Modern Approaches in Natural Language Processing

Contents

Preface **v**

Foreword **1**

1 Introduction **3**

2 Introduction: Deep Learning for NLP **5**

3 Foundations/Applications of Modern NLP **13**

4 Recurrent neural networks and their applications in NLP **25**

5 Convolutional neural networks and their applications in NLP **33**

6 Introduction: Transfer Learning for NLP **45**

7 Transfer Learning for NLP I **49**

8 Attention and Self-Attention for NLP **65**

9 Transfer Learning for NLP II **69**

10 Introduction: Resources for NLP **85**

11 Resources and Benchmarks for NLP **87**

12 Software for NLP: The huggingface transformers module **95**

13 Use-Bases for NLP **97**

14 Natural Language Generation **99**

15 Use-Case II **109**

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 (**?**). 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

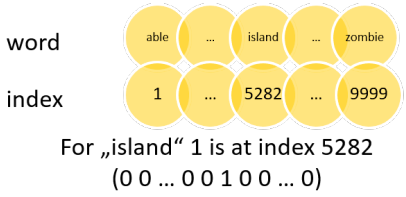
Authors: Viktoria Szabo, Marianna Plesiak, Rui Yang

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 **?**). 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 **?**). 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 **?** and GloVE from **?**. 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 (**?**). 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

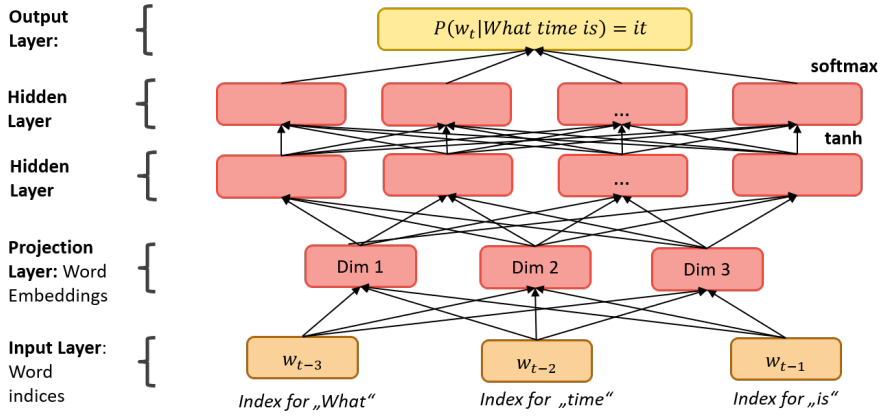


FIGURE 2.2: Feed-forward Neural Network. Source: Own figure based on Bengio et al. 2013.

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

Basically, a simple RNN is a for-loop that reuses the values which are calculated in the previous iteration (**?**). 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 (**?**).

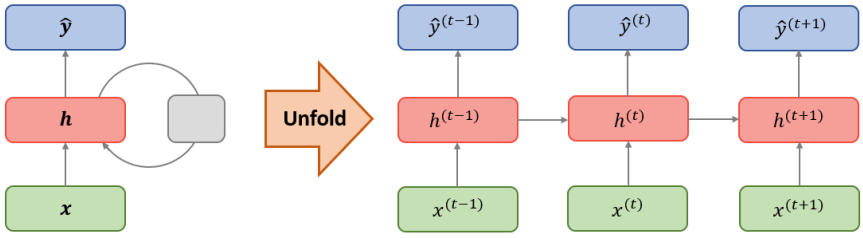


FIGURE 2.3: Right: Circuit diagram (left) and unfolded computational graph (right) of a simple RNN. Source: Own figure.

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 (**?**).

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 (**?**). 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 **?**) and Gated Recurrent Units (GRUs, presented by **?**).

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 (**?**). Bidirectional RNNs feed sequences in their original as well as reverse order because the prediction may depend on the future context, too (**?**). 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 **?**). 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 (**?**). It can be seen that the application of CNN or RNN is not restricted to a specific field.

As the Word2vec algorithm from **?** and the GloVe algorithm from **?** 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.

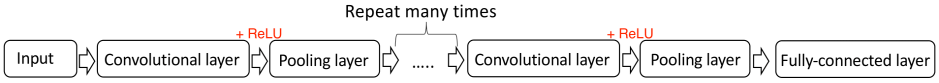


FIGURE 2.4: Basic structure of CNN. Source: Own figure.

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.

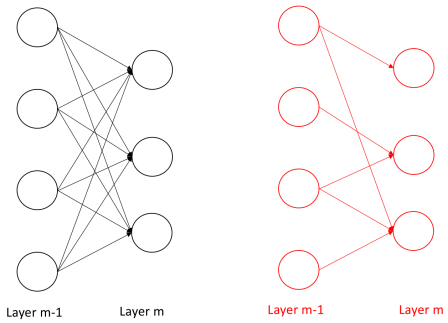


FIGURE 2.5: Comparison between the fully-connected and partial connected architecture. Source: Own figure.

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 (**?**). 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 **?**) and based on multiple data sets to detect the Very Deep Convolutional Networks (VD-CNN) for text classification (see **?**).

3

Foundations/Applications of Modern NLP

Authors: Viktoria Szabo

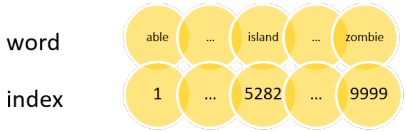
Supervisor: Christian Heumann

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.

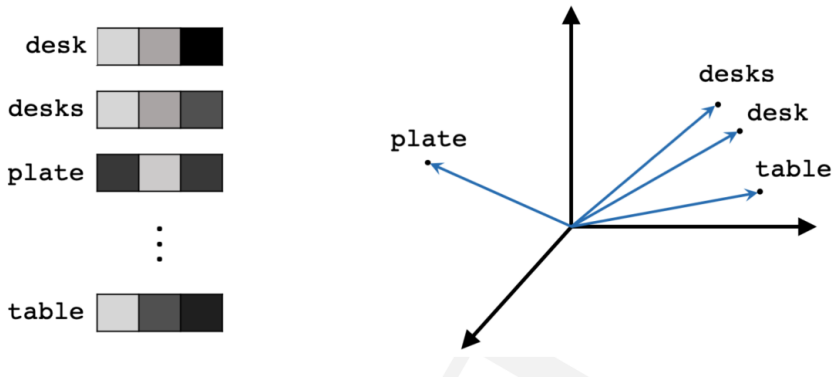


FIGURE 3.2: Example for word embeddings with three dimensions. Source: ...

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.

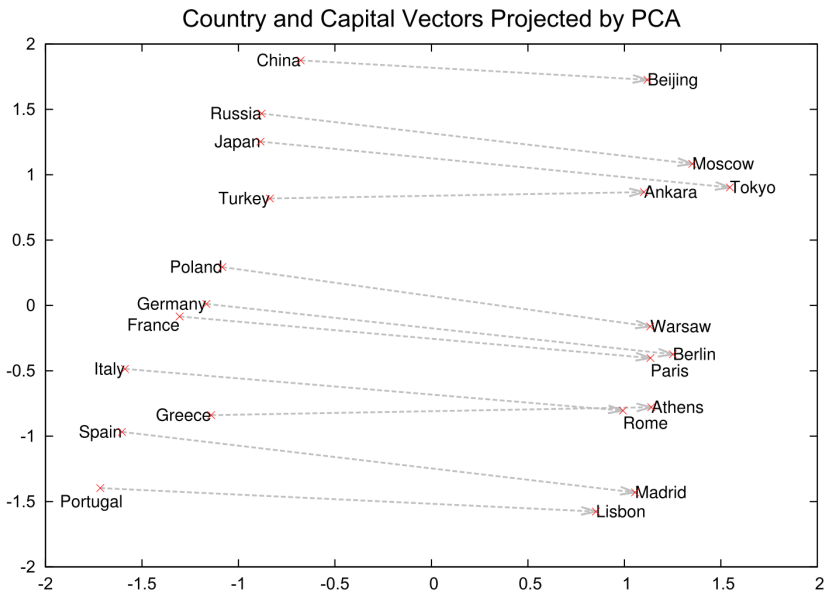


FIGURE 3.3: Two-dimensional PCA projection of 1000-dimensional word vectors of countries and their capital cities. Source: Mikolov et al. 2013 (3)

3.2 Methods to Obtain Word Embeddings

The basic idea behind learning word embeddings is the so called "distributional hypothesis" (Harris, 1954). It states that words that occur in the same contexts tend to have similar meanings. For instance, 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:

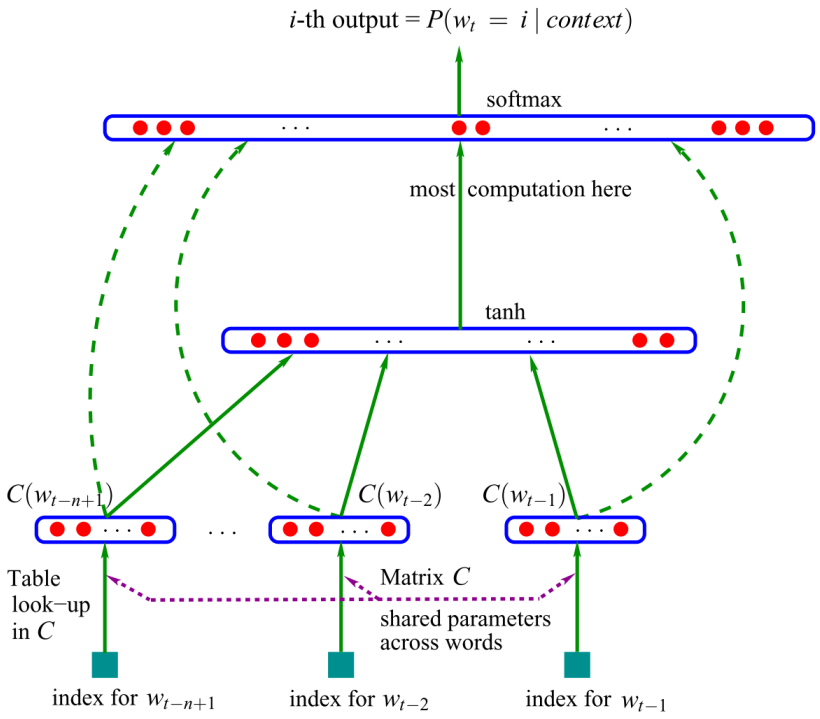


FIGURE 3.4: Architecture for NNLM proposed by Bengion et al. in 2003. Source: Bengio et al. 2003

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 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 hidden 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

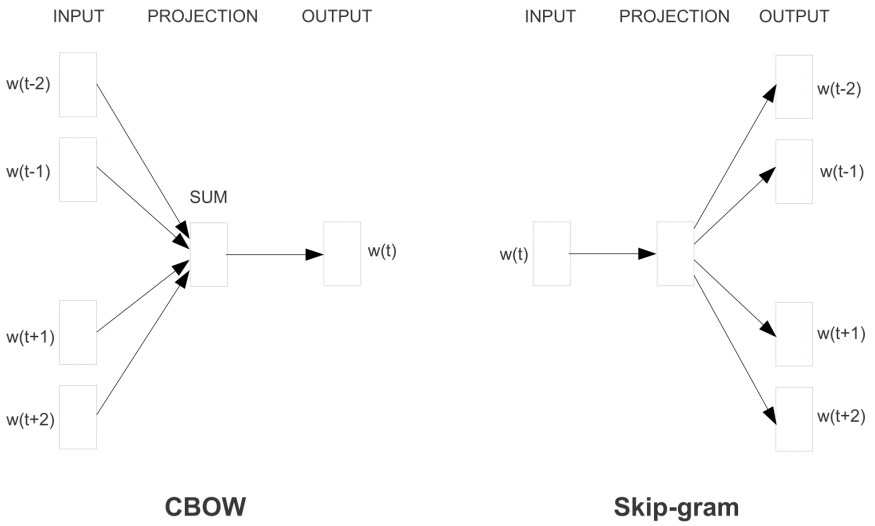


FIGURE 3.5: Learning word embeddings with the model architecture of CBOW and Skip-Gram. Source: Mikolov et al. 2013 (3)

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 context of word i. In figure 3.6 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.

