Universität Leipzig
Faculty of Mathematics and Computer Science
Degree Program Computer Science

Neural Netspeak – Exploring the Performance of Transformer Models as Idiomatic Writing Assistants

Bachelor's Thesis

Fabian Thies

1. Referee: Jun.-Prof. Dr. Martin Potthast

Submission date: December 2, 2020

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Leipzig, December 2, 2020

.....
Fabian Thies

Abstract

Since writing is a difficult task, authors often resort to sophisticated tools - writing assistants - to help them with spelling, grammar, style checks, or word choice. Query-based writing assistants offer the experienced writer an easy way to work with complex resources, like dictionaries, phrase-books, and idioms from a corpus, by providing a query language for context-sensitive search and a transparent ranking. Netspeak uses an index over the Google N-gram dataset to retrieve matching phrases ordered by frequency of occurrence in the dataset. However, the use of n-grams imposes a limit on Netspeak, requiring the queries not to exceed five words, limiting the context that can be captured in a single query. Also, the longer the queries are, the lower is the probability of finding matching n-grams. Here we show that using the Transformer-based BERT language model, trained on masked word prediction, we can circumvent these two limitations and, depending on the type of query, increase the number of answered queries dramatically. On 140,000 queries generated from four corpora of different domains, our language-model based result retrieveal strategy increases the number of answered queries by up to 81 % compared to Netspeak on the same queries. Given additional context through longer queries, we are able to answer up to 100 % queries more than Netspeak. However, we are not able to rank the expected result as high as Netspeak most cases, even with additional context. Our results demonstrate how certain capabilities of language models can independently be used even without further fine-tuning to improve certain aspects of a query answer retrieval process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since writing texts is a challenging task, authors often resort to sophisticated tools - writing assistants. There is a wide variety of different assistants available at the writer's fingertips to help them with spelling, grammar, style checks, or word choice. They are specialized in one or more of these tasks and provide appropriate interfaces with which people can interact most intuitively.

Query-based writing assistants offer the experienced writer an easy way to work with complex resources, like dictionaries, phrase-books, and idioms from a corpus, by providing a query language for context-sensitive search and a transparent ranking. Netspeak (Potthast et al. [2010]) is such a query-based assistant and lets users search the Google N-gram dataset using a custom query language. Its result retrieval works by searching a reverse index over the n-grams for possible matches, which are then shown sorted by the n-gram's occurrence frequency in the corpus in descending order. Netspeak thereby allows users to search for phrases commonly used on the web with their frequency as a measure. However, since the Google N-gram dataset only contains n-grams with $1 \leq n \leq 5$, the queries Netspeak can process are limited to a length of five words. Also, the longer the query, the less likely it is to find matching results because n-grams get sparse the larger the n is due to the number of different word combinations. This out-of-vocabulary problem, where no results can be retrieved, is further amplified for queries contains uncommon or phrases, or ones that are not contained in the n-gram dataset.

In this work, we propose **NeuralNetspeak**, a new result retrieval strategy based on an NLP language model to circumvent these two issues. We use a pre-trained version of the BERT language model (Devlin et al. [2018]) to process Netspeak queries and return a sorted list of results. This language model is based on the Transformer architecture (Vaswani et al. [2017]), which unlike previously dominant language model architectures, such as Recurrent Neural Networks (RNNs, Mikolov et al. [2010]) or Long-Short-Term-Memory

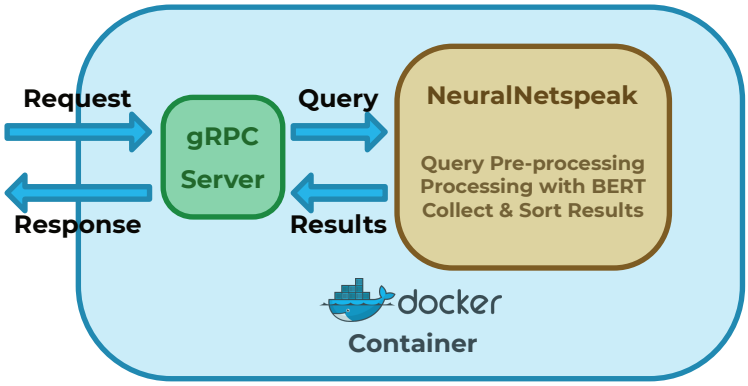


Figure 1.1: A macroscopic overview of the NeuralNetspeak software.

models (LSTMs, Peters et al. [2018]), process the whole input sequence in one step and allows for pre-training generic language models. The pre-trained language models have learned different features of natural language while being trained on vast amounts of texts. However, to take advantage of the model's capabilities, we have to pre-process the Netspeak queries before letting them run through the language model. Then, we use the model's ability to predict masked words and to calculate a score for the inputs to process the queries further and rank the resulting phrases by the calculated score. Finally, we show these phrases to the user in the same user-interface Netspeak uses.

To evaluate our approach and compare its results to Netspeak's n-gram based result retrieval, we generate Netspeak queries from four corpora of different domains. Then, we send these queries to both Netspeak and our new backend for the Netspeak web user-interface, which answers the same queries with the help of BERT as described above. Our results show, that using our approach we can lift the limitation of Netspeak queries to five words and obviate the out-of-vocabulary problem, answering up to two times the number of queries compared to Netspeak. However, NeuralNetspeak is not a complete replacement for the n-gram based result retrieval strategy of Netspeak yet.

Figure 1.1 shows an overview of the NeuralNetspeak software product we built and deployed to evaluate our approach. It is enclosed in a docker container, which can easily be deployed on any server or computer, accepting gRPC requests following the same specification as Netspeak. This way, it can be used as a drop-in replacement or alternative for the current Netspeak backend with very few changes to the existing web interface. A working version of NeuralNetspeak can be found at `https://netspeak.org/demo/`.

Chapter 2

Theoretical Foundation

2.1 Transformer Model

The Transformer architecture, as proposed by Vaswani et al. in 2017, marked a significant leap forward in natural language processing (NLP).

Similar to other sequence to sequence language models, such as ELMo (Peters et al. [2018]) or OpenAI's GPT (Radford [2018]), the Transformer has an encoder-decoder structure. The encoder takes the input sequence and produces a fixed-length representation, which is then fed into the decoder, where the output sequence is constructed from that representation. The core of the encoder and decoder is a stack of attention layers, which allow the model to individually weigh the influence of different words in the input sequence have when processing a word from the sequence. Although the concept of attention is not new to this architecture (Bahdanau et al. [2016]), the Transformer is the first language model that only relies on the attention mechanism. Previous language models included recurrent components such as RNNs (Mikolov et al. [2010]) or later LSTMs (Peters et al. [2018]). While models based on LSTMs can achieve a similar effect as attention, it only works on words the model has previously already seen left to the word being processed. A solution to this is the introduction of language models using two separate LSTMs, one processing the input forward while the other processes it backward. However, they still have to process the input sequentially word by word because of the recurrent nature of LSTMs. Transformers, on the other hand, can process the entire input at once, with the attention layers attending to the words of the input sequence based on the learned weights. Since the attention does not rely on the results of the previous or next words, the attention calculations can be done at once using matrix operations. This significantly improves the degree of parallelization compared to recurrent architectures and makes it possible to train a state of the art translation model in twelve hours on eight P100 GPUs.

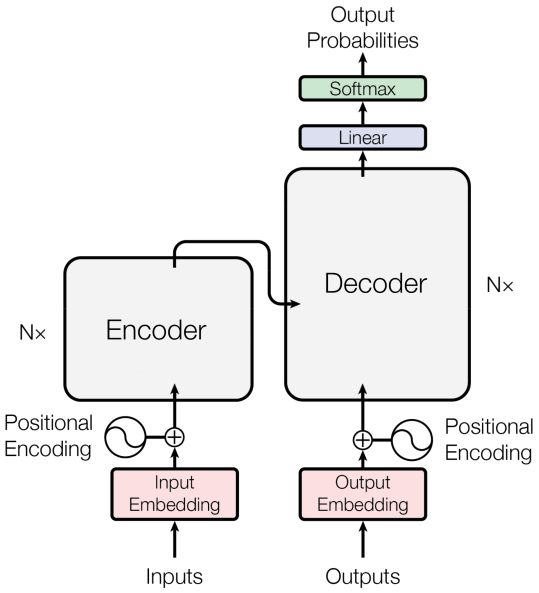


Figure 2.1: A simplified overview of the Transformer model architecture, derived from the complete illustration of the architecture from the paper by Vaswani et al. [2017].

2.1.1 Model Architecture Overview

Figure 2.1 shows a simplified overview of the Transformer model architecture, with the encoder (left) and the decoder (right). The encoder is a mapping from the input symbol sequence $(x_1, ..., x_n)$ to a fixed-length, continuos sequence of internal representations $z = (z_1, ..., z_n)$. By analogy, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols from z, one element at a time.

Both the encoder and the decoder consist of multiple layers stacked on top of each other. Before an input (output) enters the encoder (decoder), the model generates an embedding of the text sequence into tokens of the model's vocabulary and adds an encoding to the embedding, which the model then can use to infer the relative position of the words in the input sequence. After the model has processed the input, it converts the resulting token sequence back to a sequence of words and assigns a probability to each word of the output.

2.1.2 Attention

In the realm of the transformer, attention provides the answer to the question, on which part of the input it should focus most while encoding or decoding a given piece of the input. It allows the transformer to look at other parts of the

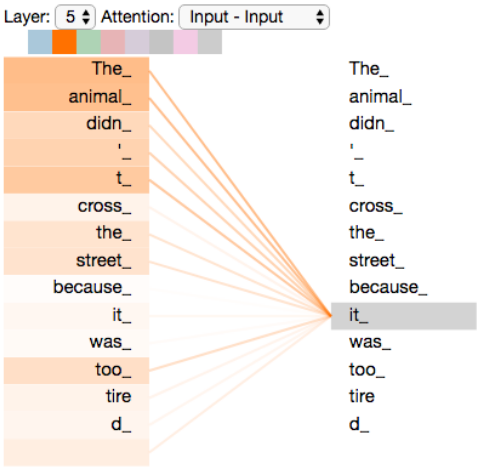


Figure 2.2: A visualization of the self-attention mechanism on an example sentence (Alammar [2018]). When processing the word "the animal", the focus of the attention mechanism is most strongly on the word "it". This information will be encoded in the output of the attention layer.

input sequence to improve the encoding of a given word. The self-attention, therefore, is a similarity measure, describing how similar or relevant another part of the input is in terms of context, and captures contextual relationships between words in the input sequence. A visualization of this behavior can be seen in Figure 2.2.

The self-attention function takes an input vector, calculates a set of vectors (composed query, key, and value vector), and returns an output vector. This multi-step process can be broken down into six parts (Alammar [2018]).

First, three new vectors are calculated by multiplying the input with trained weight-matrices, as shown in Figure 2.3. These new vectors are a query vector q, a key vector k, and a value vector v. They are calculated for each input vector x by multiplying it with W^Q, W^K, and W^V for the query, key, and value vector respectively. These three result vectors are smaller in dimension than the input vector, with the query and key vector having the same dimensionality d_k and the value vector being of dimension d_v. The difference in dimensionality is not necessary but is a conscious decision to make the calculations for the multi-headed attention easier.

Then, a score gets calculated for each word in the input sequence for the given input x. The higher the score, the more focus should be placed on that part of the input sequence while encoding x. The score is calculated by taking the dot product of the query vector q with the key vector k of the respective word the score should be calculated for. In summary, given a word at index i,

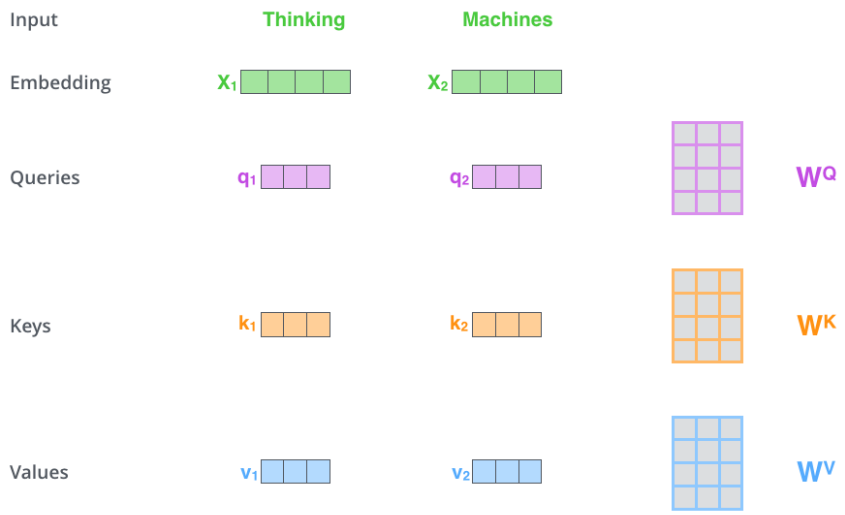


Figure 2.3: A vector and matrix-based visualization of the self-attention used in the Transformer. The query, key, and value vectors are created for each of the two vectors x_1 and x_2, which are derived from the two input words after the embedding, by multiplying the input vectors with the corresponding weight matrices W^Q, W^K and W^V. Source: Alammar [2018]

the score for the word in position j would be calculated as

$$\text{score}_i(j) = q_i \cdot k_j.$$

Once the score is calculated, it gets divided by $\sqrt{d_k}$ (the square root of the key/query vector's dimensionality) to reduce the differences between the individual scores, resulting in more stable gradients. The result is then passed through a softmax operation, normalizing the results so they are positive and add up to 1. The score of a word for itself is higher than for all other words in the input sequence, as it is most similar to itself, without adding any information.

The normalized score is then used to multiply each value vector by it. This will bring the components of the value-vectors representing words, which are irrelevant for the considered position, close to zero, and give more weight to those vectors, whose words are of greater relevance.

Finally, the sum of the weighted value vectors is calculated, which is the output z_i of the self-attention layer at the position i. This whole process is illustrated in Figure 2.6 for a short example sentence.

Since the operations to calculate the score for each position are the same, they can be calculated simultaneously by packing the query vectors into a matrix

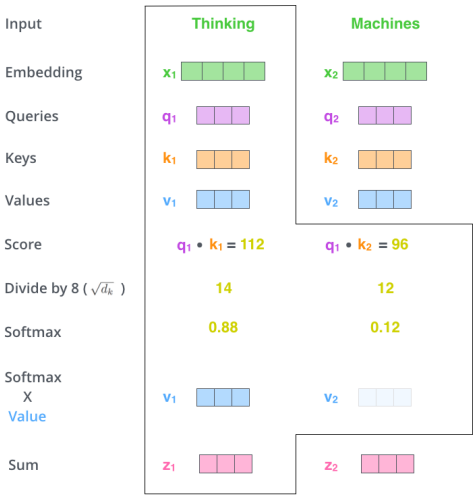


Figure 2.4: A breakdown of the steps required to calculate the self-attention vector for a single word of an example sentence. Source: Alammar [2018]

Q, and the key and value vectors into a K and V matrix respectively. Now dealing with matrices only, all the steps described above can be condensed into one single function to calculate the outputs Z of the attention layer:

$$\text{Attention}(Q, V, K) = \text{softmax}(\frac{Q \times K^T}{\sqrt{d_k}}) \cdot V = Z$$

Multi-Headed Self-Attention As mentioned above, the self-attention scoring function will assign the highest score to the word itself. However, this high score doesn't add any useful information but rather outweighs scores indicating a strong relationship to other words. To counteract this, the queries, keys, and values are projected linearly on h times with different, learned linear projections on the dimensions d_k, d_k, and d_v respectively. The resulting projections are h different sets of W^Q, W^K, and W^V matrices. These projections give the attention layer multiple "representation subspaces" to encode the attention differently, but also result in having h output matrices Z_1 to Z_h, which are called attention heads. Figure 2.5 shows the positions of the input sequence the different color-coded attention heads attend to when processing the word "it". The different words the individual attention heads attend to show their specialization on detecting different learned features in the input sequence.

