# *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, we are planning to review 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. As Attention-based models and transfer learning approaches are the foundation of most of the recent state-of-the-art models, we will cover these two topics extensively in the second part of our seminar.

We will furthermore talk about software implementations of these methods and benchmark tasks/data sets for evaluating state-of-the-art models.

This book is the outcome of the seminar "Modern Approaches in Natural Language Processing" which took place in summer 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: Christoph Molnar*

This book is the result of an experiment in university teaching. Each semester, students of the Statistics Master can choose from a selection of seminar topics. Usually, every student in the seminar chooses a scientific paper, gives a talk about the paper and summarizes it in the form of a seminar paper. The supervisors help the students, they listen to the talks, read the seminar papers, grade the work and then … hide the seminar papers away in (digital) drawers. This seemed wasteful to us, given the huge amount of effort the students usually invest in seminars. An idea was born: Why not create a book with a website as the outcome of the seminar? Something that will last at least a few years after the end of the semester. In the summer term 2020, some Statistics Master students signed up for our seminar entitled "Limitations of Interpretable Machine Learning". When they came to the kick-off meeting, they had no idea that they would write a 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 part of a book, and not an isolated seminar paper.
2. We gave challenges to the students, instead of papers. The challenge was to investigate a specific limitation of interpretable machine learning methods.
3. We designed the work to live beyond the seminar.
4. We emphasized collaboration. Students wrote some chapters in teams and reviewed each others 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 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[2].

[2] link/to/repo

# 1

# *Introduction*

*Author:*

*Supervisor:*

## 1.1 Intro About the Seminar Topic

## 1.2 Outline of the Booklet

# 2

# *Introduction: Deep Learning for NLP*

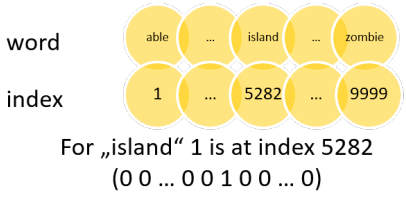
*Authors: Viktoria Szabo, Marianna Plesiak, Rui Yang*

*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. (2013) 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 previous elements. This information is stored in hidden states $h^{(t)}$ as the network

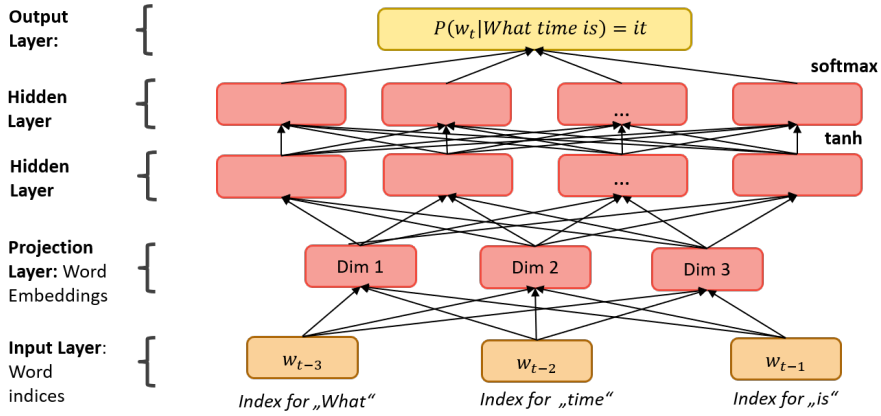



**FIGURE 2.2:** Feed-forward Neural Network. Source: Own figure based on Bengio et al. 2013.

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).

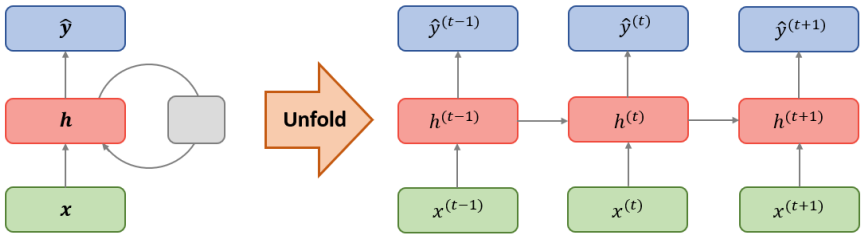



**FIGURE 2.3:** Right: Circuit diagram (left) and unfolded computational graph (right) of a simple RNN. Source: Own figure.

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. (2013) 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.

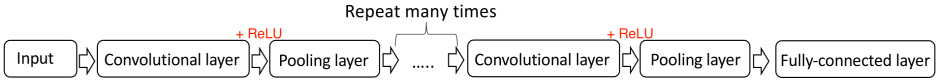



**FIGURE 2.4:** Basic structure of CNN. Source: Own figure.

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.

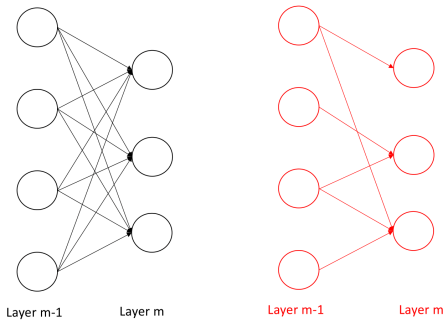



**FIGURE 2.5:** Comparison between the fully-connected and partial connected architecture. Source: Own figure.

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 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*

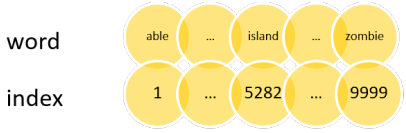
*Supervisor: Christian Heumann*

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 you can download and use pretrained word embeddings. With this, you can save a lot of time for training the final model. But if the task is not a standard one you probably want to train your own embeddings to get a better model performance for your specific task. In the following I will first outline the evolution from sparse representations of words to dense word embeddings. After that I will describe calculation methods for word embeddings within a neural network language model and with word2vec and GloVe. In the third part I will show how to improve the model performance regardless of the chosen model class based on hyperparameter tuning and system design choices and explain 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. Last but not least I will present some resources to download pretrained word embeddings.

## 3.1 The Evolution of Word Embeddings

Since computers work with numeric representations, converting the text and sentences we want to analyze into numbers is unavoidable. One-Hot Encoding and Bag-of-Words (BOW) are two simple approaches how this could be down. These methods are usually used as input for calculating more elaborate word representations called word embeddings. Only use these approaches without calculating word embeddings if you have a small amount of distinct words in your document, the words are not meaningfully correlated and you have a lot of data to learn from. The **One-Hot Encoding** labels each word in the vocabulary with size n with an index. After that each word is represented by a vector with dimension n. Every dimension 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. You can see an example for this representation in figure 2.1. 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. You can see an example for this approach on the right side in figure 2.1.




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. First of they are very simple to construct. Furthermore, they are robust to changes and it was observed that simple models trained on huge 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. Nevertheless, the **problems** coming from these approaches usually outweigh the positive points. The most obvious one is that these approaches give you 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. Neural networks in particular struggle with this type of data. And with growing vocabulary the feature size vectors also increase by the same length. So, the dimensionality of these approaches is the same as the number of different words in your text. That means you need to estimate more parameters and therefore you require exponentially more data to do well enough to build a reasonably generalizable model. This is called 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 word, such as "the" 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. It's very difficult to make predictions about things

unlike you've ever seen before – much easier if it's related to something you have seen.