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

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

FIGURE 3.6: Co-occurrence probabilities for target words ice and steam with selected context words from a 6 billion token corpus. Source: Pennington et al. 2014

, 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 examples 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. (**?**)

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.

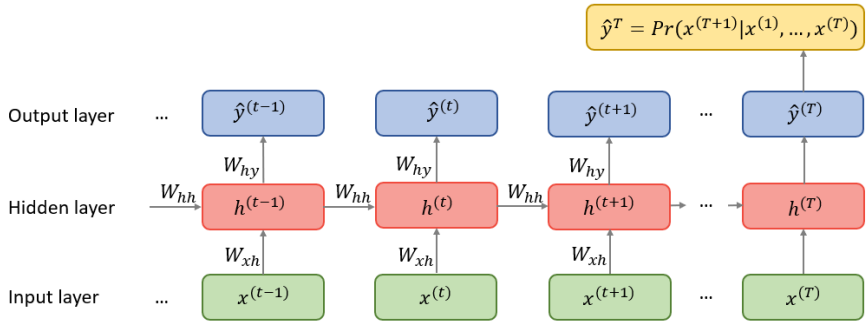


FIGURE 4.1: Unfolded computatinal graph of a RNN. Source: Own figure.

Given the hidden sequences, output vectors $\hat{y}^{(t)}$ are used to build the predictive distribution $Pr(x^{(t+1)}|y^{(t)})$ for the next input (**?**). 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. (**?**)

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. (**?**)

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 (**?** and **?**) 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 (**?**)
- a, b: input and output bias vectors.

(**?**)

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 (**?**). 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 $Pr(x)$:

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

$$\tag{4.9}$$

(**?**)

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. (**?**)

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:

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

(**?**)

4.1.3 Vanishing and Exploding Gradients

4.2 Gated RNNs

4.2.1 LSTM

Long Short Term Memory networks were introduced by **?** 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:

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

(**?**)

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.(**?**, **?**)

4.2.2 GRU

Invented by **?**. 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

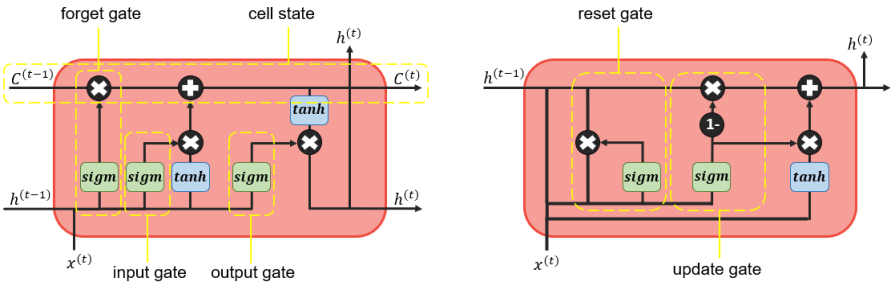


FIGURE 4.2: Structure of a hidden unit. LSTM on the right and GRU on the left. Source: Own figure inspired by http://colah.github.io/posts/2015-08-Understanding-LSTMs/

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 **?**, 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. (**?**)

? 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 (**?**, **?**), the seq2seq approach achieved state-of-the-art results also in speech recognition (**?**) and video captioning (**?**). According to **?**, the seq2seq model consists of two parts as illustrated below:

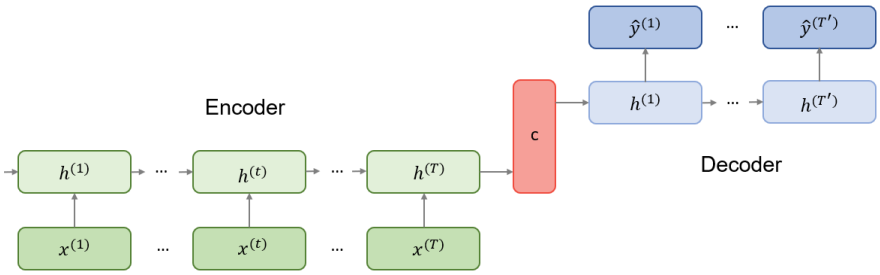


FIGURE 4.3: Encoder-Decoder architecture. Source: Own figure.

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.

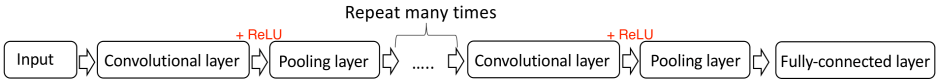


FIGURE 5.1: Basic structure of CNN

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. **?**

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.

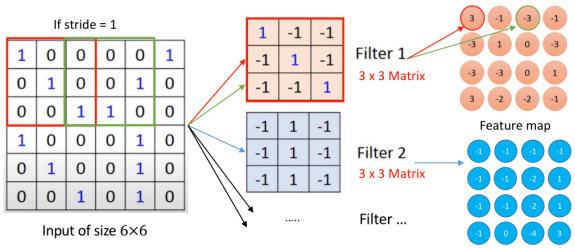


FIGURE 5.2: Basic operational structure of the convolutional layer

The input of the Neural Networks might be assumed as a 6×6 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 3×3 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 3×3 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 6×6 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 3×3;

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

where β 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 4×4 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 (**?**) 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.

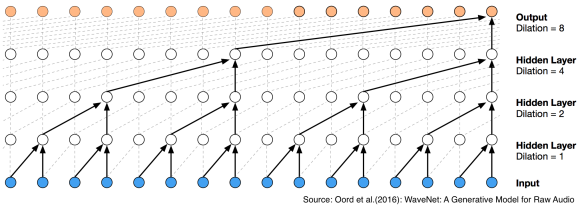


FIGURE 5.3: Visualization of dilated causal convolutional layers

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(**?**) 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(**?**) 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.

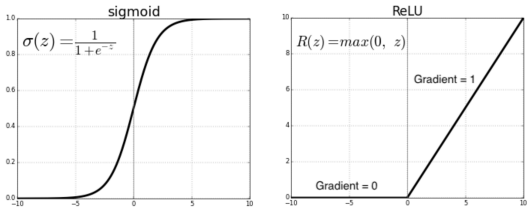


FIGURE 5.4: Comparison between saturating and non-saturating activation function

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.

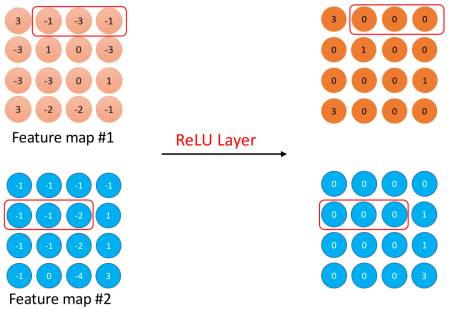


FIGURE 5.5: Basic operational structure of the ReLU layer

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(**?**), 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 (**?**). Simultaneously, this process helps to control overfitting to a certain extent. The following Figure 5.6 illustrates an example that constructs a basic operational structure of the max pooling.

The above mentioned example shows that two feature maps are generated according to two different filters. In this case, these feature maps of size 4×4 are separated into four non-overlapping sub-regions of size 2×2, 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 4×4 to 2×2.

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

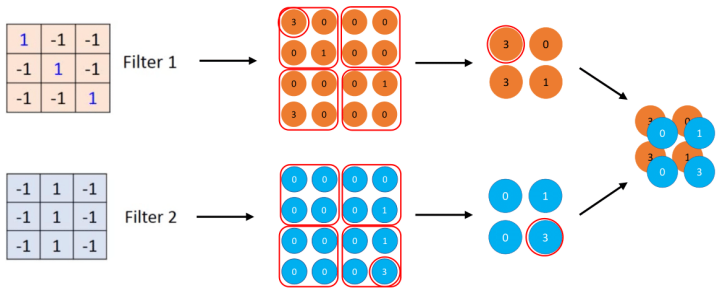


FIGURE 5.6: Basic operational structure of the max pooling layer

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 (**?**) 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.

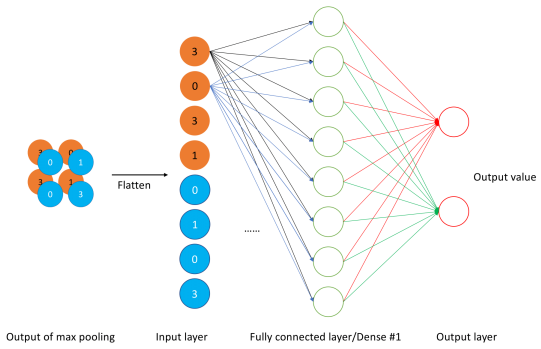


FIGURE 5.7: Basic operational structure of the fully connected layer

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 (**?**), 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, **?** utilizes a simple CNN architecture of (**?**) 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

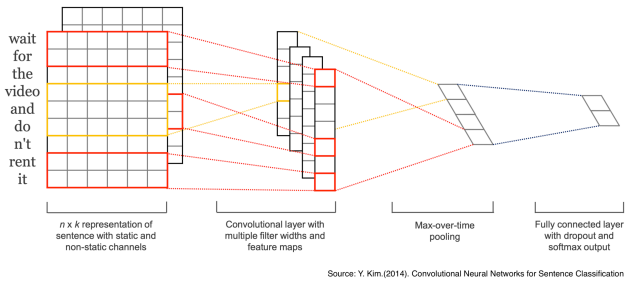


FIGURE 5.8: Model architecture of CNN for sentence classification

$x_i \in \mathbb{R}^k$ to the k-dimensional word vector. Therefore, a sentence can be represented as:

$$X_{1:n} = X_1 \oplus X_2 \oplus ... \oplus X_n$$

Where $\oplus$ is the concatenation operator.

2. **Convolutional layer**: Let a filter denote as $w \in \mathbb{R}^{hk}$, which is used to a window of h words. A feature map $c = [c_1, c_2, ..., c_{n-h+1}]$ can be generated by:
$$c_i = f(w \times x_{i:i+h-1} + b)$$
where $b \in \mathbb{R}$ is a bias term.
3. **Max-over-time pooling**: Pooling operation has been applied for 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 $\widehat{w}$ are trained by using

$$\widehat{w} = pw$$

On the basis of the model architecture described above, four derivative CNN models are introduced by (**?**) 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 (**?**), the two major differences of which compared with the previous model from (**?**) 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.

