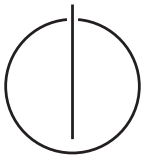
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

Transfer and Multitask Learning for Aspect-Based Sentiment Analysis using the Google Transformer Architecture

Felix Schober



DEPARTMENT OF INFORMATICS

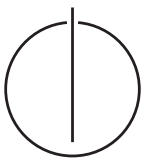
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

Transfer and Multitask Learning for Aspect-Based Sentiment Analysis using the Google Transformer Architecture

Transfer- und Multitask-Lernen für aspektbasierte Sentimentanalyse mit der Transformer-Architektur von Google

Author:	Felix Schober
Supervisor:	PD Dr. Georg Groh
Advisor:	Gerhard Hagerer M.Sc.
Submission Date:	15.05.2019



I confirm that this master's thesis in information systems is my own work and I have documented all sources and material used.

Munich, 15.05.2019 Felix Schober

Acknowledgments

Abstract

Contents

1 Introduction

1.1 Motivation

die zahl der personen die online reviews schreiben steigt stetig
manche produkte auf e-commerce seiten wie amazon können hunderte oder tausende reviews enthalten
nur ein paar reviews lesen -> time consuming und nur biased
eingedampft auf sterne wertung machnche aspekte wichtig und manche aspekte unwichtig. für andere personen anders
good motivation: "Reviews for products online are seldom fully negative or positive in sentiment. Rather, they describe the positive and negative core aspects of a product. To demonstrate this issue, consider an excerpt from this 3/5 star review for a laptop: "The faux leather cover is a wee bit cheesy for my taste, but I loved the price and the performance." For a purchaser, this review may not be useful when viewed only as a contribution to a mean score. The aspects "performance" and "price" have highly positive sentiment, while "appearence" receives a slightly negative sentiment. For a customer indifferent to the aesthetic of a laptop, this review should contribute higher than a 3/5 score to the mean. More useful to the consumer are summary statistics for each of a product's features. Our goal is to bring this structure to Amazon product reviews using deep learning." [**Marx2015**]
" However, the reader's taste may differ from the reviewers'. For example, the reader may feel strongly about the quality of the gym in a hotel, whereas many reviewers may focus on other aspects of the ho-tel, such as the decor or the location. Thus, the reader is forced to wade through a large number of reviews looking for information about particular features of interest." [**Popescu2005**]
Example: " In order to illustrate the task at hand, let us consider a text snippet expressing a customer's opinion about a particular beer. "This beer is tasty and leaves a thick lacing around the glass" This snippet discusses multiple aspects such as the taste of the beer and its appearance. The review expresses positive sentiments about both the aspects. It is interesting to note that the word "tasty" serves both as an aspect as well as a sentiment word in this case. The phrase "leaves a thick lacing" suggests that the snippet is discussing about the appearance of the beer and usage of "thick lacing" can be attributed to positive sentiment. This example demonstrates the intricacies involved

in the task of aspect specific sentiment analysis." [**Lakkaraju2014**]
"interleaving these two phases in a more tightly coupled manner allows us to capture subtle dependencies." [**Lakkaraju2014**]
"We conclude by examining factors that make the sentiment classifica-tion problem more challenging." [**Pang2012**]
it is very expensive to collect annotated data for absa. Therefore models have to be trained with less data.
Sales forecast based on sentiment prediction of customer reviews [**Shen2015**]

1.2 Outline

2 Related Work

This chapter will outline some of the past approaches to solve the task of Sentiment Analysis and Aspect Based Sentiment Analysis, as well as more recent approaches to give a general overview of the field. While the first section will deal with the traditional task of sentiment extraction, section **??** will discuss linking sentiment to specific aspects.

2.1 Sentiment Analysis

Traditionally, sentiment analysis has been approached by carfully engineering features which were handtuned to a specific work task. Most approaches used either rule based systems [**Popescu2005**] or lexicons to find the sentiment polarity of a document. Huettner and Subasic use a word lexicon for semantic categiues and a word affection lexicon with 4000 words. Each word in the lexicon is then assigned one category [**Huettner2000**].
Das and Chen use a similar approach where they manually compile a word lexicon. They use those words in combination with scoring functions to classify user generated posts on stock-market message boards as *bullish* (positive for the stockmarket) or *bearish* (negative for the stock market) [**Das2007**].
Hu and Liu improve upon the tedious task of building a lexicon by using only a small ammount of seed adjectives which is grown by the use of wordNet [**Hu2004**].

Lexicon based approaches are often manually build for a specific task like movie reviews [**Tong2001**, **Thet2010**] and therefore, generally domain dependet. For instance, sentiment in movie reviews is expressed very differently compared to a product or restaurant review. On the other hand, general sentiment lexicons like SentiWord-Net [**Baccianella2010**] are mostly to general to successfully capture domain specific sentiment.

Tourney uses an unsupervised system which classifies the sentiment of reviews as thumbs up or thumbs down. This is done by computing the closest association of adjectives and adverbs in a sentence. They then comopare the number of positive associations (associations to words like *good* or *romantic*) in a sentnece to the amount of negative associations (words like *horrific*) [**Turney2007**].

In 2012 Pang et. al. compare traditional machine learning techniques (Naive Bayes classifier, maximum entropy classification and Support Vector Machines (SVMs)) with human feature engineered baselines on movie reviews [**Pang2012**]. They still use word lists to classify the polarity of a sentence but they conclude that a sentiment lexicon generated by a machine learning algorithm always outperforms manually created lexicons.

Usually, lexicon approaches used to work in combination with scoring functions where the most trivial ones would just sum up the negative and positive words in a document. The algorithm would classify a document as positive if the sum of positive words is positive. However, there are limitations to this technique since sentences are often tree structured and may contain negated sub-trees. Just counting the positive and negative words in the sentence "*This film doesn't care about cleverness, wit or any other kind of intelligent humor*" would yield in a highly positive sentiment since it does not take the negation into account. Socher et al. introduce a "recursive neural tensor network" [**Socher2013**]. "Recursive neural networks" or "tree-structured neural networks" use the output of a forward pass as the input and some additional information as a new forward pass. By treating each word as a node in a tree they are able to input a subtree into the network, get the output and then use this output and the words on the next layer as the new input until the root node is reached. With this technique they are able to capture contrastive sentences where the first half contains negative sentiment and the second half positive sentiment in addition to negated sentiment.

2.2 Aspect Based Sentiment Analysis

Aspect Based Sentiment Analysis (ABSA) takes Sentiment Analysis one step further. Instead of just predicting sentiment for a sentence of document, Aspect Based Sentiment Analysis (ABSA) predicts sentiment for a specific aspect. This solves a major dilemma for normal sentiment analysis when a sentence or document contains multiple contradictive sentiments. For instance, the sentence "*The food tasted great but the price was too high*" contains two conflicting sentiment instances (Positive: great food - Negative: High Price). A classifier just modelling sentiment anaylsis would have to label this sentence as beeing neutral whereas an ABSA system could classify the aspect *food* as positive and *price* as negative. In addition ABSA is also much more useful for consumers and companies alike as discussed in the previous section.

2.2.1 Datasets

There are several datasets which provide tasks that include ABSA. Most widely used are the SemEval datasets. Task 4 of SemEval-2014 contains 7686 annotated sentences about restaurants and laptops [**Pontiki2014**].
Task 12 of SemEval-2015 expands the dataset of the previous year with the category hotels [**Pontiki2015a**].
Lastly, Task 5 of SemEval-2016 adds additional subtasks for ABSA and text-level aspect-sentiment extraction as well as two new domains in different languages [**Pontiki2015a**]. In addition they also provide a task where classifiers have to perform out-of-domain ABSA. Unfortunatelly, there were no submissions for this subtask. This is one of the most challenging tasks. Even modern deep learning architectures still struggle to successfully classify outside of the domain they were trained on.

Since annotating ABSA datasets is very expensive in term of annotation costs, even the biggest SemEval datasets only contain less than 10,000 sentences over all splits. In contrast, GermEval-2017 is a shared task dataset which contains more than 25,000 comments [**Wojatzki2017**].

2.2.2 Deep Learning on ABSA

One crucial element for deep neural network training on natural language is the ability to transform a sequence of words from sparse high dimesional vectors to dense sentence embeddings. The majority of recent publications on ABSA use sentence embeddings as their first input layer [**Ruder2016**, **Tang2016**, **Lee2017**, **Schmitt2018**, **Ma2018**, **Xue2018**] in combination with Long short-term memory (LSTM) architectures [**Ruder2016**, **Tang2016**, **Ma2018**].

In most approaches reasearches use a pipeline procedure to perform ABSA. During the first step, the models usually try to predict the aspects of the document. Then, in the second step of the pipeline the polarity is classified [**Lakkaraju2014**]. The winner of GermEval-2017 task-C use a pipeline approach [**Lee2017**] as well as three of the top performing approaches of SemEval-2016 [**Brun2016**, **Kumar2016**, **Xenos2016**].

Lakkaraju, Socher and Manning investigate the influence of pipeline approaches against joint aspect and sentiment classification [**Lakkaraju2014**]. They argue that combining aspect and sentiment detection gives a model the possibility to capture dependencies between aspect and sentiment more efficiently. They demonstrate that their joint aspect sentiment model using parse trees and Recurrent neural networks (RNNs) outperforms their pipeline baseline. To explain this they give several examples from the beer [**McAuley2012**, **McAuley2013b**] and camara-reviews dataset they use.

- "[...] delicious beer that's highly drinkable" - (Palate : Positive)
- "high carbonation level, kinda thin" - (Palate : Negative)
- "Display quality ofthe camera is high" - (Display : Positive)
- "This camera is highly expensive" - (Price : Negative)

Each of the four sentences contains the word "*high*" but in each sentence it appears in a different form and may lead to both positive and negative sentiment depending on the aspect. Lakkaraju et. al. state that due to the aspect-sentiment linkage the single sentiment approach was not able to correctly classify those sentences whereas their joint aspect sentiment model predicted correctly.

Ruder et. al. use a hierarchical, bidirectional LSTM for ABSA [**Ruder2016**]. Although they also assume a pipeline approach where aspects have to be fed in one after another they propose a interesting architecture. Usually, sentiment analyisis in combination with aspect detection is either done by having multiple heads for each aspect [**Schmitt2018**] or by using a big softmax layer with every possible combination of aspect and sentiment which runs into issues when classifying multiple aspects at the same time [**Lakkaraju2014**]. Ruder et. al. propose a third alternative where the aspect itself is fed into the network as an embedding alongside the actual text. After feeding the sentence embeddings through one bidirectional LSTM the text features and the aspect embedding are combined again. Together they are passed to the last bidirectional LSTM layer and classified in a last output layer. They report competitive results against two non-hierarchical baselines and achieve state of the art for multi-domain datasets.

In 2018 Schmitt et. al. propose a novel approach to classify aspect and sentiment jointly [**Schmitt2018**]. Similar to Lakkaraju they also compare joint-approaches and pipeline approaches. In both instances the joint approaches significantly outperform their corresponding pipeline approaches.
The architecture they use consists of two main components. The first layers are either bi-LSTM or Convolutional Neural Network (CNN).
Sentiment and aspect classification is then performed with aspect heads (Figure **??**). For each aspect in the dataset one aspect head is constructed. Each aspect head consists of a final softmax layer which produces a four-dimensional vector where three dimensions are polarity labels (negative, neutral, positive) and the last label indicates whether or not the aspect is relevant for this specific sentence.

By using aspect heads this architecture is not only able to classifiy aspect and sentiment together but also cabalble of multilabel classification.

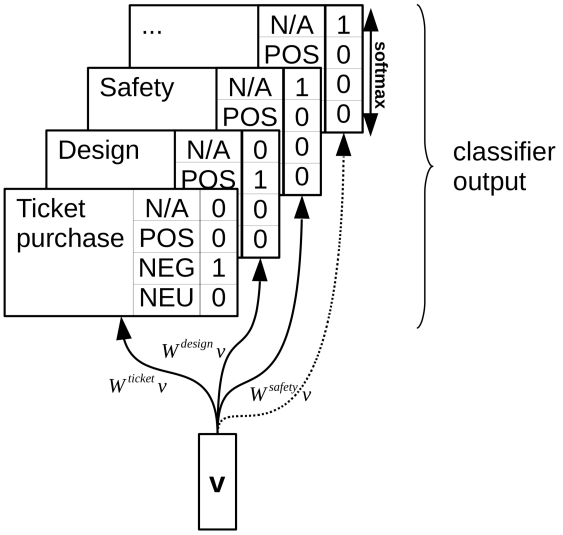


Figure 2.1: Aspect Heads proposed by Schmitt et al. Source: Schmitt et. al. [**Schmitt2018**]

Ruder et. al. tested this architecture on the GermEval-2017 [**Wojatzki2017**]. They report results that significantly outperform any submission on this task for this dataset. Their End-to-end CNN is able to achieve an F1-score of 0.423 and 0.465 on both test sets which is huge increase over the former State-of-the-art (SOTA) of 0.354 and 0.401 reported by Lee et. al. [**Lee2017**].

To the best of our knowledge there are no papers which use a transformer based architecture to perform joint aspect based sentiment analysis. There is one very recent publication which uses a BERT [**Devlin2018**] model to perform a pipeline ABSA task where they model those tasks as a sequence labeling problem [**Xu2019**]. Peters et al. use BERT for binary sentiment anaylsis with the Stanford Sentiment Treebank [**Socher2013**] on movie reviews [**Peters2019**] but none of those researches use the transformer directly.

Recently, a new extension to ABSA has emerged which combines ABSA with target-dependent sentiment classification [**Tang2016**]. The task of Target dependent sentiment classification is to infer sentiment based on a sentence and a given target. This is very similar to aspects but in contrast to ABSA the targets are not predefined. A target string resemlbles closely resemlbles a search string which makes this problem even more challenging since a model has to first understand the target string.

3 Theoretical Background

This chapter attends to the theoretical background for the technologies used in this thesis.

3.1 Convolutional Neural Networks

3.2 Word Representations

While it is trivial to use images as input for machine learning techniques the same can not be said for language. Images already come in matrix form while words appear as discrete objects. Before language can be used for machine learning, words have to be transformed into a matrix first. A function $W(\text{word}) \rightarrow \mathbb{R}^n \quad \text{word} \in V$ which encodes words from a vocabulary V into a n-dimensional matrix is called a word embedding.

The easiest way to achieve this is a one-hot encoding of a vocabulary. We just take the vocabulary which is a list of N words and number the words sequentially. A word vector is created by getting the index of the word from the vocabulary and setting the value of the word vector at the word-index to one and all other entries to zero.

$$W(\text{"dog"}) = [1, 0, 0]$$
$$W(\text{"cat"}) = [0, 1, 0]$$
$$W(\text{"bee"}) = [0, 0, 1]$$

While this works for small vocabulary sizes, with larger vocabularies one will encounter problems. When having a vocabulary size of $N = 10,000$ this approach will produce $10,000$-dimensional vectors for each word. What makes matters even worse is that those vectors are extremely sparse since they only have one non-zero entry.

3.2.1 Vector Space

In 1992 Schütze (and to some extend Hinton in 1986 [**Hinton1986**]) pioneered the idea of an universal word vector space [**Schutze1992**]. The general idea is that instead of having sparse, high dimensional word encodings it would be better to have low-dimensional,

dense word vectors where semantically similar words appear close together in the continuos word vector space. Unrelated words should be far away from each other. For example, the distance between the words "dog" and "cat" should be low while the distance between "bed" and "rocket" should be high. Figure **??** shows a visualization of a word vector space which has been reduced by t-SNE to two dimensions.

To achieve this, Schütze suggested to count co-occurrence of words and their neighbors and construct a co-occurrence matrix [**Schutze1992**]. While this makes the word-matrix dense (only 2% of the entries are zero) the resulting matrix is still extremely large. To get more useful vectors dimensionality reduction techniques like Latent Semantic Analysis (LSA) have to be applied.

Of course, this task can also be solved by neural networks. Imagine W to be a matrix $W \in \mathbb{R}^{N \times n}$ where each row corresponds to a word and n is the dimension of the desired word vector equivalent to a lookup table. We initialize the weights of W randomly and use W to transform words into dense vectors so that they look something like this:

$$
\begin{aligned}
W(\text{"dog"}) &= [0.3, -0.8, 0.1, \dots] \\
W(\text{"cat"}) &= [-0.1, 0.0, 0.2, \dots] \\
W(\text{"bee"}) &= [0.3, 0.6, -0.9, \dots]
\end{aligned}
$$

Instead of improving W directly, we use W to train a network on an auxiliary task like predicting the next word in a sequence. Instead of training W supervised we use an unsupervised task and train W indirectly. This is the basic principle how many modern word embeddings are trained and interestingly, this produces fascinating results shown in figure **??**.

Word vectors even allow for basic vector arithmetic. $W(\text{"King"}) - W(\text{"Man"}) + W(\text{"Woman"}) = W(\text{"Queen"})$ is a famous example for this demonstrated by Mikolov et. al. [**Mikolov2013d**].