To get a single output matrix as expected by the feed-forward layer, the output matrices are first concatenated and then multiplied by yet another trained weight-matrix W^O.

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

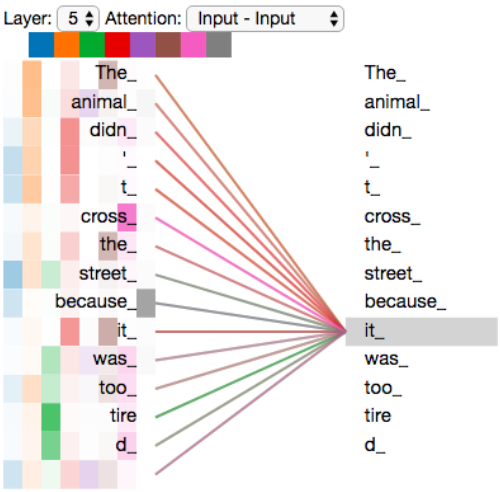


Figure 2.5: A visualization of the different attention heads that attend to different positions of the input sequence when processing the word "it". Source: Alammar [2018]

where $\text{head}_i = \text{Attention}(Q \cdot W_i^Q, K \cdot W_i^K, V \cdot W_i^V)$. The resulting output matrix Z contains information from all the attention heads.

Feed-Forward Network The output generated by the multi-headed attention layer gets passed into a fully-connected feed-forward neural network. This network is applied identically and independently from each other to the attention vectors of each word of the input sequence, allowing for parallel computation.

Add and Norm Layers Each self-attention and feed-forward sublayer has a residual connection around it and is followed by a layer-normalization step, where the output gets normalized by taking into account the sublayer's input vector.

2.1.3 Encoder and Decoder

Figure 2.7 shows the whole Transformer model architecture, with the encoder (left) and the decoder (right) with all its sublayers.

The encoder is a stack of $N = 6$ identical layers, which consist of two sublayers each: the multi-head self-attention layer and a simple fully-connected feed-forward layer. All outputs of these sub-layers are normalized (Ba et al. [2016]) before they are handed off to the next layer. Since these operations work on vectors, all outputs of the sub-layers and embedding-layer are of dimension

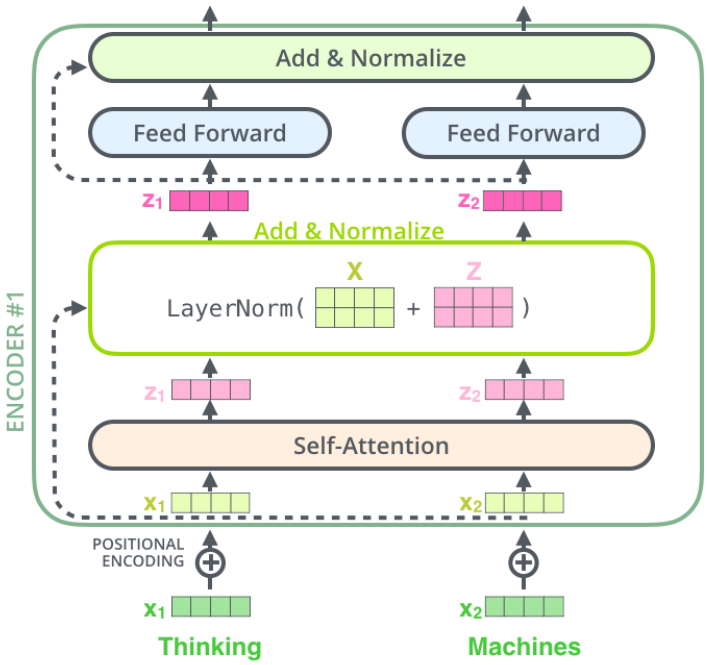


Figure 2.6: A visualization of a Transformer encoder with its sublayers and intermediate vectors. The LayerNorm operation found in the "Add & Norm" sublayers is shown in detail. Source: Alammar [2018]

$d_{model} = 512$. In contrast to other language models, which read the input sequentially, the Transformer encoder reads the entire input sequence of words at once.

Similar to the encoder stack, the decoder consists of $N = 6$ identical layers as well. In addition to the two sub-layers of the encoder layers, the encoder adds another multi-head self-attention layer to the beginning of each layer. In this additional layer, masking is introduced, so that a given position can't attend to any subsequent positions, making the predictions for position i only depended on the known outputs for the previous positions.

Figure 2.8 shows one of the encoders in the encoder stack with its self-attention layer and feed-forward neural networks. In this example, the encoder receives the two input vectors (x_1 and x_2), which are first passed into the self-attention layer, then into the individual but same feed-forward neural networks. The output (r_1 and r_2) of these layers is then sent to the next encoder in the encoder stack as input.

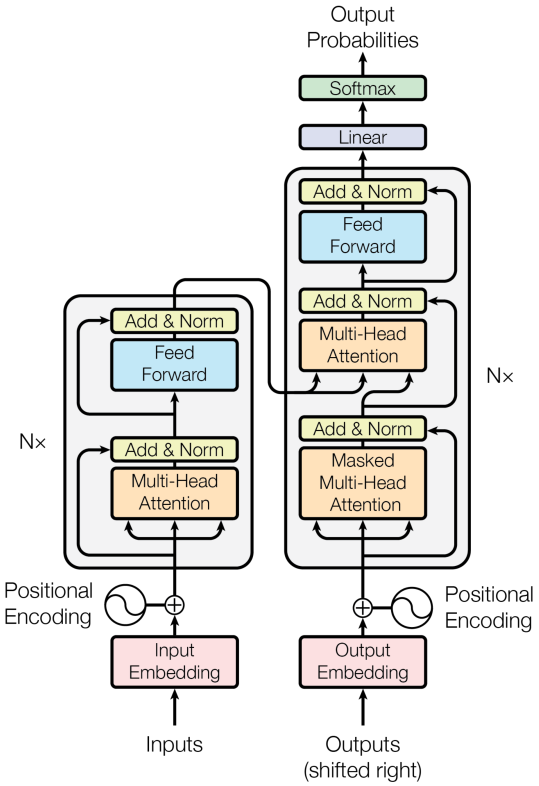


Figure 2.7: The Transformer model architecture. Source: Vaswani et al. [2017].

2.1.4 Embedding and Positional Encoding

To process the input or output sequence tokens, they get mapped to vectors of dimension d_{model}. Only using the embedding as is would result in losing essential information, since the same word can have a different meaning depending on its position in the sentence. Therefore, a positional encoding vector is added to the embedded input vector. This allows the model to retain information about each word's position in the input sequence and the relative distance between words.

The positional encoding vector is built with the following two sine and cosine functions:

$$\mathrm{PE}_{(\mathrm{pos},\ 2i)} = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right) \text{ and}$$
$$\mathrm{PE}_{(\mathrm{pos},\ 2i+1)} = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right),$$

where *pos* is the position of the token in the sequence and i is the dimension of the vector component. The resulting vector has values of alternating sines and cosines of increasing frequency in its components of ascending dimension,

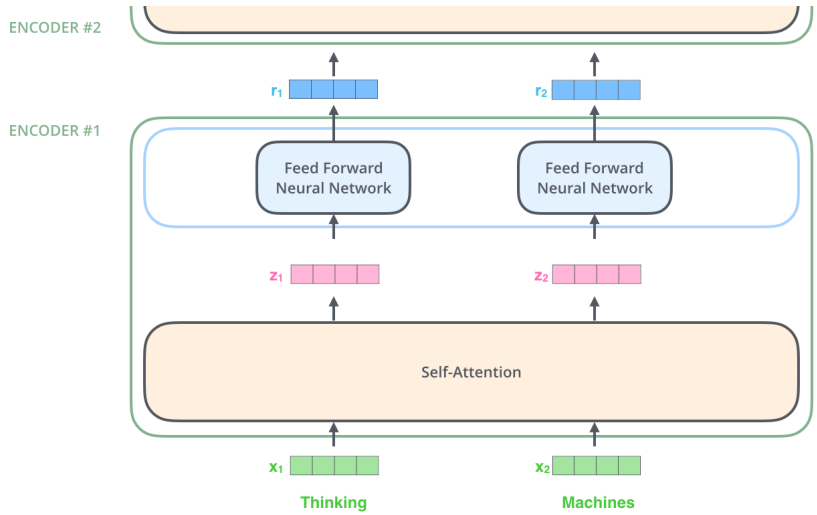


Figure 2.8: The structure of an encoder layer with its sublayers from the encoder stack of a Transformer, showing the flow of two example input vectors x_1 and x_2 through the encoder. Source: Alammar [2018]

which allows the model to infer the relative position of two words of the input. A visualization of the encodings for 20 tokens are shown in Figure 2.9. Each token (from top to bottom) has a similar structure as the one above but is pushed (left) into greater embedding dimensions.

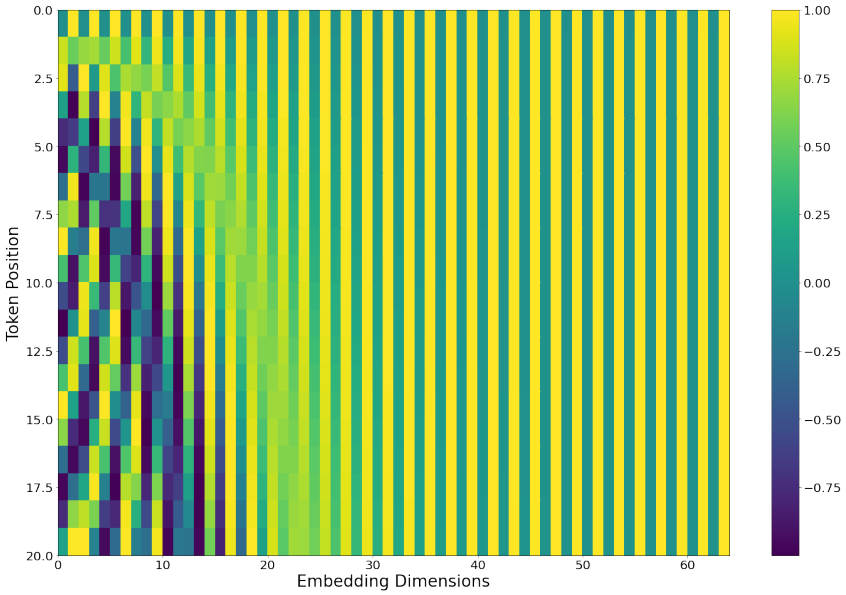


Figure 2.9: A visualization of the position encoding for 20 tokens with a dimension of 64 with the sine and cosine functions presented by Vaswani et al. [2017].

2.2 BERT

In 2018 Google open-sourced their BERT model, an NLP language model focussed on pre-training using plain text from Wikipedia articles and built upon the Transformer (see section 2.1) architecture. The name BERT is an initialism for Bidirectional Encoder Representations from Transformers. Taking advantage of these pre-trained models, one can easily fine-tune existing models to fit their specific task's needs with very little data and computational resources[1]. The models originally provided by Google Brain themselves, $\text{BERT}_{\text{BASE}}$ and $\text{BERT}_{\text{LARGE}}$, were trained on the BooksCorpus and English Wikipedia articles respectively on both masked-word and next-sentence prediction tasks.

BERT leaves the underlying Transformer structure almost completely unchanged from the original structure described in section 2.1. What distinguishes the two models, are the number of layers in the encoder and decoder stack, the hidden size, and the number of self-attention heads. $\text{BERT}_{\text{BASE}}$ uses half the layers in the encoder and decoder stack and half the hidden size compared to $\text{BERT}_{\text{LARGE}}$, while also using one quarter fewer attention heads than the large model, to match the OpenAI GPT language model (Radford [2018]).

In contrast to all previous Transformer-based language models, including the OpenAI GPT model, "BERT is the first deeply bidirectional, unsupervised language model, pre-trained only using a plain text corpus" (Devlin et al. [2018]). Pre-trained language models can either be context-free or contextual. Context-free models generate a single word embedding for each word in the vocabulary, independently from the surrounding words. As an example, the word "bank" has the same embedding in "bank account" as in "bank of the river", while the meaning of this one word in both phrases is completely different. Contextual models use context clues from surrounding words to generate an embedding for each word. They can be divided into two categories: unidirectional and bidirectional language models, as seen in Figure 2.10. Unidirectional models, like the OpenAI GPT, can only use the words either left or right of the word when generating an embedding for it, while bidirectional models, like BERT, can use all words of the input sequence to generate the embedding. Similar to the example above, unidirectional models represent the word "bank" in "I accessed the bank account" based on "I accessed the" but not "account". Bidirectional models can also use the word "account" to generate an embedding for the word "bank". There have also been efforts to achieve the same using a concatenation of two independent unidirectional LSMTs, as e.g. ELMo (Peters et al. [2018]), but their performance has been exceeded by

[1] "[...] about 30 minutes on a single Cloud TPU, or in a few hours using a single GPU" (https://ai.googleblog.com/2018/11/open-sourcing-bert-state-of-art-pre.html)

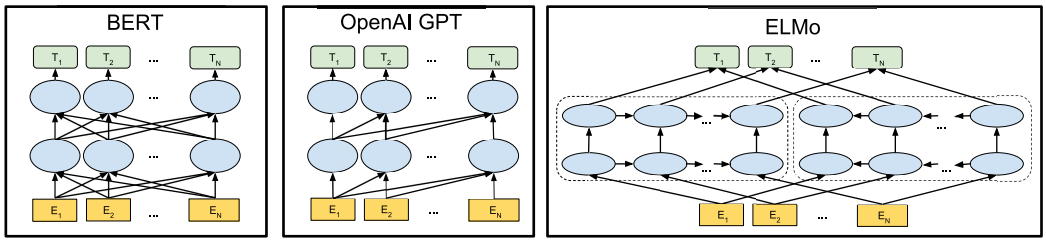


Figure 2.10: Differences between bidirectional (BERT, left) and unidirectional (OpenAI GPT, middle) models, and a language model concatenating two independently trained unidirectional LSTMs, one left-to-right and one right-to-left (ELMo, right).

BERT.

BERT is the first successful bidirectional language model. Bidirectionality allows the model to indirectly see the word it should predict through the previous prediction steps, as the model can see not only the words left of the word to predict but also the words to the right, including the word that should be predicted in the next prediction step. BERT's solution to this issue is to mask some words in the input, then condition each word to predict the masked word. This idea has been around since the 1950s under the name of the Cloze task (Taylor [1953]), but it was first applied successfully in BERT.