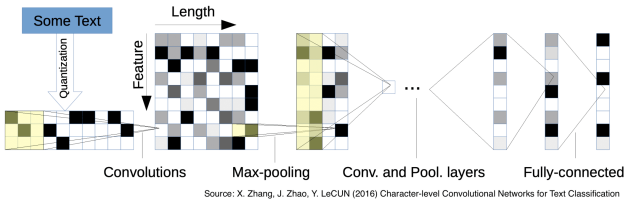


FIGURE 5.9: Model architecture of character-level CNN

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 $f_{i,j} \in [1, k] \rightarrow \mathbb{R}$ $((i = 1, ..., m \text{ and } j = 1, ..., n))$ 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 (**?**), 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 **?**, 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

Introduction: Transfer Learning for NLP

Authors: Carolin Becker, Joshua Wagner, Bailan He

Supervisor: Matthias Aßenmacher

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

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

6.1 What is Transfer Learning?

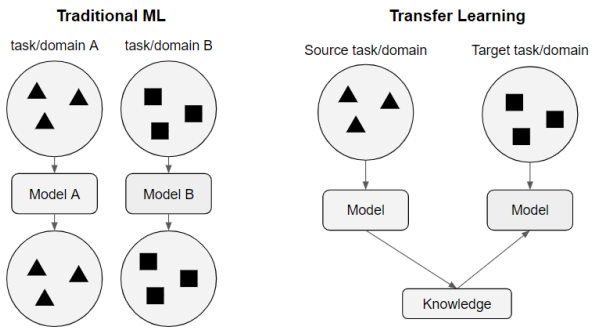


FIGURE 6.1: Classic Machine Learning and Transfer Learning

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

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

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

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

6.2 (Self-)attention

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

? 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" **?** lead to the eventual development of **B**idirectional **E**ncoder **R**epresentations from **T**ransformers by **?**. **BERT** and its successors are, at the time of writing, the state-of-the-art models used for transfer learning in NLP. The concepts attention and

self-attention will be further discussed in the **"Chapter 9: Attention and Self-Attention for NLP"**.

6.3 Overview over important NLP models

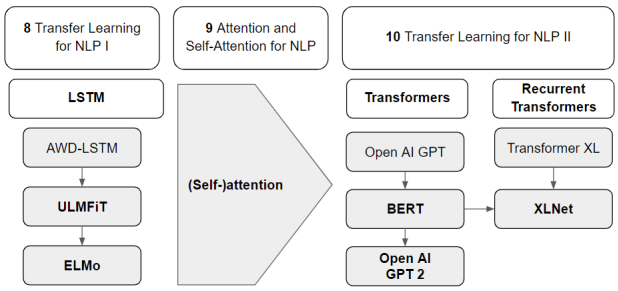


FIGURE 6.2: Overview of the most important models for transfer learning

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

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

- **ELMo** (Embeddings from Language Models) first published in **?** 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 **?**) 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 ?** and **Roberta ?** also reach great success.

- **GPT2** (Generative Pre-Training-2, **?**) 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(**?**). It borrows ideas from autoregressive language modeling (e.g., Transformer-XL **?**) and autoencoding (e.g., BERT) while avoiding their limitations. By using a permutation operation during training, bidirectional contexts can be captured and make it a generalized order-aware autoregressive language model. Empirically, XLNet outperforms BERT on 20 tasks and achieves state-of-the-art results on 18 tasks.

7

Transfer Learning for NLP I

Author: Carolin Becker

Supervisor: Matthias Aßenmacher

The field of natural language processing (NLP) has seen rapid advancements in recent years. Primarily due to the growing usage of transfer learning where the model does not have to be trained from scratch for every task and domain.

The first "wave" of transfer learning where the trained models **ELMo** (Embeddings from Language Models), **ULMFiT** (Universal Language Model Fine-tuning for Text Classification), and **GPT** (Generative Pre-Training) were published.

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

7.1 Outline

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

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

- The Allen Institute introduced with **ELMo** a state-of-the-art model that can distinguish between the varying meaning of the same word in different contexts (**contextual word embedding**).
- Howard and Ruder introduced **ULMFiT**, an adaption of **fine-tuning** in the context of NLP.

- Open AI published **GPT**, which is the first **transformer model**, which is an urge for the upcoming chapters. This model architecture is mainly used in the most recent developments.

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

7.2 Sequential inductive transfer learning

To get a better understanding of the models introduced in this chapter, the taxonomy is introduced. Especially, sequential inductive transfer learning will be explained further.

7.2.1 Types of transfer learning

? divided **transfer learning** in different types (figure 7.1).

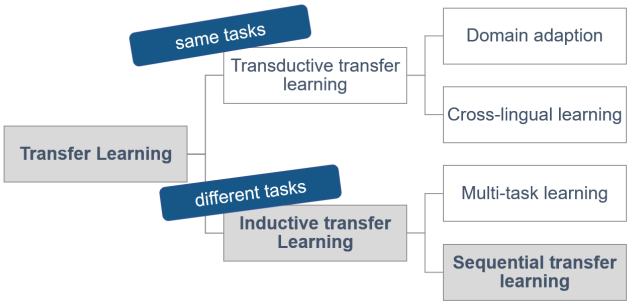


FIGURE 7.1: Four types of transfer learning (source: based on (**?**)).

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

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

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

7.2.2 Sequential inductive transfer learning

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

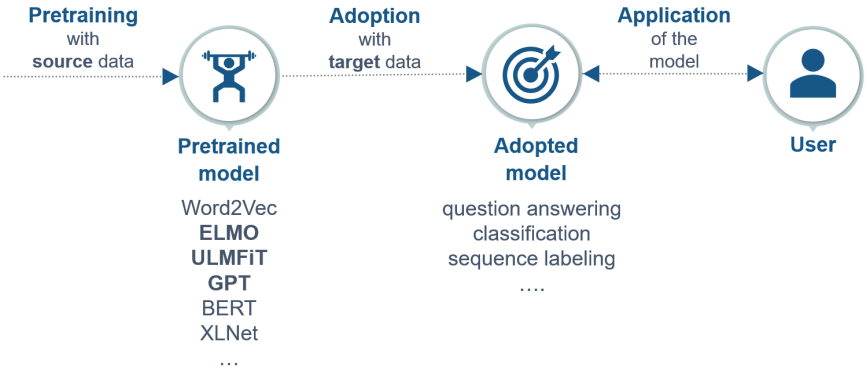


FIGURE 7.2: Steps in sequential inductive transfer learning (source: based on **?**).

As you see in figure 7.2, first, the model is pretrained with the source task and second the model is adapted to the target task. This adopted model can be used by the user for the designated task. (**?**)

In the first step, all models are **pretrained** on an extensive source data set, which is, in the best case, very close to the target task (**?**). The pre-trained language models in this chapter are uni-directional models that predict the next word during pretraining (figure 7.3).

In a second step, follows the **adoption** on the target task. Here, the main distinction is, if the pre-trained model weights are kept (**embedding** or **feature extraction**) or the pre-trained weights are adjusted to the target task (**fine-tuning**). (**?**)

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

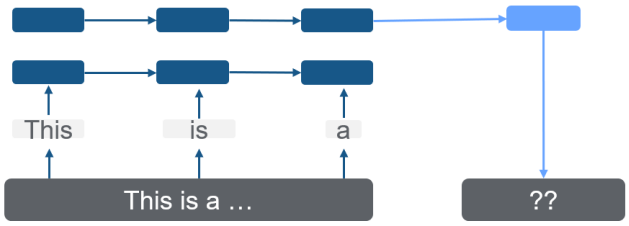


FIGURE 7.3: Unidirectional language model

On the other side, **fine-tuning** adjusts the weights of the pretrained model on a specific task, which makes it much more flexible and needs no specific adjustment. The disadvantage of this method is that the general knowledge and relationship between words can get lost in the adjustment phase. The term for that is the "catastrophic forgetting" (McCloskey & Cohen, 1989; French, 1999). Techniques for preventing this are freezing, learning rates, and regularization which will be mainly explained in the ULMFiT chapter.

7.3 Models

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

7.3.1 ELMo - The "new age" of embeddings

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

For instance, the meaning of the word "mouse" in the context of computers is a device with which you can control the cursor of your PC. Instead, in the context of nature, it often means the small animal. Until ELMo, this could not be captured by an NLP model. For this purpose, ELMo can model the

semantical and the synthetical characteristics of a word (like word2vec) but also the varying meanings in different contexts.

In the context of sequential transfer learning, ELMo can be divided in the steps of pretraining and adoption:

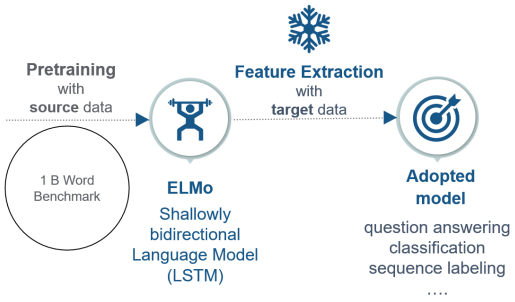


FIGURE 7.4: Inductive sequential transfer learning (ELMo)

ELMo is pretrained on the 1 Billion Word Benchmark, which is a standard dataset for language modeling from Google. The model architecture of Elmo is a shallowly bidirectional Language model, which will be explained in the further sections. Afterwards, the pretrained model is frozen. That means that all learning rates are set to 0, and an additional layer is added. In the adoption phase, a linear combination of the internal states is learned. This representation is different for every task.

7.3.1.1 Pretraining: Bidirectional language model (biLM)

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

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

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

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

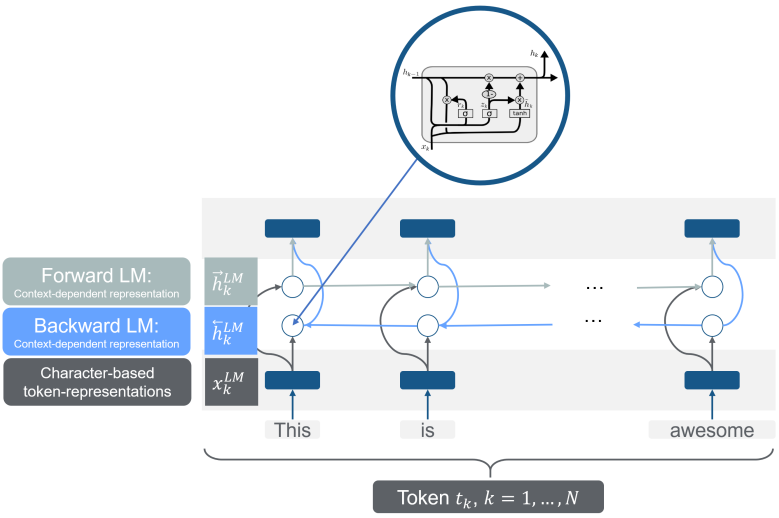


FIGURE 7.5: Bidirectional LSTM as pretrained model (source: based on **?** and **?**.

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

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

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

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

? have chosen a bidirectional LSTM with two layers which is shown in the figure 7.6. As with every layer two additional context-dependent representations are added, A two-layer bidirectional model has five representations per token $(2L + 1)$.