Figure 3.1: Three regions of a word embedding visualized by t-SNE. The left region contains names of countries, the middle region consists of days and the right regions is made up of financial terms. Source: Turian et. al. [**Turian2010**][1]

For many Natural Language Processing (NLP) tasks, word representations have become indispensable and one of the factors of success for many systems [**Luong2013**]. The next sections will showcase a few of the most popular embedding techniques.

3.2.2 Word2Vec

Word2Vec by Mikolov et. al. is a very popular method which learns to produce word embeddings unsupervised from raw text [**Mikolov2013**]. The authors propose two approaches to learn an embedding. Both are very similar and rely on shallow neural networks to generate good embeddings. Moreover, these methods are computationally much more efficient than previous architectures without sacrificing performance. This allows word2vec to be applied on much bigger corpora with billions of words which was not possible before [**Mikolov2013c**].

Skip-gram Model

The training objective for the skip-gram model is to predict the neighborhood context of a given word. For example, given the word w_t and a window size of 2 the objective is to predict the two words before w_t (w_{t-2}, w_{t-1}) as well as the two words after w_t (w_{t+1}, w_{t+2}). The objective can be formalized as maximizing the average log probability [**Mikolov2013e**]:

$$\frac{1}{T}\sum_{t=1}^{T} \sum_{-c\leq j\leq c, j\neq 0} logp(w_{t+j}|w_t) \tag{3.1}$$

T is the number of words in the corpus, c is the context window size and $p(w_{t+j}|w_t)$ is the softmax function. Simply put, skip-gram predicts a context given a source word.

Continuous Bag-of-Words Model (CBOW)

Continuous Bag-of-Words Model (CBOW) is very similar to the skip-gram model. However, instead of predicting context words for a source word, CBOW predicts the next word w_t in a sequence, given the previous words w_{t-j} [**Mikolov2013c**]. In some instances the surrounding words are taken as the context instead of just the previous words. In other words, the objective of CBOW is to predict a word given it's context.

Skip-gram works better for smaller datasets than CBOW since it is possible to generate more training pairs from a sequence. In addition, it is also able to represent rare words

[1]Full source for image `http://metaoptimize.s3.amazonaws.com/cw-embeddings-ACL2010/embeddings-mostcommon.EMBEDDING_SIZE=50.png`

better than CBOW. However, CBOW is faster to train than skip-gram and yields slightly better accuracy for more frequent words.

3.2.3 Global Vectors (GloVe)

There are two methods for create word vectors: word co-occurrence counting (section **??**) and the window based skip-gram / CBOW methods (section **??**). Co-occurrence is relatively fast and efficient to collect and train unless the vocabulary is very big. A co-occurrence matrix also captures a lot of global statistical information whereas local window based models only capture the statistics for the window. However, most of the information a co-occurrence matrix provides is about the word similarity and not necessarily about the semantics. In addition, frequent words like "the" or "a" co-occur with a lot of other words. Therefore, they have huge co-occurrence counts with many words without providing any useful information [**Pennington2014a**].
Local window based models like word2vec have the disadvantage that training is not as efficient and a lot of statistical information is not captured [**Pennington2014a**].

Global Vectors (GloVe) which was introduced by Pennington et. al. tries to combine the advantages of both model families [**Pennington2014a**]. GloVe captures the global corpus statistics directly by creating a co-occurence matrix X where the entry X_{ij} denotes how often the word j occurs together with i. The probability that i appears in the context of j is

$$P_{ij} = P(j|i) = \frac{X_{ij}}{X_i} = \frac{X_{ij}}{\sum_k X_{ik}} \quad . \tag{3.2}$$

To put it simply, P_{ij} is just the number of times i and j appeared together divided by the sum of all words that i appeared with [**Pennington2014a**].
Pennington et. al. demonstrate that the ratio of probabilities P_{ik}/P_{jk} encodes a lot of the meaning that is lost in traditional co-occurence based methods [**Pennington2014a**].

The authors use F to describe this ratio as

$$F(w_i, w_j, \widetilde{w}_k) \approx \frac{P_{ik}}{P_{jk}} \tag{3.3}$$

w are just word vectors of i, j and k ($\widetilde{w}$) is a context vector. In the end the goal is to get a good candidate for F that fulfills all our requirements. By adding vector differences ($w_i - w_j$) and linear relations (dot-product) F becomes

$$F((w_i - w_j)^T \cdot \widetilde{w}_k) \approx \frac{P_{ik}}{P_{jk}} \quad . \tag{3.4}$$

By replacing the probability ratio with logarithms we arrive at

$$w_i^T \cdot \widetilde{w}_k = log(P_{ik}) = log(X_{ik}) - log(X_i) \quad . \tag{3.5}$$

The weighted least squares objective function is then finally

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \cdot \widetilde{w}_k + b_i + \widetilde{b}_j - log(X_{ij})^2) \tag{3.6}$$

where V is the vocabulary size and b and $\widetilde{b}$ are bias terms that are added.

The last problem is solved by a weighting function $f(X_{ij})$. Co-occurences which are infrequent will contain a lot of noise. Therefore it is important to reduce the influence of noisy words and prevent frequent words like "the" from dominating the model. Therefore, they weight each word to make sure that words are not over or underrepresented.

3.2.4 FastText

FastText is an embedding technique which focuses on efficiency but is still on par with other word embedding techniques [**Joulin2016**].

FastText is based on Word2Vec. However, whereas word2vec treats each word as an atomic entity, FastText splits words into several character n-grams. The word "where" consists of the following 5 n-grams for $n = 3$ [**Bojanowski2017**]:

"<wh", "whe", "her", "ere", "re>"

The special characters "<" and ">" are boundary symbols to differentiate prefixes and suffixes from the beginning and ending of words [**Bojanowski2017**].

This notion is very powerful since it provides several advantages over previous models. For one, out of vocabulary words are better recognized since their n-grams are still known. Unknown words can therefore be partly constructed by known n-grams. [**Bojanowski2017**].

Some languages like German which heavily rely on compound words also profit from this technique since those words can also be constructed using known n-grams [**Bojanowski2017**].

Lastly, FastText represents rare words a lot better than word2vec. Even if a word is very rare and only occurs in a sentence a few times in total, there is a high chance that the model consists of n-grams which appear in other, more frequent words [**Bojanowski2017**].

3.3 Google Transformer Architecture

The Google transformer architecture is a novel neural network architecture for automatic machine translation tasks by Vaswani et. al. [**Vaswani2017d**]. Instead of relying on RNN based architectures with recurrence, the transformer uses an attention mechanism to achieve State-of-the-art (SOTA) results on machine translation tasks. Like most of sequence-to-sequence models the transformer also uses the encoder-decoder pattern to translate a sentence from the source to the target language. We will mostly focus on the encoder part of the model since this is the part, we use to perform ABSA classification.

The issue when training many RNN-based architectures is the problem of long-range dependencies. These dependencies may lead to either vanishing or exploding gradient [**Hochreiter2009**]. The dilemma is, that language translations features a lot of long dependencies. For some translations like "I like planes" to "Ich mag Flugzeuge" each token exactly matches the position of the translation. However, in many cases the positioning of words is different. Take the sentence "I think I saw you at the airport *yesterday*". This would translate to "Ich glaube, dass ich dich *gestern* im Flughafen gesehen habe".
When translating this sentence from English to German, a recurrent architecture has to keep the dependency of "saw" of all the way until the end of the German sentence when it has to output the tokens "gesehen habe".

Besides the issue with long-term dependencies, RNN architectures are also almost impossible to parallelize which is especially important for long sequences [**Vaswani2017d**].

The transformer model tries to fix both problems. It is the first transduction model to completely gives up on recurrence or convolutions to compute sequence representations [**Vaswani2017d**].
Instead it relies on attention and specifically multi-head attention. Attention allows a model to focus on relevant information and the idea of multi-head attentions enables parallelization as well as attention heads which focus on their attention on different aspects [**Vaswani2017d**].

3.3.1 Encoder-Decoder Architecture

A high level architecture overview of the transformer is depicted in figure **??**. As already mentioned the transformer resembles an encoder-decoder architecture. Yet, in contrast to traditional encoder-decoders, the transformer consists of n encoder- and n decoder blocks [**Vaswani2017d**].

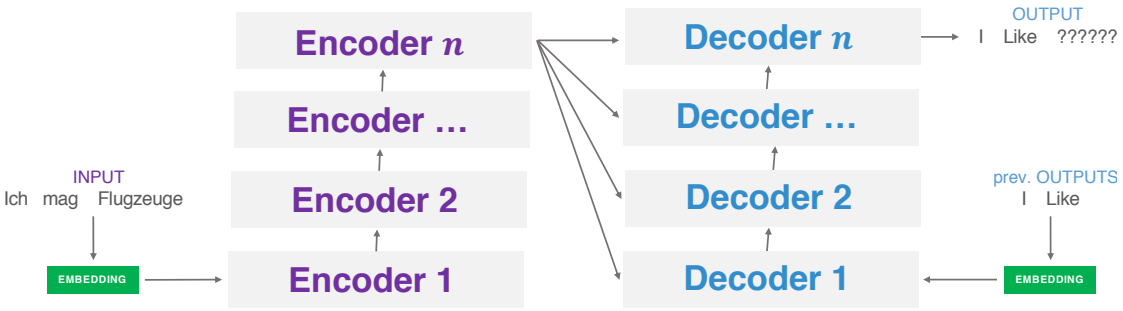


Figure 3.2: High-level overview of the transformer Encoder-Decoder architecture, translating the sentence "I mag Flugzeuge" from German to English. The model is at the step where it translates the last word "Flugzeug" to plane.

The encoder blocks always get the whole input sequence encoded as word vectors. This sequence is then propagated until it reaches the top encoder. From there, the encoder output is passed to all decoder blocks. Furthermore, the decoder blocks also receive the output of the decoder blocks below them. In addition, the first decoder block (in figure **??** this block is named "Decoder 1") receives the sequence of tokens the transformer already predicted [**Vaswani2017d**].

Following the example in figure **??** the translation would start by passing the sentence "Ich mag Flugzeuge" through all encoder blocks until it reaches the top block. All 1 to n decoder blocks receive the encoded sentence. The encoded sentence then flows up from the bottom to the top where it outputs the first token "I".
Decoder 1 now gets two inputs:

1. the encoded source sentence it previously also received
2. the previous predicted output which is the token "I"

Decoder 2 to n now get the encoded input (as before) as well as the output of the decoders below. The top encoder will then output "like" as the next token in the sequence.
The step that would follow this one is shown in figure **??** where "I like" is already predicted and only the last token is missing.

Figure **??** shows a detailed overview of one encoder and one decoder block. This is useful to comprehend the information flow on the decoder side.

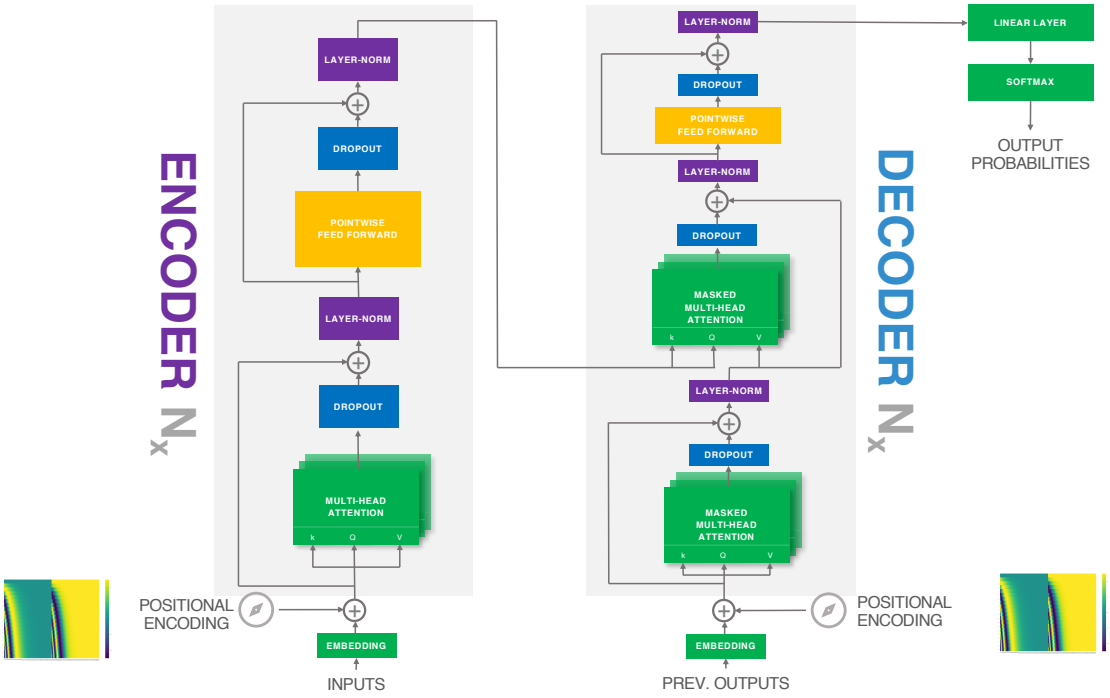


Figure 3.3: Overview of the whole Transformer architecture.

3.3.2 Attention Mechanism

The transformer uses attention to focus on the important and relevant information in a sequence. The question what is regarded as important is learned [**Vaswani2017d**]. The specific attention mechanism, the transformer uses is called "Scaled Dot-Product Attention" (Figure **??** left). The input for the attention function are three matrices called "Queries" Q', "Keys" K' and "Values" V'. We get these matrices from the input embeddings. We just multiply the embeddings with W^Q, W^K and W^V. These projection matrices are learned and project our input to a lower dimension so that we get Q', K', V' [**Vaswani2017d**].

We then take the dot-product of Q' and K' and scale the result by dividing by the square root of d_k where d_k is the dimension of Q' and K'. Next, we can finally apply the attention. From the softmax we get a vector of probabilities which are the attention values. When we apply the dot product on those attentions and the values we scale the values with the attentions [**Vaswani2017d**].

Words which are deemed important by the attention mechanism are multiplied with values near 1 while unimportant words are multiplied with small values and therefore

become unimportant for the next layers [**Vaswani2017d**].

One aspect which was not mentioned is the "mask"-step. After the scaling operation we set some values to 0. This is done because every input sequence needs to have the same length which we achieve by adding <pad> tokens to sentences which are not long enough. Those tokens carry no information so we can safely discard them in this step [**Vaswani2017d**].
In the decoder part we need to mask specific parts of the sentence which the transformer has not predicted yet. This is done to prevent the model from cheating by just replaying the input sequence [**Vaswani2017d**].

The transformer is constructed to use multiple "attention-heads" where each "head" performs its own dot-product attention (Figure **??** right). To get different values, each head i has its own set of W_i^Q, W_i^K and W_i^V matrices. The main idea behind the concept of multi head dot-product attention is that each head specializes on some aspect. One head might pay attention to entities while another head looks out for actions [**Vaswani2017d**]. This being said in practice this is not as clear and the distinction is more fuzzy.
The following equation summarizes the process which is described above:

$$\text{Attention}(Q', K', V') = \text{softmax}(\frac{Q' \cdot K'^T}{\sqrt{d_k}}) \cdot V' \tag{3.7}$$

$$\text{AttentionHead}_i(Q, K, V) = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3.8}$$

3.3.3 Positional Encoding

The transformer always gets the complete sequence of words as the input. However, since the transformer has no recurrence it has no memory. This means that it does not know which word is at which position. Yet, this information is very important to understand sentences [**Vaswani2017d**].

This issue is solved by a "positional encoding". This encoding is added to the word embeddings at the bottom of the encoders and decoders. It has the same dimension as the embedding so it can be added element-wise [**Vaswani2017d**].
For the positional encoding the authors use a combination of sine and cosine functions. They also experimented with a learned encoding but this approach did not perform significantly better [**Vaswani2017d**].

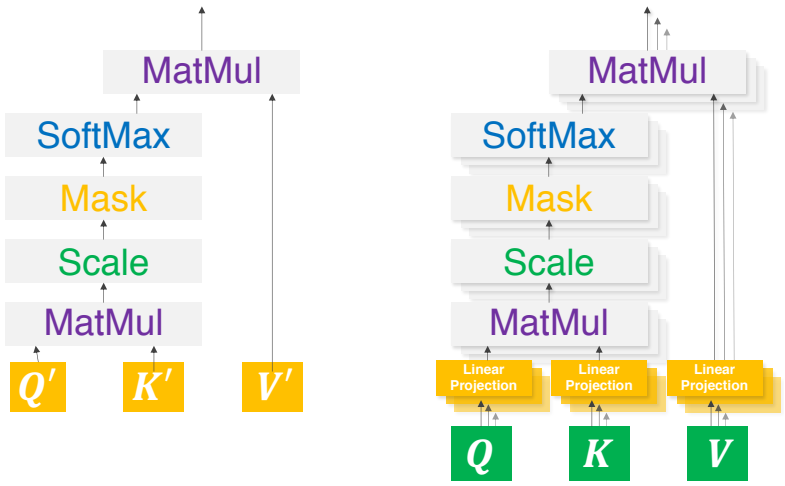


Figure 3.4: left: single scaled dot-product attention mechanism. right: multiple scaled dot-product attention heads stacked together. Each head receives its representation of Q, K and V by using projection layers.