2.2.1 Pre-Training

The BERT models were pre-trained on two different tasks, one requiring only one sentence as input and the other requiring two sentences concatenated to one input sequence. BERT uses WordPiece embeddings (Wu et al. [2016]) rather than whole word embeddings, to improve the handling of rare words. Using WordPieces, a rare word can be disassembled into multiple common subword units consisting of multiple characters. As an example, the word "embeddings" is split up into the 4 WordPiece tokens "em" "##bed" "##ding" "##s". WordPieces that should be concatenated to the previous WordPiece are prefixed with two hash symbols. This way, the vocabulary can be dramatically reduced compared to whole-word embeddings. BERT uses a vocabulary of 30,000 tokens. Each input sequence starts with a `[CLS]` token (classification token) and ends with a `[SEP]` token (separator token). The separator token is also used to separate two sentences from each other in the input sequence. Also, there are two special tokens more in BERT's vocabulary: (1) the `[MASK]` which is used for masking a specific position, and (2) the `[UNK]` token which is used when a word of the input sequence can't be tokenized with any of the WordPieces in the vocabulary.

Let s_1 and s_2 be sentences from which a input sequence should be created, which $s_{1,1} \dots s_{1,m}$ and $s_{2,1} \dots s_{2,n}$ being the tokenized WordPieces of s_1 and s_2 respectively. An input embedding only containing s_1 would be

$$x = (\texttt{[CLS]}, s_{1,1}, \dots, s_{1,m}, \texttt{[SEP]}).$$

Similarly, when both sentences should be packed into one input sequence, that sequence would be

$$x = (\texttt{[CLS]}, s_{1,1}, \dots, s_{1,m}, \texttt{[SEP]}, s_{2,1}, \dots, s_{2,n}, \texttt{[SEP]}).$$

While prior language models only transferred sentence embeddings to downstream tasks, the pre-trained BERT models expose all internal parameters for fine-tuning.

Masked Word Prediction

The first task used for pretraining the BERT models is masked word prediction. Preparing the dataset of unlabeled sentences, 15 % of WordPiece tokens of each input sequence are selected for prediction. These tokens are replaced with a mask token 80 % of the time, a random token 10 % of the time, or left unchanged 10 % of the time. The input sequence is then passed to the BERT model, which tries to predict the original tokens at the masked positions, based on the context given by the other non-masked words in the input sequence. After the input went through the encoder, the linear vocabulary embedding layer, and the softmax layer, each position of the output has scores for each word of the vocabulary assigned to it. For the next word prediction, only the positions of the mask tokens are considered, and the word with the highest score at this position is the word that fits best according to BERT.

Next Sentence Prediction

The other task the BERT models were pre-trained on is next sentence prediction. The training data for this task consists of pairs of sentences. Half of these pairs of sentences are two subsequent sentences from the original document, for the other half of sentence pairs a random sentence from the corpus is chosen as the second sentence. The assumption here is that random sentences are contextually disconnected from each other. Receiving these pairs of sentences as the input, the model has to predict if the second sentence in the pair is the subsequent sentence in the original document.

Additional information is added to the sentence pairs is added before training to help the model to distinguish between the two sentences. First, a `[CLS]` token is inserted at the beginning of the input sequence, and a `[SEP]` token

is inserted at the end of each sentence. Then, a sentence embedding is constructed to assign each token of the input to one of the sentences. Sentence embeddings are similar in concept to token embeddings with a vocabulary of 2. As the last step, a positional encoding is applied to each token to indicate its position in the sequence using the mapping described in section 2.1.4.

When predicting if the second sentence of a given input could be the sentence subsequent to the first one, the entire input sequence is passed through the transformer. Then, the output of the classification token is transformed into a 2x1 shaped vector, from which the probability of the second sentence being contextually connected to the first sentence is calculated.

2.2.2 Fine-tuning

Due to the underlying attention mechanism from the Transformer, a pre-trained language model can easily be fine-tuned to accomplish downstream tasks involving single word sequences or a pair of word sequences. Given a general pre-trained BERT model and task-specific training data, all that needs to be done to fine-tune the model to that specific task is to feed the inputs and outputs from the training data to BERT and fine-tune all parameters at once. The outputs of the BERT model are fed into an output layer for token-level tasks, while the output of the `[CLS]` token is used for classification, e.g. sentiment analysis or entailment, as input for another output layer. For the results presented in the paper, BERT was fine-tuned on four different tasks (GLUE[2] benchmark, SQuAD[3] v1.1 and SQuAD v2.0 question answering, as well as choosing the best from four sentence continuations from the SWAG[4] dataset). However, fine-tuning a BERT model is not part of this work, since we want to evaluate the performance of the BERT model itself for answering Netspeak queries.

[2]General Language Understanding Evaluation
[3]SQuAD is an acronym for the Stanford Question Answering Dataset
[4]Situations With Adversarial Generations

Chapter 3

Related Work

3.1 Writing Assistants

There is a wide variety of software solutions at an author's disposal that assist them in improving the quality of their texts. Some of them oversee the written text directly, others assist the writer in choosing the right words or phrases to convey the intended meaning. The resulting tasks and types of writing assistants can be organized in an ontology of increasing complexity, expanding on Potthast et al. [2010]:

1. Spell checkers
2. Next-word suggestion
3. Word choosing assistants
4. Grammar checkers
5. Style checkers
6. Discourse organizer and text structuring assistants

Using the least amount of context, *spell checkers* can further be divided into (1) spelling checkers themselves and (2) spelling correctors (Peterson [1980]). While the former can only be used to detect spelling mistakes, the latter can also show the user correction options or, as is common with many smartphones today, automatically replace misspelled words. However, both do not require any context beyond the checked word itself and are expected to provide results in real-time. There are many different approaches on how to detect misspelled words, e.g. using a dictionary for searching each word using tree traversal and informing the user of any words that don't match. Also, more sophisticated

solutions have been developed in recent years, e.g. by training a model Besides being used as a writing assistant, spell checkers can also be used to correct OCR[1] errors (Zhuang et al. [2004]) or search engine queries (Martins and Silva [2004]) and are included in popular text processing programs like Microsoft Word®.

Using the previous few words as context, writing assistants for *next-word suggestion* present the user with an ordered list of fitting words. Due to the minimal context constraints, this task can be achieved using an indexed list of n-grams, which are searched for the past few words. The next words of the best matches are then presented to the user. With the rise of neural network-based language models in recent years, they have also been used for next-word suggestion tasks.

Word choosing assistants suggest synonyms and other words to the writer that might fit in the context better. To show relevant alternative words, those assistants require at least the surrounding words as context. An example of word choosing assistants is the search-engine Netspeak Potthast et al. [2010]. It uses an index of n-grams searched using a custom query language that can be used to search for synonyms, whole words, and more to present the user with a list of results sorted by frequency of occurrence in the Google N-gram dataset.

Requiring even more context, some writing assistants help the writer to correct grammatical errors. These *grammar checkers* require parts of the sentence or even the whole sentence as context to check for grammatical errors, as certain words can influence the grammatical structure of distant parts of the same sentence. In general, there are three approaches to checking a given word order for grammatical errors. First, there is syntax-based checking (Jensen et al. [1993]), which parses the sequence into a tree structure. If this parsing fails, the sequence is considered to have grammatical errors. Then there is statistics-based checking (Atwell and Elliott [1987]). In this approach, a corpus representative for a given language is POS-tagged and using these tags, a list of POS tag sequences with their frequency is built. When checking a sentence, the less frequent a sequence of POS tags within that sentence is, the more likely it contains an error. However, the most common grammar checkers nowadays, such as LanguageTool[2], use rule-based grammar checking. Using a set of pre-defined rules, a given sequence of POS tags is matched against each rule. If one match fails, the sequence is considered grammatically incorrect. Since these set of rules can be quite extensive, even for a rather simple language like English (Quirk et al. [1985]), there have been efforts to automatically derive rules from vast amounts of texts using machine learning, such as

[1] Optical Character Recognition

[2] `https://languagetool.org`

Miłkowski [2012].

Requiring a deep understanding of the language for different idioms, *style checkers* aid the writer to use the correct tone of voice when addressing a specific audience. Therefore, they need to oversee the whole text, spanning across multiple sentences. One popular example of style checkers is the service Grammarly[3]. It not only incorporates a style checker but can also check for grammatical errors, provide alternative words, and correct misspellings. Also, Grammarly is related to this work as it is using Transformer models for grammar checking (Alikaniotis and Raheja [2019]), improving upon earlier rule-based approaches.

Lastly, there are the classes of *discourse organizer* and *text structuring assistants*.

The former can be used to help a moderator to organize discussions, such as reviewing discourses on Wikipedia articles, and can prevent spam from interrupting the discussion. To accomplish this task, they have to have a broader view of the whole discussion. One example that is widely used by well-known companies is Discourse[4].

The latter is more of a hypothetical type of writing assistant. It has to understand the whole text of a long document completely to be able to identify different parts of the text and help the writer improve the overall structure of the text. Since this is a very comprehensive task requiring not only to understand the context of each word in each sentence, but also the greater context of the sentences within the document, there is no example for this type of assistant at the time of writing.

In conclusion, there are many different kinds of writing assistants available, and some of them have switched from more traditional algorithms and metrics for detecting linguistic mistakes to using neural network-based approaches, as we do in this thesis. In the field of search engines assisting in word choice, this work will also explore the capabilities of neural networks for a task formerly accomplished by using n-grams.

[3] `https://grammarly.com`

[4] `https://www.discourse.org`

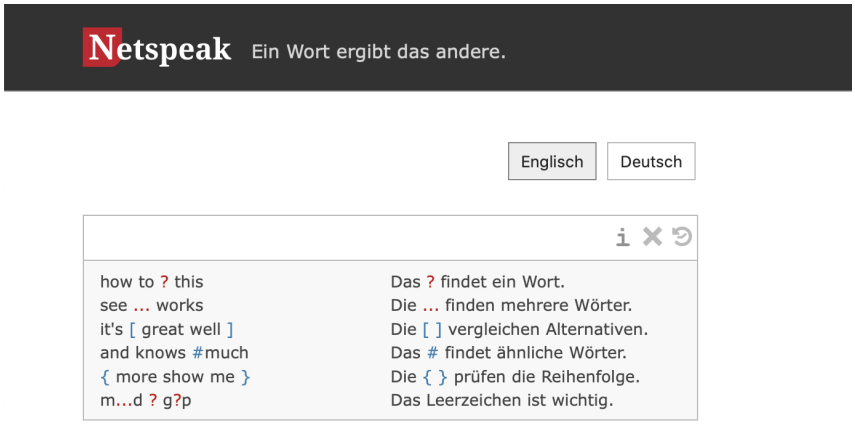


Figure 3.1: The Netspeak search engine web interface.

3.2 Netspeak

As mentioned in the last section, Netspeak is a search engine for commonly used word phrases on the internet. Figure 3.1 shows the web-accessible user-interface for Netspeak with the search bar for queries and some example queries. These examples show the seven basic operators of Netspeak queries, which are described in detail in section 3.2.1.

The frontend fetches data from an API that is made available by the Netspeak backend. Both systems communicate using gRPC[5], with functions and data types defined in Google's protobuf format.

The backend has access to multiple billion pre-indexed n-grams, with $n \leq 5$, together with their occurrence frequency in the Google N-gram Corpus[6]. When the API receives a Netspeak query, it searches the indexed n-grams for matches and returns a list of results, sorted descending by the n-gram's occurrence frequency. With its specialized index and search algorithm, Netspeak achieves retrieval times in the milliseconds.

Netspeak consists of these three main components:

1. A query language to formulate n-gram patterns,

2. an index of frequent n-grams on the web,

[5] `https://grpc.io`, an open-source high-performance framework for RPC (Remote Procedure Call)

[6] `http://storage.googleapis.com/books/ngrams/books/datasetsv3.html`

3. a probabilistic top-k retrieval strategy to find n-grams that match a given query.

3.2.1 Valid Netspeak Queries

Netspeak queries consist of words and operators to formulate patterns, which Netspeak uses to search the indexed n-grams for matches. They can include one or more of the seven basic different operators, which can be built using the tokens shown in Table 3.1. The seven basic Netspeak query operators we will refer to throughout this work are:

- single-word wildcard (example: `how to ? this`)
- multi-word wildcard (example: `see ... works`)
- single-character wildcard (example: `it fl?w away`)
- multi-character wildcard (example: `m...d the gap`)
- synonym operator (example: `and knows #much`)
- word options operator (example: `it's [ great well ]`)
- order operator (example: `{ more show me }`)

There are also other operators available to formulate valid Netspeak queries, but their functionality can be reproduced using the seven basic operators from above and are therefore ignored in this thesis.

Valid Netspeak queries can be divided into two groups: (1) fixed-length queries and (2) variable-length queries. The former only contains queries with words and operators that represent a fixed number of words or characters, such as the single-word wildcard or order operator, while the latter contains queries with one or more operators that are expanded into a variable number of words. This can be best described with the following example: The query `fine ? me` is a fixed-length query, since it can only match n-grams with a length of 3, while the query `fine ... me` can match n-grams with lengths of 2, ..., 5 and is therefore considered a variable-length query. To be able to process fixed-length and variable-length queries the same, variable-length queries are reformulated into a set of fixed-length queries and processed in parallel. The results are then merged in the end. For the aforementioned example the reformulated queries are the following:

- `fine me`,
- `fine ? me`,

Table 3.1: Netspeak query language tokens

Token	Description
?	Wildcard for either a character or a whole word
...	Wildcard for either multiple characters or two to three words
#	Compare synonyms for the word after the operator
[ ]	Compare the word alternatives enclosed by the square brackets
{ }	Find the best order of the words enclosed by the curly brackets

- `fine ? ? me` and
- `fine ? ? ? me`.

3.2.2 Result Retrieval

To search the billions of n-grams online in a reasonable time, the n-grams have to be indexed offline. Netspeak uses an inverted index μ, that is sorted by decreasing frequency of the n-grams in the corpus and enables an O(1) access to n-gram sets that fulfill the following constraints: (1) the n-gram has to contain the word w, (2) have the length n and (3) w has to be at position p in the n-gram. Including these three attributes as a tuple in μ allows for direct access through the indices. Let V denote the set of all words found in the n-grams D and let $D^\wedge$ denote the set of integer references to the storage positions of the n-grams in D on the hard disk. The inverted index μ maps each word $w \in V$ onto a postlist π_w, containing references to the n-grams that contain w:

$$\mu : V \times \{1, \ldots, n\} \times \{1, \ldots, n\} \to \mathcal{P}(D^\wedge).$$

For a given query q, the matching n-grams are at the storage positions

$$\mu_q = \cap_{w \in q} \mu(w, |q|, p)$$

with p being the position of w in q.

With the example from above, the indices for the query $q =$ `fine ? me` are given by $\mu_q := \mu(\texttt{fine}, 3, 0) \cap \mu(\texttt{me}, 3, 2)$. First, all indices are selected for n-grams of length 3, which have the word `fine` at position 0. Then, all indices from n-grams of the same length, which have the word `me` at position 2, are removed from the first list. The resulting indices μ_q contain references to 3-grams that start with `fine` and end with `me`, which are already sorted by frequency as the index list itself is also sorted as described above.