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 semantic similarity can be maintained in the representation and generalization can be achieved. Through the vectors the words are mapped to a continuous vector space (in NLP usually called 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 example the words are represented by a three-dimensional vector.

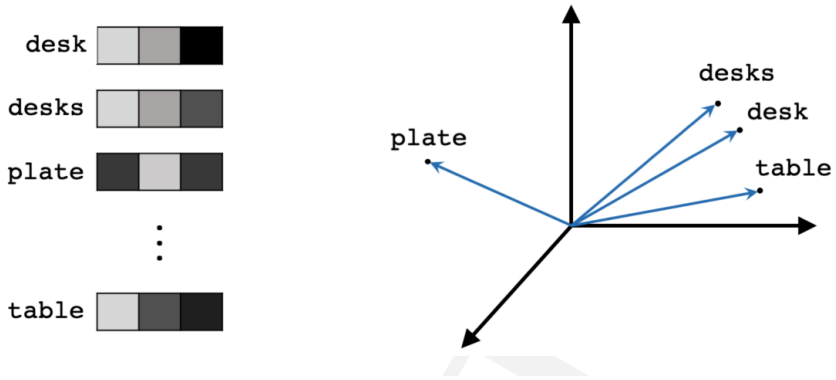



**FIGURE 3.2:** Example for word embeddings with three dimensions. Source: ...

If you want to represent higher dimensional word vectors you 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. You can see an example of this for a few country names and their capitals in figure 3.3. You can see that 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 similar y-axis values as their corresponding capitals.

## 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

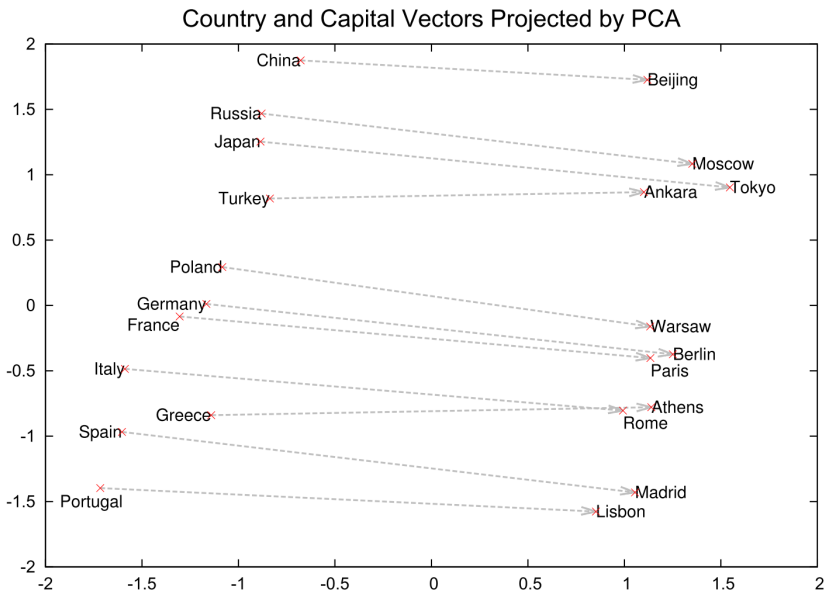



**FIGURE 3.3:** Two-dimensional PCA projection of 1000-dimensional word vectors of countries and their capital cities. Source: Mikolov et al. 2013 (3)

contexts tend to have similar meanings. For instance, Jupiter and Venus tend to have similar semantics since they usually appear in similar contexts, e.g., with words such as solar system, star, planet, and astronomy. That's why 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 and is one of the winning points when compared to other knowledge representation approaches. That means word embeddings can be thought of unsupervised feature extractors for words. However, the methods to find such similarities in the context of words vary. First finding word representations started out with the nowadays so called more traditional count-based techniques, which collected word statistics like occurrence and co-occurrence frequencies of words as seen above with BOW. But these representations are often large and need 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, word embeddings supplanted the traditional count-based word representations. It is also said that these models learn "dense representations" since they directly learn low-dimensional word representations, without needing to resort to the additional dimensionality

reduction step. In the following I will give an introduction to the best-known predictive approaches to model word embeddings.

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

Bengio et al. in 2003 were the first to propose learning word embeddings within a statistical neural network language model (NNLM). That means the model first learns a distributed representation for each word and in a second step the probability function for word sequences. The probability function is expressed as a product of conditional probabilities of the target word given the context words:

Formula

This function has parameters that can be iteratively tuned in order to maximize the log-likelihood of the training data. One of the main advantages of these models is that the distributed representation achieves a level of generalization that is not possible with traditional models. By training a neural network language model, one obtains not just the model itself, but also the learned word representations, which may be used for other, potentially unrelated, tasks. Generally, as said above a statistical model of language can be represented by the conditional probability of the target word given some context words. A good model is a function that gives high out-of-sample likelihood. Therefore, parameters are searched via the neural network which maximize the training corpus penalized log-likelihood including a regularization term. The proposed neural network architecture has an input layer with one-hot encoded word inputs, a linear projection layer for the word embeddings, a hidden layer, where most of the computation is done, followed by a softmax classifier output layer, which guarantees positive probabilities summing to 1. Funktion softmax. The output of the model is a vector whose i-th element estimates the probability P(w_t=i|c). You can see the model architecture in figure 3.4 .

The model is trained using the **stochastic gradient descent** optimizer and **backpropagation**. The optimization function is used to identify how the weights are adjusted. During training, the minimum loss function is sought. The gradient descent optimizer tries to find the direction of the strongest descent via partial derivation. (In contrast to maximum likelihood) The learning rate defines the size of the step in this direction. For the NNLM above the gradient descent optimizer performs the following iterative update:

Formula

where e denotes the learning rate. The method by which weight adjustments are made during training so that they can be optimized is called backpropagation. Backpropagation essentially consists of six steps: 1. (Random) Initialization of the weights of the network 2. Calculation of y(x_i) for the inputs