3.3.4 Point-wise Layer

After the input is passed through the attention heads, a pair of fully connected layers with nonlinearities are used [**Vaswani2017d**].
The first linear layer scales the input to the inner layer dimensionality of 2048 and the second layer scales the input back to the previous model size of 512 [**Vaswani2017d**].

3.3.5 Adaptive Moment Estimation (Adam)

Vaswani et. al. propose to use the Adaptive Moment Estimation (Adam) optimizer [**Kingma2014**]. Adam is an extension to the popular Stochastic gradient descent (SGD) optimizer. However, instead of having a fixed learning rate like SGD, Adam computes a learning rate for each trainable network parameter. Those learning rates are directly obtained by the decaying mean and uncentered variance of the past gradients [**Kingma2014**].

This is different to AdaGrad [**Duchi2011**] and RMSProp [**Hinton**] which also maintain a learning rate for each parameter. However, RMSProp only use the mean and AdaGrad updates parameters based on how frequent the specific parameter is used as a feature.

3.4 Multi-Task Learning

Rich Caruana first introduced Multi-Task Learning (MTL) in 1993. Conventional machine learning approaches break a problem down in smaller tasks and solve one task at a time (e.g., word-by-word Part-of-Speech (POS)-tagging [**Toutanova2007**], word-by-word Named-entity recognition (NER) [**Sang2003**] or handwritten image classification [**LeCun;1990**]). In each of these tasks a classification algorithm solves exactly one task (Assigning a 'part-of-speech' or entity type to a word, or the classification of handwritten digits). Caruana shows that combining multiple related tasks improves model performance [**Caruana1993**][**Caruana1997a**].
In Multi-Task Learning (MTL), multiple related tasks are learned in parallel and share a common representation. Generally speaking every machine learning model which optimizes multiple objectives for a single sample can be considered as Multitask Learning. This includes multi-label classification where one sample can have multiple labels as well as instances where different sample distributions or datasets are used for different tasks.
MTL is similar to how humans learn. Generally, humans learn new tasks by applying knowledge from previous experiences and activities. For instance, it is easier to learn ice skating when someone previously learned inline skating. This is because all the underlying important aspects of the tasks are very similar.
When tasks are related this also holds true for machine learning. When learning these tasks in parallel model performance is improved compared to learning them individually since the additional knowledge that a related task carries, can be used to improve on the original task [**Caruana1997a**].
There are four important aspects one can use to determine if MTL can bring performance boosts for a specific objective:

1. Multi Label Task: Multi Label classification task where one sample can have more than one label are almost always inherintly solved using MTL if labels are predicted by one model. Multiple authors show that adding tasks always improves performance compared to a separate model for each task as an alternative [**Ramsundar2015**].

2. Shared low-level features: MTL only makes sense if the tasks share low level features. For instance, image classification and NLP do not share common features. In this case the model would not benefit from MTL because one task can not help to improve the other task. Therefore, it is important to choose tasks that are related to each other [**Zhang2017a**]. In most cases MTL will work with NLP tasks because they usually share at least some kind of sentence or word embedding as a common layer.

3. Task Data Amount: Several authors have suggested that it is important for the success of MTL training that the amount of data for the tasks is similar. Otherwise the model will mainly optimize for the task with most training samples.

4. Model Size: Finally, the multi-task model needs to have enough parameters to support all tasks [**Caruana1997a**].

3.4.1 Differentiation against Transfer Learning

Training samples from one task can help improve the other task and vice versa. This is important for the differentiation against transfer learning [**Pratt1993**]. In MTL each task is equally important. In transfer learning the source task is only used to improve the target task so the target task is more important than the source task [**Zhang2017a**]. In addition, Transfer Learning uses a linear training timeline. First, the source task is learned and then after learning is completed this knowledge is applied to boost the learning process of the target task. MTL, in contrast, is learning both tasks jointly together instead of one after the other.

3.4.2 Improvements through Generalization

There are several reasons why the MTL paradigm performs so well. For instance, the generalization error is lower on shared tasks [**Caruana1993**]. MTL acts as a regularization method and encourages the model to accept hypothesis that explain more than one task at the same time [**Ruder2017**]. The model is forced to develop a representation that fits the data distributions for all tasks. In the end this creates a model that generalizes better because it must attend to different objectives.

3.4.3 Improvements through Data Augmentation

Secondly, Multi-Task Learning increases the number of available data points for training. All tasks share a common representation. While training one task all other tasks are also implicitly trained through the common representation.

Statistical Data Amplification

Each new task also introduces new noise. Traditionally, a model tries to learn by ignoring the noise from its data. However, if the model does not have enough training samples it will overfit because it focuses too much on the noise to explain the data. By introducing additional tasks, new data and therefore new noise is introduced which

the model has to try and ignore [**Ruder2017**]. This aspect is called *Statistical Data Amplification*[**Caruana1995a**].

Blocking Data Amplification

Blocking Data Amplification occurs when there is little or no noise. Consider the simple example from Caruana [**Caruana1995a**] that there are two tasks T and T' and common features F. The first task T is $T = A \wedge F$ and the second task T' is $T' = \neg A \wedge F$. For $A = 0$ only T uses feature F and T' does not and for $A = 1$ it's the other way around. By training on both tasks F is used no matter what value A takes.
Rei makes use of these aspects and proposed a sequence labeling framework which uses a secondary, unsupervised word prediction task to augment other tasks such as NER or chunking. They show that by including the word prediction the auxiliary task performance is improved for all sequence labeling benchmarks they tried [**Rei2017**].
Similarly, Plank et al. show that learning to predict word-frequencies along with POS-tagging also improves the total model performance [**Plank**]. They argue that predicting word frequencies helps to learn the differentiation between rare and common words.

3.4.4 Architecture

The most common architecture for multitask learning is shown in figure **??**. It is called hard parameter sharing and consists of at least one layer which is shared among all tasks. In addition, each task has at least one separate layer. This approach is also the one we used for our model which is described in chapter **??**.
The easiest way to compute the loss for a hard parameter sharing MTL architecture is to take the sum of all losses for the individual tasks which is shown in equation ...

3.5 Transfer Learning

In 1991, Pratt et al. suggested to transfer information encoded in a neural network by reusing the network weights in a new network [**Pratt1991**]. They show that even accounting for the training time of the source network they achieved significant speedups when training a target network compared to random weight initialization.
Yosinski et al. provide a more modern definition: First, a base network is trained on a base dataset. Then, the learned features (the knowledge) of the base network is transfered to a second target network which is then trained on the target dataset and task [**Yosinski2014**]. This process works well if the base and target dataset and tasks are similar.

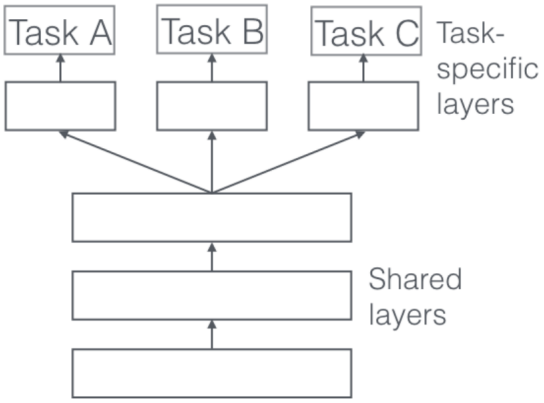


Figure 3.5: Hard parameter sharing. The first three layers are shared among tasks A, B and C. Each task also has one or more layers. Source: Ruder 2017 [**Ruder2017**]

Goodfellow et al. give a more general definition. They define transfer learning as the transfer of previously learned knowledge from one or multiple sources to a target domain with fewer examples [**Goodfellow2016**].
Figure **??** communicates those definitions.
In practice it is very expensive to collect or recollect training data for every new domain. Transfer learning makes it possible to transfer knowledge from a larger dataset to a smaller dataset which greatly reduces the labeling effort [**Blitzer2007**]. When the target dataset has significantly fewer examples than the base dataset studies showed that it is possible to train large networks without overfitting [**Donahue2013**][**Zeiler2014**]. Usually, after the base model has been trained on the large dataset, the first n layers of the base model are copied over as the first n layers of the target model. The remaining layers of the target model are then randomly initialized and trained. The weights of the n layers from the base model can either be *frozen* or *finetuned* along the rest of the target model. If the target dataset has few samples compared to the number of parameters in the first n layers, finetuning can actually result in overfitting which is a reason why the error during target training is often not backpropagated to the first n layers [**Yosinski2014**].

Pre Training

The most common way to employ transfer learning is pre-training. Pre-training is often used in image recognition were interestingly, the first few layers generally form into the same feature regardless of the domain or task [**Yosinski2014**]. Consequently,

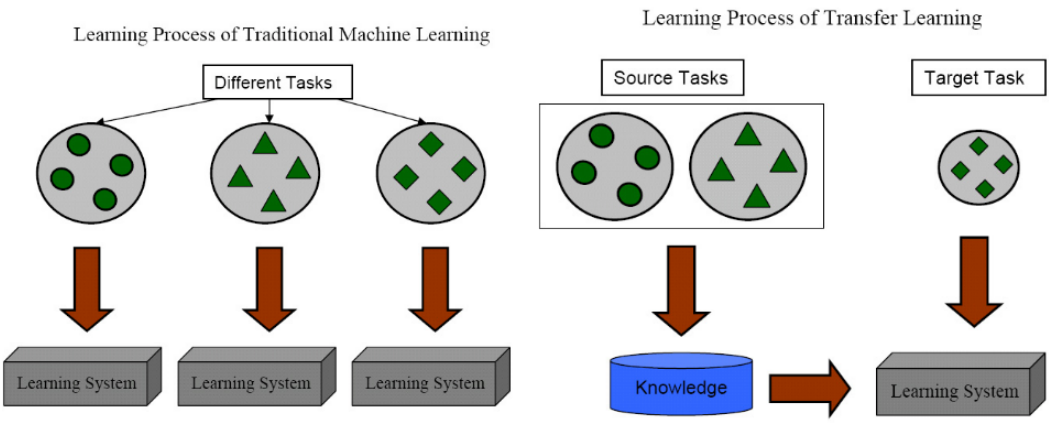


Figure 3.6: Difference in traditional machine learning were each model uses its own dataset and task. In contrast, in transfer learning a model is first trained on source tasks and part of the features are transformed to the target model to facilitate training. Source: Pan and Yang [**Pan2010**]

researchers are able to exploit this by taking the first layers from a model which was previously trained on a large dataset like ImageNet [**Russakovsky2015**] and use these weights for their tasks which might have less examples.
This paradigm can also be applied on natural language processing. Understanding what words mean is the fundamental problem every NLP model has to solve. Therefore, it is sensible to use an embedding layer which has been pre-trained on large datasets like common crawl which contain petabytes of information [**commonCrawl**]. This pre-trained embedding layer can then be used in a model trained on a much smaller dataset.

3.6 Hyperparameter Optimization

Generally, there are two sets of parameters in machine learning: learned parameters and hyperparameters which are used to configure various aspects of the training process. Learned parameters such as neural network weights are optimized during training whereas hyperparameters are usually defined at the beginning and without a few exceptions (e.g. learning rate) do not change during training.
It has been demonstrated that there are a few hyperparameters which have an enormous impact on the overall model performance but identifying those parameters among a big set of possible candidates is difficult [**Bergstra2012a**]. However, correctly setting

these parameters is crucial for achieving a good model performance. Cox and Pinto demonstrated that hyperparameters make the difference between a state of the art model and a model which does not perform better than a random classifier [**Cox2011**]. Therefore, hyperparamter tuning is critical for the model performance.
Hyperparameters are either hand tuned by reviewing similar literature or by the researchers understanding of the underlying architecutre and how he expects certain parameters to influence the architecture. Another way is to semi automatically optimize or fully automatically optimize the search for good hyperparameters.
This section presents three approaches to optimize the hyperparameter space. The first two are naive, semi-automatic approaches where a set of possible parameters is tried and the success is recorded. The researcher then selects the most promising results. The third example, HyperOpt, is an algorithm which treats the hyperparamter search as an optimization problem and identifies critical parameters and tries to find their optimum value for the given model and task [**Bergstra2013**].

3.6.1 Grid Search

Grid search is a very easy method to search for optimal hyperparameters. When performing a grid search for a parameter a new value is sampled from a predefined parameter subset at a fixed interval. Each trial with the new parameter value is then evaluated on the model. For multiple parameters each distinct parameter value is tested against all other parameter values, therefore creating a *grid* of parameter values to test. This approach is very easy to implement and is trivial to parallelize.

3.6.2 Random Search

Surprisingly, Bergstra and Bengio proofed that randomly choosing hyperparameters is more efficient than performing a grid search [**Bergstra2012a**] for high dimensional search spaces. Instead of defining values in a grid, they randomly sample from the grid space.
The problem with grid search is that by increasing the number of dimensions the number of trials has to increase exponentially to provide the same number of distinct trials for a single parameter [**Bergstra2012a**]. When performing a grid search on a one-dimensional parameter space, three runs on the model have to be performed in order to test three distinct values of the parameter. Optimizing two parameters (shown in figure **??**) increases the number of runs to 3^2 and optimizing n parameters m times will lead to m^n runs.
Grid search is set up on the assumption that each parameter is equally important. However, it has been shown that not all parameters are equally significant for the

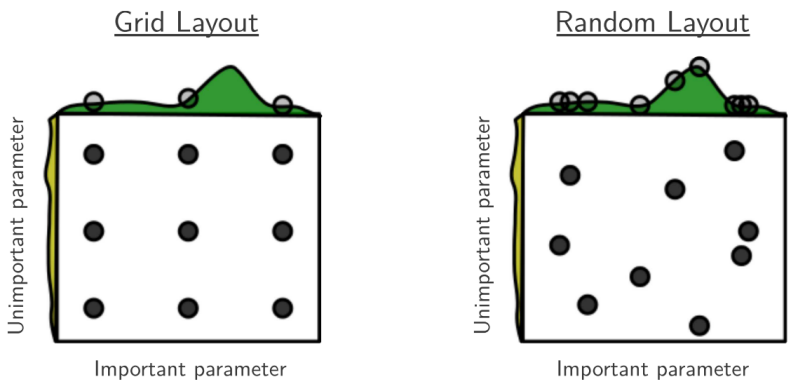


Figure 3.7: This figure from Bergstra and Bengio demonstrates the advantage of random searches over grid searches in a two-dimensional space. Nine trials are performed to optimize a function $f(x, y) = g(x) + h(y) \approx g(x)$. $g(x)$ shown in green has a bigger impact compared to $h(y)$ shown in yellow on the left. Each gray circle represents a trial. Because of the two-dimensional space, grid search can only test $g(x)$ in three places. Random search tries a different x in every trial and is therefore able to find a value close to the optimum. Source: [**Bergstra2012a**]

model performance [**Bergstra2012a**]. Figure **??** demonstrates why this is an advantage for random search over grid search. In this specific example one parameter constitutes more towards model performance than the other. However, grid search can only sample three values for the important parameter and is therefore not able to find the optimum value. According to Bergstra and Bengio this situation is the norm rather than the exception for grid search [**Bergstra2012a**].

3.6.3 HyperOpt

Hyperopt is an open source[2] hyperparameter optimization package by Bergstra et al. [**Bergstra2013a**]. It treats the hyperparameter search as an optimization problem. Bergstra et al. show that by using Tree of Parzen Estimators (TPEs) and Gaussian processes Hyperopt is able to find hyperparameters that outperform random searches and traditional manual hyperparameter tuning [**Bergstra2011**].

[2]Official repository `https://github.com/hyperopt/hyperopt`

Challenges

There are certain challenges when treating hyperparameter tuning as an optimization problem. For instance, the parameter search space is often high dimensional and may contain a mix of continuous (e.g. learning rate), discrete (e.g. hidden layer size), boolean (e.g. preprocessing steps [**Hutter2009**]) and even conditional variables [**Bergstra2013**]. For instance the choice of a optimizer or even the machine learning algorithm itself can be seen as a hyperparameter. Each choice then has its own set of parameters which are independent of the other choices. For instance, the Adam optimizer uses certain parameters like β_1 which the SGD optimizer does not use. Hyperopt generates a graph from these conditional parameters and then uses a tree like structure for solving the optimization problem [**Bergstra2011**].

Another difficulty is the limited "fitness evaluation budget" [**Bergstra2011**]. This means that for each evaluation of a hyperparameter set the model has to be trained which potentially takes a long time. Therefore, Hyperopt has to cope with less evaluation steps than a normal optimization algorithm.