Postlist Pruning

Netspeak uses postlist pruning to reduce the number of operations needed for intersecting each word's postlist π_w to get the final list of indices μ_q for a query q. Let $f : D \to \mathbb{N}$ be a mapping from the n-grams D to their respective occurrence frequency in the dataset. Using this mapping f, each postlist $\mu(w, \cdot, \cdot)$ is sorted in decreasing order of f. This allows for two types of postlist pruning: (1) head pruning and (2) tail pruning.

Head Pruning Given a query q, let τ denote an upper limit for the frequencies of the n-grams in the result set of q. τ is the minimal frequency of all non-terminal n-grams (n-gram not containing any Netspeak operators) in q. Since no n-gram matching q can have a higher frequency than τ, as the resulting n-grams have to contain that n-gram n of q with the lowest occurrence frequency, and n-grams containing n have to be less frequent since each longer n-gram containing n also contributes to the frequency of n.

For example, in the query q = "sounds fine ? me", the two maximum, non-terminal n-grams are the 2-gram "sounds fine " and the 1-gram "me", which have the occurrence frequencies f("sounds fine") = 45,817 and f("me") = 566,617,666 Since no n-gram matching q can have a frequency larger than the minimal frequency of the two maximum, non-terminal n-grams, which is in this example $\tau = f(\text{"sounds fine"}) = 45,817$, all entries of μ("sounds"), μ("fine"), and μ("me") whose n-grams have a higher frequency than τ can be skipped.

Tail Pruning Since Netspeak users look for n-grams commonly found on the web, very rare n-grams are of less interest. If a postlist is too long to be loaded into memory at once, tail pruning is applied. Netspeak uses three different strategies for tail-pruning:

1. stop after a specified number of matching n-grams have been found,
2. stop after a specified number of entries from a postlist have been read,
3. stop after a specified quantile of a postlist has been read.

Using the described offline indexing and postlist pruning, Netspeak can not only retrieve results in milliseconds but gives the user also the ability to extend the pruned search results to retrieve the complete result list.

3.3 Writing Assistants using Transformers

Since the Transformer architecture has only been released in 2017, few attempts have been made to use it in a writing assistant besides the original use as a translator or more general sequence to sequence conversions.

One prominent example where the performance of Transformers has been compared to a part of a traditional writing assistant is the already mentioned tool Grammarly. The research team at Grammarly Inc. used a Transformer to build a grammatical error correction system for their assistant (Alikaniotis and Raheja [2019]). In their work, they used BERT to calculate the probabilities of different versions of a sentence from an automatically generated confusion set and use this probability as an indication for potential grammatical errors. The proposed yet not ideal approach to calculate this probability is to iteratively mask each word in the sequence and sum the log probabilities of those words. Additionally to BERT, the researchers tested OpenAI's GPT and GPT-2 models as well and compared the results to BERT and previous work. Concluding the test results, the researchers state that Transformer-based language models, trained on vast amounts of unlabeled texts, achieve nearly state-of-the-art performance in grammatical error correction with only a very limited amount of annotated data.

Chapter 4

Methodology

4.1 NeuralNetspeak

NeuralNetspeak is what we call our version of Netspeak based on neural language models. In contrast to Netspeak, NeuralNetspeak doesn't use an indexed n-gram dataset but pre-processes the queries so a language model can predict word-placeholders and assign a score to the output. The synonym retrieval required for processing Netspeak queries also makes use of a neural network.

This should allow NeuralNetspeak to show more relevant results based on the context given in the query while also lifting the query length limitation of Netspeak. In this iteration of NeuralNetspeak, we use the BERT language model (see section 2.2), because the training objective it was pre-trained on, namely whole word masking, closely correlates with one of Netspeak's tasks: predicting word wildcards. We can also use BERT to score a given input and use that score to make a statement about the linguistic quality of that input. Then, we use this score to sort the results, showing the best result first. This score can also be shown to the user as a replacement for the frequency measure used by Netspeak to give the user a feeling for how good a particular result is compared to other results. Further changes to the user interface and differences in the featureset of NeuralNetspeak compared to Netspeak are discussed in section 4.1.5 and section 4.1.6 respectively.

As described in the introduction, the goal of NeuralNetspeak is to be a drop-in replacement or a supplement for the n-gram based backend Netspeak. To accomplish this, NeuralNetspeak uses the same API definition as Netspeak. A working example of NeuralNetspeak with Netspeak's frontend can be found at `https://netspeak.org/demo/`.

4.1.1 Query Processing with BERT

We use BERT for processing Netspeak queries in two ways: (1) to calculate a score for several or single words in the context provided by the query, and (2) to predict a set of words at certain positions, using the score of each prediction as an indication for the linguistic quality of the resulting output. The scores are retrieved from BERT by averaging the probabilities it assigned to the several words or single word in BERT's final softmax layer. In this thesis, we refer to the probability of a single word w with $\text{score}_{\text{word}}(w)$.

The whole result retrieval process for a valid Netspeak query can be divided into five steps: (1) tokenization and pre-processing, (2) scoring, (3) mask prediction, (4) synonym retrieval, and (5) result collection. First, the query is tokenized based on regular expressions to determine the positions of the different Netspeak operators in the query. This tokenized query is then pre-processed, resulting in multiple subqueries that are derived from the original query by replacing specific operators with possible results and only contain mask and synonym tokens, as well as words considered final. These subqueries are then processed in parallel by first calculating a base score for the words considered final, then predicting all masked words, and finally retrieving synonyms for words marked with the synonym operator. The synonyms are retrieved and scored after the mask prediction to minimize the work needed to be done for mask prediction since that task is very expensive in terms of computational cost and retrieving the synonyms first would result in multiplying the number of mask predictions that have to be made by the number of synonyms found for each word. As the synonyms of a word are typically quite close to the original word's meaning, this shouldn't affect the masked-word prediction results. Finally, all results are collected and sorted by the score assigned by the language model before the result list is returned.

Tokenization and Query Pre-Processing

To tokenize a query, it is matched word by word against a list of regular expressions, each identifying one of the seven operators supported by valid Netspeak queries. Words that do not match any of the regular expressions are marked as final since they only provide context and should appear unchanged in the output. The result of the tokenization step is a sequence of words, each assigned a number, which either identifies it as final or assigns it one of the operators.

Using the tokenized query, multiple pre-process subqueries are generated sequentially by iterating over all words. Each iteration step uses the partial pre-process subqueries of the previous iteration and appends one or more words or tokens to it. Depending on the word or query operator at a given position in

the query, one of the following operations is executed for that word or operator:

- Words tokenized as final are just appended to each of the previous step's partial pre-process subqueries.
- Multi-mask operators are appended once as two and once as three BERT `[MASK]` tokens, which we call a mask group. This doubles the number of partial pre-process subqueries compared to the previous step.
- Single-mask operators are appended as a mask group containing only one BERT `[MASK]` token, as only one word should be predicted. Since the language model we use operates on WordPiece and not whole words, using a single `[MASK]` token may cause longer words to not being predicted. In future works, it could be evaluated if adding multiple `[MASK]` tokens here and only considering WordPieces (indicated by the prefix `##`) as valid predictions for these additional tokens in the mask prediction step yields better results or improves the recall.
- There are two types of in-word mask operators, that Netspeak queries can contain: (1) a placeholder for exactly one character and (2) a wildcard for multiple characters. Both of them are processed similarly. First, regular expressions are generated from words containing those in-word mask operators by replacing the single in-word mask operator with an expression to match exactly one word-character and the multi in-word mask operator with an expression matching more than one word-characters. These expressions are then used to retrieve matching words from the vocabulary of the language model. Although the model's vocabulary is limited to about 30.000 words and contains not only words but also word-pieces and single characters, it also allows us to test the suitability of the language model alone for answering Netspeak queries. The results could likely be improved either by searching a dictionary of words for matches or relying on Netspeak for retrieving fitting words. However, using the latter option would only allow statements about the result ranking of Netspeak and NeuralNetspeak, since the results themselves would be the same. Another approach would be to mask the word which includes the placeholder operator and filtering the words predicted by the language model for valid matches, though this would also further decrease the amount of context-providing words, especially for short queries or queries containing multiple operators which result in mask tokens. Or, since BERT operates on WordPiece-level, masking the character(s) in question and filtering the predictions for these masks for WordPieces could be a conceivable yet for this work too elaborate approach.

Each matching word is appended once at a time to each previous pre-process subquery, multiplying the number of resulting pre-process subqueries by the number of matching words. To prevent this number from exploding, the number of possible matches could be limited using heuristics, e.g. by using only those up to x words, which are most similar to the original word.

- Words marked with the synonym operator are appended to the pre-process subquery without the operator.
- Word alternatives to be compared are appended one by one to each of the partial pre-process subqueries from the previous iteration, multiplying the number of resulting subqueries by the number of word alternatives.
- To determine which order of the words enclosed by a word order operator is the best, each one of the possible permutations is appended to each pre-process subquery generated from the previous step, multiplying the number of resulting subqueries by the number of possible permutations.

After the last iteration step, the subqueries don't contain any Netspeak operators anymore and can directly be used as inputs for BERT for masked word prediction and query scoring, which are the next steps in processing the subqueries.

4.1.2 Query Scoring Strategies

The different scoring strategies used by NeuralNetspeak are (1) sum scoring and (2) batch scoring. In both strategies, $\text{score}_{\text{word}}(w)$ is the score of a specific word w from a word sequence q as determined by BERT. In practice, $\text{score}_{\text{word}}(w)$ is the probability of the word w at its position in the softmax layer of BERT after giving it q as an input.

Sum Scoring

With sum scoring, the score for a single query q is given by

$$\text{score}_{\text{sum}}(q) = \sum_{w \in W \subseteq q} \text{score}_{\text{word}}(w),$$

where $W \subseteq q$ are the words which should be considered when calculating the score. This way, we can ignore specific words in our score calculation.

We use this scoring strategy among other things to calculate the **base score** $\text{score}_{\text{base}}(s)$ of a word sequence s with

$$\text{score}_{\text{base}}(s) = \sum_{w \in W_f \subseteq s} \text{score}_{\text{word}}(w).$$

For the base score, we only take into account the final words $W_f \subseteq s$, which are all words that aren't either a `[MASK]` token or a word marked with the synonym operator in the initial Netspeak query from which s was built as described in section 4.1.1. Following the definition of sum scoring, the base score is defined as If all words of s are final, so if $W_f = s$, the base score is the final score of the result and is displayed to the user without further modifications.

Batch Scoring

We use **batch scoring** to score multiple results at once. For this, we leverage BERT's ability to process multiple sentences within a single input by separating them with a `[SEP]` token (sentence/sequence separation token used by BERT). Given a list of queries $q_1, \ldots, q_n$ that should all be scored at once, we generate a **query batch** which can be used as the input for BERT. For this, we start the input sequence with a `[CLS]` token as required by BERT, then append the first query and a separation-token. Then, we append the next query and another separation-token. We repeat this last step until (1) all queries are contained in the query batch or (2) the query batch would exceed a maximum length of l tokens if another query and a separation-token would be appended, so the query batch only contains queries. In this work, the maximum length l of the query batch tokens is 512, as this is the input length limit of our model.

After this process, we have a query batch token sequence b is in the following form:

$$b = (\texttt{[CLS]}, q_{1,1}, \ldots, q_{1,m}, \texttt{[SEP]}, \ldots, q_{k-1,n}, \texttt{[SEP]}, q_{k,1}, \ldots, q_{k,o}, \texttt{[SEP]}),$$

where $q_{i,j}$ is the j-th token of the i-th query, and k is the last query that can fit completely in b with a separation token at the end so that $|b| \leq l$.

Then, we generate a list of sequence ids which provide information about the start and end of each query to the language model. Because the model was trained on tasks involving only up to two sentences per input sequence, it only accepts the sequence ids 1 and 0. This requires us to use alternating sequences of only ones and only zeros in the sequence id list, each sequence s_i containing as many same sequence ids as its corresponding query q_i.

Having generated the query batch token sequence and the sequence id list, we feed both lists into BERT. Then, we take the resulting probabilities of the

softmax layer and split them using the sequence id list to assign a probability score to each token of each query. With the scores assigned, we then use sum scoring to calculate a score for each query based on the probability scores of each token of the query.

The advantages of batch scoring multiple queries at once are two-fold. First, by concatenating all queries into a single input for BERT, we only have to let the prediction run once to be able to calculate a score for each query. This dramatically reduces the computation required for calculating the score for multiple queries compared to scoring each query individually only using sum scoring, which is critical for operation in the context of a search engine. Second, by retrieving all scores from the same output of the same prediction, we ensure the comparability of the results because the score for one result depends on the entire input sequence. When the results are fed into BERT one at a time, the scores differ from the score it assigns to the same result when it is packed together with other results into a single input sequence.

4.1.3 Subquery Processing

After the input query has been pre-processed as described in section 4.1.1, we calculate a base score for each subquery using the aforementioned sum scoring approach. Note that the base score calculation ignores mask tokens and words marked for synonym retrieval. For subqueries whose input query contained only in-word wildcard, order, or alternatives tokens, no further processing is required and the subquery processing is done. These subqueries are therefore shown to the user as results with the base score as their final score. The subqueries that have to be processed further using BERT are only the ones containing mask tokens, generated from single or multi-word wildcard tokens, or words marked for synonym retrieval and ranking. These further processing steps create additional subqueries, which we call **mask-subqueries** in the maks prediction and scoring step and **synonym-subqueries** in the synonym retrieval and scoring step.

4.1.4 Mask Prediction and Scoring

After calculating a base score for a subquery containing mask tokens, we split the positions of mask tokens into distinct mask groups M of tokens generated by a single query operator. As an example, the query `how ... use this ?` would produce multiple subqueries including `how [MASK] [MASK] use this [MASK]`, where the first two mask tokens and the last mask tokens would be divided into two mask groups.

All mask groups are processed sequentially from left to right. Each mask

group processing step results in multiple mask-subqueries, which are created by replacing the mask tokens of the mask group with predicted word tokens and used as inputs for the next mask group prediction.

The subquery example from above would therefore be processed in two steps. First, the first two mask tokens are predicted with our BERT language model and replaced by predicted word tokens. Since BERT calculates a probability for each word of the vocabulary for each mask token, we take the k words with the highest score for each mask token and generate a mask-subquery for each combination. These mask-subqueries are then used to predict the last mask token.

Mask Prediction and Scoring Strategies

We tested the following three different strategies for predicting and scoring mask groups consisting of multiple mask tokens:

1. **Combination Sum Scoring:** Predict all mask tokens of the mask group at once and build all possible combinations of the predicted words, respecting their relative position. To calculate the score for the word combination, sum the scores of the individual words. Add this score to the base score.

   Problem: The top predicted words for each mask token are not contextually connected, so a result constructed from these top words although not being a sensible combination will have the highest score, as each word has the highest score at its position, and will subsequently be shown as the first result to the user. This could result in a high score being assigned to word combinations that make no sense linguistically.

2. **Sequential Prediction and Individual Scoring:** Predict each mask token of the mask group sequentially from left to right, taking the best k results from the previous and use them to create new mask-prediction subqueries which are then used to predict the next mask token.

   Problem: Scoring each mask-subquery individually takes a lot of time, which is not acceptable in the domain of search engines like Netspeak and therefore NeuralNetspeak.

3. **Combination Batch-Rescoring:** Predict all mask tokens of the mask group at once, then build all possible combinations of the predicted words while respecting their relative position. Sort these results by the sum of the scores of the individual words and re-score the best results using batch-scoring.