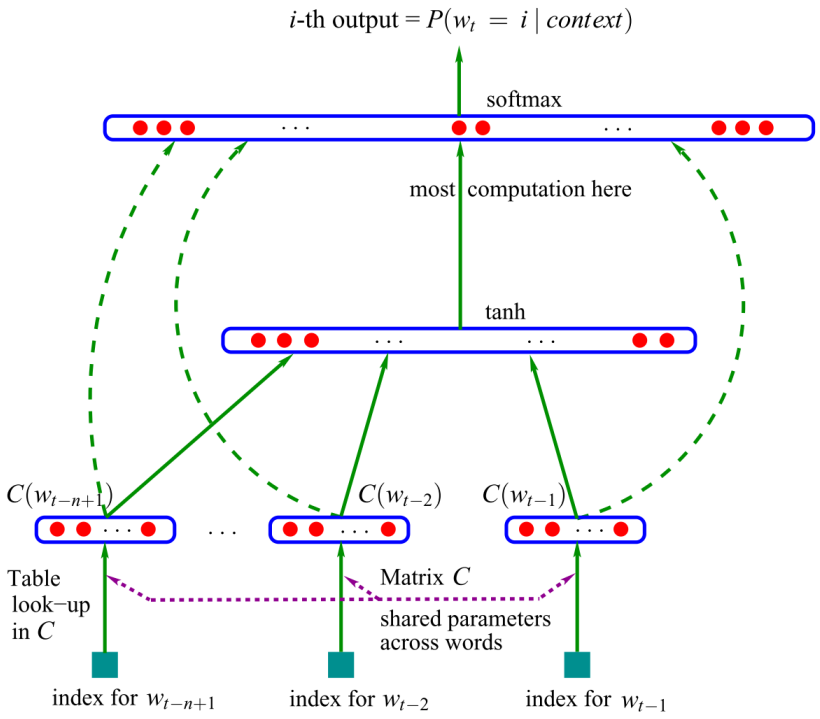



**FIGURE 3.4:** Architecture for NNLM proposed by Bengion et al. in 2003. Source: Bengio et al. 2003

x_i 3. Determining the cost of the inputs x_i with the loss function 4. Calculation of the partial derivatives of the loss for each weight (gradient descent optimizer) 5. Update the weights 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 proposed the two word2vec algorithms for learning word embeddings which led to a huge wave in NLP popularizing word embeddings. In contrast to the NNLM model above the word2vec algorithm first learns the embeddings in a specific model and then they are used as input in the final neural language model. 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. That means compared to the previously explained feedforward NNLM the non-linear hid-

den layer in between is removed. The general idea behind CBOW is to predict the target word based on a window of context words. The order of context words does not influence the prediction, thus the name Bag-of-Words. While Skip-Gram tries to predict the context words given a target word. This is done while adjusting the initial weights during training so that a loss function is reduced. These weights are used in the end as word embeddings. 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 and N the number of future and past words. The projection layer is a standard fully connected (dense) layer with D neurons whose weights are the word embeddings and D is the number of dimensions for the word embeddings. The projection layer is shared for all words; thus, all words get projected into the same position (their vectors are averaged) in a linear manner. The output layer outputs probabilities for all words from the vocabulary and has therefore a dimensionality of V. That means the output is a probability distribution over all words in the vocabulary. A log-linear softmax classifier is used for the calculation of the probabilities, where the training criterion is to correctly classify the target word. The model architecture is shown in figure 3.5 on the left side. The continuous Skip-gram architecture also uses a log-linear classifier with a continuous projection layer, but the input is only one "current" word, and the output layer consists of words within a certain range before and after the current word. Also, since the more distant words are usually less related to the current word, it weighs nearby context words more heavily than more distant context words by sampling less from those words in the training examples. You can find the model architecture on the right side of figure 3.5.

Skip-gram works well with small amounts of training data and represents even 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 Global Vector 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 different 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 equals the log of the number of times the two words will occur near each other. Formel? The model builds on the idea that you can 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 their semantic association than a direct co-occurrence probability. Let P_ij = P(j|i) = X_ij/X_i be the probability that word j appears in the

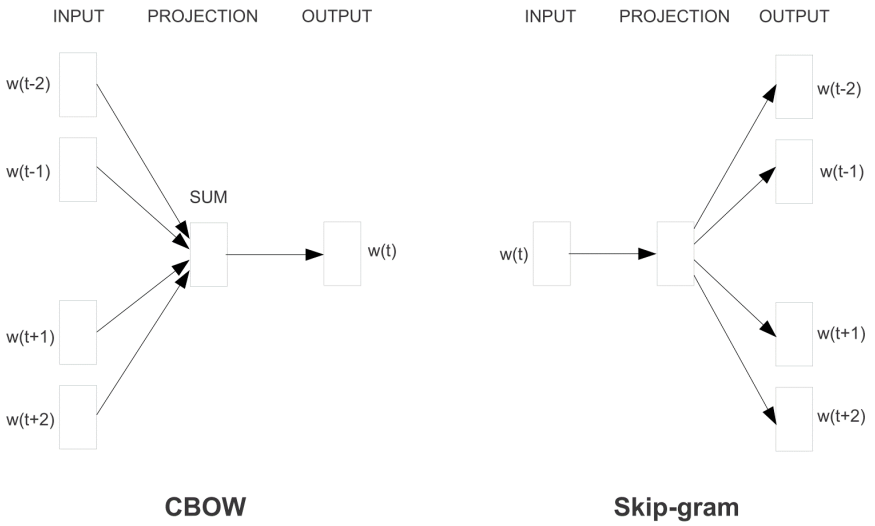



**FIGURE 3.5:** Learning word embeddings with the model architecture of CBOW and Skip-Gram. Source: Mikolov et al. 2013 (3)

context of word i. In figure **??** you can see 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. Therefore, we can see that the comparison of co-occurrence with a third word is more indicative for the semantic meanings and makes a better job at distinguishing relevant words (solid and gas) from irrelevant words (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 |

That's why Pennington et al. tried to find a way to incorporate the ratio P_ik/P_jk into computing the word embeddings. They proposed a new weighted least squares regression model with the following equation:

Formel

, where V is the size of the vocabulary, f(X_ij) is a weighting function that looks like this:

Formel2

GloVe does not use neural networks, but is still categorized as predictive model since it is different from conventional count-based models in that it uses stochastic gradient descent to optimize a non-convex objective.

## 3.3 Model Improvements

In this chapter I will describe some system design choices, which will tackle some of the problems posed by the algorithms mentioned above.

### 3.3.1 fastText

GloVe and word2vec can't cope with completely unseen words. To overcome this problem fastText was introduced (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, "artificial" with n=3, the fastText representation of this word is <ar, art, rti, tif, ifi, fic, ici, ial, al>, where the angular brackets indicate the beginning and end of the word. That means fastText learns embeddings for n-grams of words in addition for the whole word. This helps capture the meaning of shorter words and allows the embeddings to understand suffixes and prefixes. Then, in the case of an unseen word, the corresponding embedding is induced by averaging the vector representations of its constituent character n-grams. Once the word has been represented using character n-grams, a skip-gram or CBOW model is trained to learn the embeddings. That's why fastText works well with rare words. So even if a word wasn't seen during training, it can be broken down into n-grams to get its embeddings.

### 3.3.2 Word Phrases

### 3.3.3 Multiple Meanings per Word

## 3.4 Hyperparameter tuning

Once adapted across methods, hyperparameter tuning significantly improves performance in every task. In many cases, changing the setting of a single hyperparameter yields a greater increase in performance than switching to a

better algorithm or training on a larger corpus. Levy et al. 2015 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. In fact, there is no consistent advantage to one algorithmic approach over another.

