but it is certainly impossible to succeed without resources now.

[8] https://github.com/openai/gpt-3

# 10

# *Introduction: Resources for NLP*

*Authors: Nico Hahn*

*Supervisor: Daniel Schalk*

As natural language processing has become one of the hottest topics in data science/statistics in recent years, tons of new resources have been created. As a result, NLP has become accessible to a wide range of people, making it easier to train and compare models. Competitions have emerged to determine who can develop the best model for specific tasks such as question answering.

In the following chapter, we will take a look at some of these benchmark datasets to see what tasks they are used for and how performance is assessed on these datasets.

In addition to that we will be talking about pre-training resources, what they are and why they are useful.

Lastly, we will be introducing the huggingface transformers module.

# 11

# *Resources and Benchmarks for NLP*

*Authors: Nico Hahn*

*Supervisor: Daniel Schalk*

Frameworks such as TensorFlow or Keras allow users to train a wide range of different models for different tasks. Let us assume that two models for a simple question-answer system are trained, one with attention and one without attention. How can these models be evaluated in order to find the model better suited to the task? Quite simply, through benchmarking. This section looks at some of the most commonly used benchmarking datasets and at pre-training resources.

## 11.1 Metrics

For many of the benchmarking datasets in natural language processing, a leaderboard exists in which different models are compared with each other. Depending on the task, the models are evaluated with different metrics. In this section we will introduce those used for the benchmarking datasets presented later.

**Exact match (EM):** The percentage of predictions that match any one of the answers exactly.

**(Macro-averaged) F1 score (F1):** Each answer and prediction is tokenized into words. For every answer to a given question, the overlap between the prediction and each answer is calculated and the maximum F1 is chosen. This score is then averaged over all the questions. Formally speaking:

$$F1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

$$\text{precision} = \frac{\text{number of same tokens}}{\text{length(predicted tokens)}}$$

$$\text{recall} = \frac{\text{number of same tokens}}{\text{length(labeled tokens)}}$$

**Perplexity:** Perplexity is a measurement of how well a probability model predicts a sample. A low perplexity indicates the probability distribution is good at predicting the sample. In NLP, perplexity is a way of evaluating language models. A model of an unknown probability distribution $p$, may be proposed based on a training sample that was drawn from $p$. Given a proposed probability model $q$, one may evaluate $q$ by asking how well it predicts a separate test sample $x_1, x_2, ..., x_N$ also drawn from $p$. The perplexity of the model $q$ is defined as

$$b^{-\frac{1}{N}\sum_{i=1}^{N}\log_b q(x_i)}$$

where $b$ is customarily 2. (Martinc et al., 2019)

**BLEU:** BLEU (**Bi**lingual **E**valuation **U**nderstudy) is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another. Scores are calculated for individual translated segments—generally sentences—by comparing them with a set of good quality reference translations. Those scores are then averaged over the whole corpus to reach an estimate of the translation's overall quality. Intelligibility or grammatical correctness are not taken into account. (Papineni et al., 2002)

**Accuracy:** Accuracy is the ratio of number of correct predictions to the total number of input samples.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

**Matthews correlation coefficient**: The MCC is used as a measure of quality of binary classifications. It takes true and false positives and negatives into account and is regarded as a balanced measure which can be used even if the classes are imbalanced. The MCC can be calculated directly from the confusion matrix using the following formula:

$$\text{MCC} = \frac{\text{TP} \cdot \text{TN} - \text{FP} \cdot \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}}$$

(Boughorbel et al., 2017)

## 11.2 Benchmark Datasets

### 11.2.1 SQuAD

The first Version of the **S**tanford **Qu**estion **A**nswering **D**ataset was released in 2016. The dataset was created with the aim of advancing the field of reading comprehension. Reading text and answering questions about it is a demanding

task for machines and requires large data sets of high quality. Most of the datasets before the release of the first version of SQuAD were either of high quality or of large size, but not both.

With the help of crowdworkers, 107.785 question-answer pairs were created for 536 Wikipedia articles. For each question, the answer is a segment of text, or span, from the corresponding reading passage. Pairs were collected in a two-step process. In the first step the crowdworkers were asked to generate five questions and their answers per paragraph.

In the second step, each crowdworker was shown only the questions along with the paragraphs of the corresponding article and was asked to choose the shortest span in the paragraph that answered the question. As a result of this process, questions in the dev-set multiple answers.

The goal of this procedure was to get a more robust evaluation and to obtain an indicator of human performance on SQuAD.

One shortcoming of reading comprehension systems is that they tend to make unreliable guesses on questions to which no correct answer is possible. With this in mind, the second version of SQuAD was released in 2018. In addition to the approximately 100.000 questions from the first version, 53.775 new, unanswerable questions on the same paragraphs are contained in this dataset.

The accuracy of models trained on SQuAD is evaluated using two different metrics, exact match and (Macro-averaged) F1 score, both ignoring punctuation and articles.

To evaluate human performance, the second answer to each question is treated as the human prediction. (Rajpurkar et al., 2016, 2018)

Humans achieve an **EM** score of 86.831 and a **F1** score of 89.452.

Currently, the best performing model achieves an **EM** score of 90.386 and a **F1** score of 92.777.

Examples of SQuAD and the leaderboard and can be viewed here:

`https://rajpurkar.github.io/SQuAD-explorer/`

### 11.2.2 CoQA

CoQA is a dataset for building **Co**nversational **Q**uestion **A**nswering systems. Humans are capable of gathering information through conversations that include several interrelated questions and answers. The aim of CoQA is to enable machines to answers conversational questions.

The data set is made up of 127k Q/A pairs, covering seven different domains such as Children's Stories or Reddit. Five of these domains are used for

in-domain evaluation, meaning models have already seen questions from these domains, and two are used for out-of-domain evaluation., meaning models have not seen any questions from these domains. To create the Q/A pairs, two people received a text passage, with one person asking the other person questions about the text and the other person answering. Using multiple annotators has a few advantages:

1. A natural flow of conversation is created.
2. If one person gives an incorrect answer or a vague questions is asked, the other person can raise a flag. Thus bad annotators can easily be identified.
3. If there is a disagreement, the two annotators can discuss it via a chat window.

Similar to SQuAD, three additional answers are collected for each question. However, since the answers influence the flow of the conversation, the next question always depends on the answer to the previous question. For this reason, two different answers to the same question can lead to two different follow-up questions. In order to avoid incoherent discussions, annotators are shown a question that they must answer first. After answering, they are shown the original answer, and they must then confirm that their answer has an identical meaning.

Compared to SQuAD 2.0, there is a greater variety of question types in CoQA. While almost half of the questions in the SQuAD start with *what*, less than a quarter of the questions in the CoQA begin with this token. Another major difference is that questions in CoQA are on average 5.5 words long, compared to an average length of 10.1 in SQuAD. It is also worth mentioning that about 10% of the answers in CoQA are either yes or no, whereas there are no such answers in SQuAD.

Like SQuAD, trained models are evaluated using a macro-average F1 score. Models are evaluated separately on the in-domain dataset and the out-of-domain dataset. (Reddy et al., 2018)

Humans achieve a **F1** score of 89.4 for in-domain and a **F1** score of 87.4 for out-of-domain.

Currently, the best performing model achieves a **F1** score of 91.4 for in-domain and a **F1** score of 89.2 for out-of-domain.

Examples of CoQA and the leaderboard and can be viewed here:

https://stanfordnlp.github.io/coqa/

### 11.2.3 (Super)GLUE

Most models in NLP are designed to solve a specific task, such as answering questions from a particular domain. This limits the use of models for understanding natural language. In order to process language in a way that is not limited to a specific task, genre, or dataset, models should be able to solve a variety of tasks well.

The **G**eneral **L**anguage **U**nderstanding **E**valuation benchmark dataset is a collection of tools created with this in mind. It is designed to encourage and favor models that share common linguistic knowledge across tasks. These tasks include textual entailment, sentiment analysis and question answering. Some tasks come with a lot of training data, others with less. Common to all datasets is that they were not created specifically for GLUE, but are existing datasets. Models that are evaluated on GLUE only need to have the ability to process single-sentence and sentence-pair inputs and make appropriate predictions. This test suite contains a total of nine sentence or sentence-pair NLU tasks, built on established annotated datasets. There are three distinct types of tasks in GLUE: Single-Sentence Tasks, Similarity and Paraphrase Tasks and Inference Tasks.