Problem: Due to the limited input sequence length in the batch rescoring process, some of the results may get discarded. However, since these results have a low score due to them being sorted by score, they are probably not as relevant as the first results anyway.

We ran all three tests with the same 4,000 queries from our data set, which contained only the multi whole-word wildcard operator. The results can be found in the appendix in section A.1. While the first strategy answered the queries the fastest (845.5 seconds per 1,000 queries), it also resulted in the worst average rank of the expected result with an average rank of 25.93 for short and 10.06 for long queries. The second strategy ranked the expected result considerably higher, with an average rank of 15.59 and 6.53 for short and long queries respectively, but the execution time of 2722.5 seconds per 1,000 queries is inacceptable for our use-case as a search engine. Finally, the third strategy ranked the expected result only slightly lower than the second strategy, with an average rank of 16.58 and 6.51 for short and long queries respectively, while nearly matching the execution times of the first strategy with 904.5 seconds per 1,000 queries.

Following these results, we decided to use the third option, combination batch-rescoring, because it constitutes a reasonable trade-off between speed and precision. The processing speed is of great importance because Netspeak, and therefore NeuralNetspeak, is a search engine, which makes response times of multiple seconds not feasible.

Mask Prediction

For each mask group, the top k predictions with the highest score are retrieved from BERT for each masked position within that group. We denote the collection of sets of predicted words P for a mask group i as W_i, so that $W_i = \{P_1, \ldots, P_n\}$, where n is the number of masked positions within the mask group i and $\forall P \in W_i : |P| = k$.

Since the n-th word predicted for a `[MASK]` token at a certain position is not contextually related to the n-th word predicted for the subsequent position, even within the same mask group, we build all possible combinations of predicted words within a mask group, respecting the position they were predicted for in the original query. We define this set of word combinations for a mask group i as

$$M_i := \{(w_1, ..., w_{|W_i|}) \mid w_j \in P_j, P_j \in W_i\}.$$

After a mask group has been processed, the outputs are used to replace the `[MASK]` tokens in q, resulting in a set Q_m of mask-subqueries. These mask-subqueries are then used to predict the next mask group prediction step instead

of the subquery q. While increasing the number of predictions needed for the next mask group prediction by k^n, where n is the number of `[MASK]` tokens within the current mask group, we also provide BERT with additional context to improve the quality of word predictions in subsequent steps. Since we are using a k of 10 in the current implementation and are only testing queries with a single mask group, we consider this a reasonable trade-off.

However, if queries containing multiple mask groups are allowed, adjustments must be made to prevent the number of new sentences from exploding. One simple approach to limit the number of newly generated mask-subqueries Q_m would be to decrease k with each processed mask group. Another possibility would be to introduce a threshold to consider only those predicted words whose score is greater than the threshold. We have implemented the first approach in NeuralNetspeak. But since we consider in this work only queries with a single operator token, the efficacy of this approach still needs to be evaluated.

Applying this procedure to all n mask groups, we get a set M, which contains sets of all possible combinations of predicted words for all mask groups:

$$M := \{M_i \mid i \in \{0, ..., n\}\}.$$

Mask Scoring

We use combination-batch-rescoring to assign a score to each mask-subquery. To do so, we sort all mask-subqueries after the mask group prediction by the score of the predicted words used in the mask-subquery. With $C_j \in M_i$ being a combination of predicted words, the score for this combination is calculated with

$$\text{score}_{\text{predictions}}(C_j) = \frac{1}{|C_j|} \cdot \sum_{w \in C_j} \text{score}_{\text{word}}(w).$$

The score of a combination of words predicted for a mask group i is the sum of those word's scores, weighted by the number of words in that combination. The weighting makes two and three-word results, caused by a multi-word mask Netspeak query operator, comparable.

After sorting the mask-subqueries, a new score is calculated for each mask-subquery. We do this because a combination of the first predictions of each masked token, sorted by the score assigned by BERT, will always have the highest combined score, but those predictions aren't contextually related. This leads to a high score being assigned to word combinations that make no sense linguistically. To combat this, we re-score the mask-subqueries using batch scoring. Since some mask subqueries can be discarded during batch scoring due to the limitation of the input sequence length of the language model, the above-mentioned sorting step is necessary, although the score is not displayed to the

user and is not used in later processing steps. We call this recalculation of the result scores after filling the masked positions with predictions **combination batch-rescoring**.

We calculate the score of a mask-subquery q_m given a list of mask queries $Q_m = (q_1, ..., q_m, ..., q_n)$ with

$$\text{score}_{\text{mask}}(q_m|Q_m) = \sigma_m(\text{batchScore}(Q_m)),$$

where batchScore(Q_m) is the function for batch scoring the list of mask-subqueries Q_m, resulting in a list of n scores for the n subqueries in Q_m. From this list, we select the m-th score using the selection function σ_m. Since the batch scoring function calculates a score for each mask-subquery in Q_m, its results can be cached in practice and all scores can be selected from this cached list of scores.

Synonym Retrieval and Scoring

The last step in processing the query is to find and compare synonyms for the words marked with the synonym operator in the Netspeak query. Since BERT itself can't be used to retrieved synonyms for a given word, we first use spaCy[1] for POS tagging and use the POS tag of the word to filter the sets of synonyms wordnet[2] provides for that word. We have tested and evaluated three different strategies for synonym retrieval and scoring:

1. **Retrieval and individual scoring**: Retrieve all synonyms for the word as described above and use them to build new synonym-subqueries by replacing the word with its synonyms one by one. Then, use sum-scoring to calculate a score for each synonym-subquery.

   Problem: Very slow when many synonyms are retrieved for a word as each new synonym-subquery has to be processed individually.

2. **Retrieval and batch scoring**: Same procedure as in the first strategy, but using batch scoring to score the new subqueries to decrease the time needed for rescoring significantly.

   Problem: The synonym retrieval delivers an *unsorted* list of synonyms. Since batch scoring may discard some subqueries due to the limited input sequence length of our model, we can't guarantee that we include the most fitting synonym in the query batch which in turn means that the results may not contain the most relevant result, which is not acceptable for a search engine.

[1] `https://spacy.io`

[2] `https://www.nltk.org/howto/wordnet.html`

3. **Filter predictions for synonyms**: Replace the word with a `[MASK]` token and use BERT to predict words at that position. Retrieve the synonyms the same way as in the first two strategies and filter the predicted words for those synonyms. Add the scores of the predicted words to the score of the synonym-subquery, which is either still the base score or was assigned in the mask prediction and scoring step, and return the results sorted by score.

   Problem: The mask prediction might not find all synonyms for the original word, and no synonyms consisting of multiple words. However, the synonyms that are found, should fit the query quite well as they are predicted based on the context of the synonym-subquery.

We ran all three tests with the same 4,000 queries from our data set, which contained only the synonym operator. The results of each test run for each strategy can be found in the appendix in section A.2. While the first two strategies answered the most queries (17.8 %), the test of the first strategy had the longest execution time (355 seconds for 1,000 queries). The second strategy, while being the fastest (212 seconds per query), ranked the expected result considerably lower on average (rank 2.41). Finally, the third strategy ranked the expected result the highest on average (rank 1.21 for short and 1.12 for long queries), while having similar execution times compared to the second strategy (262 seconds per 1,000 queries) but also the lowest number of answered queries (13.8 %).

Following these results, we decided to use the third option, filtering the predictions for synonyms, because, although having the lowest recall for both short and long queries, it achieves the best result ranking across short and long queries. This drastic improvement of the average rank is worth the decrease in recall.

We retrieve the synonyms after the mask prediction step to ensure that no other mask tokens are included in the synonym-subquery if the original query contained not only the synonym operator but also a word wildcard operator. Using the original word in the mask-subqueries when predicting masked tokens should not affect these predictions too much, since synonyms are close in meaning. However, some synonyms may need a different preposition as the original word, which in turn is not predicted by using mask-subqueries with the original word. Since we only consider queries including a single query operator in this work, investigating this behavior is out of scope for this thesis.

To calculate the score for a synonym-subquery with a predicted synonym, we use sum scoring to calculate a score for the synonym-subquery, considering all words other than the predicted synonym, and simply add the score of the predicted synonym. Let q be the query, s the predicted synonym and $W = q$

s all words of q without the predicted synonym s. Then, the score $\text{score}_{\text{syn}}(q|s)$ for q given s is

$$\text{score}_{\text{syn}}(q|s) = \frac{1}{|q|}\left(\sum_{w \in W \subseteq q} \text{score}_{\text{word}}(w) + \text{score}_{\text{word}}(s)\right).$$

Final Result Score

In summary, the final score of a particular result r, given the predicted words selected for this particular result M_r and synonyms S_r, is calculated in one of three ways, depending on the operators contained in the original query q:

1. If q contains neither whole-word wildcard operators nor synonym operators, r is one of the unaltered subqueries generated from q with the base score as the final score.

2. If q contains whole-word wildcard operators but no synonym operators, the final score of r given the predicted words M_r is the score of the mask-subquery after the batch-rescoring process.

3. If q contains synonym operators, the final score of r given the synonyms S_r (and the predicted words M_r if q also contained whole-word wildcards) is the score of the synonym-subquery after calculating $\text{score}_{\text{syn}}(q|s)$. Since this score also includes all previously predicted words, if any, the words M_r are also indirectly re-scored through the synonym score calculation.

Sorting the results by score concludes our BERT-based query processing.

4.1.5 Implementation and Integration in Netspeak

A product of this work is a version of NeuralNetspeak implemented as a python package, which is also used for all tests and performance evaluations in the next chapters. Figure 4.1 shows an overview of the data flow through the various software components of the NeuralNetspeak backend.

When a user enters a query in the search field of the web interface (see Figure 4.2) it is sent as a request to the first docker container[3] running an envoy proxy[4], which then forwards the request to a server running in the NeuralNetspeak docker container. Since the frontend uses gRPC-Web[5] to communicate

[3] `https://www.docker.com`
[4] `https://www.envoyproxy.io`
[5] `https://grpc.io/docs/platforms/web/basics/`

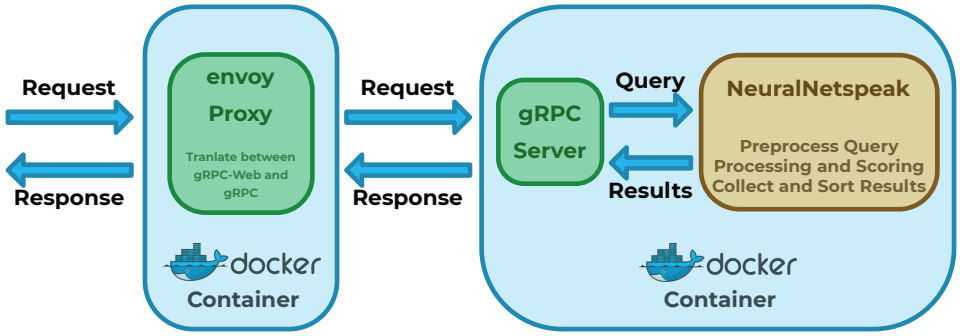


Figure 4.1: A macroscopic overview of the NeuralNetspeak software.

with the backend, but the server in our NeuralNetspeak container only accepts gRPC[6] bitstreams, the envoy proxy is needed to translate between the JSONP-based gRPC-Web and bitstream-based gRPC protocols. It also can be used to separate the server running NeuralNetspeak from the internet and can be configured to access multiple servers running NeuralNetspeak instances in a cluster. The gRPC server in the NeuralNetspeak container then extracts the query from the request and uses a NeuralNetspeak instance to process it. All results are then sent in a response back to the envoy proxy which translates the gRPC response to gRPC-Web and sends it to the frontend.

NeuralNetspeak

To allow NeuralNetspeak to be used as a drop-in replacement for the current Netspeak backend, the `NeuralNetspeak` instance is managed by a gRPC server using the same protobuf API specification as Netspeak. NeuralNetspeak uses a version of the large BERT model trained on whole word masking[7], offered by the huggingface transformers package[8]. We use this specific model because we use BERT to only predict whole words for the whole-word operators, as the possible words for the in-word operators are already inserted into the input in the query pre-processing step. If we were to predict in-word operators with BERT as well, we could split the word containing the operator into individual WordPieces and use a different model to predict single or multiple word pieces.

The whole software can be deployed completely self-contained via a docker container. To achieve maximum performance, the docker image for the container is based on the `nvidia/cuda` image provided by Nvidia, which already comes with the CUDA Toolkit and drivers needed to access any Nvidia GPUs connected to the host system from within the docker container. This way,

[6] `https://grpc.io/`
[7] Model name: `bert-large-uncased-whole-word-masking`
[8] `https://github.com/huggingface/transformers`

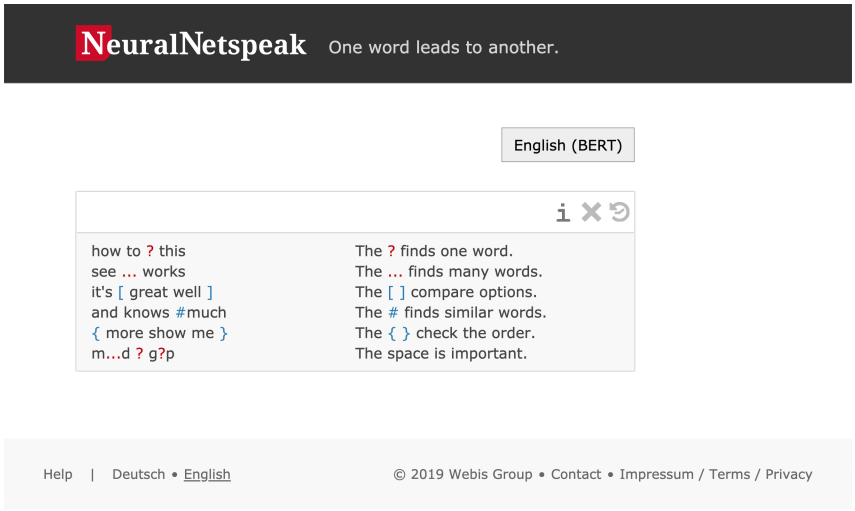


Figure 4.2: The slightly changed web interface for NeuralNetspeak. Instead of the two corpus selection buttons, there is only a button for "English (BERT)", referencing the language model used by NeuralNetspeak.

NeuralNetspeak can leverage a connected GPU to accelerate all BERT-related matrix calculations.

Web Interface

Figure 4.2 shows the updated web interface for NeuralNetspeak. Compared to the standard netspeak interface, instead of "English" and "German" there is only one language to select, which we named "English (BERT)" to differentiate our BERT-backed result retrieval process from Netspeak's existing corpus-based retrieval strategies. Future updates to NeuralNetspeak may add other languages or variants of English language models which can be differentiated using the same naming scheme.

4.1.6 Limitations

The goal of NeuralNetspeak is to reproduce most of the core functionality the current Netspeak backend provides while using a Transformer for predicting missing words and scoring the results. However, due to fundamentally different retrieval strategies of Netspeak and NeuralNetspeak, the feature set of NeuralNetspeak may be a bit limited compared to the current implementation of Netspeak.

One of the most important metrics for Netspeak, the frequency of the retrieved phrases, is based on the n-gram dataset. Since NeuralNetspeak doesn't

use a dataset of phrases but sorts the results based on the scoring from the Transformer by relevancy, no statement can be made about the frequency of a given result. To still be able to give the user a feeling for the relative quality of the results, instead of the frequency, NeuralNetspeak returns the score of each result together with the result itself to be shown to the user.