**Number of dimensions for word embeddings**

Intuitively, you probably want more dimensions the more types of input that you have. So, you would want more dimensions if you're calculating embeddings for words, of which there are probably many tens of thousands of types in your corpus, versus if you're calculating embeddings for parts-of-speech tags (like "noun", "verb", "adjective"), of which there are not that many types. The question of how many embedding dimensions is best actually turns out to be an empirical question, and the optimal number has not been worked out theoretically. The trade-off here is between accuracy and computational concerns. More dimensions means the potential to compute increasingly accurate representations of words. But more dimensions also means a higher demand for computational resources (processing speed, memory requirements) — this is more apparent during the training phase, but it does also affect inference speed. In practice, people use word embedding vectors with dimensions around 50 to 500 (you run across 300 a lot), and sometimes more for very competitive systems where you're trying to squeeze as much performance out of the model as possible.

**Size of context windows**

The traditional approaches usually use a constant-sized unweighted context window. For instance, if the window size is 5, then a word five tokens apart from the target is treated the same as an adjacent word. Following the intuition that contexts closer to the target are more important, context words can be weighted according to their distance from the focus word. Both GloVe and word2vec employ such a weighting scheme.

**Subsampling of frequent words**

Mikolov et al 2013 also proposed for their word2vec algorithms to only use a subsample of the most frequent words. This leads to a significant speedup and improves accuracy of the representations of less frequent words. GloVe also uses this subsampling technique but with another probability of discarding. formula for discarding probability for Glove and Word2vec:

**Negative sampling** In their first paper mikilov et al. 2013 proposed hierarchical softmax, but soon they published a new method called negative sampling which is more efficient in the calculation. It does so by trying to maximize a function of the product word and context word pairs that occur in the training corpus, and minimize it for negative examples of word and context word pairs that do not necessarily occur in the training corpus. The negative exam-

ples are created by stochastically corrupting observed pairs – hence the name "negative sampling".

## 3.5 Evaluation Methods

**Word Similarity Task**

There are six datasets to evaluate word similarity:

- WordSim353 (Finkelstein et al., 2002) partitioned into two datasets, WordSim Similarity and WordSim Relatedness (Zesch et al., 2008; Agirre et al., 2009),
- Bruni et al.'s (2012) MEN dataset,
- Radinsky et al.'s (2011) Mechanical Turk dataset,
- Luong et al.'s (2013) Rare Words dataset,
- Hill et al.'s (2014) SimLex-999 dataset.

All these datasets contain word pairs together with human-assigned similarity scores. The word vectors are evaluated by ranking the pairs according to their cosine similarities, and measuring the correlation (Spearman's ) with the human ratings.

**Word Analogy Task**

The two analogy datasets present questions of the form "a is to a∗ as b is to b∗", where b∗ is hidden, and must be guessed from the entire vocabulary.

- MSR's analogy dataset (Mikolov et al., 2013c) contains 8000 morpho-syntactic analogy questions, such as "good is to best as smart is to smartest".
- Google's analogy dataset (Mikolov et al., 2013a) contains 19544 questions, about half of the same kind as in MSR (syntactic analogies), and another half of a more semantic nature, such as capital cities ("Paris is to France as Tokyo is to Japan").

## 3.6 Sources and Applications of Word Embeddings

# 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

Recurrent neural networks allow 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. In the second part, gated units, an improved way to calculate hidden states, are explained. The third part gives an overview of some extended versions of RNNs and their applications in NLP.

### 4.1.1 Network Structure and Forwardpropagation

The repetitive structure of RNNs can be visualised with help of an unfolded computational graph (see 4.1).

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 previous chapter. Reccurently 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 passes it through 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. Given the hidden sequences, output vectors $\hat{y}^{(t)}$ are used to build the predic-

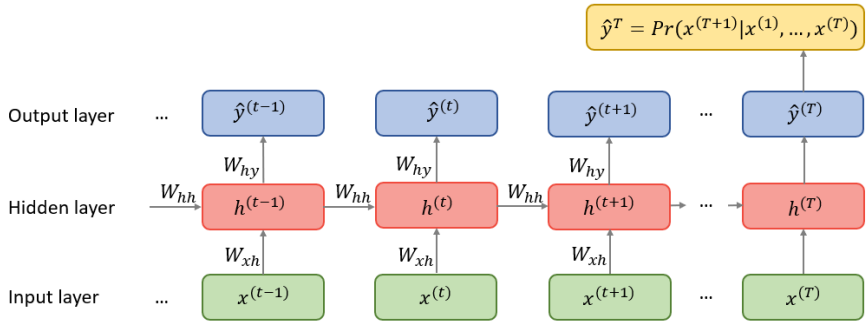



**FIGURE 4.1:** Unfolded computatinal graph of a RNN. Source: Own figure.

tive 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 this is not needed, 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 a same function $f$. This function can be considered a universal model which is shared across all time steps and is generalized for all sequence lengths. This 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)

Consider a recurrent neural network with one hidden layer that is used to predict words or characters, so the output is 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:

$$h^{(t)} = \mathcal{H}(a + W_{hh}h^{(t-1)} + W_{xh}x^{(t)}) \tag{4.3}$$
$$y^{(t)} = \mathcal{Y}(b + W_{hy}h^{(t)}) \tag{4.4}$$
$$\tag{4.5}$$

where the parameters and functions denote the following:

- $\mathcal{H}$: hidden layer function. Usually it is a sigmoid activation function ( Sutskever et al. (2014) and Mikolov et al. (2010)) or tanh or ReLu
- $W_{hh}$: weight matrix connecting recurrent connections between hidden states
- $W_{xh}$: weight matrix connecting inputs to hidden layer
- $W_{hy}$: weight matrix connecting hidden states to outputs (softmax if we want to predict next word or letter)
- $\mathcal{Y}$: output layer function. If the model is used to predict words, 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.

(Graves, 2013)

Because 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 network is used to compute the predictive distributions $Pr(x^{(t+1)}|y^{(t)})$ at each time instance $t$, the network distribution is denoted as:

$$Pr(x) = \prod_t Pr(x^{(t+1)}|y^{(t)}) \tag{4.6}$$

$$\tag{4.7}$$

and the total loss used for training is simply the sum of the losses over all time steps denoted as the negative log-likelihood of $Pr(x)$:

$$\mathcal{L}(x) = -\sum_t \log Pr(x^{(t+1)}|y^{(t)}) \tag{4.8}$$

$$\tag{4.9}$$

(Graves, 2013)

### 4.1.2 Backpropagation

In order 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 muste be applied recursively and the gradients are summed up through the network. (Boden, 2002)

Gradients w.r.t. $W_{hy}$:

- for a single time step $t$:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{hy}} = \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial W_{hy}} \tag{4.10}$$

$$\tag{4.11}$$

- for the whole sequence:

$$\frac{\partial \mathcal{L}}{\partial W_{hy}} = \sum_t \frac{\partial \mathcal{L}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial W_{hy}} \tag{4.12}$$

$$\tag{4.13}$$

Gradients w.r.t. $W_{hh}$:

- for a single time step $t$:

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

The last part $h^{(t)}$ also depends on $h^{(t-1)}$ and the gradient can be rewritten as:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{hh}} = \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \frac{\partial h^{(t-1)}}{\partial W_{hh}} \tag{4.15}$$

$$= \sum_{k=0}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{hh}} \tag{4.16}$$

- for the whole sequence:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_t \sum_{k=0}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{hh}} \tag{4.17}$$

Gradient w.r.t. $W_{xh}$ is similar to $W_{hh}$:

- for a single step:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{xh}} = \sum_{k=0}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{xh}} \tag{4.18}$$

- for the whole sequence:

$$\frac{\partial \mathcal{L}}{\partial W_{xh}} = \sum_t \sum_{k=0}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{xh}} \tag{4.19}$$

(Chen, 2016)

### 4.1.3 Vanishing and Exploding Gradients

## 4.2 Gated RNNs

### 4.2.1 LSTM

Long Short Term Memory networks were introduced by Hochreiter and Schmidhuber (1997) to deal with problems of long term dependencies. Instead of a simple hidden unit that combines inputs and previous hidden states linearly and outputs the non-linear transformation to the next step, hidden units are now extended by special input, forget and output gates which 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:

$$f^{(t)} = sigm(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f) \tag{4.20}$$
$$i^{(t)} = sigm(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i) \tag{4.21}$$
$$o^{(t)} = sigm(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o) \tag{4.22}$$
$$g^{(t)} = tanh(W_{xc}x^{(t)} + W_{hc}h^{(t-1)} + b_c) \tag{4.23}$$
$$c^{(t)} = f^{(t)}c^{(t-1)} + i^{(t)}g^{(t)} \tag{4.24}$$
$$h^{(t)} = o^{(t)}tanh(c^{(t)}) \tag{4.25}$$
$$\tag{4.26}$$

(Graves, 2013)

First, forget gate $f^{(t)}$ decides which values of the previous output $h^{(t-1)}$ to forget. The next step is deciding which information will be stored in the internal cell state $c^{(t)}$. This step consists of two parts: 1) multiplication of the old state $c^{(t-1)}$ by $f^{(t)}$ (forgetting information); 2) adding new candidates calculated in $g^{(t)}$ with help of its multiplication by values from the input gate $i^{(t)}$ (adding new information). 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 values which were chosen.(Goodfellow et al. (2016), Graves (2013))

### 4.2.2 GRU

Invented by Cho et al. (2014). Is simpler because it includes only two gates: reset and update

The hidden unit is calculated as:

$$r^{(t)} = sigm(W_{xr}x^{(t)} + W_{hr}h^{(t-1)}) \tag{4.27}$$
$$z^{(t)} = sigm(W_{xz}x^{(t)} + W_{hz}h^{(t-1)}) \tag{4.28}$$
$$\tilde{h}^{(t)} = tanh(W_{xh}x^{(t)} + W_{hh}(r^{(t)}h^{(t-1)})) \tag{4.29}$$
$$h^{(t)} = z_j h_j^{(t-1)} + (1 - z_j)\tilde{h}^{(t)} \tag{4.30}$$
$$\tag{4.31}$$

Illustration LSTM vs GRU

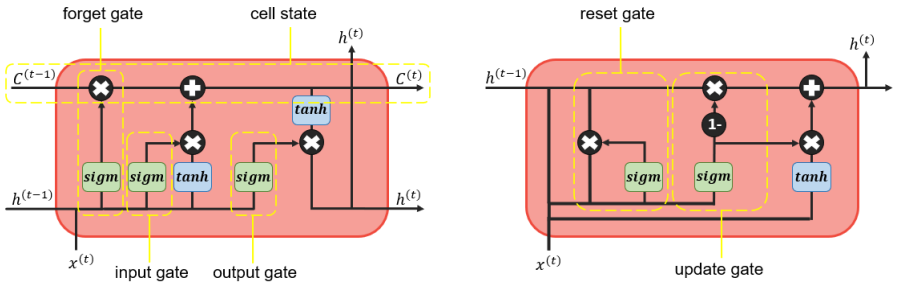



**FIGURE 4.2:** Structure of a hidden unit. LSTM on the right and GRU on the left. Source: Own figure inspired by http://colah.github.io/posts/2015-08-Understanding-LSTMs/

## 4.3 Extentions of Simple RNNs

### 4.3.1 Bidirectional RNNs

One of the limitations of simple RNNs is that they can process the information only up to a present time instance. In many NLP fields however, prediction may also depend on future inputs. Bidirectional RNNs, introduced by Schuster and Paliwal (1997), handle this issue by training on inputs in original as well as reversed order. The idea is to stack two 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. Such a composition allows using both past and future inputs at any time instance. Since both types of hidden states do not interact, the bidirectional RNNs follow almost the same training rules as a regular RNN with small differences regarding backpropagation.

### 4.3.2 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. However, a deep RNN is defined as a network with several input, hidden, or output layers stacked on top of each other. Such a structure can be more efficient at representing complex dependencies although it also requires higher computational effort because the distances between two variables at $t$ and $t+1$ become longer. (Pascanu et al., 2013)

Pascanu et al. (2013) proposed three ways to make an RNN deeper. Extending input-to-hidden functions helps to transform the raw input, e.g. one-hot encoded vectors, into a more convenient and informative representations, e.g. word embeddings, for higher levels. A deep hidden-to-hidden composition allows for the hidden states to effectively add new information to the accumulated summaries from the previous steps. For example, the model preserves useful information from the past although new inputs vary strongly and rapidly. Finally, a deep hidden-to-output function is used to make hidden states more compact and therefore enables the model to summarize the previous inputs more efficiently.

### 4.3.3 Encoder-Decoder Architecture

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 consists of two parts as illustrated below:

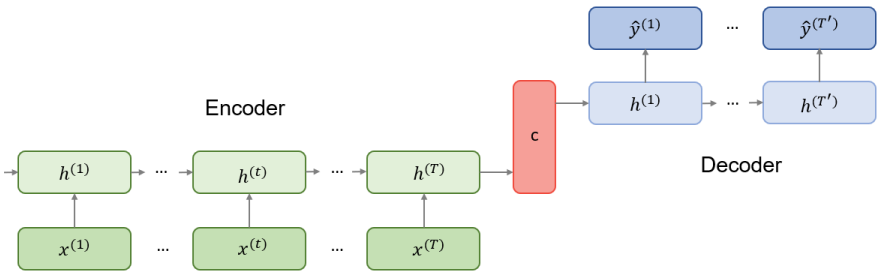



**FIGURE 4.3:** Encoder-Decoder architecture. Source: Own figure.

The first part is the encoder, an RNN which is trained on input sequences in

order to obtain a large summary vector $c$ with a fixed dimension. The second part is a decoder, another RNN which generates predictions. In contrast to a simple RNN described at the beginning of this chapter, outputs $y^{(t)}$ and decoder hidden states $h^{(t)}$ are both conditioned on the summary vector $c$ from the encoder part as well as on $y^{(t-1)}$ and are computed by:

$$h^{(t)} = f(h^{(t-1)}, y^{(t-1)}, c) \tag{4.32}$$

$$P(y^{(t)}|y^{(t-1)}, ...y^{(1)}, c) = \mathcal{Y}(h^{(t-1)}, y^{(t-1)}, c) \tag{4.33}$$

$$\tag{4.34}$$

# 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 Convolutional neural network in the field of NLP.

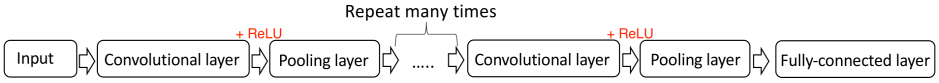



**FIGURE 5.1:** Basic structure of CNN

As illustrated in Figure 5.1, a convolutional neural networks includes successively an input layer, multiple hidden layers, and an output layer, the input layer will be dissimilar according to various application. The hidden layers, which is 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 layer of CNN and their corresponding intuitive example will be provided in detail. Scherer et al. (2010)