Tree of Parzen Estimators (TPEs)

The Hyperopt package uses TPEs to sample good hyperparaters from the hyperparameter search space [**Bergstra2013a**]. To model $p(x|y)$ TPE replaces, all distributions in the configuration space by Gaussian mixture equivalents: uniform $\rightarrow$ truncated Gaussian mixture, log-uniform $\rightarrow$ exponentiated trucanted Gaussian mixture and categorical $\rightarrow$ re-weighted categorical [**Bergstra2013a**]. The prior for the calculation - the different observations $\{x^1, ..., x^n, ..., x^k\}$ - is initialized by performing n random runs where the default value for n is 10.
The TPE defines $p(x|y)$ as

$$p(x|y) = \begin{cases} l(x) & \text{if } y < y^* \\ g(x) & \text{if } y \geq y^* \end{cases} \tag{3.9}$$

where $l(x)$ (first case) is a density formed by an observation $\{x^i\}$ where the loss y of $f(x^i) = y$ was less than a threshold y^*. $g(x)$ is the density by using all other remaining observations [**Bergstra2013a**]. y^* is higher than the best observation so that the density $l(x)$ is formed by more than just one observation. $l(x)$ and $g(x)$ model the hyperparameter search space which means that they have to be hierarchical when the search space contains conditional and discrete variables. $l(x)$ and $g(x)$ are then used to optimize the expected improvement and after each iteration the parameter set with the

highest expected improvement is chosen for the next iteration which then becomes the next observation x^{k+1} [**Bergstra2013a**].

3.7 Methodology

3.7.1 Performance Measurements

Precession - Recall

The most used measure for the precision of food classifiers is the average accuracy which is calculated by dividing the number of correct matches and the total number of samples. Accuracy, however, gives no information about the underlying conditions. It is a measure of overall performance. To have a higher chance of suggesting the correct items, future systems may present a list of options that the user can chose from. Intuitively, the accuracy is much higher if a classifier can present a list of items with high confidences instead of only one item because the problem is much easier. Accuracy, however, does not measures how easy a problem is. If a classifier were able to suggest all classes as options the accuracy would always be 100% although the results are not useful at all.
The combination of precision and recall objectively measures the actual relevance and performance of a classifier for a class of images because it includes the amount of considered items and the correct predictions. In this case the amount of considered items changes based on how many items the classifier can suggest. Precision and recall is defined as:

$$Precision = \frac{T_p}{T_p + F_p} \quad Recall = \frac{T_p}{T_p + F_n}. \tag{3.10}$$

- True positives T_P is the number of correctly classified images of a class.
- False positives F_P are all images that the classifier predicted to be positive but are in reality negative. (Type I Error)
- False negatives F_N are all images that are positive (belong to the class) but are labeled as negative (do not belong to class) (Type II Error)

A high recall means that many images were matched correctly and a high precision denotes a low number of incorrectly classified images. The bigger the area under the Precision-Recall curve the better the classifier.

Null Error Rate

The null error rate is a baseline for any classification task that calculates the accuracy if a classifier would just predict the class with the most images.

Confusion Matrix

Confusion matrices are one of the most important metrics to understand why a classifier struggles with certain classes while getting a high precision with others. As the name suggests, a confusion matrix tells if the classifier "confuses" two classes.
A confusion matrix for n classes is always a $n \times n$ matrix where columns represent the actual images classes and rows represent the predicted image classes so if the diagonal of the matrix has high values this means that the classifier makes correct predictions.

Categorical Cross-Entropy

The categorical cross-entropy L_i is an error function that is used for the training of neural networks in classification tasks as the objective function. It is more versatile than the accuracy or the Mean Squared Error (MSE) because it takes the deviations of the predicted label $p_{i,j}$ and the actual label $t_{i,j}$ into account and weights the "closeness" of the prediction with the logarithm. For classification, cross entropy is more useful than MSE because MSE gives too much emphasis on incorrect predictions. The categorical cross entropy function is defined as:

$$L_i = -\sum_j t_{i,j} \log(p_{i,j}) \tag{3.11}$$

The loss values that are used for the discussion of results for neural networks are the average values of the categorical cross-entropy (Average Cross-Entropy Error (ACE)).

3.7.2 Cross Validation

Cross validation is one of the most essential techniques to evaluate real-world classification performance. Classifiers like SVMs or neural networks are always better on data they have already seen. This is called overfitting (see section **??**). By training and testing on the same data the classification performance would be much better than the actual real world performance. To test if a classifier can actually work with samples it has not seen cross validation divides the dataset into different partitions.
For most tasks it is sufficient to divide the dataset into a training and a test set. The data in the training set is used to train the classifier and the test data is used to evaluate it with data is has not seen before.

k-fold Cross Validation

To make the classification evaluation even more robust, k-fold cross validation is used. By applying k-fold cross validation the dataset is randomly partitioned into k different parts. $k-2$ parts are used for training and two parts are used for the evaluation. This process is repeated k-times and after each iteration the parts are exchanged so that at the end, each sample was used for training and for validation. Calculating the mean of the k evaluations gives a much more robust measurement because the evaluation does not depend on the difficulty of the test partitions.

3.7.3 Early Stopping

4 Method

The transformer model has shown great potential on a variety challenging NLP tasks. Originally, the transformer was created to perform machine translation where it outperformed previous models by a huge margin [**Vaswani2017**]. In fact, at the time of writing, the transformer is the core of the Google translate engine.

OpenAI uses the transformer to perform various NLP tasks [**Radford2018**]. Their Generative Pre-Training (GPT)-model solves classification, entailment, similarity and multiple choice question answering tasks. They use a transformer encoder stack with 12 encoder blocks and task dependent classification heads.
They show that even though the same base transformer model is used for each task they achieve SOTA results for most of the tasks.

One year later, in 2019, OpenAI published GPT-2 which is an extension to GPT [**Radford2019**]. With GPT-2 Radford et. al. are able to achieve revolutionary results generating text with a transformer architecture[1]. Again, they also achieve SOTA results on other NLP task mentioned above.

We propose the ABSA-Transformer (ABSA-T) which builds on the knowledge that the transformer is a powerful architecture useful for a variety of NLP tasks. Radford et. al. already show that the transformer can be used to predict sentiment for a document [**Radford2018**].
However, sentiment analysis is one of the few tasks where GPT-1 is not able to achieve SOTA results. Moreover, GPT-1 only classifies standard sentiment and not aspect-based sentiment.

This will be the first architecture which uses a transformer to perform multi-task aspect-based sentiment analysis. The next section describes the architecture of the ABSA-Transformer (ABSA-T) model. Section **??** describes how ABSA-T is used in combination with multi task learning and finally, section **??** explains how we used transfer learning to boost the training performance.

[1] OpenAI will not release the trained model for the language generation task as they are afraid that it will be used for malicious intents - Source: OpenAI blog `https://openai.com/blog/better-language-models/`

4.1 General Model Architecture

The following section describes the proposed ABSA-Transformer (ABSA-T) architecture. As the name suggests this design is based on the transformer model [**Vaswani2017**]. The second model characteristic is influenced by the work of Schmitt et. al. and their concept of separate aspect heads [**Schmitt2018**].

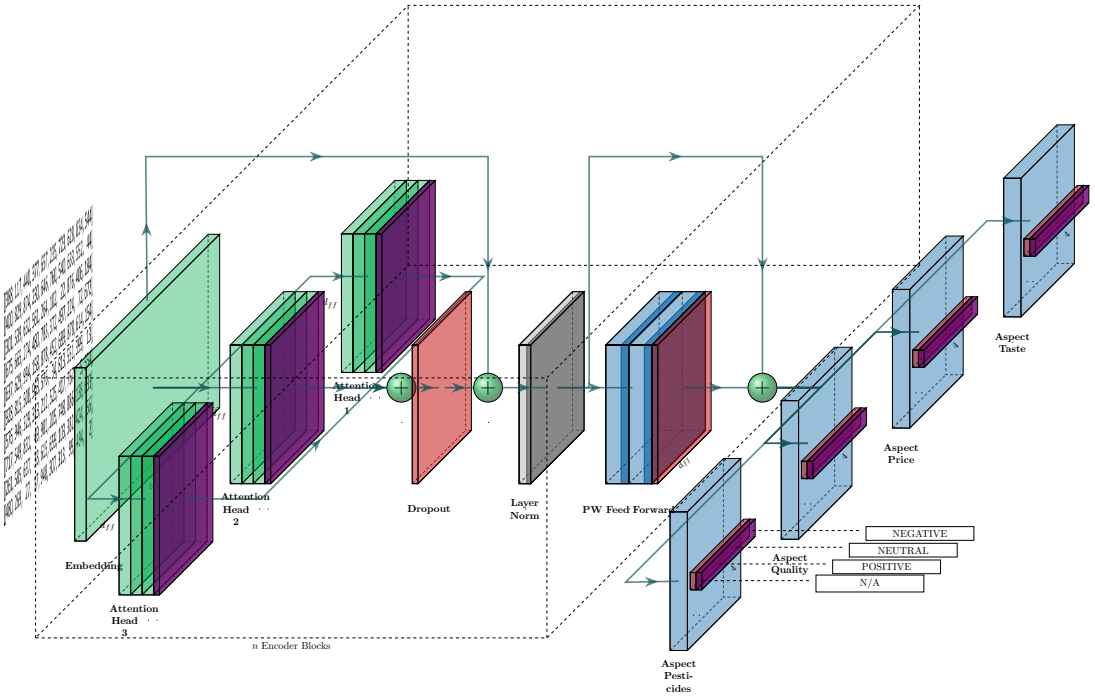


Figure 4.1: Visualization of an exemplary ABSA-Transformer (ABSA-T) model with one encoder block consisting of three attention heads and four aspect heads. The positional encoding is not visualized in this figure.

Figure **??** visualizes a simplified ABSA-T model. This specific instance consists one encoder block which contains 3 attention heads. To the right of the encoder block are four aspect heads. Each aspect head is trained to classifiy the sentiment for one aspect. In figure **??** those aspects are *GMOs*, *Quality*, *Price* and *Taste*. Each aspect has four output classes (visualized in the figure for the "Pesticides"-aspect):

- negative
- neutral

- positive
- not applicable (N/A)

Each aspect head is isolated from the rest which allows the transformer to perform multi-label ABSA. When a document or sentence contains a specific aspect the corresponding aspect head outputs either *negative*, *neutral* or *positive*. If this aspect is not part of the sentence, the output is *not applicable*.
We propose two different versions of aspect heads which are described in section **??** in detail.

4.2 Transformer

The original transformer uses undisclosed word embeddings which output a 512-dimensional vector[2]. It is possible that the original transformer does not use pretrained embeddings. We performed experiments with ABSA-T and untrained word embeddings but concluded that pretrained word embeddings outperform untrained word embeddings.
However, the difference was only a few percent and it is conceivable that a transformer with more training data would be able to train its own word embeddings.

Considering the smaller datasets that we use for ABSA, the ABSA-T model uses pretrained embeddings instead of untrained embeddings. Pretrained embeddings for both GloVe and fastText are only available for up to 300 dimensions. As a consequence, the model size of transformer is only 300 instead of 512.

Similar to the vanilla transformer, ABSA-T also uses an Adam optimizer [**Kingma2014**] and a special learning rate decay which is called NOAM[3] [**Vaswani2017**]

$$\text{NOAM:} \quad lr = d_{\text{model}}^{0.5} * min(step\_num^{0.5}, step\_num * warmup\_steps^{-1.5}) \tag{4.1}$$

Contrary to the transformer model, we do not use label smoothing as a regularization technique. Experiments showed that this impacted the F1-score negatively. Instead, ABSA-T uses weight decay with a decay value of $\epsilon_w = 1e - 8$.

[2]They also experiment with 256 and 1024 dimensional vectors

[3]It is not clear what noam stands for or where this learning rate decay schema came from. It is not mentioned or cited as noam but it is referred to as noam by the authors in discussions: `https://github.com/tensorflow/tensor2tensor/issues/280`

4.3 Aspect Heads

The transformer is designed as an encoder-decoder architecture with multiple stacks of encoders and decoders. Therefore, the input of an encoder (or decoder) has the same dimensionality as the output. Consequently, the input of an aspect head has the following dimensionality: $[batch\_size, s, d_{model}]$ where d_{model} is the model size and s is the sequence lenght. In other words, the transformer provides a d_{model} dimensional vector for each word w_i in a sequence.

The aspect heads have the role of transforming this vector to a vector which can be used for sentiment classification. For aspect-based sentiment classification this dimension is $[batch\_size, 4]$.

4.3.1 Linear Mean Head (LM-H)

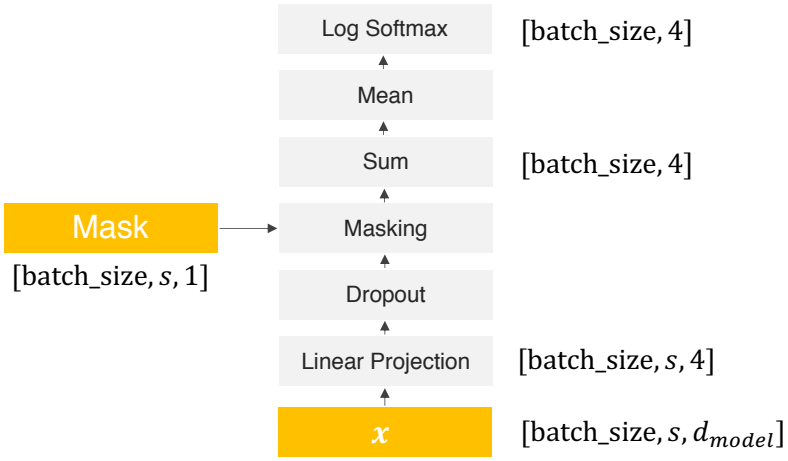


Figure 4.2: Visualization of the operations of an ABSA Linear Mean Head (LM-H)

The first aspect head design is called Linear Mean Head (LM-H). This head consists of a linear layer which projects the d_{model} dimensional word vector to a 4-dimensional word vector. This reduces the tensor to $[batch\_size, sequence\_lenght, 4]$. The tensor is then summed up along the second dimension and the mean is taken. Finally, a softmax operation calculates the log probabilities.
Similar to the transformer layers, optional masking is applied on the tensor. Predictions for words in the sequence which are only padding tokens are replaced by zeros.

Mean Operation

The mean after the sum acts as a normalization for the prediction values before they pass through the softmax. The log softmax is defined as

$$\sigma(y_i) = log(\frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}}) \quad . \tag{4.2}$$

As a consequence, large values for y_{ij} result in very large negative values after the log softmax. This has two effects on the NllLoss:

1. Long sequences will get a larger loss than shorter sequences. The reason for this is, that the sum-function will sum up all predictions for every word in a sequence. Of course, a longer sentence will have a larger sum. Therefore, the result of the softmax will be more negative which results in a higher loss.
2. It is not possible to compare the loss of Linear Mean Head (LM-H) and Convolutional Head (CNN-H). CNN-H uses a combination of convolutions and max pooling. Therefore, the numerical value of a prediction before the log softmax will usually be lower than for CNN-Hs, since max pooling takes the maximum and not the sum.

Using a mean operation before the log softmax solves both problems.

4.3.2 Convolutional Head (CNN-H)

The Convolutional Head (CNN-H) uses convolutions in combination with max pooling to perform the final prediction. Figure **??** visualizes the operations and how the tensor size changes during a forward pass through the head.

First, the tensor is passed through a convolutional layer. The filter size, the number of filters, stride and padding are controlled through hyperparameters. This means the size of subsequent layers depends on these parameters. The output size c_out of the convolutional layer is given as:

$$c_{out} = \frac{s + 2 * P - F}{S} + 1 \tag{4.3}$$

where F is the filter size, S is the stride and P is the padding amount. The max pooling layer reduces the tensor from $[.., d_{d_model}, c_{out}]$ to $[.., d_{model}]$ along the first dimension. Finally, a linear layer projects the output to class predictions and the log softmax scales the values.

Figure **??** depicts the convolution and the pooling in more detail.

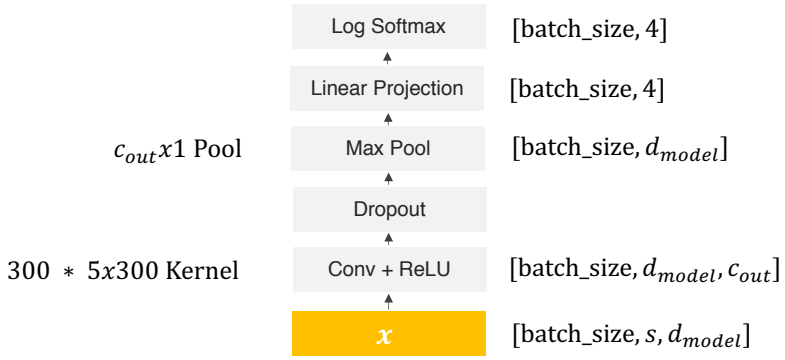


Figure 4.3: Visualization of the operations of an exemplary ABSA Convolutional Head (CNN-H). This specific head uses a kernel size of 5 and 300 filters.