4.1.7 Expected Improvements

Despite the limitations mentioned above, using language model based query result retrieval has many advantages over an indexed list of fixed-length phrases.

Due to the limitations given by the length of n-grams, only the context of a few words surrounding the query operator can be used to find relevant results, because all context providing words must be an exact match for a result to be found. Since the transformer-based language model can work with substantially longer input sequences of up to 512 tokens while maintaining context over the entire input sequence, we expect the results to be more relevant when longer queries are supplied. Also, we expect the language model to process queries that contain words in an order it has never seen before based on its understanding of natural language. This should lead to an increased number of answered queries.

Another consideration is the amount of storage required for the current Netspeak implementation and Neural Netspeak. While Netspeak needs about 200 GB to 250 GB of low-latency disk space to store its n-grams and index lists, the large BERT model used by NeuralNetspeak only needs about 1.35 GB of disk space and is loaded into memory once after starting the service. However, these advantages are somewhat diminished by the significantly higher cost of computing resources compared to storage costs.

4.2 Datasets

We use multiple text corpora of high linguistic quality to compare on both NeuralNetspeak and the current version of Netspeak. For this, we decided to use the British National Corpus (BNC Consortium [2007]), Europarl Corpus (Koehn [2005]), The New York Times Corpus (Sandhaus [2008]), and a Corpus assembled from featured Wikipedia articles[9], which ensure a higher language quality compared to other Wikipedia articles. Using these four corpora we cover multiple domains such as informative and imaginative texts, spoken language, news articles, and factual texts.

4.2.1 Sentence Selection

Due to the sheer size of the corpora, not all sentences from every corpus can be used. Since we want to have 5,000 queries for each one of the seven valid Netspeak query operators per dataset to eliminate statistical effects, we select 35,000 sentences for query generation. This amounts to a total of 140,000 queries across all four corpora. The 35,000 sentences used for generating the queries from a corpus are randomly selected from all sentences of the entire corpus, except for the New York Times Corpus, where we only consider articles from the three most recent years available, 2005, 2006, and 2007. We only use the most recent years, to minimize the effects changes in language over time have on the results. Since the objective of this thesis is to evaluate the performance of NeuralNetspeak on idiomatic language from current years and onward, data from articles from many years ago wouldn't be meaningful for evaluating NeuralNetspeak's performance in this context.

[9] https://www.kaggle.com/jacksoncrow/wikipedia-multimodal-dataset-of-good-articles

4.2.2 Query Generation

We generate two types of queries from each sentence. First, we generate a **short query** Q_s, which strictly follows Netspeak's query format as described in section 3.2.1 and contain only up to five words. Second, we generate a **long query** Q_f by replacing the words used for generating the short query in the original sentence with the short query itself. This results in longer queries, which still use the Netspeak query operators but provide more context for query processing. These long queries let us measure the performance difference given a longer context. However, due to the five-word limit of queries Netspeak can process, these long queries can only be tested with NeuralNetspeak. Table 4.1 shows examples of our auto-generated queries. Further examples can be found in the appendix in Table A.3

Since Netspeak and NeuralNetspeak work with the same queries that follow a certain format, the 35,000 queries can be generated unsupervised from the 1.5 million to more than 6 million relevant sentences of each dataset. Only a random part of each selected sentence is used to generate one short query containing only a single operator. This part of the sentence is limited to five words to respect the query length limitation of Netspeak. Further, we only generate queries from parts of the sentences, which don't contain any proper nouns or names. We also don't expect Netspeak or NeuralNetspeak to fill placeholders with numbers, as they are mostly topic-specific and won't make any statements about the general suitability of either system as a general writing assistant, so words containing non-word characters are not considered for masking and synonym operators. Before a query is generated from a given sentence, it is first cleaned by removing all special characters such as quotation marks and asterisks and characters that overlap with Netspeak query operators.

Given a sequence of words $X = (w_1, w_2, \ldots, w_n)$, we randomly (uniformly) select an index i_q, $1 \leq i_q \leq n$, for the start of the query. Since queries written by Netspeak's users typically contain only certain parts of a sentence, as seen in Netspeak's query logs, it makes sense to generate queries from different parts of a sentence rather than to use only the beginning or end of the sentence. Next, we sample a query length l from a discrete uniform distribution $l \sim \mathcal{U}(a_l, b_l)$ with values between $a_l = 3$ and $b_l = 5$, and an index i_o within the query range $i_o \sim \mathcal{U}(i_q, i_q + l)$ at which the word, or, depending on the operator, multiple words, of the sentence should be replaced with a query operator. To determine which operator should be used in the query, we then sample a number $o \sim \mathcal{U}(1, 7)$, representing one of the seven valid Netspeak operators.

Whole-word Wildcard Operators If the single-mask operator was selected, the word w_{i_o} from X is replaced by the corresponding query operator

token ?, producing the new sequence $X' = (w_1, \ldots, w_{i_o-1}, ?, w_{i_o+1}, \ldots, w_n)$.

For the multi-mask operator, w_{i_o} and either one or two surrounding words are replaced by the multi-mask operator token .... We limit the number of words replaced by the multi-mask operator to three, to provide enough context with the short query to prevent Netspeak and NeuralNetspeak from showing random results (or "guessing"), and replace at least two words to distinguish between the single-mask and multi-mask operator, as the former is a placeholder for a single word.

Example: Let $X = (\text{This}, \text{is}, \text{an}, \text{example}, \text{sentence})$ be a sequence of words with w_{i_o} = "an" as the word to be replaced with by operator. For the single whole-word wildcard query operator, the output sequence X' is $X' = (\text{This}, \text{is}, ?, \text{example}, \text{sentence})$. For the multi whole-word wildcard query operator, one possible output sequence X' is $X' = (\text{This}, \ldots, \text{example}, \text{sentence})$.

In-word Wildcard Operators Similar to the single-mask operator, queries containing a single-in-word-mask operator are produced by replacing a single random character of w_{i_o} with the respective token ?. The result is a sequence of the form

$$X' = (w_1, \ldots, w_{i_o-1}, (x?z), w_{i_o+1}, \ldots, w_n),$$

with $xyz := w_{i_o}$, where x and z are the surrounding characters of the single character y which should be replaced.

Similarly, in-word multi-mask operator queries are built by a random number of characters between the first and last character of w_{i_o} with the operator token ..., resulting in sequences with the following structure:

$$X' = (w_1, \ldots, w_{i_o-1}, (a \ldots b), w_{i_o+1}, \ldots, w_n)$$

where a and b are the first and last character(s) of w_{i_o}, $0 < |a| + |b| < |w_{i_o}| - 1$.

Example: Let $X = (\text{This}, \text{is}, \text{an}, \text{example}, \text{sentence})$ be a sequence of words with w_{i_o} = "example" as the word to be replaced with by operator. For the in-word single-mask query operator, one possible output sequence X' is $X' = (\text{This}, \text{is}, \text{an}, \text{ex?mple}, \text{sentence})$. For the in-word mulit-mask query operator, one possible output sequence X' is $X' = (\text{This}, \text{is}, \text{an}, \text{e} \ldots \text{le}, \text{sentence})$.

Synonym Operator To create a query to retrieve and compare synonyms for the word w_{i_o}, we first acquire a list of synonyms S for w_{i_o} using the same basic synonym retrieval strategy with spaCy and wordnet as described section 4.1.4. From this list of synonyms, we pick one synonym $s_{i_o} \in S$ at random which we use to replace w_{i_o} with. Then, we prefix the synonym with the hash symbol synonym operator. The resulting sequence is as follows:

$$X' = (w_1, \ldots, w_{i_o-1}, \#s_{i_o}, w_{i_o+1}, \ldots, w_n).$$

Since we use the same approach for retrieving synonyms when creating the queries as to when we process these queries, NeuralNetspeak will retrieve the synonyms for this synonym rather than for the original word. But because the relationship between synonyms is not bijective in wordnet, those retrieved synonyms might not include the original word.

Example: Let $X = (\text{This}, \text{is}, \text{an}, \text{example}, \text{sentence})$ be a sequence of words with $w_{i_o} =$ "example" as the word to be replaced with by operator. For the synonym query operator, X' is $X' = (\text{This}, \text{is}, \text{an}, \#\text{sample}, \text{sentence})$.

Alternatives Operator Queries containing the alternatives operator are constructed by retrieving a synonym w'_{i_o} of word w_{i_o} using the same approach as above. Both words are then surrounded by square brackets `[ ]`, producing the sequence

$$X' = (w_1, \ldots, w_{i_o-1}, \texttt{[}\ w_{i_o}\ w'_{i_o}\ \texttt{]}, w_{i_o+1}, \ldots, w_n).$$

Since these queries are only used for validation purposes and not for training or fine-tuning the language model, there are no negative consequences from the first alternative word always being the expected word.

Example: Let $X = (\text{This}, \text{is}, \text{an}, \text{example}, \text{sentence})$ be a sequence of words with $w_{i_o} =$ "example" as the word to be replaced with by operator. Here, X' is $X' = (\text{This}, \text{is}, \text{an}, \texttt{[}\ \text{sample example}\ \texttt{]}, \text{sentence})$.

Order Operator To create queries for which all permutations of specific words should be compared, either one or two words directly surrounding w_{i_o} are cut from the sequence together with w_{i_o} itself. These two or three words are then shuffled and inserted back into the word sequence, surrounded by braces. One example of the structure of these queries is

$$X' = (w_1, \ldots, w_{i_o-2}, \texttt{\{}\ w_{i_o+1}\ w_{i_o-1}\ w_{i_o}\ \texttt{\}}, w_{i_o+2}, \ldots, w_n)$$

with $w_{i_o \pm 1}$ being the two words which originally surrounded w_{i_o} in X.

Example: Let $X = (\text{This}, \text{is}, \text{an}, \text{example}, \text{sentence})$ be a sequence of words with $w_{i_o} =$ "example" as the word to be replaced with by operator. Here, one possible output sequence X' is $X' = (\text{This}, \texttt{\{}\ \text{an example is}\ \texttt{\}}, \text{sentence})$.

Finally, using the new sequence of words and Netspeak query operator tokens X', the subset of words $Q_s := (w_{i_q}, \ldots, w_{i_q+l-r})$ is selected from X' as the final Netspeak query, with r being the number of removed tokens.

In this work, we refer to this type of generated query consisting of up to five words as **short query**. As described in the beginning of this section,

to evaluate the potential improvements discussed in 4.1.7, an additional **long query** Q_f is generated as well. Since X' still contains all original words left and right to Q_s, we can just take it as out long query:

$$Q_f := X' = (w_1, \ldots, w_{i_q}, \ldots, w_{i_q+l-r}, \ldots, w_n)$$

Based on our assumption that the sentences are almost optimally formulated, we use that part of the initial sentence X, from which the short query was generated, as the expected output $E = (w_{i_q}, \ldots, w_{i_q+l})$ in the testing phase.

Table 4.1: Examples for auto-generated queries from our dataset. We randomly selected two and three queries for each in-word and whole-word operator respectively. The queries for different operators are separated by a horizontal line.

Single in-word mask ?					
Well I don't *really ?ut there you* are					
Netspeak	**but**				
BERT-short	out	put	cut	**but**	nut
BERT-long	**but**	nut	gut	hut	cut
Multiple in-word masks ...					
During the day, he will take samples from the ship's hold to check for *damage and also c...k the* quantity of dust and husk in the load.					
Netspeak	**check**	click			
BERT-short	**check**	cook	crack	click	chuck
BERT-long	**check**	click	chalk	clock	creek
Single whole-word mask ?					
By being clearly against an unpopular figure Mrs Thatcher has usually *rallied public ?* to her side.					
Netspeak	**oppinion**	support			
BERT-short	mood	sympathy	at	**oppinion**	interest
BERT-long	**oppinion**	sympathy	sentiment	voters	votes
Multiple whole-word masks ...					
Instead of a repeat of last year when he was *surrounded ... of* reporters he was old news.					
Netspeak	by a group	by photos	by a number	by a crowd	by some
BERT-short	by	out	inside	all	by all
BERT-long	**by dozens**	with dozens	by hundreds	with hundreds	by thousands
Synonym #					
He was *quite #sensible to* believe that he would have to invade early to avoid the worst.					
Netspeak	sensible	**reasonable**	sensible		
BERT-short	sensible	**reasonable**			
BERT-long	**reasonable**	sensible	sane		
Word Alternatives [ ]					
It took a couple of minutes for my *[breathing respiration] to steady.*					
Netspeak	**breathing**				
BERT-short	respiration	**breathing**			
BERT-long	**breathing**	respiration			
Word Order { }					
All the injured were *taken { in hospital to }* Middlesbrough but none were seriously hurt.					
Netspeak	**to hospital in**	in to hospital			
BERT-short	in to hospital	**to hospital in**	to in hospital	in hospital to	hospital to in
BERT-long	**to hospital in**	in to hospital	to in hospital	hospital in to	in hospital to

4.3 Experiment Design

To evaluate the performance of NeuralNetspeak compared to Netspeak, both systems are tested on the same sets of queries generated as described in section 4.2.2.

For testing purposes, an instance of NeuralNetspeak was deployed on a server with multiple GPUs of which one was assigned to NeuralNetspeak. During all of our tests, NeuralNetspeak was assigned a single Nvidia GTX 1080, which let us use the language model to its maximum potential compared to running all operations on a CPU.

Since Netspeak already provides an API[10], it could be used to retrieve results for the queries very efficiently. The only change that needed to be made to the queries to be processed by Netspeak's API was to replace the multi-mask and multi-in-word mask operator tokens with a plus sign.

As described in section 4.2.2, there are two different types of queries, namely (1) short queries and (2) full queries. Short queries are limited to 5 words, which is the maximum query length Netspeak can process and are sent to both NeuralNetspeak and Netspeak. Full queries are the complete sentences formed by inserting the short query back into the original sentence. Because they are too long to be processed by Netspeak, they are only sent to NeuralNetspeak and their results are only used to make statements about the improvements achieved by the additional context.

4.3.1 Performance Metrics

We use mainly two metrics to determine and compare the performance of each Netspeak variant.

Recall We measure the number of queries from a dataset whose results contain the expected response as the recall of that dataset. This number is not to be confused with the number of results for a given query. Since the language model used by NeuralNetspeak maps the probability distribution of predicted words to the model's whole vocabulary, the results would eventually include all combinations of vocabulary words at the masked positions.

Average Rank Netspeak and NeuralNetspeak respond to a query with a sorted list of results. This list is sorted by occurrence frequency in the case of Netspeak and by the score assigned by its language model in the case of NeuralNetspeak. Each of the results can be assigned a rank in ascending order,

[10] `https://netspeak.org/help.html#for-developers`

starting with rank 0 for the first result. The average rank measures the simple mean of the ranks of the expected results of a given set of queries. Based on our assumption that the data sets used for testing contain only sentences of high linguistic quality, the goal is to minimize the average rank. The closer the average rank is to zero, the better the results.

These metrics can only be used to compare the results of the short queries from NeuralNetspeak with results for the same queries from Netspeak. Since the full queries cannot be processed by Netspeak, they are only for comparison purposes between short query results and full query results from NeuralNetspeak.