### 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)** play an important

role throughout this process. The following Figure 5.2 provide a more intuitive explanation of the convolutional layer.

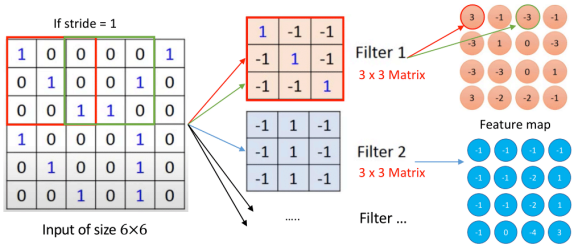



**FIGURE 5.2:** Basic operational structure of the convolutional layer

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 back and forth 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 parameter 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 been calculated through dot product operation shown as 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 generate 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 correspond to different filters are stacked together along the depth dimension.

Another improved convolutional layer was proposed by (Kalchbrenner et al. (2016)) and this kind of convolution is named Dilated convolutionn, 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, although there is no pooling layer, the original input information is still increased as the layers are deeper.

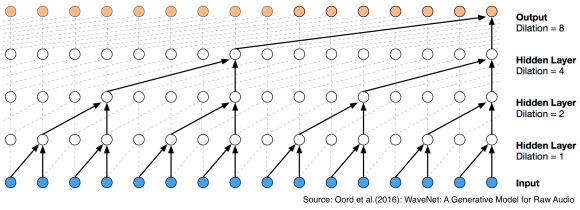



**FIGURE 5.3:** Visualization of dilated causal convolutional layers

### 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 map 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 input, ReLU activation function do 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 activation function. To be more specific, a neuron's weights will stop updating if its gradient is close to zero. It is obviously problematic if such scenario appears too early in the training process.

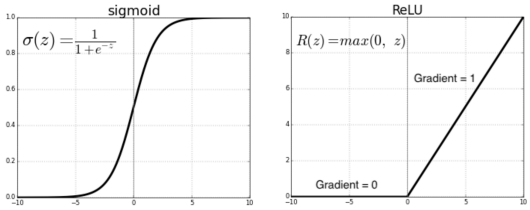



**FIGURE 5.4:** Comparison between saturating and non-saturating activation function

The following Figure 5.5 based on Figure 5.2 indicates a simplified version of the ReLU layer. Each 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 the identical networks structure with the feature map from previous convolutional layer, will be used as an input for subsequent convolutional layer.

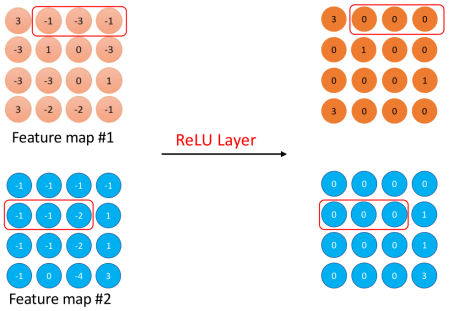



**FIGURE 5.5:** Basic operational structure of the ReLU layer

### 5.1.3 Pooling layer

Pooling layer is a concept that can be intuitively understood. The purpose of pooling layer is to reduce progressively the spatial size of 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.

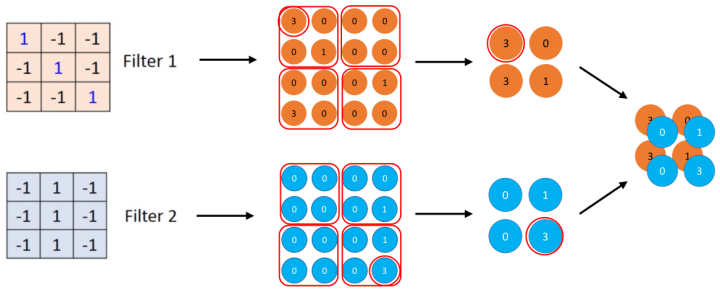



**FIGURE 5.6:** Basic operational structure of the max pooling layer

The above mentioned example 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 each single sub-region is names as depth slice. The Maximum value of each sub-region will be stored in the output of pooling layer. As a result, the input dimensions are further reduced from $4 \times 4$ to $2 \times 2$.

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 amount of parameters in the neural networks and consequently reduces computational load.
- **Controlling overfitting**: Overfitting appears when the model 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 feature from each sub-region are extracted. Therefore, max pooling reduces the probability of overfitting to the 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 specific, 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.

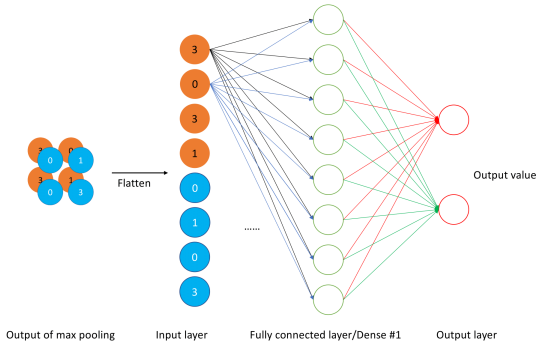



**FIGURE 5.7:** Basic operational structure of the fully connected layer

## 5.2 CNN for sentence classification

The explanation of CNN basic architecture provided in the first sub- is based on a general example. Many researchers constructed their own specific CNN models based on this basic architecture in recent years and achieved outsanding results in the field of NLP. Therefore, this section explores five superior CNN architecture with some technical detail and their performanc comparison will be provided in later sub-chapter 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, Kim (2014) utilizes a simple CNN architecture of (Collobert et al. (2011)) with single convolutional layer and the general architecture include the following sub-structure:

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

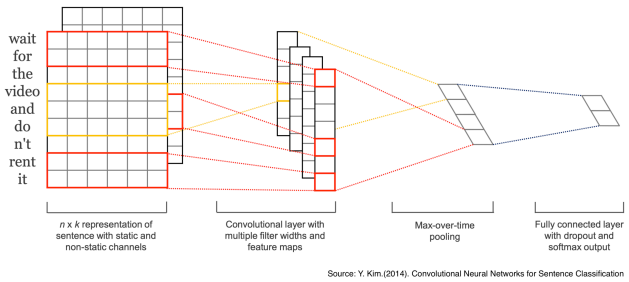



**FIGURE 5.8:** Model architecture of CNN for sentence classification

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 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 follow.