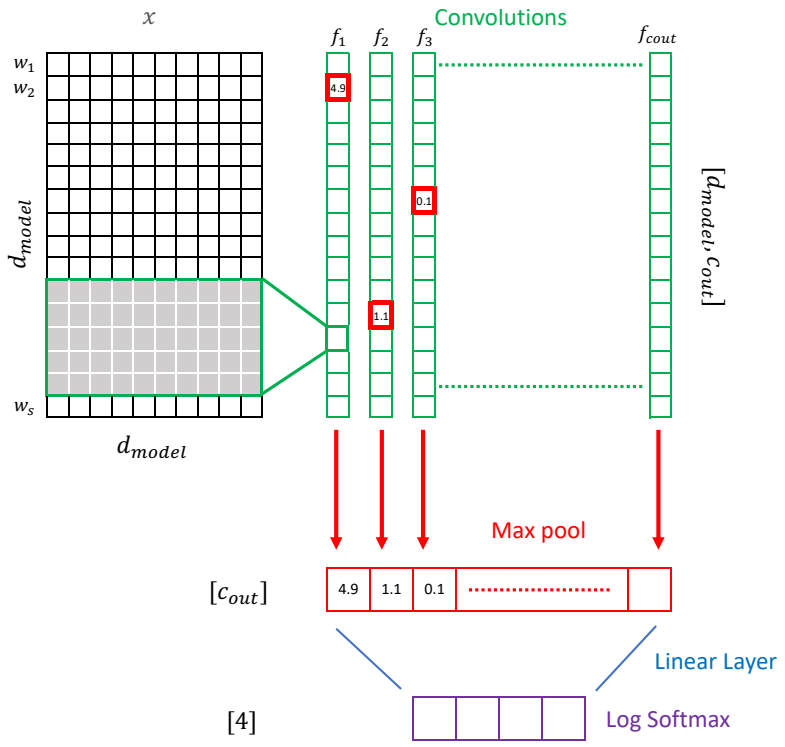


Figure 4.4: Visualization of the convolutional and max pooling operations of an ABSA-T CNN-H in detail. w_1 to w_s are the words in a sequence with length s. This figure does not include the batch size as an extra dimension.

4.3.3 Weighted Loss

Each aspect head calculates its own loss value using the predictions for the aspect the the targets. To combat data imbalance, we use a weighted NLL loss which is given as

$$\mathcal{L}_{\text{NLL}} = -\frac{1}{n}\sum_{i=1}^{n} w_i * (y_i \cdot log(\hat{y}_i)) \tag{4.4}$$

where n is the number of classes, w_i is the specific class weight, y_i is the ground truth and $\hat{y}_i$ is the prediction. Since we only use the aspect heads for sentiment classification, i will always be either a sentiment or not applicable. Therefore $i \in \{\text{negative}, \text{neutral}, \text{positive}, \text{not applicable}\}$ and $n = 4$.

The class weights are calculated during data loading. The equation for the calculation of a weight scalar for a class weight for i is given as

$$w_i = 1 - \frac{x_i}{N} \tag{4.5}$$

where x_i is the sum of all of i-s occurrences for the aspect. N is the total number of times an aspect occurred in the dataset.

Equation **??** assigns a low numerical value to classes which occur more frequently. As a consequence, the nll loss will be lower for frequent classes with low class weights which reduces the influence of those frequent classes.

4.4 Multi-Task Learning

The way the ABSA-Transformer is build, inevitably necessitates multi-task learning since for each aspect-head a separate Negative Log Likelihood (NLL)-loss is computed. For each of the m classes a loss value is calculated. In the end the mean of all losses is taken as shown in equation **??**.

$$\mathcal{L}_{\text{MultiTask}} = \frac{1}{m}\sum_{j=1}^{m} \mathcal{L}(f(x), y_m) \tag{4.6}$$

where $\mathcal{L}(f(x), y_m)$ is the NLL-loss of the model f with input the x defined in equation **??**.

Multitask Task Data Augmentation

As described in section **??** it is also possible to augment the data by using an auxiliary task in addition to the regular classification tasks.
There are three possible auxiliary tasks to consider:

1. Predict additional label from the source data
2. Use an additional dataset B in combination with the source dataset A and predict labels for the classes in A and the classes in B simulatanously. This approach is similar to transfer learning but instead of training the models sequentially, this approach would train them together.
3. Predict additional aspects of the source data which can be trained unsupervised.

This thesis will focus on the first type of auxiliary task where we will try predict an additional label for the source dataset. As the source dataset we choose the GermEval-2017 data since this dataset provides an additional document-wide sentiment label. This label was choosen since the other aspect heads already perform sentiment ananlisis so this task is very similar on the one hand but can provide additional datapoints for the training of the model. In addition the dataset provides a reasonable amnount of training data.
Training with the auxiliary sentiment label is performed by adding an additional sentiment head to the model. During training the loss of the auxiliary tasks contributes to the improvement of the transformer base.
However, during evaluation this task is ignored when calculating the F1-score for the model.

4.5 Transfer Learning

comparison to image first layer features which are very similar regardless of target domain. [**Yosinski2014**] -> Embedding layer, Transformer
However, last layer usually very dependet on domain and dataset -> good for model because last layer -> heads only domain relevant. So keep Embedding and transformer and exchange heads.

5 Experimental Setup

The following chapter describes the experimental setup for the discussion of results in chapter **??**. The first section of the chapter deals with data preprocessing. Section **??** lists all datasets used for evaluations of the models and finally section **??** provides detail about the training and evaluation process used to generate the results.

5.1 Data Preprocessing

The following section describes the general data preprocessing steps which were taken for all datasets described in section **??**. Some of the preprocessing steps are specific to certain datasets and will be described there. All data preprocessing steps can be enabled or disabled to evaluate the impact on the performance of these preprocessing steps. Some of those results will be discussed in section **??** in chapter **??**.

5.1.1 Text Cleaning

The main goal of the text cleaning step is

1. Reduce the number of words which are out of vocabulary
2. Keep the vocabulary size as small as possible.

without changing the semantics of the text.
The first step of the data preprocessing pipeline is the removal of all unknown characters which are not UTF-8 compatible. Those characters can occur because of encoding issues or words outside of the target language.

Contraction Expansion

Before we remove any special characters all contractions are expanded with the goal of reducing the vocabulary size and language normalization. Contractions are shortened versions of several words or syllables. In the English language, vowels are often replaced by an apostrophe. Especially in social media and spoken language a lot of contractions are used. *'I'll've'* and *'I will have'* have the same meaning but if they are

not expanded they produce a completely different embedding. *'I'll've'* will produce a (300)-dimensional vector (for glove and fasttext) whereas *'I will have'* will be interpreted as 3 300-dimensional vectors.

The contraction expansion is followed by the replacement of Uniform Resource Locators (URLs) with the token '<URL>' and e-mail addresses with the token '<MAIL>'. E-Mails and URLs are always out-of vocabulary and contain very little information that is worth encoding.

In addition any special characters are completely removed. Dashes ('-') are kept because there are compound-words which rely on dashes (e.g. non-organic).

Spell Checking

When writing comments in social media people tend to make spelling mistakes. Unfortunately, each spelling mistake is an out-of vocabulary word which we want to reduce as much as possible.
Therefore, a spell checker is used to prevent these mistakes. The first spell checker[1] which was evaluated relies on the Levenshtein Distance [**Levenshtein1966**] and a dictionary to determine if a word is spelled incorrectly and to make suggestions which word was meant originally. Although, word replacement suggestions are good, the spell checking is slow especially with large dictionaries.

The second spell checker is called Hunspell developed by László Németh[2]. Hunspell is used in a variety of open- and closed sourced projects such as OpenOffice, Google Chrome or macOS. Hunspell also utilizes the Levenshtein Distance in addition to several other measurements. Both spell checkers suffer from false positives (word is incorrectly flagged as negative) as well as incorrect suggestions. Below are examples of Hunspells suggestions for words it did not recognize:

- taste/flavor -> flavorless
- GMOs -> G Mos
- Coca Cola -> Chocolate
- didn -> did

All of the above replacements are very bad because they change the meaning of the entire sentence.

[1] PySpellchecker: `https://pyspellchecker.readthedocs.io/en/latest/`
[2] Hunspell: http://hunspell.github.io/

Nevertheless, in terms of vocabulary size reduction they are clearly outperforming other techniques as table **??** demonstrates. Running Hunspell on the Amazon dataset reduces the original vocabulary size of 1.6 Million by over 80% to about 311,000 unique words. In addition, as column *SP + TR-1* shows there are no tokens which only appear once. The reason for this is, that Hunspell always suggests something. Even words like *ˆ_ˆb4* are replaced by new words even if it would make more sense to delete those words altogether.

Stemming and Lemmatization

Stemming were also briefly explored, however, they did not provide a significant performance improvement.

Stopword Removal

5.1.2 Comment Clipping

The transformer works with different input sequence lengths within one batch. Therefore, it is possible to group similar sequence lengths together and have arbitrary sequence lengths. Unfortunately, in each dataset there is a small percentage of sequences which are longer than other sequences. Due to the limited computational resources a batch of those long sequences does not fit into Graphics Processing Unit (GPU) memory. Therefore, all sentences are either padded or clipped to a fixed length. This is also a requirement for the CNN-based transformer aspect head since CNN-layers need a fixed number of input channels.

5.1.3 Sentence Combination

Some datasets feature sentence annotations instead of comment annotations. In this case important information for the aspect and sentiment classification could be encoded in previous sentences. Refer to figure XX for an example.

Therefore, n previous sentences are prepended to the current sentence where n is a hyper parameter which can be optimized. Similar to the clipping of comment wise annotations described in the previous section, these sentence combinations are also clipped and padded.

The process starts by repeatedly adding sentences to a stack. All $n-1$ sentences which are too long are cut at the front. The n-th sentence is cut in the back instead. This is done so that in the case of $n = 2$
See section **??** for the evaluation of this preprocessing step.

5.2 Data

This section describes the four datasets which were used for the evaluation of the ABSA-Transformer architecture described previously.
The first dataset - CoNLL-2003 - is used to evaluate just the transformer model without the use of aspect heads. The task of this dataset is word level NER prediction. Since the original transformer model provides predictions on the word level this is good task to evaluate just the transformer part.

GermEval-2017 described in section **??** is a dataset for aspect-based sentiment analysis and contains over 25,000 review documents from social media.

Organic-2019 is a very recent dataset, also providing an aspect-based sentiment analysis task in the domain of organic food. Whereas, GermEval-2017 contains document level annotations, Organic-2019 contains word level over 10,000 annotated sentences. Organic-2019 is described in section **??**.

Finally, section **??** describes a new dataset consisting of Amazon reviews which was created with the goal to provide a big dataset as the source for transfer learning. The dataset contains almost 1.2 million reviews with 20 domains spanning the Amazon product catalog.

5.2.1 CoNLL-2003 - Named Entity Recognition

The CoNLL-2003 shared task contains datasets in English and German for Named-entity recognition (NER) [**Erik2003**]. NER describes the task of assigning labels to individual words. The four labels which are used for CoNLL-2003 are *persons*, *locations*, *organizations* and *names* [**Erik2003**]. For example the sentence "Gerry is a researcher at TUM in Munich" would be labeled as "[*PER* Gerry] is a researcher at [*ORG* TUM] in [*LOC* Munich]".

The English data which was used for this research consists of news stories which occurred between August 1996 and August 1997 [**Erik2003**]. The English dataset contains a total of 22,137 sentences with 301,421 tokens and is reasonably balanced in comparison to the datasets described in the next sections. Table **??** shows the distribution of the labels and the number of samples for each data split.

5.2.2 GermEval-2017 - Customer Feedback on Deutsche Bahn

GermEval 2017 is a dataset for Aspect-Based Sentiment Analysis on customer feedback about *"Deutsche Bahn"* in German [**Wojatzki2017**]. *"Deutsche Bahn"* is the largest railway

	Articles	Sentences	Tokens	LOC	MISC	ORG	PER
Train	946	14,987	203,621	7140	3438	6321	6600
Validation	216	3,466	51,362	1837	922	1341	1842
Test	231	3,684	46,435	1668	702	1661	1617

Table 5.1: Number of samples and labels for each split in the CoNLL-2003 English NER dataset

operator in Europe[3]. All data is collected from social media, blogs and Q&A pages over the course of one year from May 2015 till June 2016. Each document is annotated with a relevance flag, a document-level sentiment polarity as well as up to 19 different aspect-sentiment combinations such as atmosphere (*Atmosphäre*) or the experience of buying a ticket (*Ticketkauf*).

GermEval-2017 is a shared dataset for four different tasks:

1. Task-A: Relevance Detection
2. Task-B: General Document Sentiment Classification
3. Task-C: Aspect-Based Sentiment Analysis
4. Task-C: Opinion Target Extraction

This work focuses on Subtask C and results for the aspect-based sentiment analysis are reported in section **??**.

Beating the baseline systems of GermEval is not trivial since the dataset is extremely skewed towards the dominant category 'general' (*Allgemein*). This category makes up 62.2% of all the samples in the dataset. Some categories contain less than 50 samples which is only 2% of the whole data. Almost half of the aspects have less than 1% share of the total amount of samples. There is even one aspect *QR-Code* which has a total of two samples and none in the training split. Table **??** provides the detailed breakdown of the number of samples per aspect.
This imbalance is the reason why the GermEval-2017 majority class baseline is extremely strong. In fact, during the GermEval-2017 challenge there was only one other model submission from Lee et al [**Lee2017**] that could outperform the baseline models [**Wojatzki2017**].

[3]Financial Earnings Presentation 2014: `https://ir.deutschebahn.com/fileadmin/Deutsch/2014/Anhaenge/2014_finanzpraesentation_asien_de.pdf`

Aspect	Test-1	Test-2	Train	Val	Total	Ratio
Allgemein	1398	1024	12138	1475	16035	62,16%
Atmosphäre	148	53	1046	139	1386	5,37%
Auslastung & Platzangebot	35	20	251	33	339	1.31%
Barrierefreiheit	9	2	64	17	92	0.36%
Connectivity	36	73	257	23	389	1.51%
DB App & Website	28	18	185	23	254	0.98%
Design	4	2	31	4	41	0.16%
Gastronomisches Angebot	3	3	44	4	54	0.21%
Gepäck	2	6	18	3	29	0.11%
Image	0	3	51	7	61	0.24%
Informationen	58	35	330	34	457	1.77%
Komfort & Ausstattung	24	11	153	21	209	0.81%
QR-Code	1	0	0	1	2	0.01%
Reisen mit Kindern	7	2	44	4	57	0.22%
Service & Kundenbetreuung	63	27	486	49	625	2.24%
Sicherheit	84	42	429	63	618	2.40%
Sonstige Unregelmässigkeiten	224	164	1335	145	1868	7.24%
Ticketkauf	95	48	593	70	806	3.12%
Toiletten	7	4	44	5	60	0.23%
Zugfahrt	241	184	1798	190	2413	9.35%
Total	2467	1721	19,297	2310	25,795	100%

Table 5.2: Number of samples for each aspect per split in the GermEval-2017 shared task dataset.

In addition, there are some issues with the evaluation metric that the organizers of GermEval-2017 provide. Section **??** deals with this issue in detail.

5.2.3 Organic-2019 - Organic Comments

This dataset was collected and annotated in the end of 2018 and the beginning of 2019. It contains 1,373 comments and 10,439 annotated sentences from Quora, a social question-and-answer website.

Each sentence is annotated with a domain relevance flag, a sentiment and at least one entity-attribute-sentiment triplet. Out of the 10,439 sentences, 5560 sentences are marked as domain relevant. Out of the relevant sentences 668 contain two or more

aspect triplets.

There are 9 possible entities, each entity can have one of 14 attributes and the entity-attribute combination is annotated with a three-class sentiment polarity. In theory this combines to a total of 378 possible triplet combinations and 126 entity-attribute combinations. However, there are only 113 actual entity-attribute combinations and some of these combinations only have a few examples in total which makes this dataset even harder to train than GermEval-2017. The appendix contains two figures which show the distribution of the entities (figure **??**) and the attributes (figure**??**).

Since training on the full number of entities and attributes is very challenging the dataset also provides a coarse-grained version which combines both aspects and entities into a total of 18 bigger sets. The distribution for this dataset version is visualized in figure **??**.

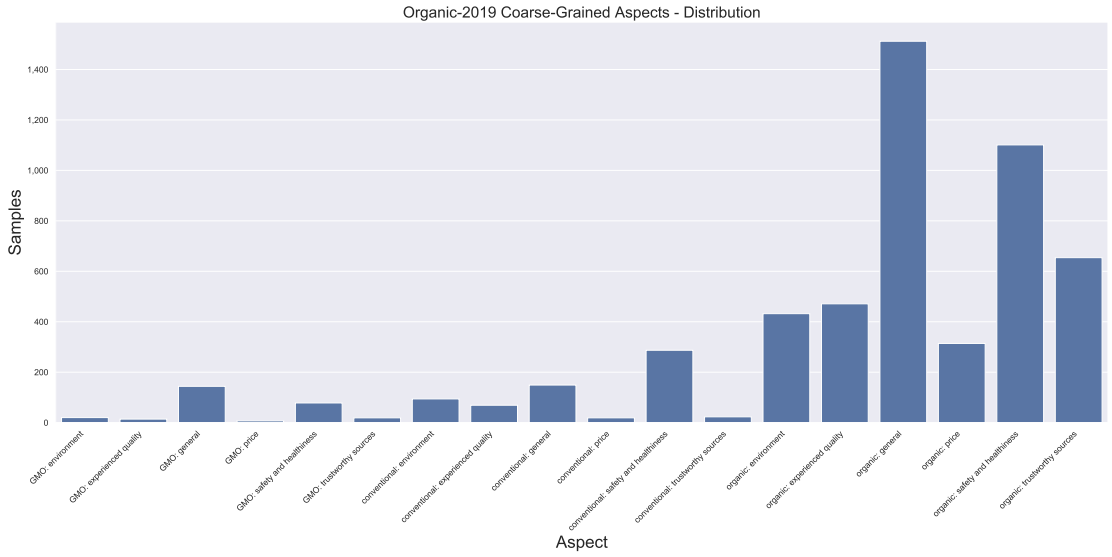


Figure 5.1: Distribution of the *coarse-grained* aspects in the Organic-2019 dataset

5.2.4 Amazon Reviews Dataset

The Amazon Reviews Dataset consists of over 130 million Amazon product reviews from 1995 until 2015. Therefore, this dataset is one of the richest data sources for sentiment analysis or other related NLP tasks. The raw data is available directly through amazon.[4] The reviews are grouped into 45 product categories such as "Grocery", "Luggage" or "Video Games".