**Single-Sentence Tasks**:

The first single-sentence task is CoLA, the **Co**rpus of **L**inguistic **A**cceptability, which consists of English acceptability judgments derived from books and journal articles on linguistic theory. Each datapoint consists of a sequence of words and an annotation as to whether this sequence is a grammatical English sentence. Matthews correlation coefficient is used as the evaluation metric.

The **S**tanford **S**entiment **T**reebank task consists of sentences from movie reviews and the corresponding sentiment (positive/negative). Accuracy is used for evaluation.

**Similarity and Paraphrase Tasks**

The **M**icrosoft **R**esearch **P**araphrase **C**orpus consists of pairs of sentences and the goal is to predict whether two sentences are semantically equivalent. For evaluation, F1 score and accuracy is used.

**Q**uora **Q**uestion **P**airs is similar to MRP in that the aim is to predict whether two questions are semantically equivalent and F1 and accuracy is used for evaluation.

The **S**emantic **T**extual **S**imilarity **B**enchmark consists of sentence pairs human-annotated with a similarity score from 1 to 5. The goal is to predict these scores. Pearson and Spearman correlation coefficients are used for evaluation.

**Inference Tasks**:

The **M**ulti-Genre **N**atural **L**anguage **I**nference Corpus is a collection of sentence pairs with textual entailment annotations. Based on a premise sentence and a hypothesis sentence, the aim is to predict whether the premise entails the hypothesis, contradicts the hypothesis, or neither of the two. Models are evaluated using accuracy.

**R**ecognizing **T**extual **E**ntailment is akin to MNLI, only this time with a two-class split.

The Winograd Schema Challenge is a reading comprehension task in which a system must read a sentence containing a pronoun and pick the speaker of that pronoun from a list of choices. To transform this task into a classification problem, pairs of sentences are constructed by replacing the ambiguous pronoun with any possible referent. The task is to predict if the sentence with the pronoun substituted is entailed by the original sentence. Evaluation is done using accuracy.

The last task in GLUE is based on the first version of SQuAD. The task is converted into a sentence pair classification task by pairing each question and each sentence in the respective context, with the aim of determining whether or not a sentence contains the answer to the question. The task is evaluated on accuracy.

The models are scored separately for each task and then a macro-average of these scores is calculated to determine a system's position on the ranking. If a task has multiple metrics, an unweighted average of these metrics is used as the score for the task when calculating the overall macro average. (Wang et al., 2018)

The human baseline score is 87.1, while the best model score is currently 90.6.

Roughly one year after the release of GLUE, models surpassed human performance. In response to this, a new benchmark, SuperGLUE, was introduced. It follows the same principles as GLUE, however the tasks included are more challenging. The two hardest tasks in GLUE, Recognizing Textual Entailment and the Winograd Schema Challenge, remain, the rest were selected based on difficulty for current NLP approaches. There are a total of eight different tasks in this benchmark.

**Bool**ean **Q**uestions consists of a text passage together with a corresponding yes/no question. Models are evaluated using accuracy.

**C**ommitment **B**ank is a three-class textual entailment task. Accuracy and F1 is used for evaluation, where for multi-class F1 the unweighted average of the F1 per class is calculated.

**C**hoice **o**f **P**lausible **A**nswers is a causal reasoning task in which a model is given a premise sentence, a question and two possible answers. It must then decide which answer is the correct one. Accuracy is used for the evaluation.

**Multi**-Sentence **R**eading **C**omprehension is a QA task where each example consists of a paragraph, a question and a list of answers. Models must predict which answers are correct. Evaluation metrics are F1 over all answer choices and the exact match of the answer set of each question.

**Re**ading **Co**mprehension with **Co**mmonsense **R**easoning **D**ataset is a multiple choice QA task. Each data point consists of a paragraph, a fill-in-the-gap sentence in which an entity is masked, and a list of possible entities to choose from. The entities can be expressed using several different surface forms, all of which are considered correct. Models are evaluated using max (over all mentions) token-level F1 and exact match.

**W**ords **i**n **C**ontext is a word sense disambiguation task in which a model is given two sentences and a polysemous word. Models must decide whether the word is used with the same meaning in both sentences. Accuracy is used for evaluation.

(Wang et al., 2019)

For SuperGLUE, the human baseline score is 89.8, which is above the best model score, presently 89.3.

More information about the tasks and the leaderboard for both GLUE and SuperGLUE is available here:

https://super.gluebenchmark.com/

### 11.2.4 AQuA-Rat

One task that most people know from their time at school is solving algebraic word problems. For humans, this task can be easy, depending on a person's mathematical abilities, since we only have to perform a series of arithmetic operations. However, since programs can be endlessly complicated, it is a considerable challenge to induce them directly from question-answer pairs. The **A**lgebra **Qu**estion **A**nswering with **Rat**ionales dataset attempts to make this task more feasible for machines by providing not only the correct answer but also step-by-step instructions for deriving the correct answer, the so-called rationale. Models trained on AQuA-Rat must not only predict the correct answer, but also the rationale.

The dataset contains over 100.000 questions, and each question has five different options as to what the correct answer is. It also contains the answer rationale and the correct option. The problems cover a wide range of topics, for instance probability theory or calculus, with a variety of difficulty levels. To create the dataset, examples of exams such as the GMAT (Graduate Management Admission Test) and GRE (General Test) were taken from the Internet. This part of the dataset is called the seed dataset. Besides,

crowdsourcing was used to generate further questions. For this users were presented with five questions from the seed dataset and asked to select one of these questions and write a similar question. Users were also forced to rephrase the rationales and answers to avoid paraphrasing the original questions. These created questions were then passed to another user for quality control.

The rationales are evaluated using average sentence level perplexity and the BLEU score. If a model is unable to generate a token for perplexity computation, an unknown token is predicted. The correctness of the answers is evaluated by calculating the accuracy of the predictions.

This is a relatively new dataset and as of now there is no online leaderboard for it. The authors of the original paper used an attention-based sequence to sequence model as their baseline method. The authors generated a program containing both instructions that generate output and instructions that simply generate intermediate values used by following instructions. The program uses a latent predictor network which generates an output sequence conditioned on an arbitrary number of input functions and staged back-propagation to save memory. Going into further depth about this program would be beyond this book so I'd advise to have a look at the original paper. The program outperformed the baseline model and achieved a perplexity of 28.5, a BLEU score of 27.2 and has and accuracy of 36.4. (Ling et al., 2017)

The paper and examples of the dataset can be found here:

`https://github.com/deepmind/AQuA`

### 11.2.5 SNLI

When it comes to understanding natural language, the understanding of entailment and contradiction is essential. The characterization and use of these relationships in computational systems is called natural language inference and is fundamental for tasks such as commonsense reasoning and information retrieval. The **S**tanford **N**atural **L**anguage **I**nference Corpus is a collection of sentence pairs that are labeled either as entailment, contradiction or semantic independence. While there are other datasets that try to accomplish this particular task, they all have problems of size, quality and vagueness.