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

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

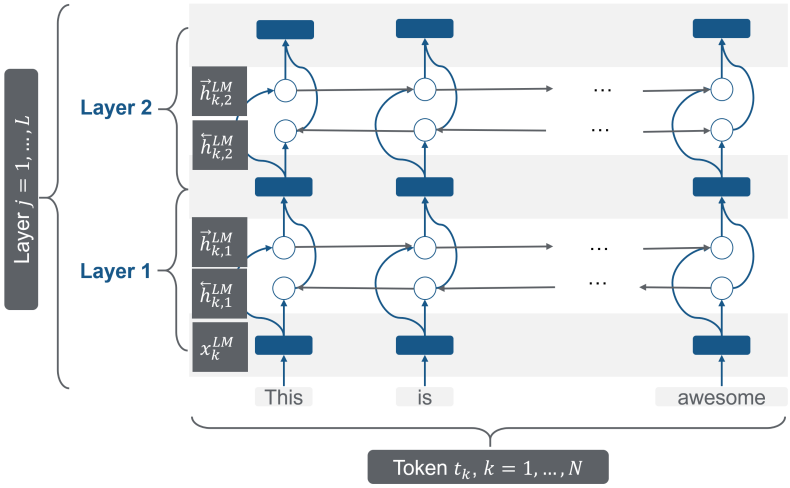


FIGURE 7.6: Bidirectional LSTM as pretrained model (source: based on **?**.

where Θ_s are the parameters for the token representations and Θ_x are the parameters of the Softmax-layer.

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

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

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

7.3.1.2 Adoption: Extracting ELMo representation

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

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

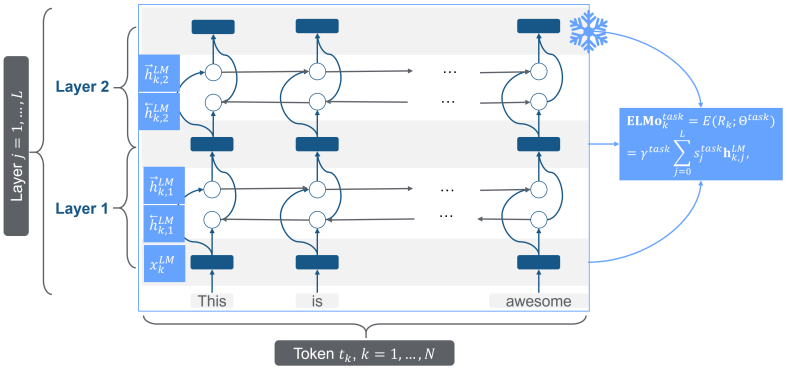


FIGURE 7.7: Bidirectional LSTM as pretrained model (source: based on **?** and **?**).

The ELMo specific task is formulated by

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

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

7.3.1.3 ELMo: Summary and further links

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

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

One interesting fact is that these representations save semantic - context-dependent features - information more in the higher layer and syntactic information more in the lower layers. That is also why the weighting in every task is different.

ELMo is suitable for every NLP task, and by adding ELMo due to **?**'s paper, a relative error reduction of 6-20% can be achieved.

Intersting Links

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

7.3.2 ULMFiT - cutting-edge model using LSTMs

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

In the context of sequential transfer learning, ULMFiT can be divided in the steps of pretraining and adoption:

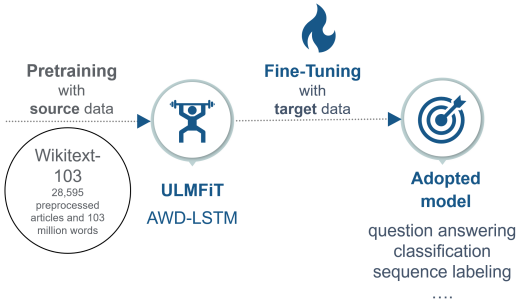


FIGURE 7.8: Inductive sequential transfer learning (ULMFiT)

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

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

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

7.3.2.1 Pretraining: AWD-LSTM

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

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

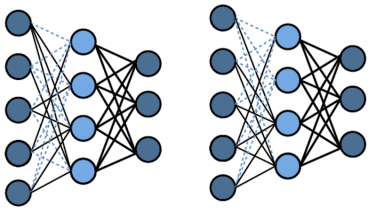


FIGURE 7.9: Dropconnect Algorithm

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

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

In addition to the enhancements above the AWD-LSTM, the authors of the paper propose several other regularization and data efficiency methods: Variable Length Backpropaation Sequences (BPTT), Variational Dropout, Embedding Dropout, Reduction in Embedding Size, Activation Regularization, Temporal Activation Regularization. For further information, the paper **?** is a great way to start.

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

7.3.2.2 Adoption I: LM fine-tuning

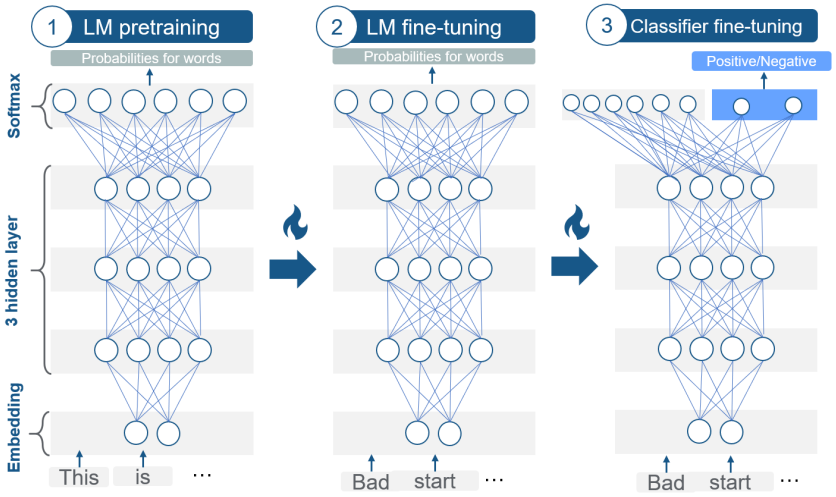
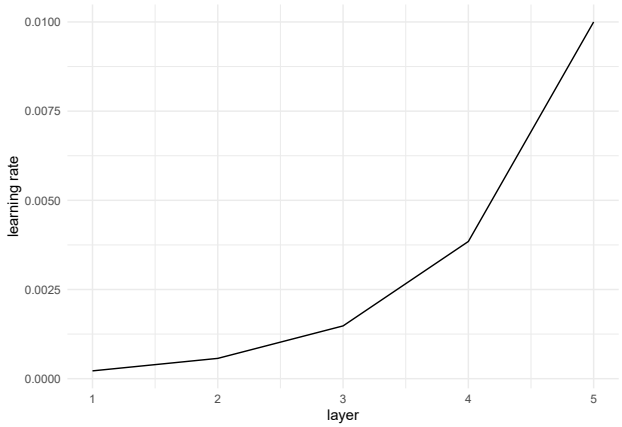


FIGURE 7.10: Three steps of ULMFiT (source: based on **?**).

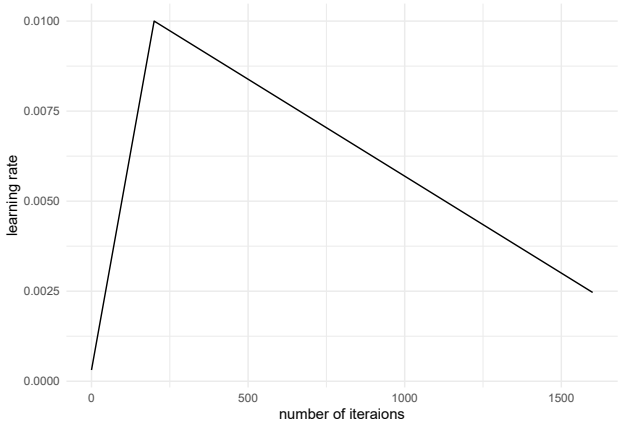
The model is fine-tuned on the tasks' dataset. For this purposed **?** proposed two training techniques to stabilize the fine-tuning process:

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



On the other side **Slanted triangular learning rates** (STLR) is particular

learning rate scheduling that first linearly increases the learning rate, and then gradually declines after a cut. That leads to an abrupt increase and a more extensive decay:



The learning rates η_t are calculated by the number of iterations T:

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

where cut_frac is the increasing learning rate factor, cut the iteration where the decreasing is started, p the fraction of the number of iterations that are increased or decreased, $ratio$ is the ratio the difference between the lowest and highest learning rate

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

7.3.2.3 Adoption II: Classifier fine-tuning

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

- **Concat pooling** As in text classification single words are very important,

concat pooling save the mean and the max representations. By that, theses signal words do not get lost. For example, good or bad are perfect indicators for sentiment. If the fine-tuning loses them, they are still saved in the GPU with concat pooling.

- **Gradual unfreezing** A common problem of retraining the model is losing information about the general data (in this case the Wikipedia dataset), which is called **"catastrophic forgetting."** Hence, with gradual unfreezing the model will be trained step by step, starting from the last layer. So first, all layers are "frozen" except the last layer. In every step, one additional layer is "unfrozen." By that the the specific knowledge which is in the higher layers is adapted more to the target task than the first layer which contains the most general knowledge.

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

7.3.2.4 ULMFiT: Summary and further links

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

Intersting Links

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

7.3.3 GPT - First step towards transformers

? from Open AI published **Generative Pre-Training** (GPT).

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

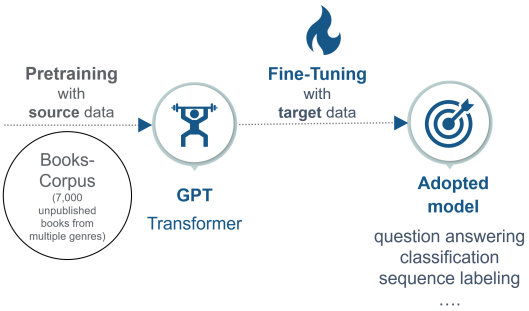


FIGURE 7.11: Inductive sequential transfer learning (GPT)

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

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

- ELMo is based on word embeddings, whereas GPT is based on fine-tuning like ULMFiT.
- GPT uses a different model architecture. Instead of the multi-layer LSTM, GPT is a multi-layer transformer decoder. A model architecture will be explained in the upcoming chapters, as it is a significant step towards the state-of-the-art NLP models.
- In contrast to ELMo, that works character-wise, GPT uses tokens (subwords) from the words.
- GPT is trained on a very large number of data (especially the subsequent models like GPT-2 or GPT-3)

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

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

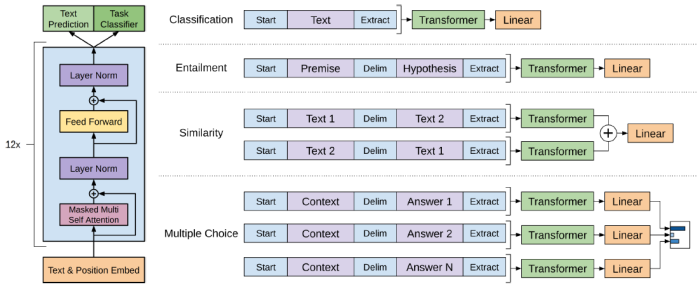


FIGURE 7.12: GPT (source: based on **?**).