[4] `https://s3.amazonaws.com/amazon-reviews-pds/readme.html`

Figure 5.2: Number of reviews per domain category in the amazon review dataset by McAuley et. al. [**McAuley2015**]

In 2013 McAuley and Leskovec compiled a subset of Amazon reviews [**McAuley2013**]. This dataset contains 34,7 million reviews ranging from 1995 till 2013 grouped into 33 categories[5]. The authors also created a "Fine Food" Dataset from Amazon reviews [**McAuley2013a**] [6]. This dataset consists of 568,454 Amazon reviews from 1995 till 2012. The domain of this specific dataset is related to the organic domain with 273 occurrences of the word 'organic'. Unfortunately, it does not contain predefined aspects so ABSA is not possible without extensive pre-processing to generate aspects out of the reviews.

The datasets created in 2013 contains duplicates so McAuley et. al. generated an improved Amazon Reviews dataset in 2015 without duplicates [**McAuley2015**][**He2016**]. This iteration of the dataset contains 142.8 million reviews from 1996 till 2014[7]. Due to the size of this dataset the authors provide a smaller dataset which only contains reviews from users who wrote exactly 5 reviews. This 5-core subset features 18 million reviews. The distribution of the domain categories is visualized in figure **??**. As one can observe the dataset is substantially skewed towards the largest domain 'books' which makes up of 49% of the data.

[5] Available through Stanford `https://snap.stanford.edu/data/web-Amazon.html`
[6] Available through Kaggle `https://www.kaggle.com/snap/amazon-fine-food-reviews`
[7] Available here: `http://jmcauley.ucsd.edu/data/amazon/`

To combat data imbalance and the sheer size of the dataset we propose a balanced subset of the 5-core dataset with 60000 reviews for each domain aside from *Musical Instruments*, *Amazon Instant Video*, *Automotive* and *Patio, Lawn and Garden*. These categories contain less than 50000 reviews so including them would skew the dataset again. In addition, we also transformed the star-rating to the common negative-neutral-positive rating schema. Similar to Blitzer et. al. we interpret $1-2$ stars as negative, 3 stars as neutral and $4-5$ stars as positive sentiment [**Blitzer2007**].

To create a balanced dataset not only on domains but also on sentiment we sampled 20000 reviews for each sentiment for each domain. Overall, there are more positive reviews than neutral or negative reviews. Thus, some domains contain less than 20000 reviews per sentiment category. To prevent data imbalance, reviews from the remaining other sentiment categories are sampled so that each domain contains 60000 reviews in sum. This distribution and additional statistics about the dataset are documented in table **??**.

Token Removal

There are over 145 million words in the dataset. These words combine into a vocabulary size of 1.6 million unique tokens and consequently into a very large embedding layer. (In comparison: the Organic2019 dataset has a vocabulary size of just 11,685.) Two techniques were used to reduce the vocabulary size:

1. Spell checking words
2. Removing rare tokens

The process for the first technique is described in section **??**. Another way to reduce the vocabulary size is by removing tokens, that only occur once or twice. These tokens make up the majority of the vocabulary size but only a small percentage of the overall word count. Table **??** shows the proportion of tokens which only occur 1, 2, or 3 times. As demonstrated in the table, infrequent tokens are very rare (all the tokens with one occurrence make up only 0.33% of the whole dataset). Yet, infrequent tokens make up over 74% of the total vocabulary size. Removing all tokens with one occurrence, therefore reduces the vocabulary size by 74% but only 0.33% of information is lost.
Most of these rare tokens are either incorrectly written (*nthis*), are part of structural elements such as headings (*review=======pros*) or are other unidentifiable characters and digits (*ˆ_ˆb4*).

Domain Category	helpful mean	Pos. Count	Neu. Count	Neg. Count	stars mean	# words mean	# words std
Apps for Android	0.22	20000	20000	20000	3.03	47	50
Baby	0.29	17012	17255	17012	3.33	105	106
Beauty	0.32	20000	20000	20000	3.10	90	94
Books	0.43	20000	20000	20000	3.08	176	201
CDs & Vinyl	0.44	20000	20000	20000	3.11	172	168
Cell Phones & Accessories	0.19	20000	20000	20000	3.06	93	138
Clothing Shoes & Jewelry	0.26	20000	20000	20000	3.11	67	70
Digital Music	0.53	47410	6789	5801	4.19	202	190
Electronics	0.43	20000	20000	20000	3.06	122	138
Grocery & Gourmet Food	0.33	28790	17514	13696	3.53	99	97
Health & Personal Care	0.35	20000	20000	20000	3.09	95	126
Home & Kitchen	0.44	20000	20000	20000	3.08	104	110
Kindle Store	0.35	20000	20000	20000	3.07	111	131
Movies & TV	0.39	20000	20000	20000	3.07	184	198
Office Products	0.29	45342	5060	2856	4.35	148	164
Pet Supplies	0.27	26412	15933	17655	3.35	91	96
Sports & Outdoors	0.30	20751	20000	19249	3.14	94	111
Tools & Home Impr.	0.40	39126	10769	10105	3.90	111	134
Toys & Games	0.32	11005	16357	11005	3.70	108	114
Video Games	0.41	20000	20000	20000	3.07	226	267
Total	0.35	506202	349677	337379	3.31	122	151

Table 5.3: Dataset statistics for the generated Amazon review subset for the domain categories. This table contains mean helpfulness rating; number of positive reviews; number of neutral reviews; number of negative reviews; mean star rating; mean number of words per review; standard deviation of the number of words per review

	Original	SP	SP + TR-1	TR-1	TR-2	TR-3
Word Count	148,129,490	-	0%	0.329%	0.389%	0.414%
Vocabulary Size	1,594,742	80.51%	80.51%	62.97%	74.41%	79.32%

Table 5.4: Different vocabulary size reduction techniques. This table shows the proportion of tokens that occur only 1, 2 or 3 times in relation to the total word count and the vocabulary size. *SP* is the spell checked dataset; *TR-n* is the token removal technique were *n* is the number times, tokens can occur in the dataset.

5.3 Training and Evaluation

5.3.1 Evaluation

The models that are used in this thesis are stochastic models since model parameters are randomly initialized. In addition, samples within the training batches are randomly shuffled. Therefore running the model multiple times leads to different results.

This means that it is necessary to collect model results multiple times. Unfortunately, k-fold cross validation is not possible for three out of the four datasets since the creators of the datasets provide a predefined split and changing the split during k-fold cross validation would prevent comparability with other results.
Therefore, for each dataset-result we repeat the experiment 5-times and report the mean and standard deviation. Iyer and Rhinehart suggest to run an experiment up to a 1000 times to get an optimal result [**Iyer1999**]. However, this is not possible for our models due to computational constraints.

All experiments on hyper parameters are performed once with a fixed seed of 42. This should make sure that all experiments on hyper parameters are reproducible. There are however some cudnn functions which are non-deterministic which means that even though a random seed is set the results could differ when running the same model with the same parameters multiple times.

GermEval 2017 - Evaluation

Wojatzki et al. [**Wojatzki**] provide an evaluation script for their dataset GermEval-2017. All results from the GermEval 2017 challenge were evaluated using this dataset. Therefore, all results reported in this thesis also use the evaluation script to calculate the f1 score. This is done to be able to compare the results on this datasets to other approaches on this data.

Table 5.5: Example for GermEval-2017 evaluation. None sentiment is not shown. Document 1 is predicted correctly. Document 2 has a correct prediction for aspect A but an incorrect prediction for the sentiment of aspect B (in bold).

	Gold	**Prediction**
Document 1	A : negative	A : negative
Document 2	A : positive B : **positive**	A : positive B : **negative**

Unfortunately, there are irregularities in the calculation of the micro f1 score. The evaluation script first creates every possible permutation of the combination of aspect and sentiment. If there are just two aspects (Aspect A and Aspect B) and four sentiments (n/a, negative, neutral, positive) this will generate 8 combinations (A-n/a, A-negative, ..., B-positive). This is used as the first input (*aspect_sentiment_combinations*) of the GermEval-2017 evaluation algorithm shown in **??**.

In the next step, all gold-labels and predictions are paired together for each document based on the specific aspect-sentiment combination. The example in table **??** will produce the following combinations where the left side represents the gold labels and the right side the predictions. This would be the second input parameter *golds_predictions* for algorithm **??**:

1. A:neg - A:neg (Document 1)
2. A:pos - A:pos (Document 2)
3. B:pos - B:n/a (Document 2)
4. B:n/a - B:neg (Document 2)

Using these inputs the algorithm will compute the following results:

- True Positives: 2
- False Positives: 2
- False Negatives: 2
- True Negatives: 26

which results in an f1-score of 0.5. In this example there is one misclassification where instead of predicting a pos. sentiment for aspect B the classifier predicted a neg.

sentiment. When looking at the combination B:pos as the *'true class'* the model predicts a negative (NOT pos. sentiment) when in reality this is a positive (pos. sentiment) which is the definition of a *'False Negative'*. When looking at the combination B:neg as the *'true class'* the model predicts a positive (neg. sentiment) when in reality this is a negative (NOT neg. sentiment) which is the definition of a *' False Positive'*.

One could therefore argue that instead of producing two False Positives and two False Negatives the correct evaluation should be one False Positive and one False Negative.

Algorithm 1: GermEval-2017 Evaluation script.

Input : *aspect_sentiment_combinations*: List of all possible combinations between aspects and sentiments including n/a, *golds_predictions* List of all comment wise pairs between gold labels and prediction labels

Output: $(tp, fp, tn\ fn)$

```
tp = 0 fp = 0 tn = 0 fn = 0
foreach (aspect, sentiment) in aspect_sentiment_combinations do
    foreach (gold), (pred) in golds_predictions do
        if gold matches current aspect and sentiment then
            if gold matches prediction then
                tp++
            else
                fn++
            end
        else
            if prediction matches current aspect and sentiment then
                fp++
            else
                tn++
            end
        end
    end
end
return (tp, fp, tn, fn)
```

5.3.2 Hardware

Training and evaluation of the models was done on four different machines. One of the servers belongs to the faculty of applied informatics, one is a local desktop machine

Table 5.6: Hardware used for model training

	OS	CPU	RAM	GPU
Desktop	Windows 10 (17134)	Core i5-6500 @ 3.20GHz	16 GB	GTX 1060
Social 5	Ubuntu 16.04.5	Xeon E5-2643 v3 @ 3.40GHz	126 GB	GTX 970
Azure	Ubuntu 16.04.5	Xeon E5-2690 v3 @ 2.60GHz	55 GB	Tesla K80
Google	Ubuntu 16.04.5	Xeon E5-2670 v3 @ 2.60GHz	15 GB	Grid K520

and the last two are cloud instances. One is an Azure virtual compute instance with 8 Central Processing Unit (CPU) cores and 28 Giga Bytes (GB) of Random Access Memory (RAM) and the other is a Google Cloud GPU compute instance instance with an Intel Xeon E5-2670 processor, 15 GB of RAM and a NVIDIA Grid K520 GPU. See table **??** for more details.

5.3.3 Docker

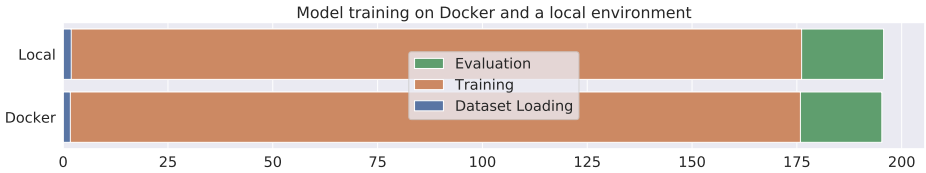


Figure 5.3: Docker vs. Local environment - Comparision of model training times.

Docker[8] is a framework for container virtualisation. Docker containers use the same kernel as the host system but an isolated file system with own system libraries.

Since training was performed on four different environments a Docker image was created which automates the installation of all required frameworks, environments, drivers and versions. An automated build pipeline builds a new image as soon as a new code version is pushed to the repository. Users can install or update an image directly from Docker Hub without rebuilding it every time locally.

The main concern of using docker for resource intensive task is the loss of performance due to the virtualization overhead. To evaluate this, epoch training time was measured with and without docker. The experiment was performed on machine-1 displayed in table **??**. For both experiments a complete model was trained for 5 epochs on the Organic2019 dataset. Figure **??** visualizes the time each part of the training took. For

[8]Docker: `https://www.docker.com`

both environments the mean execution time was around 195 seconds. This means that there is no difference between running a model inside a docker container or just locally. However, this is only the case when the host is running on a linux environment. On Windows and macOS, Docker has to virtualize part of the linux kernel. Therefore, there is no advantage of running on the exact same kernel as the host system. In addition, At the time of writing, the NVIDIA-runtime[9] is only supported for linux environments.

[9]NVIDIA Docker Runtime: `https://github.com/NVIDIA/nvidia-docker`

6 Discussion of Results

6.1 Hyper Parameter Optimization

The following presents the results of the hyper parameter optimization on GermEval-2017 Task C and the Organic-2019 Coarse category dataset. We evaluate the performance of Hyperopt compared to a random search. Then, the next sections show how certain parameters impact the model performance.

6.1.1 Hyperopt Evaluation

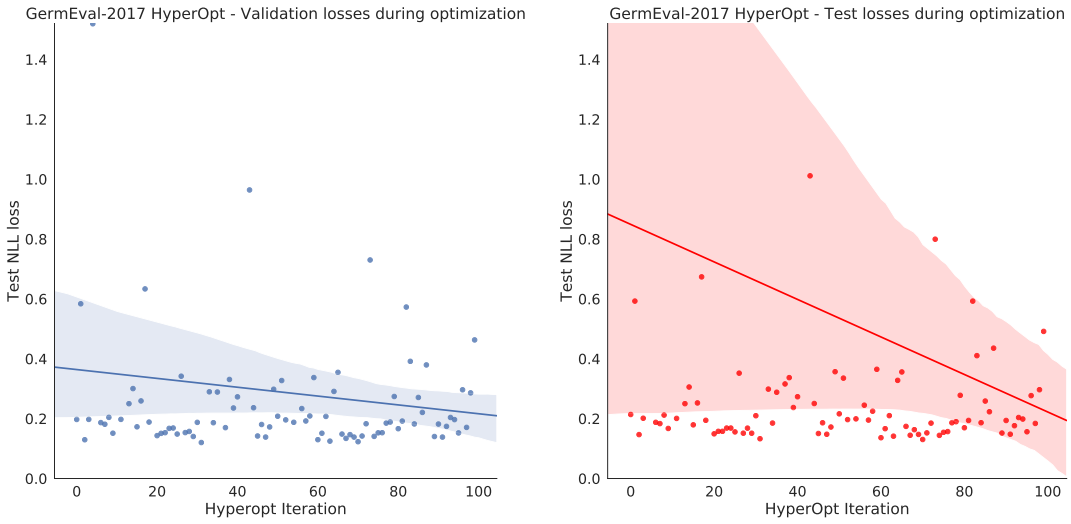


Figure 6.1: Validation- (blue - left) and test (red - right) losses during 100 Hyperopt iterations on GermEval-2017 dataset

To evaluate Hyperopt three evaluation runs with 100 iterations were performed.

1. GermEval-2017 - TPE on validation loss
2. GermEval-2017 - Random search

3. Organic Coarse Grained - TPE on validation loss

Figure **??** visualizes the improvement of the validation- and test losses on the GermEval-2017 dataset after 100 Hyperopt iterations. Is seems as if the regression line is negative in both cases which means that the TPE algorithm Hyperopt uses suggests better results as the time moves on.
Unfortunately, the Ordinary Least Squares (OLS) analysis **??** and **??** in the appendix show that the negative correlation is in fact not significant. This implies that the TPE algorithm does not sample parameters from the space which actually improve the loss of the model. This is even more obvious in figure **??**. This figure shows the development of F1-Score during optimization. While the results on the left for GermEval might look like they improve over the course of the optimization the results[1] for the optimization of the coarse organic dataset clearly show no improvement.