SNLI consists of about 570k record pairs. Again, crowdworkers were used to construct the dataset. For this purpose they were shown the caption of a photo but not the photo itself, and asked to write three alternative captions: One that is definitely a true description of the photo, one that could be a true description of the photo, and one caption that is definitely a false

description of the photo. By not showing the photo, the authors wanted to ensure that each pair of sentences could be reconstructed based on the available text alone. To quantify the quality of the corpus, about 10% of all created sentence pairs were validated. For this purpose, each crowdworker was shown five pairs of sentences and asked to mark them with one of the three labels. Each set was shown to a total of five crowdworkers. For each pair, a gold label was awarded if at least three of the five annotators chose the same label. About 98% of all sentence pairs received a gold label, the rest were given a placeholder label. (Bowman et al., 2015)

The models are evaluated again with the accuracy of the predicted label. There is no measurement of human performance for the SNLI corpus. At present, the most accurate model is a semantics-aware BERT (SemBERT) with an accuracy of 91.9. The paper and examples of the dataset can be found here:

`https://nlp.stanford.edu/projects/snli/`

### 11.2.6 Overview

Below is a brief overview of all of the datasets discussed in this chapter, including some other interesting datasets. If you would like to learn more about one of the datasets, for each dataset the corresponding paper is linked.

| Name | Task | Size | Description |
|---|---|---|---|
| SQuAD 2.0[1] | Question Answering, Reading Comprehension | 150,000 | Paragraphs w questions and answers |
| CoQA[2] | Question Answering, Reading Comprehension | 127,000 | Answering interconnected questions |
| GLUE[3] | General Language Understanding | — | Nine different NLU tasks |
| SuperGLUE[4] | General Language Understanding | — | Eight different NLU tasks |
| AQuA-Rat[5] | Question Answering, Reading Comprehension, Mathematical Reasoning | 100,000 | Solving algebraic word problems |

| Name | Task | Size | Description |
|---|---|---|---|
| SNLI[6] | Natural Language Inference | 570,000 | Understanding entailment and contradiction |
| Irony Sarcasm Analysis Corpus[7] | Classification, Sentiment Analysis | 33,000 | Ironic, sarcastic, regular and figurative tweets |
| WikiText-103 & 2[8] | Language Modelling | 100M+ | Word and character level tokens from Wikipedia |
| WMT 14 English-German[9] | Language Translation | 4.5M | Sentence pairs for translation |
| VOiCES[10] | Speech Recognition | 3,900 | Voices in complex environmental settings. 15h material |

## 11.3 Pre-Trained Models

In the last chapters we've already heard quite a bit about pre-trained models like BERT or GPT-3. But exactly on what data are they trained on? Let's find out.

### 11.3.1 BERT

The pre-training corpus used for BERT consists of the BookCorpus and the entirety of the English Wikipedia.

[1] https://arxiv.org/pdf/1806.03822.pdf
[2] https://arxiv.org/pdf/1808.07042.pdf
[3] https://openreview.net/pdf?id=rJ4km2R5t7
[4] https://w4ngatang.github.io/static/papers/superglue.pdf
[5] https://arxiv.org/pdf/1705.04146.pdf
[6] https://nlp.stanford.edu/pubs/snli_paper.pdf
[7] http://www.roman-klinger.de/publications/ling2016.pdf
[8] https://arxiv.org/pdf/1609.07843.pdf
[9] https://nlp.stanford.edu/pubs/luong-manning-iwslt15.pdf
[10] https://arxiv.org/pdf/1804.05053.pdf

**The BookCorpus**: This dataset was released in 2015. To create the corpus, 11,038 free books were collected from the Internet. All of these were written by authors who have not yet been published. To be included, a book had to have more than 20,000 words to filter out the shorter stories that might be noisy. The dataset includes over 16 different genres, for example *Romance*, *Science Fiction* or *Fantasy*. In total there are about 1 billion words, 1.3 million unique words and 74 million sentences with an average sentence length of 13 words in the BookCorpus. (Zhu et al., 2015)

**English Wikipedia**: For Wikipedia only text passages were extracted, while lists, tables and headings were ignored. In total, this dataset contains 2.5 billion words.

According to the authors, it is crucial to use a document-level corpus rather than a shuffled sentence-level corpus like the Billion Word benchmark dataset to extract long sentences. (Devlin et al., 2018)

### 11.3.2 OpenAI GPT-3

The dataset used for pre-training GPT-3 consists of a filtered version of the Common Crawl dataset and multiple curated high quality datasets, including an extended version of WebText, two books corpora and the English language Wikipedia.

**Common Crawl**: The Common Crawl corpus contains petabytes of data collected over 8 years of web crawling. The corpus contains raw web page data, metadata and text extracts. To improve the quality of Common Crawl, two techniques are used: (1) filtering Common Crawl and (2) fuzzy deduplication.

1. To improve the quality, the original WebText was used as a proxy for high quality documents. A classifier was trained to distinguish these documents from the raw text in the Common Crawl. This classifier was used to re-sample Common Crawl by prioritizing documents for which higher quality was predicted. A logistic regression was trained for the classifier using characteristics from the standard Spark and HashingTF tokenizer. A document was kept in the dataset, if

$$\text{np.random.pareto}(\alpha) > 1 - \text{document\_score}.$$

A value of 9 was chosen for $\alpha$ in order to obtain both high and low scoring, but mostly high scoring documents.

2. To prevent overfitting, documents were fuzzily deduplicated

using Spark's MinHashLSH implementation with 10 hashes. WebText was also fuzzily removed from Common Crawl. This decreased dataset size by around 10%.

**WebText**: The WebText dataset is a dataset created by web scraping that emphasizes the quality of the documents. Only websites that have been curated/filtered by humans have been scrapped. To simplify this task, all outbound links from Reddit, a social media platform, which received at least 3 Karma, were used. The resulting dataset contains the text subset of these 45 million links. Fuzzy deduplication was also used here.

**Books1 and Books2**: These are two internet-based books corpora on which fuzzy deduplication was performed. Nothing more is known about these datasets.

The datasets used to train GPT-3 are shown in the table below.

| Dataset | Quantity (tokens) | Weight in training mix |
|---|---|---|
| Common Crawl (filtered) | 410 billion | 60% |
| WebText2 | 19 billion | 22% |
| Books1 | 12 billion | 8% |
| Books2 | 55 billion | 8% |
| Wikipedia | 3 billion | 3% |

(Brown et al., 2020)

### 11.3.3 Google 5T

Google 5T also uses a dataset based on Common Crawl for pre-training their model, called the "Colossal Clean Crawled Corpus" (C4). To improve the quality of Common Crawl, the following heuristics were used:

- Only keep lines that end in a period, exclamation mark, question mark, or closing quotation mark.
- Remove any page that contains a word from the "List of Dirty, Naughty, Obscene or Otherwise Bad Words".
- Remove any line containing the word Javascript to remove warnings about enabling Javascript.
- Remove any page containing the phrase "lorem ipsum".
- Remove all pages that contain "{" because some pages may have accidentally contained code.
- To deduplicate the dataset, discard all but one on any three-sentence span occurring more than once in the dataset.

Furthermore, **langdetect** was used to filter out any pages that were not classified as English with a probability of at least 99%.

(Raffel et al., 2019)

## 11.4 Resources for Resources

If you are interest in further NLP tasks or dataset, there are two websites worth checking out.

Papers With Code highlights trending Machine Learning research and the code to implement it. Their mission is to create a free and open resource with ML papers, code and evaluation tables. Anyone can contribute by downloading data, training their own model and comparing their model to others.

To see the newest trends in NLP, check out the link below.

`https://paperswithcode.com/area/natural-language-processing`

If you want to refine your natural language processing (NLP) skills, finding accessible and relevant datasets can be one of the biggest bottlenecks. A lot of time can be spent searching for accessible datasets for the learning task at hand or trying to curate your own data instead. This is where The Big Bad NLP Database, managed by Quantum Stat, comes in. It is a central location for NLP datasets. Currently there are over 500 data entries for general NLP tasks, such as question answering or language modeling. While most of the datasets are in English, there are also a number of datasets in other languages. Just have a look for yourself!

`https://datasets.quantumstat.com/`

# 12

# *Use-Cases for NLP*

*Author: Matthias Aßenmacher*

Since it is all fine and good to know the theory and how everything's working on the inside, we thought that this booklet could also benefit from showcasing some exemplary use case(s). Initially we had planned to include two chapters on different use cases, but during the course a student assigned to one of these chapters dropped out of the course. So we were left with only one (but highly motivated) student willing to pursue this issue.

The following chapter will contain a use case on Natural Language Generation (NLG) which is a task that (oftentimes) strongly relies on an encoder-decoder style architecture. Recently, these types of models could benefit **a lot** from the use of Attention mechanisms (Bahdanau et al. (2014)) and the proposal of the Transformer architecture (Vaswani et al. (2017)) presented in Chapter 8. Nevertheless the task of NLU requires more than just good architectures, it is also necessary to handle the output of these models in a suitable way in order to generate meaningful text.