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

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

$$\hat{w} = pw$$

On the basis of 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 kept them static.
- **CNN-non-static**: A model with pre-trained word vectors by using word2vec and these word vectors are fine-tuned for each tasks.
- **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 implement character quantization.

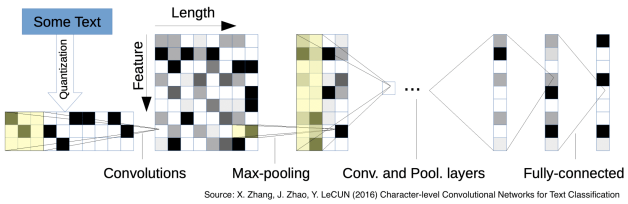



**FIGURE 5.9:** Model architecture of character-level CNN

The Figure 5.9 above 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 conmolutional 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 modelis similar to ReLU $h(x) = max\{0, x\}$. More spesific, the algprithm is stochastic gradient descent (SGD). However, SGD is inflenced by 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 initial step size 0.01 are established to reach the minimum more quickly.

### 5.2.3 Very Deep CNN

### 5.2.4 Deep Pyramid CNN

## 5.3 Datasets and Experimental Evaluation

### 5.3.1 Datasets

### 5.3.2 Experimental Evaluation

## 5.4 Conclusion and Discussion

# 6

# *References*

# 7

# *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

## 7.1 What is Transfer Learning?

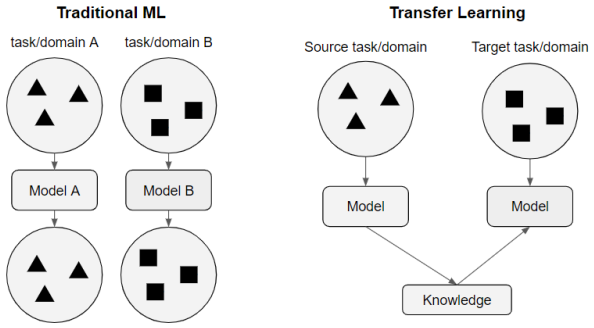



**FIGURE 7.1:** Classic Machine Learning and Transfer Learning

In figure 7.1 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)

## 7.2 (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., 2015).

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"**.

## 7.3 Overview over important NLP models

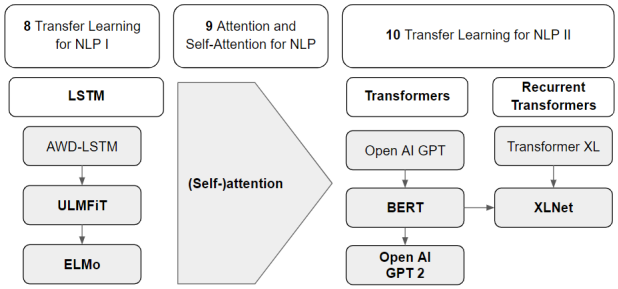



**FIGURE 7.2:** Overview of the most important models for transfer learning

The models in figure 7.2 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.

# 8

# *Transfer Learning for NLP I*

*Authors: Author 1, Author 2*

*Supervisor: Supervisor*

# 9

# *Attention and Self-Attention for NLP*

*Authors: Author 1, Author 2*

*Supervisor: Supervisor*

# 10

# *Transfer Learning for NLP II*

*Authors: Author 1, Author 2*

*Supervisor: Supervisor*

# 11

# *Introduction: Resources for NLP*

*Authors: Author 1, Author 2*

*Supervisor: Supervisor*

# 12

# *Resources and Benchmarks for NLP*

*Authors: Author 1, Author 2*

*Supervisor: Supervisor*

# 13

# *Software for NLP: The huggingface transformers module*

*Authors: Author 1, Author 2*

*Supervisor: Supervisor*

# 14

# *Use-Bases for NLP*

*Authors: Author 1, Author 2*

*Supervisor: Supervisor*

# 15

# *Natural Language Generation*

*Author: Haris Jabbar*

*Supervisor: Matthias Aßenmacher*

## 15.1 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. That's a much harder problem than just identifying a horse in a picture or fitting a line through a set of datapoints.

In this chapter, I will tackle this generative process in Natural Language Processing domain. Understandably, it's called Natural Language Generation (NLG).

## 15.2 Definition

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"

## 15.3 Domains of NLG Systems

NLG systems have a wide variety of application areas.

1. Text-to-Text
   - Machine Translation
   - Text Summarization
2. Data-to-Text
   - Image Captioning
   - Business Intelligence
3. Ideas-to-Text
   - Poetry/Song Generation
   - Mimic an artist
   - Fake News
4. Dialog Systems (Chatbots)
   - Goal Oriented
   - Open ended conversations

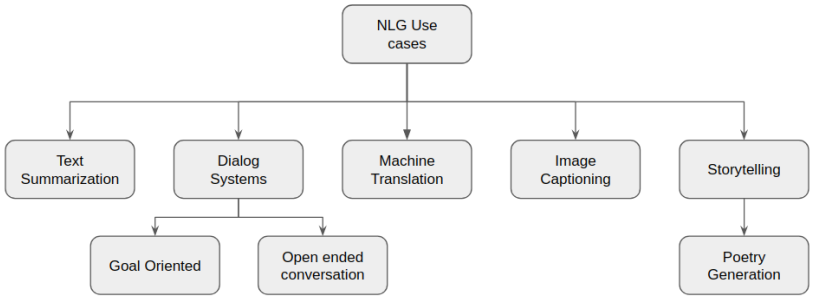



**FIGURE 15.1:** NLG Use Cases

## 15.4 Commercial Activity

1. https://www.narrativescience.com
2. https://automatedinsights.com
3. http://www.arria.com

# 16

# *The Architectures*

There are many architectures that are peculiar to an NLG system. While some are used in other domains as well, in the following sections I will explain them with a focus on language generation.