In our tests, we have limited the number of predictions considered at every mask position to 30. We are also only taking the first 100 results for each query into consideration, allowing us to compare the recall with different rank thresholds. In this work, we focussed on thresholds of 5, 10, 20, and 100 results. We call the recall with the threshold n "recall@n" in the following sections.

4.3.2 Result Evaluation

The results from our test runs are evaluated in two ways.

First, we evaluate the results quantitatively using the metrics described in section 4.3.1. We compare the results from NeuralNetspeak and Netspeak for short queries directly, using the recall first to show how many of the queries have delivered the expected result and then the average rank to determine the quality of those results. Next, we use the long queries to see if adding more context to the queries improves the results of NeuralNetspeak. The long queries can't be used to compare the performance of NeuralNetspeak to Netspeak though, as Netspeak can't handle those queries with more than five words.

Lastly, since the recall and average rank metrics only make statements about the presence of the expected result, extracted from the original sentence, and its ranking, we also evaluate selected results qualitatively by determining up to which rank the results are useful. We will concentrate this part of the evaluation on queries that either didn't return the expected result at all or where the expected result was ranked exceptionally low.

Chapter 5

Experiment Results and Discussion

In this chapter we will show and discuss the results of the experiments we conducted. First, we present an overview of the results at different recall thresholds and evaluate the differences in recall and ranking between Netspeak and NeuralNetspeak quantitatively. Next, we compare the performance of both systems on the five whole-word and the two in-word query operators. Finally, we analyze the actual results for a few selected short and long queries in a qualitative evaluation to compare the performance of NeuralNetspeak to Netspeak when not only the expected result from our dataset is considered.

5.1 Quantitative Evaluation

To compare the overall performance of NeuralNetspeak to Netspeak's performance on a macroscopic level, we use the two metrics described in section 4.3.1: (1) recall at different thresholds and (2) average ranking of the expected result. Additionally, we include each system's average execution times for query processing despite NeuralNetspeak not being optimized to its full potential. Since Netspeak is a query-based search engine, this last metric can't be omitted as the execution time is a critical factor for the suitability of a given result retrieval strategy for use in a search engine.

Table 5.1 shows an overview of the results of our experiments for Netspeak and NeuralNetspeak on short queries (denoted as *Netspeak* and *BERT-short* respectively) as well as NeuralNetspeak on long queries (denoted as *BERT-long*). The results from each model for each measure are presented in two groups: (1) grouped by the whole-word operator used in the query, and (2) the micro and macro averages of the results. Results for in-word operator queries are shown separately in Table 5.2 in the same format.

Table 5.1: Experiment results of the 100,000 of the 140,000 queries from the four corpora which only contain whole-word operators, showing average rank, recall and response times for Netspeak, NeuralNetspeak with short queries (BERT-short) and NeuralNetspeak with long queries (BERT-long). The execution time is the time in seconds needed for processing 1,000 queries. The best result of each row in each column is printed bold.

Measure	Model	Operator					Average	
		?	...	#	[ ]	{ }	Micro	Macro
Average Rank	Netspeak	**7.87**	11.65	1.32	1.09	**1.02**	**2.80**	4.59
	BERT-short	15.14	16.43	1.25	1.25	1.19	4.34	7.05
	BERT-long	10.98	**6.63**	**1.12**	**1.09**	1.04	3.21	**4.17**
Recall@5	Netspeak	0.23	0.13	**0.23**	0.44	0.49	0.34	0.30
	BERT-short	0.10	0.08	0.11	**1.00**	**0.88**	0.54	0.43
	BERT-long	**0.26**	**0.25**	0.11	**1.00**	**0.88**	**0.62**	**0.50**
Recall@10	Netspeak	0.26	0.15	**0.23**	0.44	0.49	0.35	0.31
	BERT-short	0.17	0.11	0.11	**1.00**	**0.88**	0.57	0.45
	BERT-long	**0.42**	**0.33**	0.11	**1.00**	**0.88**	**0.65**	**0.55**
Recall@20	Netspeak	0.28	0.17	**0.23**	0.44	0.49	0.35	0.32
	BERT-short	0.34	0.15	0.11	**1.00**	**0.88**	0.61	0.50
	BERT-long	**0.64**	**0.39**	0.11	**1.00**	**0.88**	**0.70**	**0.60**
Recall@100	Netspeak	0.31	0.20	**0.23**	0.44	0.49	0.36	0.33
	BERT-short	0.5	0.20	0.11	**1.00**	**0.88**	0.65	0.54
	BERT-long	**0.76**	**0.41**	0.11	**1.00**	**0.88**	**0.72**	**0.63**
Time [s]	Netspeak	456	**610**	148	260	149	**282**	324
	BERT-short	**115**	729	**97**	**134**	**98**	648	**234**
	BERT-long	173	886	101	138	133	671	286

Average Rank

Netspeak outperforms NeuralNetspeak on both short and long queries when considering the average ranking of the expected result. It is most evident when considering the macro average, which is the mean of the average ranks per operator. This metric does not consider the number of answered queries for a certain operator but weights the performance of each operator as a whole equally. Here, Netspeak is around 35 % better than NeuralNetspeak when both have to process short queries. NeuralNetspeak's performance, however, increases drastically when asked to answer the longer version of the queries, allowing the underlying language model to use more context when processing the query. With this additional context, NeuralNetspeak can outrank Netspeak, ranking the expected result around 10 % higher in total than Netspeak (4.17 vs. 4.59) when weighing the average ranks of each operator equally.

A similar picture is emerging when looking at the micro average. This

Table 5.2: Experiment results of the remaining 40,000 of the 140,000 queries from the four corpora which only contain in-word operators, showing average rank, recall and response times for Netspeak, NeuralNetspeak with short queries (BERT-short) and NeuralNetspeak with long queries (BERT-long). The execution time is the time in seconds needed for processing 1,000 queries. The best result of each row in each column is printed bold.

Measure	Model	Operator		Average	
		? in-word	... in-word	Micro	Macro
Average Rank	Netspeak	**1.03**	**1.08**	**1.06**	**1.10**
	BERT-short	1.27	5.46	3.20	3.37
	BERT-long	1.06	1.83	1.43	1.45
Recall@5	Netspeak	0.49	0.38	0.34	0.43
	BERT-short	0.93	0.74	0.54	0.54
	BERT-long	**0.94**	**0.89**	**0.62**	**0.92**
Recall@10	Netspeak	0.49	0.38	0.35	0.43
	BERT-short	**0.94**	0.81	0.57	0.88
	BERT-long	**0.94**	**0.91**	**0.65**	**0.93**
Recall@20	Netspeak	0.49	0.38	0.35	0.43
	BERT-short	**0.94**	0.86	0.61	0.90
	BERT-long	**0.94**	**0.92**	**0.70**	**0.93**
Recall@100	Netspeak	0.49	0.38	0.35	0.43
	BERT-short	**0.94**	0.92	0.65	0.93
	BERT-long	**0.94**	**0.93**	**0.72**	**0.94**
Time [s]	Netspeak	**220**	**250**	**290**	**230**
	BERT-short	260	2530	630	1400
	BERT-long	270	2630	680	1450

metric is the unweighted mean of the expected result's average rank across all query results. For all three models, the micro average is considerably lower than the macro average, suggesting that all three models have a higher recall for queries whose expected result they also rank the highest. The relative difference between the two averages is by far the highest for NeuralNetspeak on short queries, with a 4.34 micro average and a 7.05 macro average, indicating that the effect is be most pronounced here. NeuralNetspeak on long queries has the least relative difference (only 0.96), suggesting that this model has the most consistent recall across all whole-word operators.

Looking at the performance of each system on the five whole-word query operators individually, it is apparent that all three systems perform exceptionally low on both the single and multi whole-word operators in comparison to all other operators. While Netspeak achieves much better results on single mask operator queries (7.87) than on multi-word mask operator queries (11.65), NeuralNetspeak performs almost exactly the other way round on long queries containing those operators (10.98 for single mask operators and 6.63 for

multi-mask operators). For short queries, NeuralNetspeak ranks the expected results for queries with either one of the two whole-word mask operators nearly equally low (15.14 and 16.43 for single and multi-mask operator queries respectively).

On the other whole-word operators, all three models are head-to-head, with NeuralNetspeak on long queries almost exactly matching Netspeak's performance and outperforming Netspeak on synonym queries (1.12 vs. 1.32). Meanwhile, NeuralNetspeak performs considerably worse when it can only work with short queries with the same operators.

The results for in-word operator queries from Table 5.2 show again, that NeuralNetspeak can almost match Netspeak's performance when supplied with long queries. Both systems perform almost identically on single in-word mask operator queries (1.03 for Netspeak and 1.06 for NeuralNetspeak). However, Netspeak ranks the expected result for the multi in-word mask operator queries higher, resulting in an average rank of 1.08, while NeuralNetspeak ranks the expected result more often lower, resulting in an average rank of 1.83. When only given short queries, NeuralNetspeak falls behind Netspeak, ranking results for single in-word mask operator queries at rank 1.27 and results for multi in-word mask operator queries at rank 5.46 on average.

The results for the individual operators show that NeuralNetspeak can rank the expected results significantly higher if it can use the whole sentence as context, while Netspeak still outperforms NeuralNetspeak in most cases, even if the latter has more context from longer queries.

Recall

Comparing the recall of Netspeak and NeuralNetspeak shows the strength of NeuralNetspeak. It is evident, that NeuralNetspeak answers significantly more queries than Netspeak. NeuralNetspeak achieves an increase in recall of 59 % to 81 % for short queries, at a rank threshold of 5 and 100 respectively, and can improve this even further with long queries to 82 % to 100 % compared to Netspeak at the same rank thresholds. Here, the increase in performance of NeuralNetspeak due to the additional context provided with longer queries is very apparent as well. NeuralNetspeak can answer up to 14 % long queries more than short queries.

Figure 5.1 shows the recall of Netspeak and NeuralNetspeak at different rank thresholds. Note, that the recall of Netspeak is nearly constant with only a slight increase across the different rank thresholds, while the recall of NeuralNetspeak considerably increases with an increasing rank threshold for short and long queries alike. The near-constant recall of Netspeak could on one hand mean that when Netspeak delivers the expected result, it is ranked very

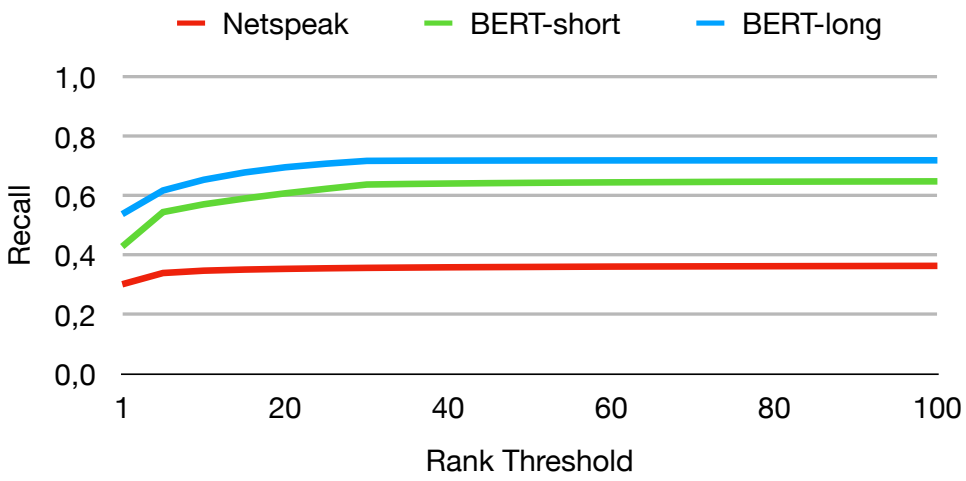
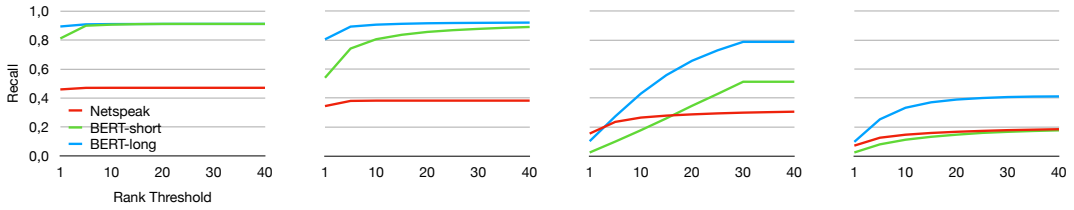


Figure 5.1: A graph showing the recall at different thresholds from 1 to 40 in steps of 5 of Netspeak compared to NeuralNetspeak on short and long queries.

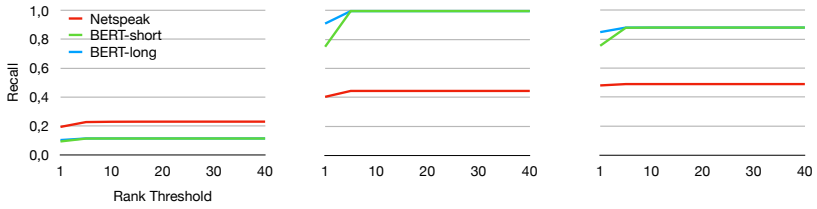
high, but on the other hand, it could also mean that Netspeak doesn't return many results which in turn ranks the expected result high. NeuralNetspeak's increasing recall shows that for some queries, the expected result can be ranked considerably low. We look at the higher-ranked results in the qualitative evaluation to find out if the results that NeuralNetspeak ranks higher than the expected result are sensible as well and the expected result was ranked low because it is specific to the context of the sentence the query was automatically generated from.

The highly increased recall of NeuralNetspeak compared to Netspeak, even when tested only with short queries, shows that we overall succeeded in eliminating the *out-of-vocabulary problem* that Netspeak has due to its n-gram based retrieval strategy.

However, if one looks at Figure 5.2, which shows the recall per operator at different recall thresholds, this conclusion is only partly correct. For both the word alternative and word order operators, NeuralNetspeak performs significantly better than Netspeak, answering almost double the number of queries compared to Netspeak. Here it should be noted that while Netspeak is nearly constant across all rank thresholds, NeuralNetspeak on long queries has a steep increase on recall when rising the rank threshold from 1 to 5. This effect is even more pronounced when tested on short queries. When looking at the results of the single and multi whole-word mask operator query results, a different picture is drawn. For single mask operator queries, Netspeak achieves the highest recall compared to NeuralNetspeak when only considering results where the expected result was ranked first. Allowing the expected result to be ranked lower, leads to a steep increase in recall for NeuralNetspeak, with the recall



(a) Single IW Mask **(b)** Multi IW Mask **(c)** Single Word Mask **(d)** Multi Word Mask



(e) Synonyms **(f)** Alternatives **(g)** Order

Figure 5.2: Graphs showing the recall at different thresholds from 1 to 40 in steps of 5 of Netspeak compared to NeuralNetspeak on short and long queries for each one of the seven query operators. From left to right, top to bottom: (a) Single in-word mask operator, (b) multi in-word mask operator, (c) single whole-word mask operator, (d) multi whole-word mask operator, (e) synonym operator, (f) alternative words operator, (g) word order operator.

for long queries always being higher than for short queries. NeuralNetspeak's recall peaks at a rank threshold of 30, answering more than double the number of short queries compared to Netspeak and even more when more context is given through longer queries. For multi-mask operator queries, NeuralNetspeak dominates Netspeak when tested on long queries. Interestingly, when asked to answer only short queries, NeuralNetspeak answers fewer queries than Netspeak, only catching up when allowing the expected result to be ranked in 35th place. This shows BERT's reliance on context, which for short queries simply don't suffice to predict multiple subsequent words. The only operator, where Netspeak consistently outperforms NeuralNetspeak in terms of recall is the synonym operator, answering around two times the number of queries compared to NeuralNetspeak.