So without further ado: The following chapter will showcase the challenges of NLU by touching upon the the tow topic *Chatbots* and *Image Captioning*, where the latter represents a hybrid task of Computer Vision and NLP.

# 13

# *Natural Language Generation*

*Author: Haris Jabbar*

*Supervisor: Matthias Aßenmacher*

## 13.1 Introduction

Machine learning systems can be differentiated into two types: Discriminative and Generative. While discriminative systems like classification, regression, clustering are the more well known type, it's the Generative systems that hold greater promise of achieving Artificial General Intelligence. In essence, a Generative system is expected to produce images, text or audio that would be meaningful to the users. Generating a picture of a horse is a much harder problem than just identifying whether there is a horse in the picture.

In this chapter, I will tackle the generative processes in NLP. Understandably, the field is called Natural Language Generation (NLG).

## 13.2 Definition and Taxonomy

Reiter and Dale (2000) defined Natural Language Generation (NLG) as "the sub-field of artificial intelligence and computational linguistics that is concerned with the construction of computer systems than can produce understandable texts in English or other human languages from some underlying non-linguistic representation of information".

Two aspects need to be highlighted. First is generation of *understandable* text in a human language and second is the input to such a generation system is *non-linguistic* representation of information. For our purposes, we

will drop the second requirement; which means that the source can be text as well. With such a definition in mind, we can have following taxonomy of NLG systems :

1. Text-to-Text
   - Machine Translation : Automatically translating between various human languages
   - Text Summarization : Summarizing a (big) text document into a shorter summary/abstract.
2. Data-to-Text
   - Image Captioning : Describe the image in a short sentence.
   - Video Captioning : Generate captions that change with the video scenes.
   - Business Intelligence : Creating text summaries of data from conventional databases (e.g SQL)
3. Ideas-to-Text
   - Poetry/Song Generation : Generating a song from a few keywords or mimicking the style of a certain artist.
   - Fake News : Automatically generating news items that look credible but are not.
4. Dialog Systems (Chatbots)
   - Goal Oriented : Chatting with a computer system (agent) with a specific purpose (e.g. booking a flight)
   - Open ended conversations : When the conversation with the agent is casual chit-chat but has the components of information, emotion and human like empathy.

## 13.3 Common Architectures

There are many architectures that are common across NLG systems. While some are used in other NLP domains as well, in the following sections I will explain them with a focus on language generation.

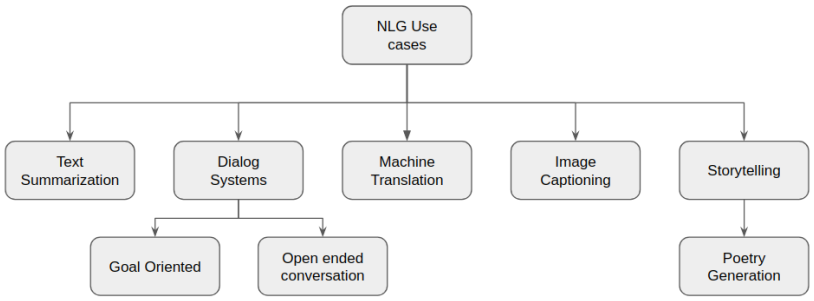



**FIGURE 13.1:** NLG Use Cases

### 13.3.1 Encoder-Decoder Architecture

The most ubiquitous architecture for NLG is the encoder-decoder architecture, and especially the decoder part of it. Hence I will explain it in some detail. The architecture is shown in the subsequent figures.

The architecture can be seen as modeling conditional probability $P(y/x)$ with $y$ being the output of the decoder and it is conditioned on $x$ (the output of the encoder). Hence the NLG task becomes generating text through decoder conditioned on some input, coming from the encoder.

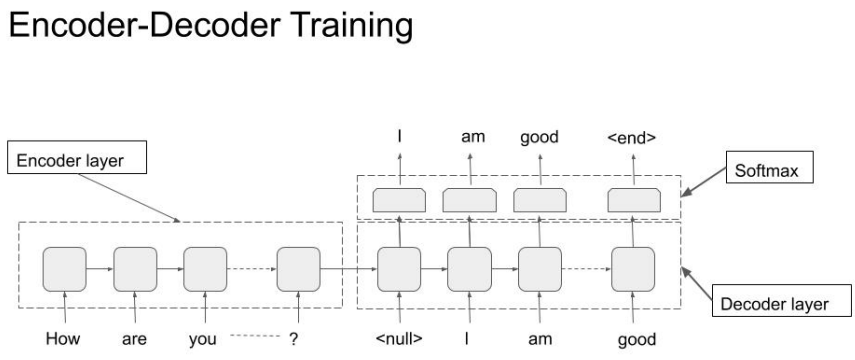



**FIGURE 13.2:** Encoder-Decoder (Training)

#### 13.3.1.1 Encoder :

As stated above, the purpose of this part of the network is to provide conditional information on which the decoder generates text. As such, this part can have **ANY** architecture that provides some form of embedding of the input. It can be a convolutional neural network to condition the generated text on some properties of an image (for example image captioning), or RNN/LSTM/Transformer architecture for text or audio based conditioning; or even a simple feed forward network to condition it on relational database for example. For the purpose of illustration we will be using an RNN/LSTM with text as input condition (as shown in the figure).

The thing to note here is that the richer the feature vector going from encoder to decoder, the more information decoder would have to generate output. This was the motivation to move from single feature vector to multiple vectors and to attention based models. This trend finally led to the transformer based models.

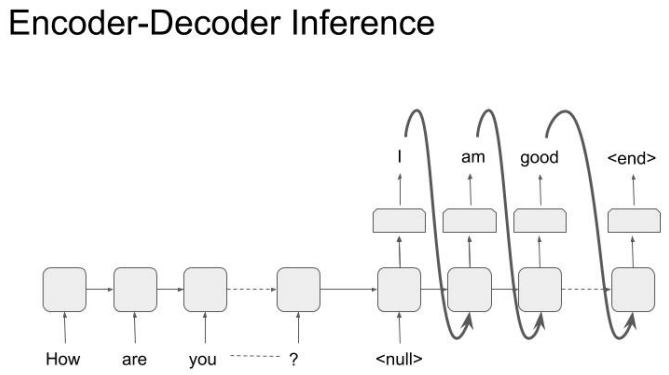



**FIGURE 13.3:** Encoder-Decoder (Inference)

#### 13.3.1.2 Decoder :

The decoder is the most distinctive part of an NLG system. Almost all decoders have the same form as shown in the figures above. The purpose is to generate text tokens one after the other until a terminating criteria is met. This termination is usually a termination token (<end> in the figures) or a max length criteria.

During training, we are given an input (text/image/audio) and the 'gold

label text' that we want the system to learn to generate for that particular input. In the given example, the input is the text "How are you?" and the gold label is "I am good". The input goes through the encoder and produces a feature vector that is used as the input to decoder. The decoder then generates tokens one by one and the loss is calculated after the softmax layer from the generated token and the gold label token. Note the inclusion of an extra token '<null>' as the first token. The last token of the gold label should produce the '<end>' token.

During inference, we don't have the gold label, so the output of one step is used as input to next step, as shown in the figure. Note that it matches with the setup during training. The generator stops when '<end>' token is emitted; thus completing the inference.

### 13.3.2 Attention Architecture

The attention architecture is introduced in detail in Chapter 9[1]. Here I will briefly mention its use in NLG systems. Looking at the picture below, we can see that the attention is from decoder to encoder.