## 16.1 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 following figures:

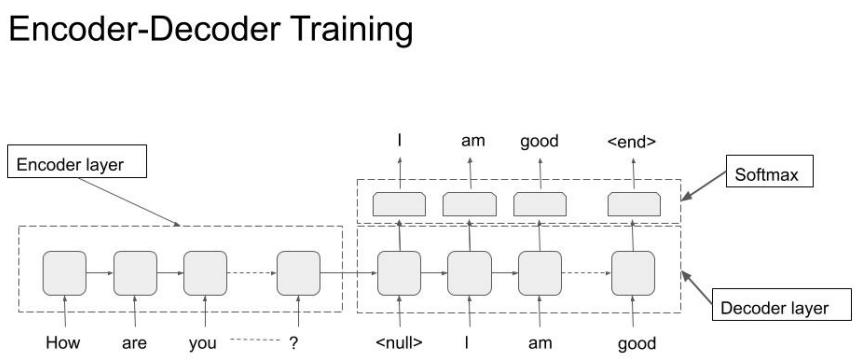



**FIGURE 16.1:** Encoder-Decoder (Training)

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

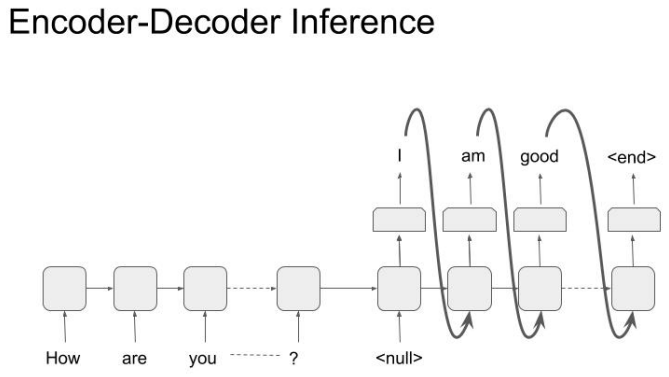



**FIGURE 16.2:** Encoder-Decoder (Inference)

### 16.1.1 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. It can be a convolutional neural network for image based conditioning, or RNN/LSTM/Transformer architecture for text or audio based conditioning. But for the purpose of illustration we are using an RNN/LSTM and text as input condition.

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 better output. This was the motivation to move from single feature vector (\cite) to multiple vectors and subsequently to a transformer based architecture.

### 16.1.2 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 (\cite) one after the other until a terminating criteria is met. This termination is usually a termination token (<end> in the figures).

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 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 than 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.

## 16.2 Attention Architecture

The attention architecture is introduced in detail in section (\cite). 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.

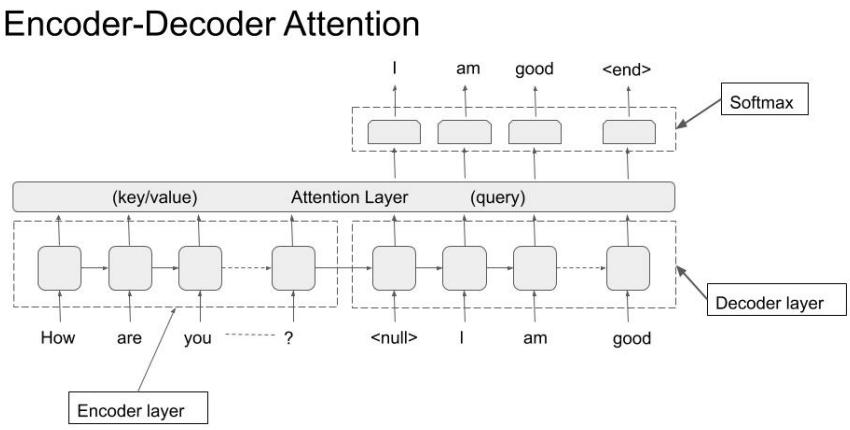



**FIGURE 16.3:** Encoder-Decoder (Inference)

In other words, before generating each token, the decoder attends to all tokens in the encoder, as shown. 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.

## 16.3 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.

### 16.3.1 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 than goes as follows:

1. Select k-tokens with highest probability at step 0.
2. Use these k-tokens to generate k softmax vectors at step 1.
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.

### 16.3.2 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.

### 16.3.3 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.

## 16.4 Memory Networks

Memory Networks is another architecture that is 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. So the network employs an external storage (the memory) that it can use during language generation.

## 16.5 Language Models

Language models are probably the most important ingredient of generating text. As the name implies, they model how the langauge probability distribution. More concretely, they model the conditional probability distribution P(w_t/w_{t-1}). This can then be used to generate text.

# 17

# *Question-Answer Systems*

The question-answer systems attempt to extract or generate answers from a given question and either a fixed or open ended context. The context here is the corpus from which the answer needs to be generated. For example in SQUAD dataset (\cite) the context is a given paragraph from wikipedia and the question is asked from that paragraph. In open ended contexts, the whole wikipedia (or other corpus) can be the contexts.

## 17.1 Datasets

- msmarco
- google natural questions
- multiple choice
  - swag/trivia qa
- conversational qa
  - coqa, wizard of wikipedia, quac.ai
- many others e.g. visual qa, KB qa etc.

## 17.2 Types

- Question Answer Systems
  - Structured knowledge source The sources can be e.g. Freebase, Wikidata, DBpedia or RDBMS systems
  - Unstructured knowledge soure Free text e.g. Wikipedia
  - FAQs Extract answers for similar questions to the given in a corpus of question-answer pairs.

## 17.3 Architectures

- Context
- Question

Five conceptual levels - Token level features (embeddings) - Context and question encoder - Attention from context to question or vice versa - Modeling layer - Output layer

- Pointer Networks
- Open Domain QA
  - DrQA
  - Distant Supervision

## 17.4 Evaluation Metrics

There are generally two metrics commonly used in QA systems. The exact match (EM) and F1 score.

# 18

# *Dialog Systems*

These are the systems where an agent chats with a human being either with a specific pupose (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.

## 18.1 Types

- Chatbots
  - Open domain
  - Goal Oriented

## 18.2 Architectures

- Information Retrieval
  - Inbuilt in the model weights
  - External Source
    - Memory Networks
    - API calls (?)
- Text Generation
  - Encoder-Decoder

# 19

# *Conclusion*

Natural Language Generation (NLG) has huge potential to be not only an academic domain for research but to affect and improve our daily lives. In this chapter I have talked about only two of its manifestations.

# 20

# *Use-Case II*

*Authors: Author 1, Author 2*

*Supervisor: Supervisor*

# 21

# *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


Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, (3):1137–1155.

Boden, M. (2002). A guide to recurrent neural networks and backpropagation. *the Dallas project*.

Boureau, Y.-L., Ponce, J., and LeCun, Y. (2010). *A Theoretical Analysis of Feature Pooling in Visual Recognition*.

Chen, G. (2016). A gentle tutorial of recurrent neural network with error backpropagation. *arXiv preprint arXiv:1610.02583*.

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.

Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555.

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*.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

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*.

Harris, Z. S. (1954). Distributional structure. *WORD*, 10(2-3):146–162.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Kalchbrenner, N., Espeholt, L., Simonyan, K., van den Oord, A., Graves, A., and Kavukcuoglu, K. (2016). Neural machine translation in linear time. *ArXiv*, abs/1610.10099.

Kalchbrenner, N., Grefenstette, E., and Blunsom, P. (2014). *A Convolutional Neural Network for Modelling Sentences*.

Kim, Y. (2014). *Convolutional Neural Networks for Sentence Classification*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). *ImageNet Classification with Deep Convolutional Neural Networks*.

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*.

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., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Malte, A. and Ratadiya, P. (2019). Evolution of transfer learning in natural language processing.

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., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, pages 3111–3119.

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2013). How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*.

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.

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., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

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*.

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.

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*.

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.

Zhang, X., Zhao, J. J., and LeCun, Y. (2015). *Character-level Convolutional Networks for Text Classification*.