7.3.3.1 GPT: Summary

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

7.4 Summary

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

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

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

- All models are **not genuinely bi-directional**, as ULMFiT and GPT are

uni-directional, and ELMo is a concatenation of a right-to-left and left-to-right LSTM. Bidirectional models can even have more precise word representations, as the human language understanding is bidirectional.

- ELMo uses character-based **model input**, and **ULMFit** uses word-based **model input**. **GPT** and following transformer-based models use **tokenized** words (subwords), which is take advantage of both other model inputs.
- ULMFiT and ELMo are **based on LSTMs**, whereas the transformer-based model architecture of GPT has many advantages like parallelization and subsequent performance improvements.
- Starting with 12 layers in GPT the models get deeper and also use more parameters and data for the pre-training.

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

8

Attention and Self-Attention for NLP

Authors: Joshua Wagner

Supervisor: Matthias Aßenmacher

Both attention and self-attention were important for the advances made in NLP. The first part is an overview of attention as it is a building block for self-attention. The second part focuses on self-attention which enabled the commonly used models for transfer learning that are seen today.

8.1 Attention

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

As seen in chapter 3[2], traditional early encoder-decoder architecture passes the last hidden state of the encoder to the decoder. This leads to the problem that information is lost in long input sequences. Especially information found early in the sequence tends to be "forgotten" after the entire sequence is processed. The addition of bi-directional layers remedies this by processing the input in reversed order. The problem still persists for mid sections of very long input sequences. The development of attention enables the decoder to attend to the entire input sequence.

8.1.1 Bahdanau-Attention

In 2015, **?** proposed attention to fix the information problem that the before seen encoder-decoder architecture faced. Early decoders are trained to predict

[1] 01-02-rnns-and-their-applications-in-nlp
[2] 01-02-rnns-and-their-applications-in-nlp

$y_{t'}$ given a context vector c and all earlier predicted words $\{y_t, \dots, y_{t'-1}\}$.

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

where $h_1, \dots, h_T$ are the the hidden states of the encoder for the input sequence $x_1, \dots, x_T$ and q is a non-linear function. **?** for example used $q(\{h_1, \dots, h_T\}) = h_T$ as their non-linear transformation which remains a popular choice for architecture without attention. Attention changes the context vector c that a decoder uses for translation from a fixed length vector c of a sequence of hidden states $h_1, \dots, h_T$ to a sequence of context vectors c_i. The hidden state h_i has a strong focus on the i-th word in the input sequence and its surroundings. Each h_i is computed by a concatenation of the forward $\overrightarrow{h_i}$ and backward $\overleftarrow{h_i}$ hidden states of the bi-directional encoder.

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

The hidden states of the decoder s_t at time-point t is computed as $s_t = f(s_{t-1}, y_{t-1}, c_t)$. The context vector c_t is computed as a weighted sum of the hidden states $h_1, \dots, h_{T_x}$:

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

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

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

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

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

were tanh is used as a non-linear activation function and v_α and W_α are the weight matrices to be learned by the alignment model. The alignment score function is called "concat" in **?** and "additive attention" in **?** because of the concatenation seen above. A nice by-product of attention is a matrix of alignment scores which can be visualised to show the correlation between source and target words.

The attention model proposed by Bahdanau et al. is also called a soft/global attention model as it attends to every input in the sequence.

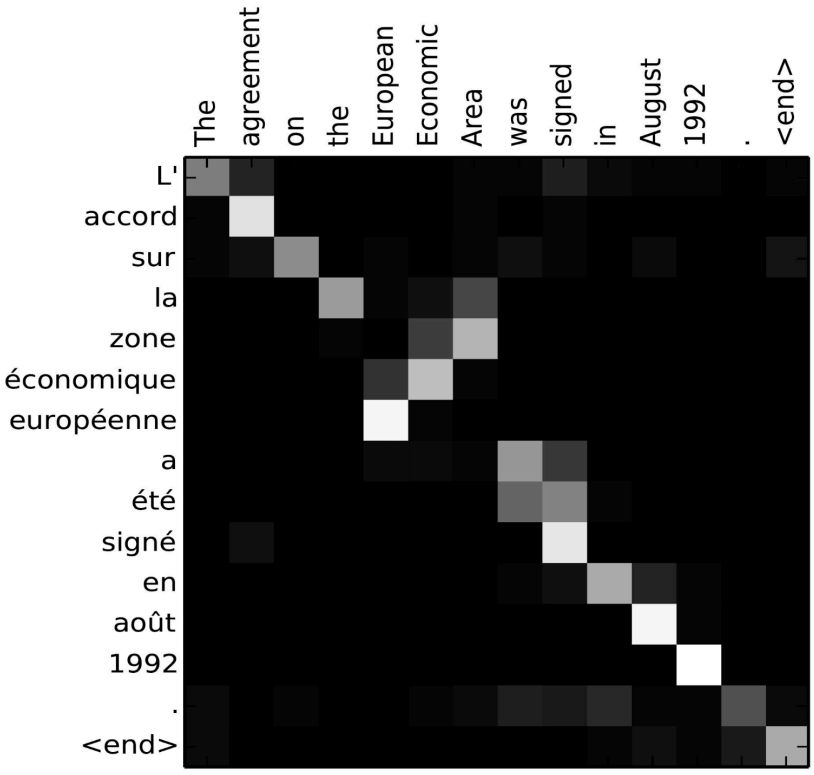


FIGURE 8.1: Alignment Matrix visualised for a French to English translation. Image source: Fig 3 in **?**

8.1.2 Luong-Attention

- intro loung
- different proposed score functions
- differences to bahdanau
- explanation for local/hard attention and differences to global/soft attention

8.1.3 Attention Models

Overview over models that use attention and different attentions used, segway to self-attention

- just a short overview over different attention models/score functions

8.2 Self-Attention

general intro to self-attention

- intro self attention, explanation is done in the transformers part

8.2.1 Transformers

explain transformer architecture, multi-head attention, dot-prod. attention, explain the differences in computational cost to rnns and conv. models

9

Transfer Learning for NLP II

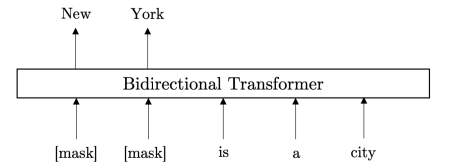
Authors: Bailan He

Supervisor: M. Aßenmacher

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

9.1 Bidirectional Encoder Representations from Transformers (BERT)

9.1.1 Autoencoding



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

FIGURE 9.1: Autoencoding

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

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

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

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

, where $mask_t$ is an indicator to show if a token is masked,**?**.

The training objective is to reconstruct the masked tokens $\bar{X}$ given the sequence $\hat{X}$. AE tries to find the best model P to predict the masked tokens.

9.1.2 Introduction of BERT

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

BERT is the notable example of AE, 15% of tokens are replaced by a special symbol [MASK], and the model is trained to reconstruct the original sequence from the masked tokens. By contrast with previous efforts that looked at a text sequence either from left to right(RNN) or combined left-to-right and right-to-left training (ELMO), the Transformer Encoder utilizes bidirectional contexts simultaneously. Therefore BERT uses the Transformer Encoder as the structure of the pre-train model and addresses the unidirectional constraints by proposing a new pre-training objective: the "masked language model"(MLM).

In conclusion, BERT uses masked language models to enable pre-trained deep bidirectional representations and after fine-tuning based the representation, BERT advances state-of-the-art performance for eleven NLP tasks.

9.1.3 Input Representation of BERT

The input representation is able to unambiguously represent both a single text sentence or a pair of text sentences in one token sequence. For a given token, its input representation is constructed by summing the corresponding token, segment and position embeddings, **?**.

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

- Use WordPiece embeddings **?** with a 30,000 token vocabulary and split word pieces denoted with ##. e.g.,[playing = play and ##ing]
- The first token of every sequence is always the special classification embedding [CLS]. For non-classification tasks, this vector is ignored.

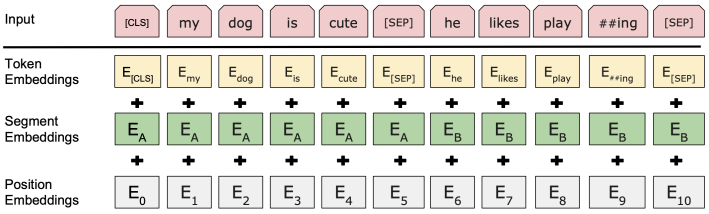


FIGURE 9.2: BERT input representation

- Sentence pairs are packed together into a single sequence and are separated with a special token [SEP]. Second, different learned sentence embedding[e.g.,A and B] will be added to every token of each sentence.For single-sentence inputs we only use the sentence A embeddings.

9.1.4 Masked Language Model

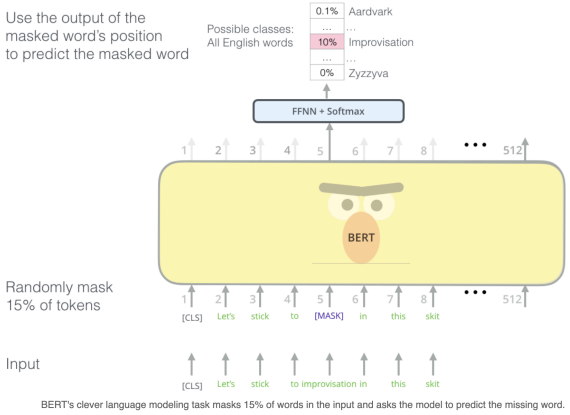


FIGURE 9.3: BERT Masked Language Model Alammar, Jay (2018). The Illustrated BERT, ELMo, and co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-bert/

Figure 9.3 is the visual representation of Masked Language Model. As shown in the above figure, when word embeddings are fed into BERT, 15% of the words in each sequence will be replaced by a special token [MASK]. The task of MLM is to predict the original value of masked tokens. The output of MLM is the word embeddings of corresponding tokens, then feed the word embedding of [MASK] token into a simple softmax classifier and get the final prediction of [MASK] token. And the most important part is the "yellow"

block, it's basically a multi-layer bidirectional Transformer encoder based on implementation described in **?**.

- What is the Masked Language Model?
  - 15% of all WordPiece tokens in each sequence will be randomly masked.
  - Input: token embedding(one sentence, begin with [CLS])
  - Output: BERT token embedding.
  - Using softmax(a simple linear classifier) to predict the masked token. (using token [CLS] to do classification task.) (words match each other may have the same BERT embedding)
- How to mask?
  - 80% of the time: Replace the word with the [MASK] token, e.g., my dog is hairy ⇒ my dog is [MASK]
  - 10% of the time: Replace the word with a random word,e.g., my dog is hairy ⇒ my dog is apple.
  - 10% of the time: Keep the word unchanged, e.g., my dog is hairy ⇒ my dog is hairy.
- Why there are two other methods to replace the word?
  - Why keep 10% of masked tokens unchanged?
    A: In some downstream task like POSTagging, all the tokens are known, if BERT only trained the Masked sequences, then the model only use the information of context, exclude the information of the masked words. It will lose a part of the information, then weakens the performance of the model.
  - Why replace 10% of masked tokens with random words?
    A: Since we keep 10% of masked token unchanged, if we do not add random noise, the model will be "lazy" in our training, the model will plagiarize current tokens, rather than learning.