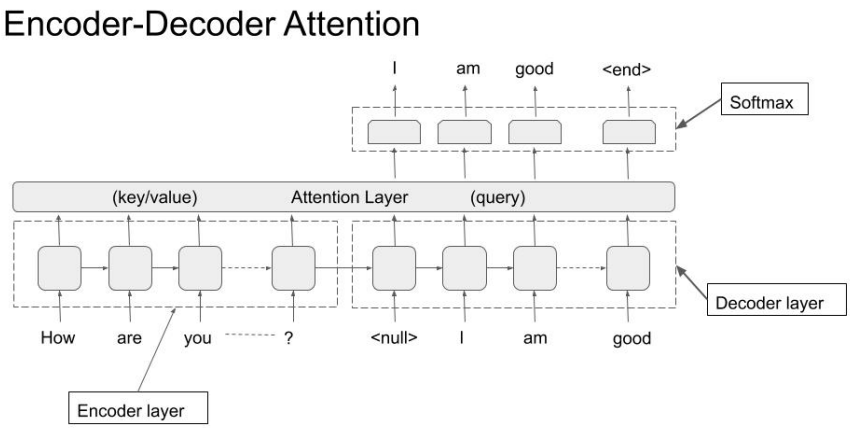



**FIGURE 13.4:** Encoder-Decoder (Attention)

In other words, before generating each token, the decoder attends to all tokens in the encoder. The query is the decoder token and key/values are all encoder token. That way the decoder has much richer information to base its output on.

[1] ./02-02-attention-and-self-attention-for-nlp.html

### 13.3.3 Decoding Algorithm at Inference

Now I will explain the decoding algorithms that are used to generate text from the softmax layer.

As explained above, during inference, the tokens are generated sequentially. In a vanilla version of decoding, at each step of the sequence, the token with highest probability in the softmax layer is generated. This is called 'greedy decoding', but it has been shown to produce suboptimal text. There are few improvements over this greedy approach.

#### 13.3.3.1 Beam Search

In greedy decoder we simply output the maximum probability token at each step. But if we track multiple words at each step and then output the sequence formed by highest probability combination, we get beam search. The number of tokens we keep track of is the length of beam (k). The algorithm then goes as follows:

1. Select k-tokens with highest probability at step 1.
2. Use these k-tokens to generate k softmax vectors at step 2.
3. Keep the k highest scoring combinations.
4. Repeat steps 2 and 3 till <end> token is generated, or a predefined max is reached
5. At each step, we have only k-hypothesis, which is the length of beam search.

While beam search tends to improve the quality of generated output, it has its own issues. Chiefly among them is that it tends to produce shorter sequences. Although it can be controlled by the max parameter (of step 4), it's another hyperparameter to be reckoned with.

#### 13.3.3.2 Pure Sampling Decoder

Here, at each step, instead of taking the token with maximum probability like in greedy search, the token is sampled from the whole vocabulary according to the probability distribution predicted by the softmax layer.

#### 13.3.3.3 K-sampling Decoder

It's like pure sampling decoder, but instead of sampling from whole vocabulary, the token is sampled only from the k-highest probability tokens.

### 13.3.4 Memory Networks

Memory Networks is another architecture that is potentially quite useful in language generation tasks. The basic premise is that LSTMs/RNNs and even Transformer architecture stores all the information only in the weights of the network. When we want to generate text that should include information from a large knowledge base, this 'storage' of network weights is insufficient. Memory networks resolve this problem by employing an external storage (the memory) that it can use during language generation. Conceptual diagram is showing in the following figure, followed by a brief description.

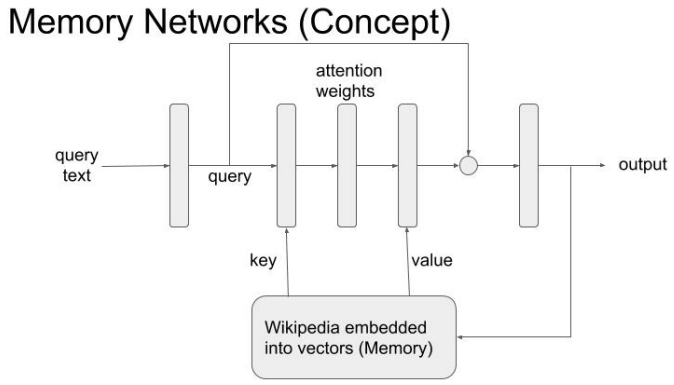



**FIGURE 13.5:** Encoder-Decoder (Attention)

Memory (M) in this context can be any database that can be queried by the network. Usually it is of the form of key-value pairs or a simple array of vectors embedding a corpus of knowledge (eg DBPedia/wikipedia). For any query input (x), first we get an embedding. This is then used to attend over the memory M in the usual attention mechanism (Chapter 9[2]). The output is the weighted sum of memory that incorporates information from complete knowledge corpus. In some cases the output can also be used to update the memory database.

### 13.3.5 Language Models

Language models are probably the most important ingredient of generating text. As the name implies, they model the probability distribution of

[2] ./02-02-attention-and-self-attention-for-nlp.html

generating words and characters. More concretely, they model the conditional probability distribution P(w_t/w_{t-1}). Thus with a given input vector coming from a source (image/database/text), this model can be used to generate words one after the other.

## 13.4 Dialog Systems

These are the systems where an agent chats with a human being either with a specific purpose (goal oriented) or it is a general open ended chat. The examples of goal oriented chats include tasks like booking an appointment or a flight ticket. Open ended chats can be talking about a general topic which may or may not include a 'personality' for the chatbot.

### 13.4.1 Types

- Chatbots
  - Open domain
  - Goal Oriented

### 13.4.2 Architectures

Chatbots are complex software systems involving many modules and components that interact with each other asynchronously and in an event-driven fashion. An example architecture is shown in the following figure. This is an example of a goal-oriented chatbot which helps the use make a flight reservation. Following could be an example sequence of steps

1. The first step is to decode the text coming from a user. The two types of decoders that are usually required are an 'intent classifier' and a 'Named Entity Recognizer (NER)'. The intent classifier tries to determine the intent of the command coming from the user. So for the given example the command is 'Book a flight to Berlin on Tuesday', the correct classification would be 'book-flight'.
2. This intent is then fed to a 'State Space' module that keeps a list of entities that are required for each intent. For 'book-flight' intent, it would need entities like destination, date of flight, time of flight, airline/cost preferences etc. The NER in step 1 tries to

extract these entities from the command. In our example, it could extract the destination and date of travel.

3. The conversation state manager keeps track of which entities have been resolved and which are outstanding. For both resolved and outstanding entities it refers to 'Action Space' module that generates API calls to external databases for requested parameters. It also refers to a 'Natural Language Understanding (NLU) model' to extract the information it needs to resolve the entities.

4. The 'Action Space' module calls various airline APIs and databases for associated tasks like determine available flights, prices and timings etc. Based on this information it refers back to the user to ask for unresolved entities.

5. The cycle continues till the system has all the required information to carry out the task.

A chatbot touches upon all areas of NLP and NLG; for example in the above case, it would need QnA ability, text generation, summarization, information retrieval and classification, to name a few.

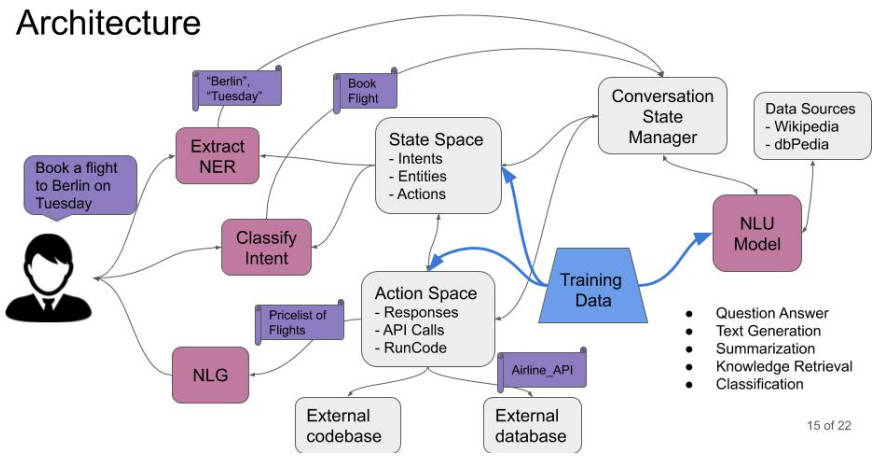



**FIGURE 13.6:** Chatbot Architecture


**FIGURE 13.7:** Image Captioning Examples

## 13.5 Image Captioning System

As the name suggests, these systems attempt to capture the caption of a given image as shown in the following figure. The input to the system is an image which is reduced to a feature vector using an appropriate architecture. Usually convolution based pretrained architectures are employed at this stege. The feature vector is then fed to a decoder network that produces the caption of the image. (See figure 2).