There are several possible explanations which could contribute to this behaviour:

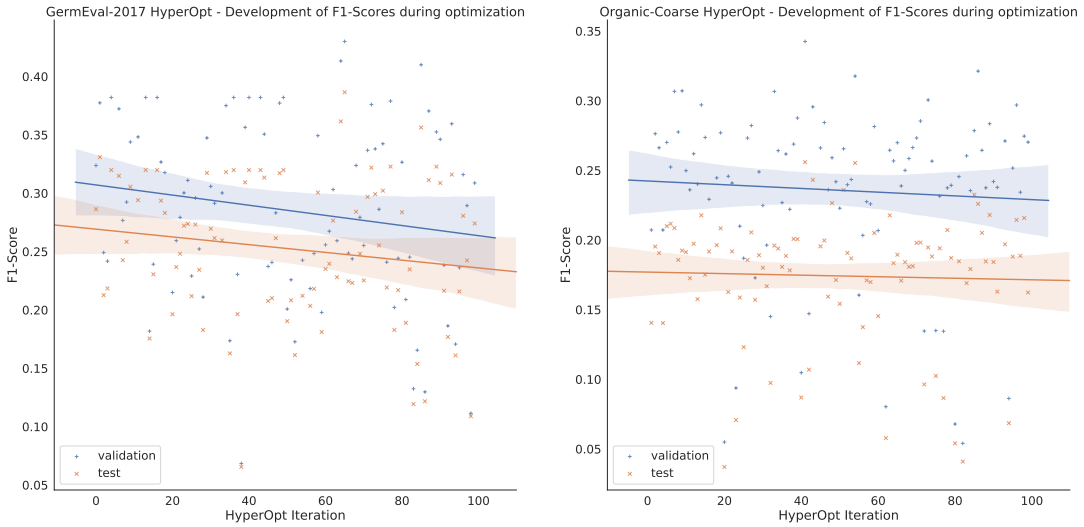


Figure 6.2: F1 Scores of the hyperparamter optimization of GermEval-2017 (left) and Coarse Organic 2019 datasets (right).

Iterations

First, it could be possible that 100 iterations are not enough to provide a stochastic model which is able to make good predictions in the hyperparameter space. It is worth

[1] The improvement is still not significant (0.340)

GermEval-2017					
Aspect Head	count	mean	std	min	max
Warmup Iterations 1 - 10*					
CNN-A	6	0.267644	0.050316	0.212655	0.330922
MLS-A	3	0.294608	0.032134	0.258432	0.319838
TPE Iterations 1 - 100					
CNN-A	67	0.249094	0.066835	0.065565	0.386465
MLS-A	23	0.261533	0.044603	0.181078	0.356296

Table 6.1: Result of sampling of Aspect Head choices during TPE Hyperopt optimization. Values show micro F1-score achieved by models on the GermEval-2017 dataset. * The 10th iteration failed which is the reason why the warmup does not sum up to 10.

noting that Bergstra et. al. show that hyperopt outperforms random search within 200 trials [**Bergstra2013**]. However, for most architectures it is not feasible to run an optimization search for much longer than 200 iterations let alone the 1000 iterations they claim as the point where hyperopt converges.

It is also worth mentioning that the Hyperopt module uses a random sampler for the first 10 iterations to get data points to initialize the TPEs. Decreasing this number could yield to better results for computational expensive models since the algorithm is forced to suggest values earlier.

Hyperparamter Search Space

It is possible that the hyperparameter search space which Hyperopt uses to generate new parameters is too large. Table **??** shows the hyperparameter search space for the optimization of the GermEval-2017 dataset. There are parameters which do not change the outcome by a huge margin and then there are parameters which decide whether or not the model trains at all. However, finding those parameters is a challenging task.

Warmup Phase

TPE supports tree structures for the search space. In the search space used for the optimization there is one tree-like parameter which is the choice of the aspect head architecture. TPE can either choose a Mean Linear Sum Aspect Head (MLS-A) or a

CNN-based Aspect Head (CNN-A). The MLS-A does not have additional parameter nodes, whereas the CNN-A has 4 additional parameters.

MLS-A has a higher mean F1-score of 0.263 compared to CNN-A which achieves 0.250. Despite the higher mean score, TPE only sampled MLS-A 23 times compared to 67 times.
This becomes even more interesting when looking at the TPE warmup phase. During warmup, MLS-A is chosen 3 times and CNN-A is chosen 6 times. This is roughly the same distribution compared to the later TPE iterations.
For greater detail refer to table **??**. There is also a violinplot which visualizes the impact of the aspect head choice in the appendix as figure **??**.

In contrast, during the warmup phase of the hyperopt run on the coarse organic dataset, Hyperopt sampled CNN-A 2 times and MLS-A 7 times which is exactly the other way around. During the TPE iterations, CNN-A was sampled 14- and MLS-A 71 times. Again, this is the exact opposite of the previous optimization on the GermEval-2017 dataset.

This leads to the following conclusion: During the warmup phase of Hyperopt the search space is randomly sampled. This random sampling distribution is adhered to during the whole TPE suggestion phase. This leads to results which are heavily dependent on the first 10 random iterations.

Comparison with a Random Search

To confirm the findings above a completely random search was performed by Hyperopt on the same number of iterations. The result of this comparison is plotted in figure **??**. This violinplot visualizes the result of both optimization runs. The light blue density curve on the left shows the F1 scores from the TPE generated models. A wider body on the density curve means that more observations for this particular score value were recorded at this part. The black dots correspond to the actual individual F1-Score observations. The dark blue parts and the white dots correspond to the F1-scores which originated by randomly generated hyperparameters.

6.1.2 Model Parameters

Due to the high dimensionality of the hyperparameter search space, statistical significance tests will not show any significant correlations between the parameter and an improvement of the F1-Score. For each single parameter change, all other parameters will also change and they influence the model as well. While not possible to evaluate

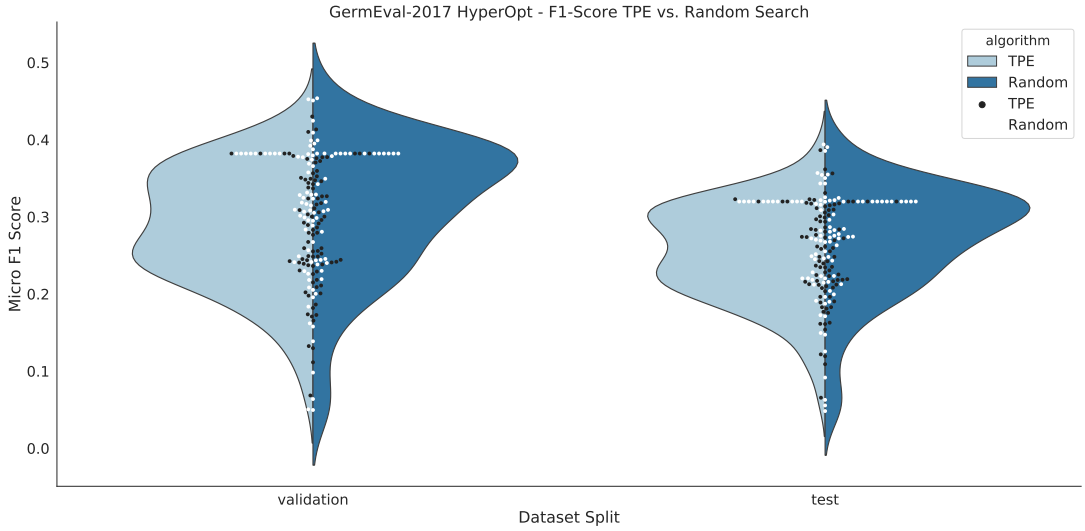


Figure 6.3: Comparison of HyperOpt TPE algorithm against a classical random search. The results for TPE are light blue on the left, whereas results for the random search are a deep blue on the right.

the results with statistical significance it is possible to derive certain assumptions from the data which will be discussed in the following sections.

Aspect Heads

As discussed in section **??** it is not entirely possible to favor one or the other aspect head. Both can provide similar results.
Figure **??** shows the impact of the two CNN-A parameters 'Kernel Size' and 'Number of Filters'. The number of filters does not seem to impact the result. However, there is a (statistical[2]) significant negative correlation between the kernel size and the model performance.
Smaller filters lead to a performance improvement compared to bigger filters. This result seems to follow the literature. For instance, Schmitt et. al. use filter sizes of 3, 4 and 5 [**Schmitt2018**].

There is no significant change for the other two parameters 'Kernel Padding' and 'Kernel Stride'. The impact on the F1-Score of both parameters is visualized in figure **??** in the appendix.

[2]Significant at a p-value of 0.05

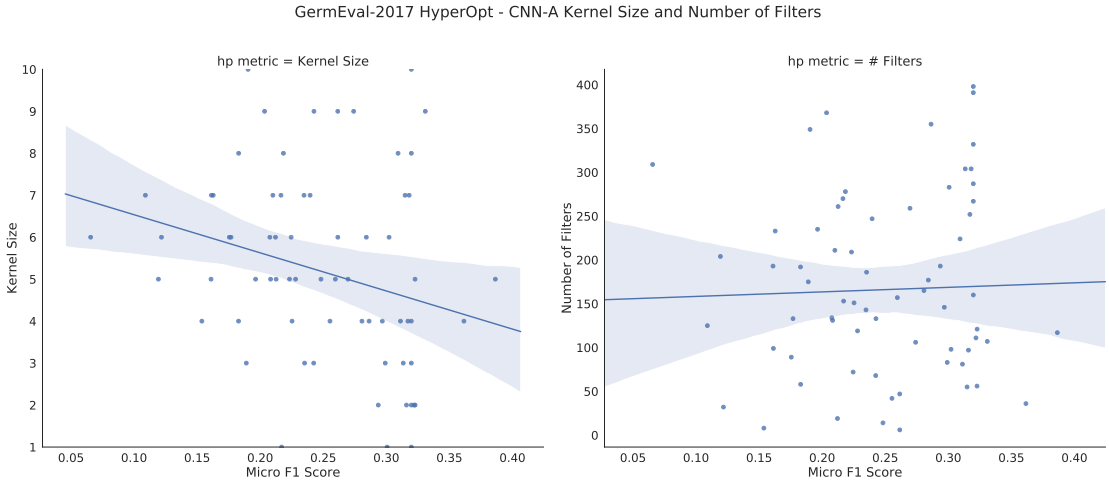


Figure 6.4: Impact of CNN parameters on micro F1-score. The graph on the left shows the impact of the kernel size on the model performance. The graph on the right depicts the influence of the number of filters on the F1-score.

Point-wise Feed-Forward Layer Size

In the original transformer model the inner Point-wise Feed Forward (PWFC) has a dimensionality of 1024 while the model size has a dimensionality of 512 [**Vaswani2017c**]. This is a 2x increase over the model size. Due to the availability of pretrained Glove or Fasttext embeddings our ABSA-T model only uses a model size of 300. Consequently, the inner Point-wise Feed Forward (PWFC) layer dimensionality should be around 600. However, layer sizes above 300-400 neurons quickly lead to interesting model behavior. After a few training iterations the PWFCs transform every input to the exact same output. In other words, no matter what the model gets as input it always predicts the same output.

The solution for this overfitting behavior is to use a smaller inner PWFC. Values from 100 to 200 neurons lead to the best results.
This is extremely interesting since it completely changes the task of the PWFCs. From a layer with a higher dimensionality than the model to a bottleneck layer with a lower dimensionality. A bigger layer may allow for more complex and expressive features to be learned while a smaller bottleneck layer limits the expressiveness and forces the network to focus on crucial features [**Ramsundar2015**].

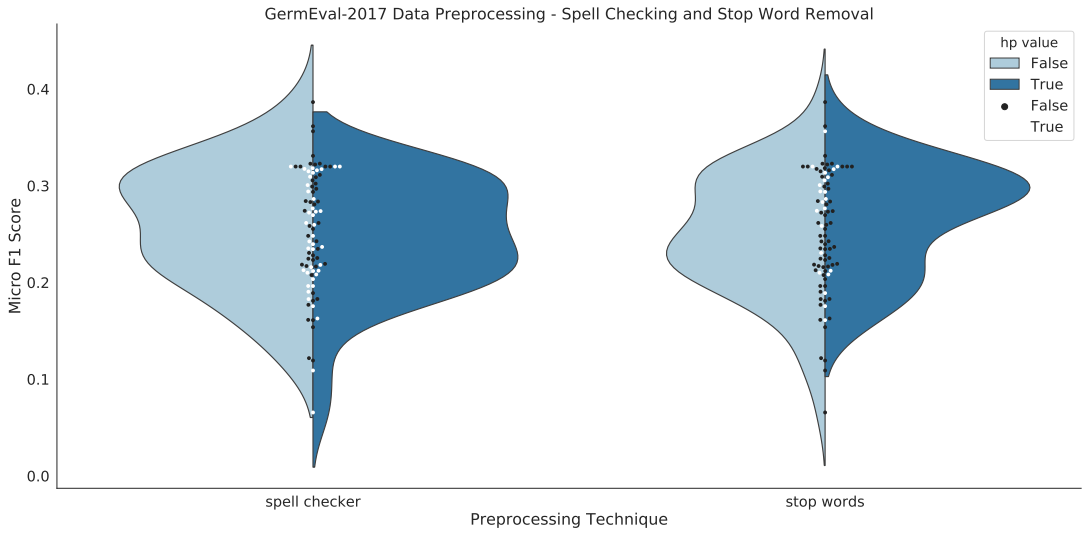


Figure 6.5: Comparison of preprocessing techniques - Impact of Spell checking and Stop Word Removal on Validation Micro F1-Score

6.1.3 Data Preprocessing

In the following section we discuss the impact of the preprocessing steps and how they affect the overall model performance.

Spell Checking

The left side of figure **??** shows the impact of spell checking on the GermEval-2017 dataset. In this instance, spell checking negatively impacted the performance of the classifier. There are a few explanations for this performance.

Social media content contains a lot of special characters and words which are not part of a regular dictionary. However, especially those words might carry the most sentiment. By replacing those words it is possible that a lot of information is lost.

Tweets and forums posts about travel contain a lot of special abbreviations that spell checkers do not recognize. Persons might be talking about the bad performance of the public transport operator Münchner Verkehrsgesellschaft (MVG) but the spell checker replaces 'MVG' with 'mag' which changes the sentence dramatically.

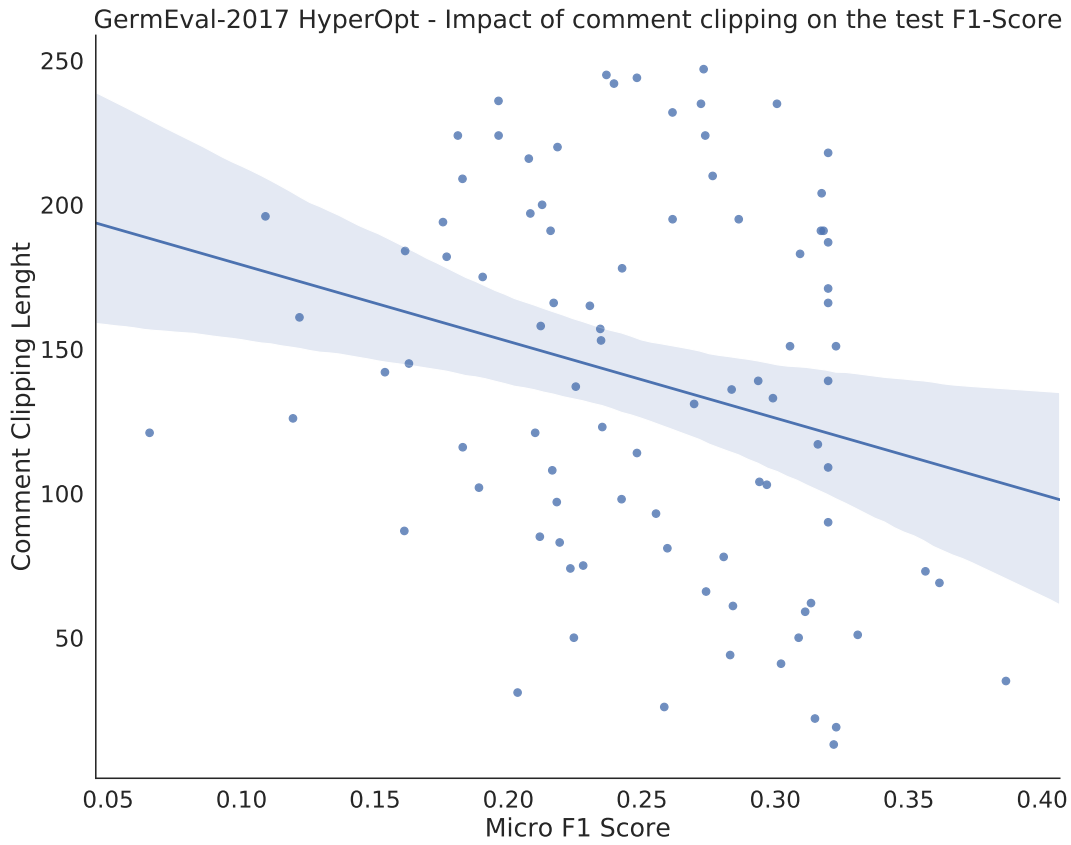


Figure 6.6: Comparison of preprocessing techniques - Impact of Comment Clipping on Test Micro F1-Score

Stop Word Removal

Stop words are a group of words which are very common in a language but carry little actual information. Examples for stop words are 'the', 'is' or 'what'. The results for stop word removal is shown in figure **??** on the right side for GermEval-2017. In this specific instance, removing them improves performance. However, this was not as clear for the organic dataset where removing stop words did not significantly improve the performance.