9.1.5 Next-sentence Tasks

Figure 9.4 is the visual representation of Next-sentence Tasks(NST). Many important downstream tasks such as Question Answering (QA) and Natural Language Inference (NLI) are based on understanding the relationship between two text sentences, which is not directly captured by language modeling. So BERT proposes the NST by using the special token [CLS] as the first token of every sequence.

- What is Next-sentence Tasks?
  - Input: token embedding (two sentence, begin with [CLS], each sentence ends with [SEP])

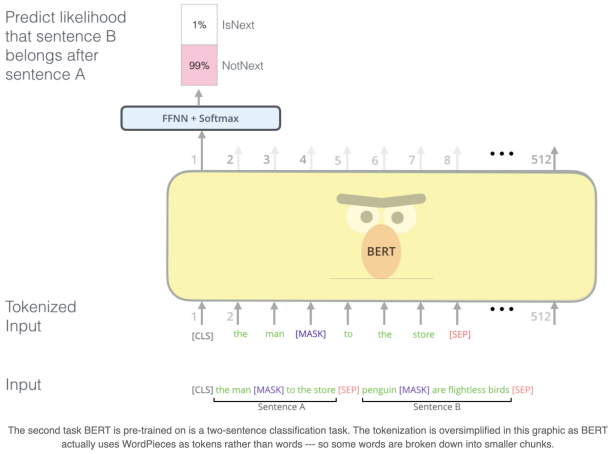


FIGURE 9.4: BERT Next-sentence Tasks Alammar, Jay (2018). The Illustrated BERT, ELMo, and co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-bert/

  - Output: BERT token embedding.
  - Using softmax (a simple linear classifier) to explain the relationship between two sentences.
  - Using [CLS] token to pre-train a binary classification tasks.
  - Sentences can be trivially generated from monolingual corpus.
  - Choose sentences A and B for each example, 50% of B is actual next sentence that follows A, and 50% of B is a random sentence from the corpus.

- For example:
  - Input = [CLS] the man went to [MASK] store [SEP] he bought a gallon [MASK] milk[SEP]
    Label = IsNext.
  - Input = [CLS] the man [MASK] to the store [SEP] penguin [MASK] are flight ##less birds [SEP].
    Label = NotNext

9.1.6 Pre-training Procedure of BERT

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

- BERT-Base: L = 12, H = 768, A = 12, Total parameters = 110M

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

9.1.7 Fine-tuning Procedure of BERT

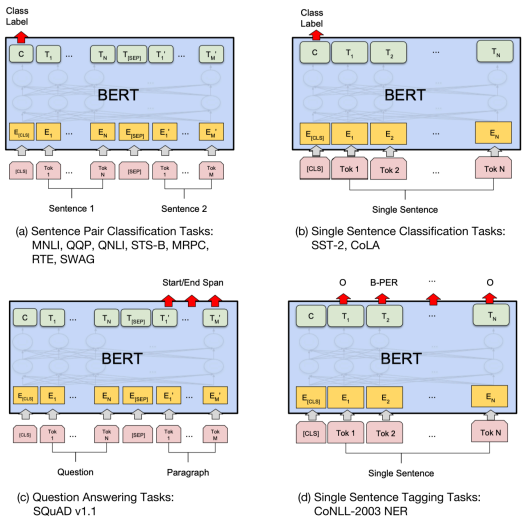


FIGURE 9.5: BERT Task Specific Models

As shown in Figure 9.5, different types of tasks require different modifications to the model, and the modification of the model before fine-tuning is quite simple. For example, for the sequence-level classification problem (such as sentiment analysis, Task (a) in the figure), take the output representation of the first token[CLS] and feed it to a softmax to get the classification result. As for token-level classification (e.g. NER, Task (d) in the figure), take the output of the last layer transformer of all tokens and feed it to the softmax layer for classification.

9.1.8 Feature Extraction

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

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

9.1.9 BERT-like models

The state-of-the-art performance of BERT reveals the deep bidirectional language model can significantly improve the model performance in NLP tasks, and BERT chart a new course that how a real bidirectional model should be. However, BERT has also the following weaknesses: first of all, the input to BERT contains artificial symbols like [MASK] that never occur in downstream tasks, which creates a pre-train-fine-tuning discrepancy problem. Secondly, BERT assumes the predicted tokens are independent of others given the unmasked tokens, which is oversimplified for natural language. Several models are inspired by BERT to solve these problems.

Roberta **?** shows hyperparameter choices have a significant impact on the final results. It improves BERT pre-training in the following aspects to get better performance:

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

ALBERT **?** mainly makes three improvements to BERT, which reduces the overall parameter amount, accelerates the training speed, and improves the model performance under the same training time.

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

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

9.2 Generative Pre-Training(GPT-2)

9.2.1 Auto-regressive Language Model(AR)

Figure 9.6 shows the modelling of Auto-regressive language model, it tries to estimate the probability distribution of a sequence with a auto-regressive pat-

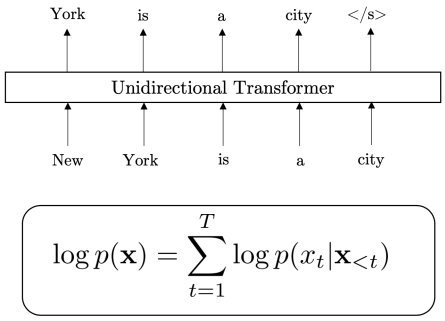


FIGURE 9.6: Autoregressive

tern. Specifically, given a text sequence $X = (x_1, ..., x_T)$, AR language model factorizes the log-likelihood into a forward sum $logp(x) = \sum_{t=1}^{T} p(x_t|x < t)$ or a backward one $logp(x) = \sum_{T}^{t=1} p(x_t|x > t)$ **?**. Since an AR language model is only trained to encode a uni-directional context (either forward or backward), it is not effective at modeling deep bidirectional contexts. On the contrary, downstream language understanding tasks often require bidirectional context information.

9.2.2 Introduction of GPT-2

GPT-2 is proposed by researchers at OpenAI in 2019. It captures the attention of the NLP community for the following characters:
First of all, instead of the fine-tuning model with specific tasks, GPT-2 demonstrates language models can perform down-stream tasks in a zero-shot setting, which means without any parameter or architecture modification. Secondly, in order to perform better under the zero-shot setting, GPT-2 becomes extremely large. On the other hand, training GPT-2 also needs enormous data, so researchers also create a new dataset "WebText", which contains millions of webpages.
With the characters above, GPT-2 achieves state-of-the-art results on 7 out of 8 tested language modeling datasets in a zero-shot setting but still underfits WebText.

Readers can experiment with GPT-2 by using AllenAI GPT-2[3]. You can input a sentence and see the prediction of next words.

[3] https://demo.allennlp.org/next-token-lm?text=AllenNLP%20is

9.2.3 Input Representation of GPT-2

GPT-2 uses a human-curated dataset called "WebText"[4], that contains text scraped from 45 million web-links. All results presented in paper use a preliminary version of WebText which after de-duplication and some heuristic based cleaning contains slightly over 8 million documents for a total of 40GB of text **?**.

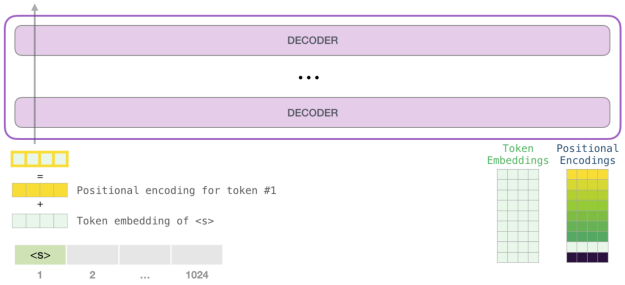


FIGURE 9.7: GPT-2 Input Representation Alammar, Jay (2018). The Illustrated GPT-2 co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-gpt2/

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

9.2.4 The Decoder-Only Block

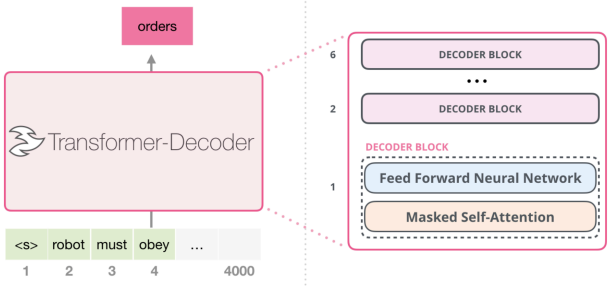


FIGURE 9.8: GPT-2 Model Alammar, Jay (2018). The Illustrated GPT-2 co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-gpt2/

Figure 9.8 shows the model of GPT-2. This model essentially is the Trans-

[4] https://skylion007.github.io/OpenWebTextCorpus/

TABLE 9.1: GPT-2 models size

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

former decoder, except they threw away the second self-attention layer. In each decoder, Layer normalization **?** was moved to the input of each sub-block, similar to a pre-activation residual network **?** and an additional layer normalization was added after the final self-attention block. A modified initialization which accounts for the accumulation on the residual path with model depth is used **?**.

GPT-2 reads from the start token [s] till the last predicted token to predict the next token. For example, in the first round, model uses [s] to predict [robot], in the next round the input is updated as {[s],[robot]} since [robot] has been predicted. This is how Masked self-Attention in Decoder block works.

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

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

For example, a translation training example can be written as the sequence (translate to french, English text, French text). Likewise, a reading comprehension training example can be written as (answer the question, document, question, answer).

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

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

9.2.5 GPT-2 Models

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

9.2.6 Conclusion

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

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

9.3 XLNet

9.3.1 Introduction of XLNet

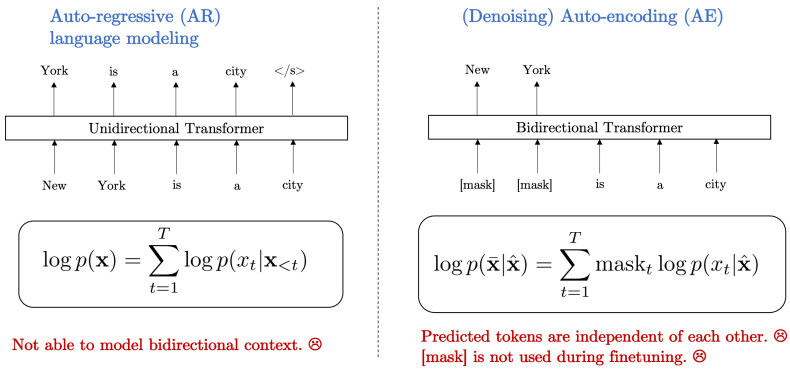


FIGURE 9.9: AR Language Modeling and AE

XLNet is proposed by researchers at Google in 2019. Since the autoregressive language model (e.g.GPT-2) is only trained to encode a unidirectional context and not effective at modeling deep bidirectional contexts and autoencoding (e.g.BERT) suffers from the pretrain-finetune discrepancy, XLNet borrows ideas from the two types of objectives while avoiding their limitations. It is a new objective called Permutation Language Modeling. By using a permutation operation during training time, bidirectional context information can be captured and makes it a generalized order-aware autoregressive language model. Besides, XLNet introduces a two-stream self-attention to solve the problem that standard parameterization will reduce the model to bag-of-words.