### 13.5.1 Experiments

For this seminar, I implemented a network for this task. For the encoder part, I used a pretrained Resnet backbone with a trainable fully connected layer appended after that. This gave me a feature vector for an image. For decoder part, I used three different kind of decoder architectures.

- Simple LSTM Layer
- BERT decoder
- GPT-2 decoder

However, these decoders expect a multi-token input feature embedding on

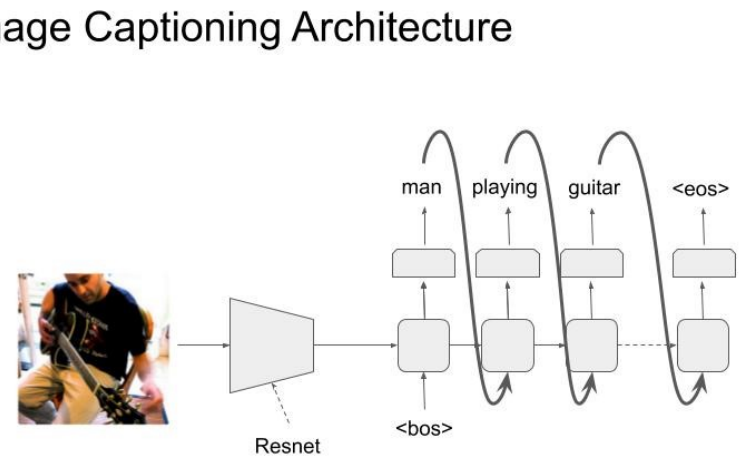



**FIGURE 13.8:** Image Captioning Concept

which attention can be implemented. But I have only one feature vector, so it cannot be used as-is in the decoder network. Therefore I came up with a different architecture in which this feature vector is added to the word and position embeddings of each output token (as shown in figure 3).

### 13.5.2 Implementation

The network was implemented in PyTorch, using the Pytorch-Lightning and Nepture.AI frameworks for experimentation. I used MS-COCO dataset (2017) which had image captioning task. The encoder and LSTM decoder code is appended below for reference, while the BERT and GPT2 decoders were used from the HuggingFace library.

```
class EncoderCNN(nn.Module):
    def __init__(self, embed_size):
        """Load the pretrained ResNet-152 and replace top fc layer."""
        super(EncoderCNN, self).__init__()
        resnet = models.resnet152(pretrained=True)
        modules = list(resnet.children())[:-1]      # delete the last fc layer.
        self.resnet = nn.Sequential(*modules)
        self.linear = nn.Linear(resnet.fc.in_features, embed_size)
        self.bn = nn.BatchNorm1d(embed_size, momentum=0.01)
```

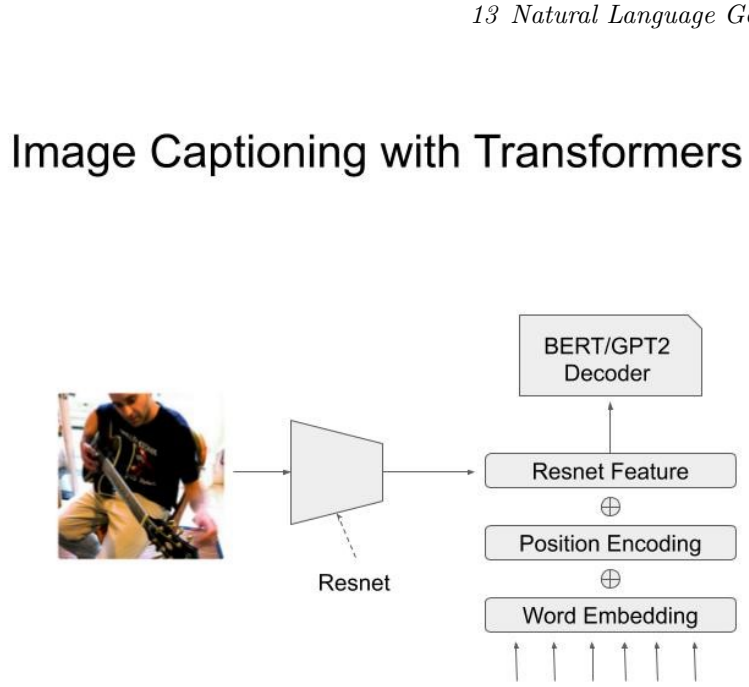



**FIGURE 13.9:** Image Captioning Architecture

```
    def forward(self, images):
        """Extract feature vectors from input images."""
        with torch.no_grad():
            features = self.resnet(images)
        features = features.reshape(features.size(0), -1)
        features = self.bn(self.linear(features))
        return features


class DecoderRNN(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size, num_layers, max_seq_length=20)
        """Set the hyper-parameters and build the layers."""
        super(DecoderRNN, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers, batch_first=True)
        self.linear = nn.Linear(hidden_size, vocab_size)
        self.max_seg_length = max_seq_length

    def forward(self, features, captions, lengths):
        """Decode image feature vectors and generates captions."""
        embeddings = self.embed(captions)
```

```
        embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
        packed = pack_padded_sequence(embeddings, lengths, batch_first=True)
        hiddens, _ = self.lstm(packed)
        outputs = self.linear(hiddens[0])
        return outputs

    def sample(self, features, states=None):
        """Generate captions for given image features using greedy search."""
        sampled_ids = []
        inputs = features.unsqueeze(1)
        for i in range(self.max_seg_length):
            hiddens, states = self.lstm(inputs, states) # hiddens: (batch_size, 1, hidden_
            outputs = self.linear(hiddens.squeeze(1))   # outputs:  (batch_size, vocab_siz
            _, predicted = outputs.max(1)               # predicted: (batch_size)
            sampled_ids.append(predicted)
            inputs = self.embed(predicted)              # inputs: (batch_size, embed_size)
            inputs = inputs.unsqueeze(1)                # inputs: (batch_size, 1, embed_si
        sampled_ids = torch.stack(sampled_ids, 1)   # sampled_ids: (batch_size, max_seq_le
        return sampled_ids
```

### 13.5.3 Results

For evaluation, I used perplexity on a held out validation set. The results for the three architectures are shown in the following figure. It can be seen that BERT outperforms LSTM with a lower perplexity and GPT2 outperforms both LSTM and BERT model.

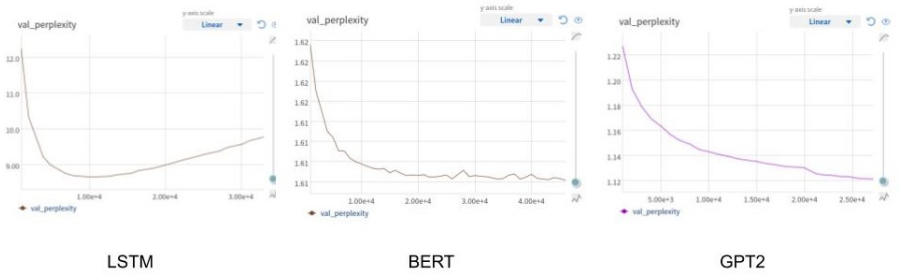



**FIGURE 13.10:** Image Captioning Results

## 13.6 Conclusion

In humanity's search for Artificial General Intelligence (AGI), natural language plays a crucial part. It is not only one of the fundamental ways to communicate with other beings, it also gives deep insight to how intelligent beings understand the world around them. And as I stated in the introduction, the best way to understand something is to be able to create/generate it. Hence natural language generation will continue to play an important part in AGI. In this chapter I have talked about only few of its manifestations; but there is a lot of scope to work in this domain.

# 14

# *Epilogue*

*Author: Matthias Aßenmacher*

Since this project was realized in a limited time frame and accounted for about one third of the ECTS points which should be achieved during one semester, it is obvious that this booklet cannot provide exhaustive coverage of the vast research field of *Natural Language Processing*.