Comment Clipping

Comment clipping refers to the technique of cutting a sentence or comment after a specific number of tokens. In other words, comments which are too long are shortened and comments which are too short are padded.
Figure **??** provides a visualization how comment clipping impacts the model performance on the GermEval-2017 dataset. The regression line shows that a shorter sequence length may improve performance of the overall model[3].

This observation seems to be reasonable since GermEval-2017 contains a mix of very short tweets (140/280 character limit) as well as long newspaper articles. Aligning them to an overall shorter length logical especially considering that most longer newspaper articles include a short summary in the beginning. Cutting those long documents helps the transformer to focus on important information instead of spreading out the attention.

6.2 Results for Named Entity Recognition

The following section contains the final results for the Named-entity recognition (NER) task of the CoNLL-2003 dataset.
Since this task was an auxiliary training task to assess the performance of the transformer part, no hyperparameter tuning was performed.

For this dataset the architecture is slightly different since CoNLL-2003 has annotations for each word. Since the transformer makes predictions per word there is no need for separate aspect heads.
Therefore this architecture follows the original transformer and just consists of a single linear layer to project the 300-dimensional per-word prediction down to the amount of classes to predict. Finally, a log-softmax is used to provide the log probabilities for the class labels.

6.3 Results for Aspect-Based Sentiment Analysis

The following sections outline the results on Aspect Based Sentiment Analysis (ABSA). The first section contains the results for the GermEval-2017 dataset. This dataset uses a special evaluation method. To be able to compare our results we also use the evaluation method that GermEval-2017 provides.
The method GermEval-2017 uses is described in section **??**.

[3]Significant at a p-value of 0.05

Section **??** reports results on the Organic-2019 dataset and section **??** discusses the results on the Amazon Reviews dataset.
Finally, section **??** and **??** review the results of multitask learning and transfer learning.

6.3.1 GermEval-2017

6.3.2 Organic-2019

6.3.3 Amazon Product Reviews

Due to the large size of the dataset no hyperparameter tuning was performed. Parameters from previous optimizations on the smaller organic dataset were chosen for the evaluation. The final model uses the MLS-A architecture with fasttext embeddings. Comments are clipped to a fixed length of 100 and spell checking as well as stop word removal is enabled to reduce the vocabulary size. The model consists of one attention head with d_k and d_v of 300 and two encoder blocks. The inner PWFC-layer acts as a bottleneck with a size of 128. Finally, a reduced batch size of 12 was chosen to be able to fit the model into GPU memory which would not have been otherwise possible.

The vocabulary size of the amazon dataset after all reductions consists of 389,371 unique tokens. This creates an embedding layer which maps the vocabulary size of 389,371 to a 300-dimensional vector. Unfortunately, a lot of those tokens are not part of the pretrained embedding so the parameters of the embedding layer can not be locked during training. Therefore, the embedding layer has a size of 116,811,300 trainable parameters. As a result, the first embedding layer makes up over 99% of the overall number of model parameters which is 117,710,780.

6.4 Impact of Multitask Learning

Difference to Multitask learning

6.5 Impact of Transfer Learning

7 Conclusion

7.1 Future Work

Unsupervised clustering of samples to aspects was briefly evaluated but this task might not Multitask

A Appendix

A.1 Datasets

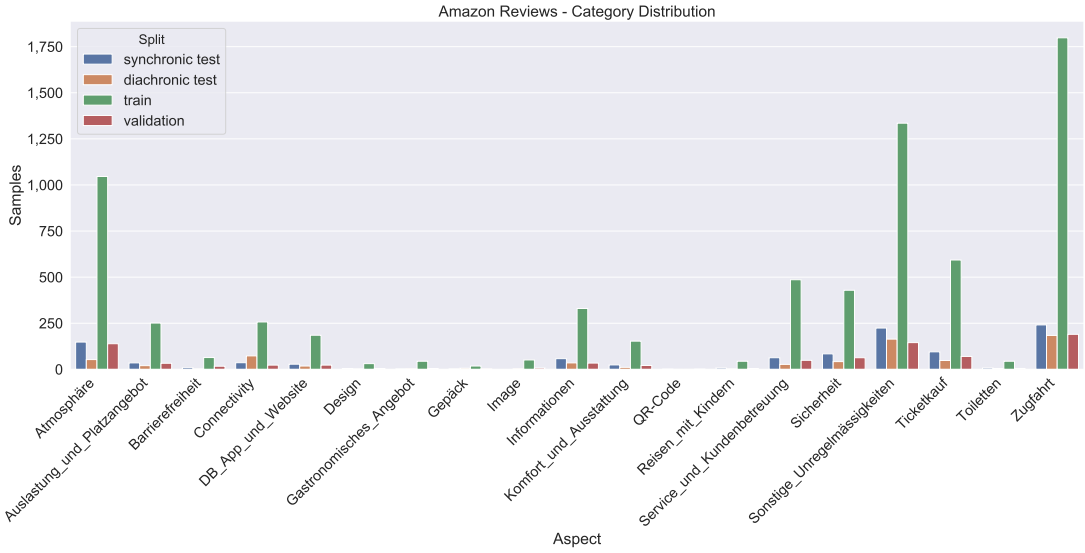


Figure A.1: Statistics on GermEval-2017 aspects

A.2 Optimization

OLS regression for Validation Loss / HyperOpt Iterations

Dep. Variable:	y	**R-squared:**	0.009
Model:	OLS	**Adj. R-squared:**	-0.002
Method:	Least Squares	**F-statistic:**	0.8307
Date:	Mi, 24 Apr 2019	**Prob (F-statistic):**	0.365
Time:	15:32:34	**Log-Likelihood:**	-51.223
No. Observations:	90	**AIC:**	106.4
Df Residuals:	88	**BIC:**	111.4
Df Model:	1		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	0.3598	0.090	3.980	0.000	0.180	0.539
x1	-0.0016	0.002	-0.911	0.365	-0.005	0.002

Omnibus:	163.475	**Durbin-Watson:**	2.062
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	11634.826
Skew:	6.940	**Prob(JB):**	0.00
Kurtosis:	56.944	**Cond. No.**	102.

Table A.1: OLS Regression statistics for a regression of HyperOpt TPE optimization losses on the validation set (Dataset: GermEval-2017)

OLS regression for Test Loss / HyperOpt Iterations

Dep. Variable:	y	**R-squared:**	0.006
Model:	OLS	**Adj. R-squared:**	-0.006
Method:	Least Squares	**F-statistic:**	0.4930
Date:	Mi, 24 Apr 2019	**Prob (F-statistic):**	0.484
Time:	16:22:21	**Log-Likelihood:**	-206.16
No. Observations:	90	**AIC:**	416.3
Df Residuals:	88	**BIC:**	421.3
Df Model:	1		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	0.8356	0.506	1.653	0.102	-0.169	1.840
x1	-0.0069	0.010	-0.702	0.484	-0.026	0.013

Omnibus:	191.171	**Durbin-Watson:**	2.030
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	25929.119
Skew:	9.027	**Prob(JB):**	0.00
Kurtosis:	84.169	**Cond. No.**	102.

Table A.2: OLS Regression statistics for a regression of HyperOpt TPE optimization losses on the test set (Dataset: GermEval-2017)

Variable	Type	Parameters
Batch Size	QUniform	Interval: [1, 100]
Comment Clipping	QUniform	Interval: [10, 250]
Replace URL Tokens	Bool	[True, False]
Use Stop Words	Bool	[True, False]
Use Spell Checker	Bool	[True, False]
Harmonize Bahn	Bool	[True, False]
Embedding Type	Choice	[Glove, Fasttext]
# Encoder Blocks	QUniform	Interval [1, 8]
# Attention Heads	Choice	[1, 2, 3, 4, 5]
PW Layer Size	QUniform	Interval: [32, 256]
TF Dropout	Uniform	Interval [0, 0.8]
Output Dropout	Uniform	Interval [0, 0.8]
Transformer Bias	Bool	[True, False]
LR Warmup	QUniform	Interval [1000, 9000]
LR Factor	Uniform	Interval [0.01, 4]
Adam $\beta 1$	Uniform	Interval [0.7, 0.999]
Adam $\beta 2$	Uniform	Interval [0.7, 0.999]
Adam EPS	LogUniform	log(1e-10), log(1)
LR	LogNormal	log(0.01, log(10)
Weight Decay	QUniform	$1e^{[-8,-3]}$
Output Layer	Choice	[CNN*, LinSum]
# CNN Filter*	QUniform	Interval [1, 400]
Kernel Size*	QUniform	Interval [1, 10]
Stride*	QUniform	Interval [1, 10]
Padding*	QUniform	Interval [0, 5]

Table A.3: Hyperparameter Search space for GermEval-2017. * marks parameters which are only sampled if CNN is chosen as the output layer.

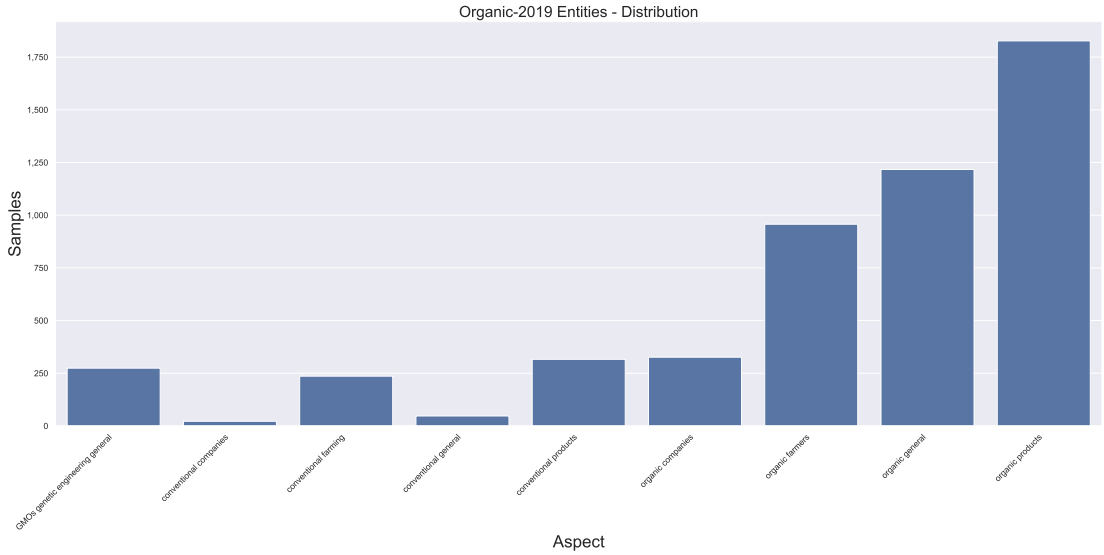


Figure A.2: Distribution of the aspect *entity* in the Organic-2019 dataset.

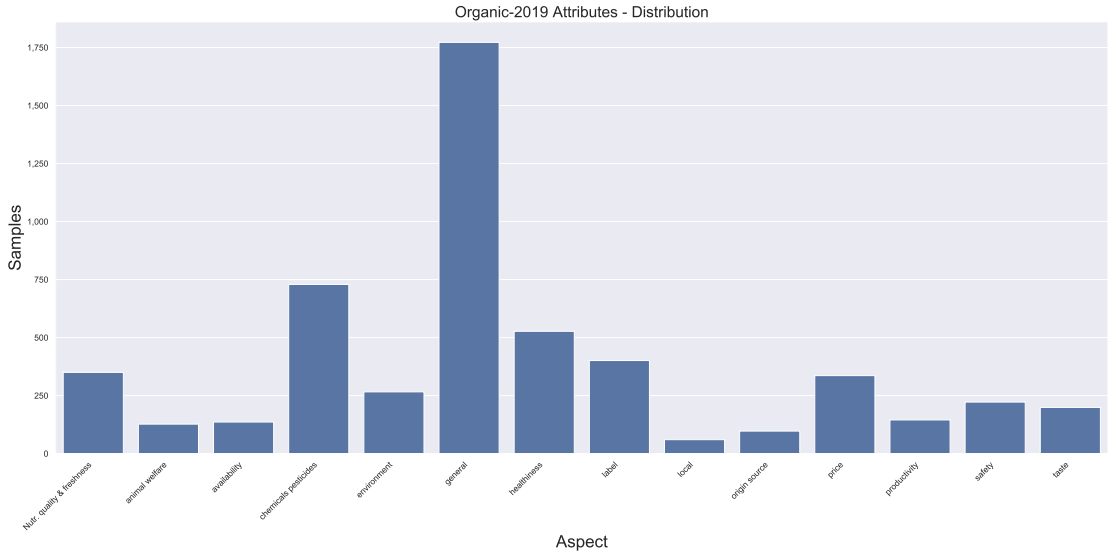


Figure A.3: Distribution of the aspect *attribute* in the Organic-2019 dataset.

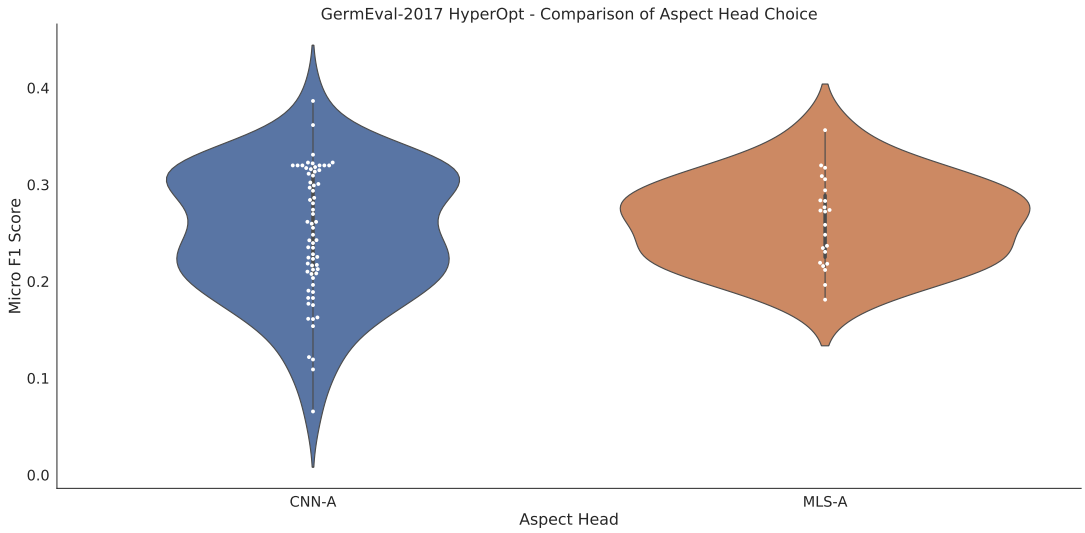


Figure A.4: Hyperopt - Comparison and impact of aspect head choices and sampling amount

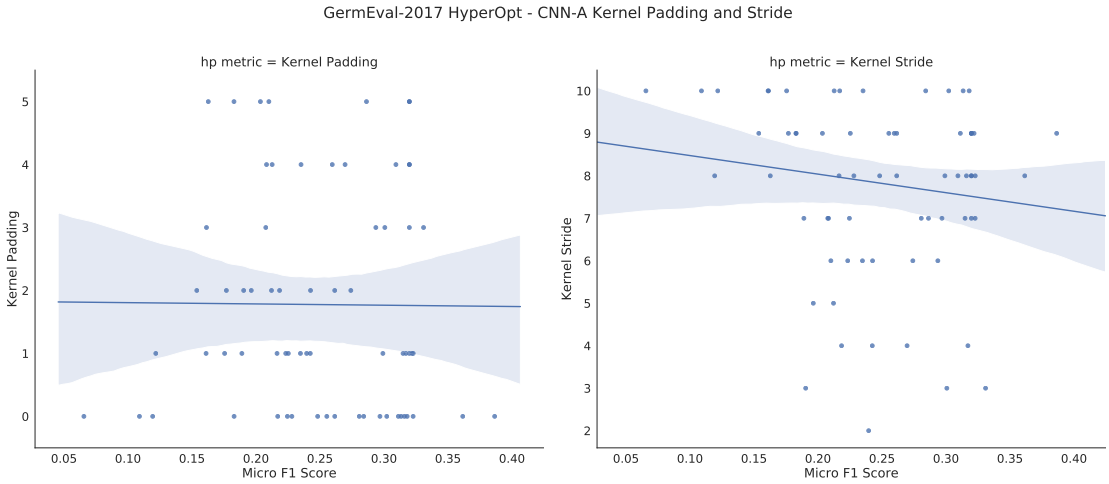


Figure A.5: Impact of CNN parameters on micro F1-score. The graph on the left shows the impact of the kernel padding on the model performance. The graph on the right depicts the influence of the kernel stride on the F1-score.

List of Figures

List of Tables