Two XLNet are released, i.e. XLNet-Base and XLNet-Large, and include the similar settings of corresponding BERT. Empirically, XLNet outperforms BERT on 20 tasks and achieves state-of-the-art results on 18 tasks.

9.3.2 Permutation Language Modeling(PLM)

Now Figure 9.10 illustrates the permutation language modeling objective.

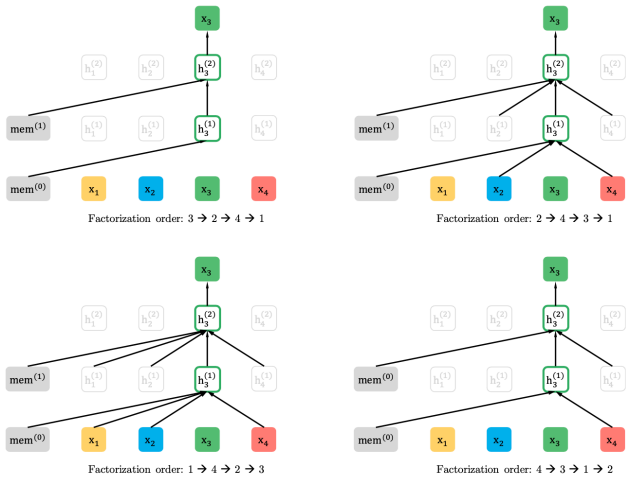


FIGURE 9.10: Illustration of the permutation language modeling objective for predicting x3 given the same input sequence x but with different factorization orders.

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

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

For instance, when we have a factorization order: { New York is a city }. The probability of sequence can be expressed as follows:

$$P(New, York, is, a, city) = P(New) * P(York|New) * P(is|New, York) * P(a|New, York, is) * P(city|New, York, is, a) \tag{9.2}$$

As for another factorization order: {city a is New York}, then the probability of sequence can be expressed differently:

$$P(New, York, is, a, city) = P(city) * P(a|city) * P(is|city, a) * P(New|city, a, is) * P(York|city, a, is, New) \tag{9.3}$$

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

9.3.3 The problem of Standard Parameterization

The Standard Parameterization can be expressed as follows:

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

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

While the permutation language modeling objective has desired properties, naive implementation with standard Transformer parameterization may not work. Specifically, let's consider two different permutations p^1:New York is a city and p^2 : New York a city is. The probability of is in p^1: P(is|New, York) and the probability of a in p^2: P(a|New, York) are identical. The model will be reduced to predicting a bag-of-words, because $h_\theta(x_{z<t})$ does not contain the position of the target.

XLNet resolves the problem by reparameterizing with positions:

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

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

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

9.3.4 Two-Stream Self-Attention

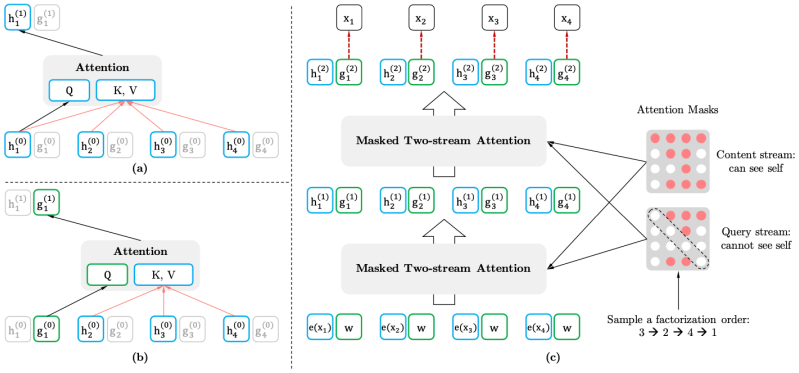


FIGURE 9.11: Two-Stream Self-Attention

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

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

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

- $h_i^{(t)}$ denotes the content representation of the i-th token in the t-th layer of self-attention. It is the same as the standard self-attention. For instance, $h_1^{(1)}$ can see all the $h_i^{(0)}$ since the 1-st token is after token $3, 2, 4$.
- $g_i^{(t)}$ denote the query representation of the i-th token on the t-th layer of self-attention. It does not have access information about the content x_{p_t}, other trivial.
- Computationally, $h_i^{(0)}$ is the word embeddings, and $g_i^{(0)}$ is a trainable parameter initialized with a trainable vector. Only $h_i^{(t)}$ is used during fine-tuning. The last $g_i^{(t)}$ is used for optimizing the LM loss. A self-attention layer $m = 1, ..., M$ are schematically updated with a shared set of parameters as follows:

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

where Q, K, V denote the query, key, and value in an attention operation **?**. More details are included in Appendix A.2 for reference **?**.

9.3.5 Partial Prediction

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

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

For unselected tokens, their query representations need not be computed, which saves speed and memory. XLNet incorporates ideas from Transformer-XL and inherits two important characters of it, Segment-Level Recurrence and Relative Position Encoding, to enable the learning of long-term dependency and resolve the context fragmentation.

9.3.6 XLNet Pre-training Model

After tokenization with SentencePiece **?**, Researchers obtain 2.78B, 1.09B, 4.75B, 4.30B, and 19.97B subword pieces for Wikipedia, BooksCorpus, Giga5, ClueWeb, and Common Crawl respectively, which are 32.89B in total, to pre-train the XLNet.

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

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

9.3.7 Conclusion

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

9.4 Latest NLP models

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

After GPT-2, researchers at Google did the same thing, they proposed a general language model called T5 **?**, which is trained on 750GB corpus - "C4 (Colossal Clean Crawled Corpus)"[5]. If you read the paper, the last page of it is a table of several experience results, which may cost millions of dollars to reproduce it.

On 28th May 2020, the "Arms race" goes into another level, GPT-3 **?** emerged, the new paper takes GPT to the next level by making it even bigger - GPT-3 has 175 billion parameters and is trained on a dataset that has 450 billion of tokens "GPT-3 Dataset"[6]. GPT-3 experiments with the three different settings: zero-shot, one-shot, and Few-shot to show that scaling up language models can greatly improve performance, sometimes even reaching competitiveness with prior SOTA approaches. However, it is conservatively estimated that training GPT-3 will cost one hundred million dollars.

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

[5] https://www.tensorflow.org/datasets/catalog/c4
[6] https://github.com/openai/gpt-3

10

Introduction: Resources for NLP

Authors: Nico Hahn

Supervisor: Daniel Schalk

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

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

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

Lastly, we will be introducing the huggingface transformers module.

11

Resources and Benchmarks for NLP

Authors: Nico Hahn

Supervisor: Daniel Schalk

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

11.1 Metrics

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

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

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

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

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

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

Perplexity: Perplexity is a measurement of how well a probability model

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

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

where b is customarily 2.

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

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

11.2 Benchmark Datasets

11.2.1 SQuAD

The first Version of the **S**tanford **Qu**estion **A**nswering **D**ataset was released in 2016. The dataset was created with the aim of advancing the field of reading comprehension. Reading text and answering questions about it is a demanding task for machines and requires large data sets of high quality. Most of the datasets before the release of the first version of SQuAD were either of high quality or of large size, but not both.

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

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

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

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

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

To evaluate human performance, the second answer to each question is treated as the human prediction.(**??**)

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

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

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

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

11.2.2 CoQA

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

The data set is made up of 127k Q/A pairs, covering seven different domains such as Children's Stories or Reddit. Five of these domains are used for in-domain evaluation, meaning models have already seen questions from these domains, and two are used for out-of-domain evaluation., meaning models have not seen any questions from these domains. To create the Q/A pairs, two people received a text passage, with one person asking the other person questions about the text and the other person answering. Using multiple annotators has a few advantages:

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

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

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

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

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

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

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

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

11.2.3 (Super)GLUE

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

The **G**eneral **L**anguage **U**nderstanding **E**valuation benchmark dataset is a collection of tools created with this in mind. It is designed to encourage and favour models that share common linguistic knowledge across tasks. These tasks include textual entailment, sentiment analysis and question answering. Some tasks come with a lot of training data, others with less. Common to all datasets is that they were not created specifically for GLUE, but are existing datasets. Models that are evaluated on GLUE only need to have the ability to process single-sentence and sentence-pair inputs and make appropriate

predictions. This test suite contains a total of nine sentence or sentence-pair NLU tasks, built on established annotated datasets. The models are scored separately for each task and then a macro-average of these scores is calculated to determine a system's position on the ranking. If a task has multiple metrics, an unweighted average of these metrics is used as the score for the task when calculating the overall macro average. (**?**) The human baseline score is 87.1, while the best model score is currently 90.6.

Roughly one year after the release of GLUE, models surpassed human performance. In response to this, a new benchmark, SuperGLUE, was introduced. It follows the same principles as GLUE, however the tasks included are more challenging. The two hardest tasks in GLUE remain, the rest were selected based on difficulty for current NLP approaches. (**?**) For SuperGLUE, the human baseline score is 89.8, which is above the best model score, presently 89.3.

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

https://super.gluebenchmark.com/

11.2.4 AQuA-Rat

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

The dataset contains over 100.000 questions, and each question has five different options as to what the correct answer is. It also contains the answer rationale and the correct option. The problems cover a wide range of topics, for instance probability theory or calculus, with a variety of difficulty levels. To create the dataset, examples of exams such as the GMAT (Graduate Management Admission Test) and GRE (General Test) were taken from the Internet. This part of the dataset is called the seed dataset. Besides, crowdsourcing was used to generate further questions. For this users were presented with five questions from the seed dataset and asked to select one of these questions and write a similiar question. Users were also forced to rephrase the rationales and answers to avoid paraphrasing the original

questions. These created questions were then passed to another user for quality control.

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

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

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

https://github.com/deepmind/AQuA

11.2.5 SNLI

Understanding entailment and contradiction is fundamental to understanding natural language, and conclusions about entailment and contradiction are a valuable testing ground for the development of semantic representations. The **S**tanford **N**atural **L**anguage **I**nference Corpus is a collection of sentence pairs that are labeled either as entailment, contradiction or semantic independence. While other datasets exist that try to tackle this specific task, they all have issues of size, quality, and indeterminacy.