Furthermore this area of research is moving very rapidly at the moment, which means that certain architectures, improvements or ideas had net yet even been published when we sat down and came up with the chapter topics in February 2020. Thus, this epilogue tries to put the content of this booklet into context and relate it to what is currently happening. Thereby we will focus on mainly three aspects:

- New influential (or even state-of-the-art) architectures
- Improvements and work on the Attention-mechanism and Transformers
- Work on proper evaluation and interpretability

## 14.1 New influentioal architectures

In **Chapter 7: "Transfer Learning for NLP I"**[1] and **Chapter 9: "Transfer Learning for NLP I"**[2] some of the most important models for *sequential transfer learning* have been presented. We chose to narrow ourselves down to this type of models, since we considered them to be most important to begin with, in order to unterstand the overall concept. Nevertheless, other influential architectures shall also be addressed:

- An architecture with a relatively interesting pre-training objective the interested reader might want to have a look at, is **ELECTRA**[3] (Clark et al. (2020)).

[1] ./transfer-learning-for-nlp-i.html
[2] ./transfer-learning-for-nlp-ii.html
[3] https://openreview.net/pdf?id=r1xMH1BtvB

- **Google's T5**[4] (Raffel et al. (2019)) (already briefly mentioned in Chapter 9[5]) does not fit into the category of sequential tansfer learning but rather belongs to *multi-task learning* models (cf. Fig. 7.1) in Chapter 7[6])) since it is trained on multiple tasks at once. This is possible due to transformation of the entire input *and* output to strings, which essentially converts every tasks to as seq-to-seq task.
- In May 2020 the **OpenAI GPT-3**[7] (Brown et al. (2020)) shook the NLP community and triggered *a lot* of subsequent research. This model puts, as already mentioned in Chapter 9[8], is by far bigger then every previous model and put a special focus on *few-shot learning*-

## 14.2 Improvements of the Self-Attention mechanism

Recently, there has been a lot effort put in improving the Self-Attention, mostly by reducing its computational cost and this enabling models to process longer sequences. One interesting article has already been discussed at the end of Chapter 8[9], while another interesting piece of work has been published recently by Wang et al. (2020): The so-called Linformer[10]

## 14.3 Evaluation and Interpretability

While "traditional" Benchmark (collections) have been discussed in Chapter 11[11], there is a lot of ongoing research about proper evaluation and interpretability of NLP models. Here are just two examples of impressive work, which was published very recently:

- Ribeiro et al. (2020) won the best paper award at ACL 2020 for their article Beyond Accuracy: Behavioral Testing of NLP Models with CheckList[12]

[4] https://arxiv.org/pdf/1910.10683.pdf
[5] ./transfer-learning-for-nlp-ii.html
[6] ./transfer-learning-for-nlp-i.html
[7] https://arxiv.org/pdf/2005.14165.pdf
[8] ./transfer-learning-for-nlp-ii.html
[9] ./attention-and-self-attention-for-nlp.html
[10] https://arxiv.org/pdf/2006.04768.pdf
[11] ./resources-and-benchmarks-for-nlp.html
[12] https://www.aclweb.org/anthology/2020.acl-main.442.pdf

- Google launched a Language Interpretability Tool[13] together with an accompanying research article[14] (Tenney et al. (2020))

We hope that this little outlook can adequately round off this nice piece of academic work created by extremely motivated students and we hope that you enjoyed reading.

[13] https://github.com/pair-code/lit
[14] https://arxiv.org/pdf/2008.05122.pdf

# 15

# *Acknowledgements*

The most important contributions are from the students themselves. The success of such projects highly depends on the students. And this book is a success, so thanks a lot to all the authors! The other important role is the supervisor. Thanks to all the supervisors who participated! Special thanks to Christian Heumann[1] who enabled us to conduct the seminar in such an experimental way, supported us and gave valuable feedback for the seminar structure. Thanks a lot as well to the entire Department of Statistics[2] and the LMU Munich[3] for the infrastructure.

The authors of this work take full responsibilities for its content.

[1] https://www.misoda.statistik.uni-muenchen.de/personen/professoren/heumann/index.html

[2] https://www.statistik.uni-muenchen.de/

[3] http://www.en.uni-muenchen.de/index.html

# Bibliography

(1966). Language and machines. *National Academy of Sciences National Research Council*.

(2015). Understanding lstm networks.

Alammar, J. (2018). The illustrated transformer [blog post]. *http://jalammar.github.io/*.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bakarov, A. (2018). A survey of word embeddings evaluation methods. *arXiv preprint arXiv:1801.09536*.

Beltagy, I., Lo, K., and Cohan, A. (2019). Scibert: A pretrained language model for scientific text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3606–3611.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, (3):1137–1155.

Bobrow, D. G. (1964). Natural language input for a computer problem solving system.

Boden, M. (2002). A guide to recurrent neural networks and backpropagation. *the Dallas project*.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

Boughorbel, S., Jarray, F., and El-Anbari, M. (2017). Optimal classifier for imbalanced data using matthews correlation coefficient metric. *PloS one*, 12(6):e0177678.

Boureau, Y.-L., Ponce, J., and LeCun, Y. (2010). *A Theoretical Analysis of Feature Pooling in Visual Recognition*.

Bowman, S. R., Angeli, G., Potts, C., and Manning, C. D. (2015). A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

Bruni, E., Tran, N.-K., and Baroni, M. (2014). Multimodal distributional semantics. *Journal of Artificial Intelligence Research*, 49:1–47.

Chen, G. (2016). A gentle tutorial of recurrent neural network with error backpropagation. *arXiv preprint arXiv:1610.02583*.

Cheng, J., Dong, L., and Lapata, M. (2016). Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*.

Child, R., Gray, S., Radford, A., and Sutskever, I. (2019). Generating long sequences with sparse transformers.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Chollet, F. (2018). *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG.

Chomsky, N. (1957). *Syntactic structures*. The Hague: Mouton.

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Davis, J., Sarlos, T., Belanger, D., Colwell, L., and Weller, A. (2020). Masked language modeling for proteins via linearly scalable long-context transformers.

Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555.

Clark, K., Luong, M.-T., Le, Q. V., and Manning, C. D. (2020). Electra: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. P. (2011). *Natural Language Processing (Almost) from Scratch*.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Deng, L. and Liu, Y. (2018). *Deep learning in natural language processing*. Springer Nature.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135.

Gao, B., Bian, J., and Liu, T.-Y. (2014). Wordrep: A benchmark for research on learning word representations. *arXiv preprint arXiv:1407.1640*.

Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. (2017). Convolutional sequence to sequence learning.

Gerz, D., Vulić, I., Hill, F., Reichart, R., and Korhonen, A. (2016). Simverb-3500: A large-scale evaluation set of verb similarity. *arXiv preprint arXiv:1608.00869*.

Gladkova, A. and Drozd, A. (2016). Intrinsic evaluations of word embeddings: What can we do better? In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pages 36–42.

Goldberg, Y. (2016). A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420.

Goldberg, Y. and Levy, O. (2014). word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *CoRR*, abs/1410.5401.

Gregor, K., Danihelka, I., Graves, A., Rezende, D. J., and Wierstra, D. (2015). Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*.

Hancox, P. (1996).

Harris, Z. S. (1954). Distributional structure. *WORD*, 10(2-3):146–162.

He, K., Zhang, X., Ren, S., and Sun, J. (2016a). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

He, K., Zhang, X., Ren, S., and Sun, J. (2016b). Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer.

Hill, F., Reichart, R., and Korhonen, A. (2015). Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 41(4):665–695.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Howard, J. and Ruder, S. (2018). Universal Language Model Fine-tuning for Text Classification. *arXiv e-prints*, page arXiv:1801.06146.

Hutchins, J. (2005). The first public demonstration of machine translation: the georgetown-ibm system, 7th january 1954. *noviembre de*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ArXiv*, abs/1502.03167.

Johnson, M. (2009). How the statistical revolution changes (computational) linguistics. *Proceedings of the EACL 2009 Workshop on the Interaction between Linguistics and Computational Linguistics Virtuous, Vicious or Vacuous? - ILCL 09*.

Johnson, R. and Zhang, T. (2016). Convolutional neural networks for text categorization: Shallow word-level vs. deep character-level. *ArXiv*, abs/1609.00718.

Johnson, R. and Zhang, T. (2017). Deep pyramid convolutional neural networks for text categorization. In *ACL*.

Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2016). Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.

Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2017). Bag of tricks for efficient text classification. *ArXiv*, abs/1607.01759.

Jurgens, D., Mohammad, S., Turney, P., and Holyoak, K. (2012). Semeval-2012 task 2: Measuring degrees of relational similarity. In ** SEM 2012: The First Joint Conference on Lexical and Computational Semantics–Volume 1: Proceedings of the main conference and the shared task, and Volume 2: Proceedings of the Sixth International Workshop on Semantic Evaluation (SemEval 2012)*, pages 356–364.

Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., and Uszkoreit, J. (2017). One model to learn them all. *arXiv preprint arXiv:1706.05137*.

Kalchbrenner, N., Espeholt, L., Simonyan, K., van den Oord, A., Graves, A., and Kavukcuoglu, K. (2016a). Neural machine translation in linear time. *ArXiv*, abs/1610.10099.

Kalchbrenner, N., Espeholt, L., Simonyan, K., van den Oord, A., Graves, A., and Kavukcuoglu, K. (2016b). Neural machine translation in linear time.

Kalchbrenner, N., Grefenstette, E., and Blunsom, P. (2014). *A Convolutional Neural Network for Modelling Sentences*.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. (2020). Transformers are rnns: Fast autoregressive transformers with linear attention.

Kim, Y. (2014). *Convolutional Neural Networks for Sentence Classification*.

Kitaev, N., Łukasz Kaiser, and Levskaya, A. (2020). Reformer: The efficient transformer.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). *ImageNet Classification with Deep Convolutional Neural Networks*.

Kudo, T. and Richardson, J. (2018). Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2019). Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*.

Le, Q. and Mikolov, T. (2014). Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196.

Levy, O., Goldberg, Y., and Dagan, I. (2015). Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225.

Ling, W., Yogatama, D., Dyer, C., and Blunsom, P. (2017). Program induction by rationale generation: Learning to solve and explain algebraic word problems. *arXiv preprint arXiv:1705.04146*.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Luong, M.-T., Le, Q. V., Sutskever, I., Vinyals, O., and Kaiser, L. (2015a). Multi-task sequence to sequence learning. *arXiv preprint arXiv:1511.06114*.

Luong, M.-T., Pham, H., and Manning, C. D. (2015b). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Luong, M.-T., Socher, R., and Manning, C. D. (2013). Better word representations with recursive neural networks for morphology. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 104–113.

Malte, A. and Ratadiya, P. (2019). Evolution of transfer learning in natural language processing.

Manning, C. D. and Schutze, H. (2008). *Foundations of statistical natural language processing*. MIT.

Martinc, M., Pollak, S., and Robnik-Šikonja, M. (2019). Supervised and unsupervised neural approaches to text readability. *arXiv preprint arXiv:1907.11779*.

McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.

Mccorduck, P. and Cfe, C. (2004). Machines who think. *A K Peters/CRC Press*.

Merity, S., Shirish Keskar, N., and Socher, R. (2017). Regularizing and Optimizing LSTM Language Models. *arXiv e-prints*, page arXiv:1708.02182.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.

Mikolov, T., Le, Q. V., and Sutskever, I. (2013b). Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013c). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, pages 3111–3119.

Mikolov, T., Yih, W.-t., and Zweig, G. (2013d). Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pages 746–751.

Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer.

Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2013a). How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013b). On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318.

Patel, K. and Bhattacharyya, P. (2017). Towards lower bounds on number of dimensions for word embeddings. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 31–36.

Pennington, J., Socher, R., Manning, and Christopher D. (2014). Glove: Global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. *arXiv e-prints*, page arXiv:1802.05365.

Peters, M. E., Ruder, S., and Smith, N. A. (2019). To Tune or Not to Tune? Adapting Pretrained Representations to Diverse Tasks. *arXiv e-prints*, page arXiv:1903.05987.

Prabhavalkar, R., Rao, K., Sainath, T. N., Li, B., Johnson, L., and Jaitly, N. (2017). A comparison of sequence-to-sequence models for speech recognition. In *Interspeech*, pages 939–943.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Rajpurkar, P., Jia, R., and Liang, P. (2018). Know what you don't know: Unanswerable questions for squad.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text.

Reddy, S., Chen, D., and Manning, C. D. (2018). Coqa: A conversational question answering challenge. *CoRR*, abs/1808.07042.

Ribeiro, M. T., Wu, T., Guestrin, C., and Singh, S. (2020). Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912, Online. Association for Computational Linguistics.

Ruder, S. (2019). *Neural Transfer Learning for Natural Language Processing*. PhD thesis, National University of Ireland, Galway.

Ruder, S., Peters, M. E., Swayamdipta, S., and Wolf, T. (2019). Transfer learning in natural language processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pages 15–18.

Scherer, D., Müller, A. C., and Behnke, S. (2010). Evaluation of pooling operations in convolutional architectures for object recognition. In *ICANN*.

Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681.

Schwenk, H., Barrault, L., Conneau, A., and LeCun, Y. (2017). *Very Deep Convolutional Networks for Text Classification*.

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Shen, Z., Zhang, M., Zhao, H., Yi, S., and Li, H. (2018). Efficient attention: Attention with linear complexities.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

Sutskever, I., Martens, J., Dahl, G. E., and Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. In *ICML*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Tenney, I., Wexler, J., Bastings, J., Bolukbasi, T., Coenen, A., Gehrmann, S., Jiang, E., Pushkarna, M., Radebaugh, C., Reif, E., et al. (2020). The language interpretability tool: Extensible, interactive visualizations and analysis for nlp models. *arXiv preprint arXiv:2008.05122*.

Tsai, Y.-H. H., Bai, S., Yamada, M., Morency, L.-P., and Salakhutdinov, R. (2019). Transformer dissection: A unified understanding of transformer's attention via the lens of kernel.

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265.

Turing, A. M. (1948). Intelligent machinery.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, LIX(236):433–460.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. W., and Kavukcuoglu, K. (2016). *WaveNet: A Generative Model for Raw Audio*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Venugopalan, S., Rohrbach, M., Donahue, J., Mooney, R., Darrell, T., and Saenko, K. (2015). Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542.

Visin, F., Kastner, K., Cho, K., Matteucci, M., Courville, A. C., and Bengio, Y. (2015). *ReNet: A Recurrent Neural Network Based Alternative to Convolutional Networks*.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1058–1066, Atlanta, Georgia, USA. PMLR.

Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. (2019). Superglue: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems*, pages 3261–3275.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. (2018). Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. (2020). Linformer: Self-attention with linear complexity.

Weaver, W. (1949). The mathematics of communication. *Scientific American*, 181(1):11–15.

Weizenbaum, J. (1966). Eliza-a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45.

Weng, L. (2018). Attention? attention! *lilianweng.github.io/lil-log*.

Winograd, T. (1972). Shrdlu: A system for dialog.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057.

Yamaguchi, K., Sakamoto, K., Akabane, T., and Fujimoto, Y. (1990). *A neural network for speaker-independent isolated word recognition*.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. (2019). Xlnet: Generalized autoregressive pretraining for language understanding. In Wallach, H., Larochelle, H., Beygelzimer, A.,

d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 5753–5763. Curran Associates, Inc.

Yeung, J. (2020). Three major fields of artificial intelligence and their industrial applications.

Zhang, X., Zhao, J. J., and LeCun, Y. (2015). *Character-level Convolutional Networks for Text Classification*.

Zhang, Z., Han, X., Liu, Z., Jiang, X., Sun, M., and Liu, Q. (2019). Ernie: Enhanced language representation with informative entities. *arXiv preprint arXiv:1905.07129*.

Zhu, Y., Kiros, R., Zemel, R. S., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. (2015). Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *CoRR*, abs/1506.06724.