SNLI consists of about 570k record pairs. Again, croworkers were used to create the data set. To do this, they were shown the caption of a photo but not the photo, and they were asked to write three alternative captions: One that is definitely a true description of the photo, one that could be a true description of the photo, and one caption that is definitely a false description of the photo. By not showing the photo, the authors wanted to ensure that each pair of sentences could be reconstructed based on the available text alone. To measure the quality of the corpus, about 10% of all created sentence pairs were validated. For this purpose, each crowdorker was shown five sentences and asked to mark them with one of the three labels. Each sentence was shown to a total of five crowdorkers. For each pair, a gold label

was assigned if at least three of the five annotators chose the same label. Around 98% of all sentence pairs were assigned a gold label, the rest was assigned a placeholder label.(**?**)

Models are once again evaluated with the accuracy of the predicted label. For the SNLI corpus there is no measurement for human performance. Currently, the best performing model is a semantics-aware BERT (SemBERT) with an accuracy of 91.9. - Understanding entailment and contradiction - https://nlp.stanford.edu/projects/snli/

11.2.6 LAMBADA

- **LA**nguage **M**odeling **B**roadened to **A**ccount for **D**iscourse **A**spects
- Word prediction task
  - Predict the last word of a sentence depending on the context
- Models can't rely on local context, must be able to keep track of information in the broader discourse
- https://www.aclweb.org/anthology/P16-1144.pdf

11.3 Pre-Training Resources

- BERT/ALBERT/CamemBERT
  - Trained on unlabeled data over different pre-training tasks
  - WordPiece embeddings with a 30k token vocabulary
- OpenAI GPT2
- GPT-3
- Google 5T

11.4 Resources for Resources

https://paperswithcode.com/area/natural-language-processing
https://datasets.quantumstat.com/

12

Software for NLP: The huggingface transformers module

Authors: Author 1, Author 2

Supervisor: Supervisor

13

Use-Bases for NLP

Authors: Author 1, Author 2

Supervisor: Supervisor

14

Natural Language Generation

Author: Haris Jabbar

Supervisor: Matthias Aßenmacher

14.1 Introduction

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

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

14.2 Definition and Taxonomy

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

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

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

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

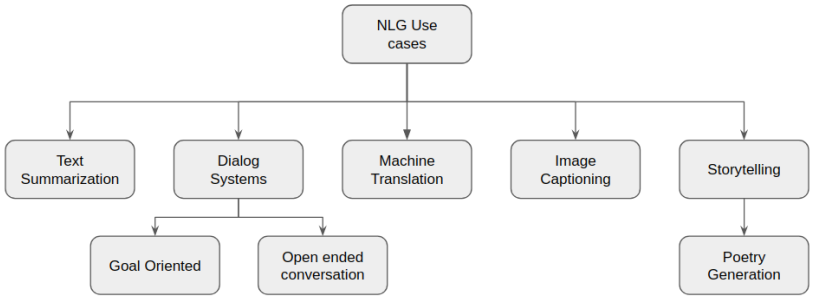


FIGURE 14.1: NLG Use Cases

14.3 Common Architectures

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

14.3.1 Encoder-Decoder Architecture

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

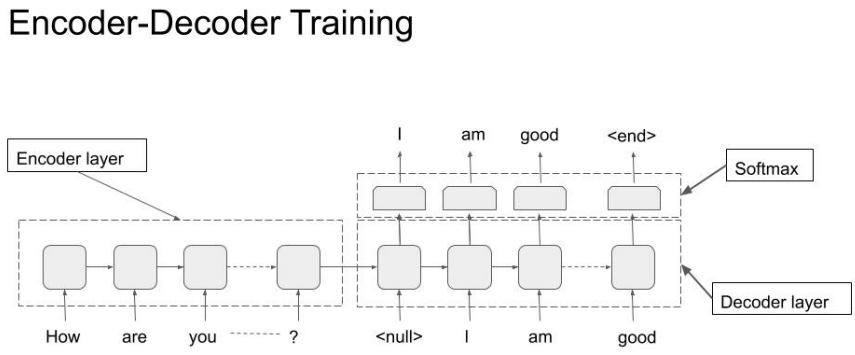


FIGURE 14.2: Encoder-Decoder (Training)

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

14.3.1.1 Encoder :

As stated above, the purpose of this part of the network is to provide conditional information on which the decoder generates text. As such, this part can have **ANY** architecture that provides some form of embedding of

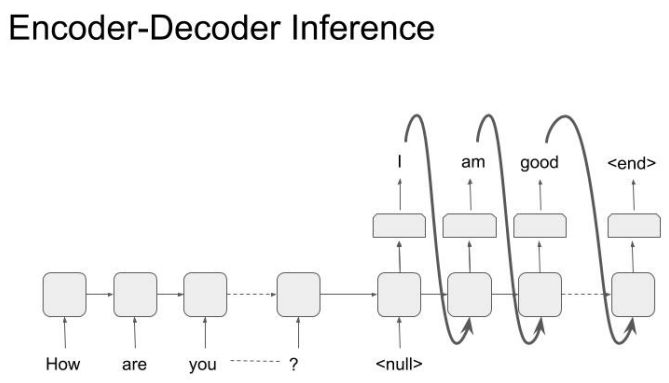


FIGURE 14.3: Encoder-Decoder (Inference)

the input. It can be a convolutional neural network to condition the generated text on some properties of an image (for example image captioning), or RNN/LSTM/Transformer architecture for text or audio based conditioning; or even a simple feed forward network to condition it on SQL database for example. For the purpose of illustration we will be using an RNN/LSTM with text as input condition (as shown in the figure).

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

14.3.1.2 Decoder :

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

During training, we are given an input (text/image/audio) and the 'gold label text' that we want the system to learn to generate for that particular input. In the given example, the input is the text "How are you?" and the gold label is "I am good". The input goes through the encoder and produces a feature vector that is used as the input to decoder. The decoder then

generates tokens one by one and the loss is calculated after the softmax layer from the generated token and the gold label token. Note the inclusion of an extra token '<null>' as the first token. The last token of the gold label should produce the '<end>' token.

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

14.3.2 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.

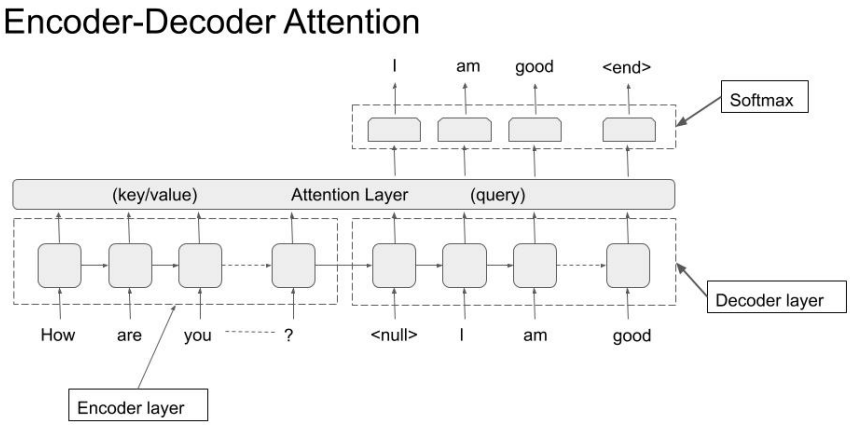


FIGURE 14.4: Encoder-Decoder (Attention)

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.

14.3.3 Decoding Algorithm at Inference

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

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

14.3.3.1 Beam Search

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

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

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

14.3.3.2 Pure Sampling Decoder

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

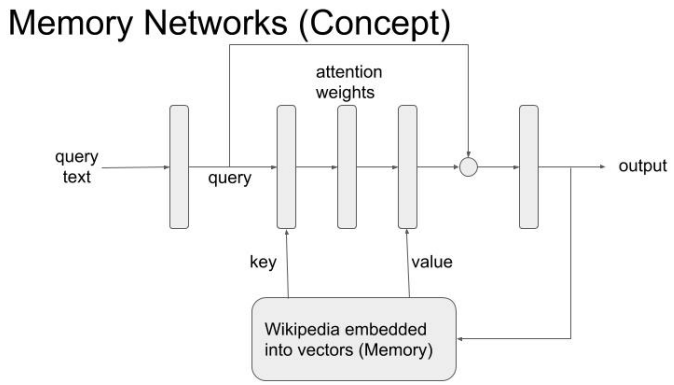
14.3.3.3 K-sampling Decoder

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

14.3.4 Memory Networks

Memory Networks is another architecture that is potentially quite useful in language generation tasks. The basic premise is that LSTMs/RNNs and even

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



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

14.3.5 Language Models

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

14.4 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.

14.4.1 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.

14.4.2 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.

14.4.3 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

14.4.4 Evaluation Metrics

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

14.5 Dialog Systems

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

14.5.1 Types

- Chatbots
  - Open domain
  - Goal Oriented

14.5.2 Architectures

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

14.6 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.

15

Use-Case II

Authors: Author 1, Author 2

Supervisor: Supervisor

16

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

Understanding lstm networks.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

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

Beltagy, I., Lo, K., and Cohan, A. (2019). Scibert: A pretrained language model for scientific text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3606–3611.

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

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*.

Bowman, S. R., Angeli, G., Potts, C., and Manning, C. D. (2015). A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

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.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer.

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

Howard, J. and Ruder, S. (2018). Universal Language Model Fine-tuning for Text Classification. *arXiv e-prints*, page arXiv:1801.06146.

Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., and Uszkoreit, J. (2017). One model to learn them all. *arXiv preprint arXiv:1706.05137*.

Kalchbrenner, N., Espeholt, L., Simonyan, K., van den Oord, A., Graves, A., and Kavukcuoglu, K. (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*.

Kudo, T. and Richardson, J. (2018). Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2019). Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*.

Ling, W., Yogatama, D., Dyer, C., and Blunsom, P. (2017). Program induction by rationale generation: Learning to solve and explain algebraic word problems. *arXiv preprint arXiv:1705.04146*.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Luong, M.-T., 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.

Merity, S., Shirish Keskar, N., and Socher, R. (2017). Regularizing and Optimizing LSTM Language Models. *arXiv e-prints*, page arXiv:1708.02182.

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.

Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (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.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. *arXiv e-prints*, page arXiv:1802.05365.

Peters, M. E., Ruder, S., and Smith, N. A. (2019). To Tune or Not to Tune? Adapting Pretrained Representations to Diverse Tasks. *arXiv e-prints*, page arXiv:1903.05987.

Prabhavalkar, R., Rao, K., Sainath, T. N., Li, B., Johnson, L., and Jaitly, N. (2017). A comparison of sequence-to-sequence models for speech recognition. In *Interspeech*, pages 939–943.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.

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

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Rajpurkar, P., Jia, R., and Liang, P. (2018). Know what you don't know: Unanswerable questions for squad.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text.

Reddy, S., Chen, D., and Manning, C. D. (2018). Coqa: A conversational question answering challenge. *CoRR*, abs/1808.07042.

Ruder, S. (2019). *Neural Transfer Learning for Natural Language Processing*. PhD thesis, National University of Ireland, Galway.

Ruder, S., Peters, M. E., Swayamdipta, S., and Wolf, T. (2019). Transfer learning in natural language processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pages 15–18.

Scherer, D., Müller, A. C., and Behnke, S. (2010). Evaluation of pooling operations in convolutional architectures for object recognition. In *ICANN*.

Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681.

Schwenk, H., Barrault, L., Conneau, A., and LeCun, Y. (2017). *Very Deep Convolutional Networks for Text Classification*.

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

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*.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1058–1066, Atlanta, Georgia, USA. PMLR.

Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. (2019). Superglue: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems*, pages 3261–3275.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. (2018). Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

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*.

Zhang, Z., Han, X., Liu, Z., Jiang, X., Sun, M., and Liu, Q. (2019). Ernie: Enhanced language representation with informative entities. *arXiv preprint arXiv:1905.07129*.