Execution Times

Although NeuralNetspeak is a proof-of-concept software product, we compare its execution times per query with the production version of Netspeak. Since NeuralNetspeak processes the pre-processing subqueries it generates mostly individually (for details, refer to section 4.1.1), there may be a lot of room for improvements. On the one hand, the processing could be parallelized further

and could be distributed across multiple machines. On the other hand, more powerful GPUs could be used to accelerate processing steps where the language model is used even more. As stated in section 4.3, we used a single Nvidia GTX 1080 for our testing. However, at the time of writing this graphics card is already two generations old and there are much more powerful offerings from Nvidia and competitors on the market.

Yet despite these circumstances, NeuralNetspeak achieves almost for every whole-word operator considerably lower result retrieval times. On average across all whole-word operators, NeuralNetspeak is 27 % and 12% faster than Netspeak when answering short and long queries respectively. However, weighing the execution times by the recall of the individual operators shows, that NeuralNetspeak in practice takes 2.3 times as long to answer whole-word queries. Similarly, NeuralNetspeak is almost able to compete with Netspeak on single in-word mask operator queries, answering those queries only 15 % slower compared to Netspeak. On multi-character in-word operator queries, NeuralNetspeak is almost an order of magnitude slower than Netspeak, which makes NeuralNetspeak uncompetitive on this specific operator. The reason for these slow retrieval times is that a large number of matching words in the vocabulary of the language model are found in the pre-processing step of the query. Each of these words is then processed in individual pre-process subqueries. This could be accelerated by scoring these resulting pre-process subqueries using our batch scoring approach. However, this still has to be tested.

Resource Utilization

During our testing on the Nvidia GTX 1080, we saw 20 % to 40 % GPU utilization. When tested locally on an Intel Core i7 9700K eight-core CPU, overclocked to 4.9 GHz on all cores, the CPU utilization approached 80 % to 90 % across all cores while taking approximately twice as long to answer a query compared to the tests run on the GPU server.

Performance differences on common queries

Here, we compare Netspeak to NeuralNetspeak when tested on long queries. Since NeuralNetspeak receives more context than Netspeak through the longer queries, the results can not be used to make statements about NeuralNetspeak's performance compared to Netspeak. However, we still want to show NeuralNetspeak's capabilities in a best-case scenario. Table 5.3 shows the average ranks of the expected results for both models for each of our datasets once by query operator and once on average. In the first four columns, the

Table 5.3: Experiment results of the 44.437 results Netspeak had in common with NeuralNetspeak when the latter was tested with long queries. The best result of each row in each column is printed bold. In-word operators are marked with the postfix *iw*.

Dataset	Model	Operator							Average
		? iw	... iw	?	...	#	[ ]	{ }	Micro
BNC	Netspeak	1.03	**1.18**	**7.56**	9.90	1.30	1.09	1.02	**2.44**
	BERT-long	**1.03**	1.55	10.81	**5.08**	**1.08**	**1.07**	**1.01**	2.55
Europarl	Netspeak	1.03	**1.13**	**7.20**	8.87	1.35	1.09	1.02	2.45
	BERT-long	**1.01**	1.33	8.42	**4.88**	**1.10**	**1.08**	**1.02**	**2.35**
NYT	Netspeak	**1.02**	**1.18**	**7.15**	11.19	1.14	1.08	1.02	**2.68**
	BERT-long	1.07	1.45	10.94	**5.08**	**1.05**	**1.07**	**1.02**	2.80
Wiki	Netspeak	**1.02**	**1.20**	**6.74**	8.82	1.26	1.10	1.02	**2.32**
	BERT-long	1.03	1.31	12.33	**5.63**	**1.08**	**1.07**	**1.01**	2.78
Total	Netspeak	**1.03**	**1.17**	**7.17**	9.67	1.28	1.09	1.02	**2.47**
	BERT-long	1.03	1.41	10.41	**4.14**	**1.08**	**1.07**	**1.02**	2.60

common results from our four datasets (British National Corpus, Europarl, New York Times, and Wikipedia) are shown, while the last column shows the combined results across all four datasets. As marked by the bold numbers in the last column, when supplied with long queries, NeuralNetspeak can outperform Netspeak on the synonym, word alternative, order, and even the multi whole-word mask operator. The results for the latter are most impressive, as NeuralNetspeak ranks the expected result about two times higher than Netspeak. By contrast, Netspeak ranks the expected result for common queries with the single whole-word mask operator significantly higher than NeuralNetspeak, with an average rank of 7.17 for Netspeak and 10.41 for NeuralNetspeak. Also, Netspeak outperforms NeuralNetspeak on both in-word mask operators, with the average rank for single in-word mask operator queries being almost the same across all datasets.

If looking at the datasets individually, it stands out that Netspeak's results are more consistent across the different datasets for each operator but the multi whole-word mask operator. Here, NeuralNetspeak is much more consistent, while having a higher variance in the results for most other operators, especially in the single whole-word mask operator query results.

In total, Netspeak ranks the expected result for most common queries higher, despite ranking the results for four of the seven results lower than NeuralNetspeak. This is mainly due to the unequal recall of common queries for the seven operators.

In conclusion, although NeuralNetspeak does not always match Netspeak's performance in grading the expected result, it instead answers nearly twice

as many queries as Netspeak can answer. Together with the ability to also answer longer queries past the five-word limit of Netspeak, which also improve the results from NeuralNetspeak significantly as expected, and the comparable execution times, this shows that NeuralNetspeak can be a viable supplement to the n-gram-based result retrieval of Netspeak. In future works, more models besides BERT could be evaluated for the use in NeuralNetspeak and further optimizations could be made, potentially allowing NeuralNetspeak to be a partial replacement for the Netspeak backend.

5.2 Qualitative Evaluation

For the qualitative evaluation, we examine the results of selected queries from our dataset. We select the queries based on one of three performance characteristics:

1. Queries whose expected results are ranked significantly higher (better) by NeuralNetspeak compared to Netspeak.

2. Queries whose expected results are ranked significantly lower (worse) by NeuralNetspeak compared to Netspeak

3. Queries whose expected results are ranked significantly higher (better) by NeuralNetspeak when supplied with the additional context of the long version of the query compared to the short query

The first kind of queries should show the improvements our language model-based NeuralNetspeak has over the n-gram based Netspeak. We examine results for queries of the second kind to see if the results NeuralNetspeak ranks higher than the expected result also make sense as independent phrases. The last kind of queries shows how the results change when the additional context from longer queries is supplied to BERT.

Looking at the results for all three kinds of queries with the highest rank differences, it is immediately clear, that queries with the single whole-word mask operator are the only ones with a difference of more than two ranks between NeuralNetspeak and Netspeak. For this reason, we focus only on these queries in this qualitative evaluation. A selection of queries including the ones we evaluate in this section can be found in the appendix in Table A.4, Table A.5, and Table A.6.

Queries where NeuralNetspeak performed better than Netspeak

All in all, the queries where NeuralNetspeak performed better than Netspeak don't follow a specific pattern aside from the presence of the single-word operator. Representative examples for these queries with the expected word in brackets are (1) "based on the `?` *[story]*", (2) "by the `?` *[fire]*" and (3) "`?` *[plan]* to improve".

While NeuralNetspeak almost exclusively suggests nouns for the wildcard operator in the first query, Netspeak also suggests other words like *geographical*, *following*, *same* or *above*. This preference for nouns indicates that BERT has learned to predict nouns following articles.

The results for the second query show a similar picture, although Netspeak now also predominantly recommends nouns. However, apart from idiomatic

expressions like *by the time*, *by the end* or *by the way*, Netspeak suggests many nouns that refer to institutions like *state*, *government*, *department* or *company*. NeuralNetspeak suggests mostly places like *sun*, *sea* and *ocean*, but also people and, ranked further down, also idiomatic expressions like *by the means* or *by the dozens*.

The result for the third query expects a verb. Both Netspeak and NeuralNetspeak correctly identify that a verb is needed at this position in the query and show mostly verbs but also some nouns that build phrases that make sense. All in all, it has to be noted, that for all queries the almost every one of the results NeuralNetspeak provides makes sense as a phrase on their own, while Netspeak often includes phrases in the results that can only make sense in whole sentences. For example, Netspeak includes results like *made to improve* and *, to improve* for the third query.

Queries where Netspeak performed better than NeuralNetspeak

Looking at the second kind of queries, one can recognize a pattern many of the queries follow. It seems that NeuralNetspeak struggles with predicting articles like *the* and *a*. While Netspeak almost always ranks the result with the article the highest, NeuralNetspeak prefers to predict words that give the surrounding words more or additional meaning. This is especially apparent in the query examples (1) "`?` *[the]* person most", (2) "to be in ? *[a]* position" (3) "the station and `?` *[the]*" and (4) "go out of `?` *[the]* room". While NeuralNetspeak tries to describe the subject of the first query (*person*) with adjectives and add more meaning to it, Netspeak mostly finds results which either have an article (e.g. *the* or *a*, the two top results) in front of the subject or quantify it (e.g. with words like *one* or *single*).

The same is true for the second query, with the difference that Netspeak only returns four results. NeuralNetspeak on the other hand is only limited by the threshold of predictions retrieved from BERT and its results for this query are plausible way past the 50th result.

The results for the third query example are even more drastic, as Netspeak only returns a single result. Only the exact expected result happens to be included in Netspeak's n-gram dataset. NeuralNetspeak meanwhile suggests over twenty different places and types of buildings, e.g. *cementery*, *platform*, *park*, and even *viaduct*, before the expected article. Given that this query ends with the single whole-word wildcard operator and an article has to be predicted, the results from NeuralNetspeak are more likely to fit real-world queries.

For the fourth query, Netspeak also only responds with a single result, while NeuralNetspeak's results include pronouns, e.g. *our*, *your*, and *my*, as well as

articles and nouns that add context to the subject of the query, e.g. *control* room, *hotel* room, *bed* room, or *hospital* room.

Queries where NeuralNetspeak performed better on long queries than on short queries

Similar to the first type of queries, the improvements of NeuralNetspeak due to the context of long queries compared to its performance on short queries don't follow a specific pattern. However, it should be noted that in many of the results NeuralNetspeak used the additional context to accurately predict stop words such as articles and other short function words. A few example queries where this is particularly obvious are (1) "`?` *[in]* a situation", (2) "whatever in `?` *[the]* context" and (3) "to `?` *[an]* end".

The first query was generated from the sentence "Especially *in a situation* like this" (the query is written in italic) and requires the word *in* to be predicted. When only given the short query, NeuralNetspeak predicts different verbs like multiple versions of the words define and *describe*, and a few prepositions as well, with the expected word *in* ranked 29th. Having the additional context from the long query, however, enables NeuralNetspeak to rank the expected result in second place and leads NeuralNetspeak to almost exclusively predict other function words fitting the context, e.g. *before*, *during* or *given*.

The second query was derived from the sentence "But while the sobriquet of Canaanite might have meant something [...] years before in Old Testament times it makes no sense *whatever in the context* of the New Testament", with the article *the* being replaced with the single word wildcard operator. Here, NeuralNetspeak ranks only the prediction *any* above the expected article *the*, which arguably makes less sense than the article from the source sentence. Compared to NeuralNetspeak's results for only the short query, however, the results fit the context of the sentence much better, as NeuralNetspeak predicts many quantification words like *one* and *each* again, ranking them higher than the expected article. For comparison: Netspeak returned only the one expected result for the query, which leads to this result being ranked first.

Lastly, the third query was auto-generated from the sentence "Dulé found himself longing this carnage this bloodshed must come *to an end* we must call a truce make a new treaty". The word *an* was replaced with the single word mask operator and has to be predicted. Since the short query doesn't provide any context whatsoever by itself, NeuralNetspeak ranked the expected result in 17th place. Adding the context from the sentence, however, resulted in the expected result to be ranked first.

When asked to predict nouns, NeuralNetspeak also performed better with the context the long queries provide. An example which demonstrates this

quite clearly, is the query "balance between these ? *[factors]*", derived from the sentence "The *balance between these factors* needs further research". When only supplied with the short query, NeuralNetspeak predicts all kinds of different plurals to words which could be balanced, including *dangers*, *extremes*, *resources* and *systems*. Being able to use the whole sentence as context, NeuralNetspeak can predict words that closely correlate with *research*. The expected word *factors* is ranked first, followed by *variables*, *concepts* and *issues*. Netspeak on the other hand, while providing several sensible results, also returns many results that don't fit in the context of the original sentence and ranks the expected result fourth.

This qualitative analysis of query examples shows that instead of relying only on auto-generated queries to evaluate the actual performance of Netspeak and NeuralNetspeak, tests involving real users have to be conducted. However, the results of many different queries give a clear understanding of how a BERT-backed result retrieval strategy compares to the n-gram-based strategy Netspeak uses in terms of context awareness and suggesting results that fit the context of the source sentence. Also, it becomes apparent, that when supplied with additional context from the sentence, NeuralNetspeak can suggest many additional fitting results and rank them high. Meanwhile, Netspeak seems limited by its five-word query-length restriction and thus often predicts many non-fitting words for the context alongside the expected result.

Chapter 6

Conclusion

In this work, we investigated the performance of a retrieval strategy based on the BERT Transformer model to find out if it could be a viable replacement or supplement for a query-based writing assistant. Focussing on the Netspeak writing assistant, our goal was to eliminate the query length limitation and out-of-vocabulary problem the use of an n-gram index imposed on it. We developed a strategy that uses the masked word prediction and word scoring capabilities of a pre-trained BERT language model and created a dataset of 140,000 queries that we auto-generated from corpora with high-quality texts of different domains. Using this dataset, we tested both Netspeak and our NeuralNetspeak for recall and the average ranking of our ground-truth result. Based on our results, we can conclude that the use of language models can positively contribute to existing word search engines like Netspeak, but such a strategy cannot entirely replace existing approaches at this time.

A language model backed result retrieval strategy can be used to enrich results found by the n-gram retrieval strategy or provide results when that strategy couldn't find any matching results. It is especially true for queries comparing different alternatives, like ones including the order or word alternative operator, due to the language model's learned understanding of natural language. When provided enough context, neural language model based query answering can solve Netspeak's out-of-vocabulary problem while also improving results for queries with specific operators, that the n-gram-based retrieval strategy Netspeak uses can answer as well.

However, the results could be different for other language models. For example, the SpanBERT language model by Facebook (Joshi et al. [2020]), trained to predict multiple masked subsequent words, was not yet available at the time of this work. Also, the evaluation based solely on auto-generated queries is not entirely conclusive, as we saw in our qualitative evaluation of the results since it only respects a single result and may differ from queries users

search for.

To eliminate these uncertainties, future work on this topic could include user studies and more extensive qualitative evaluations, e.g. focussing on BERT's behaves for different kinds of queries containing specific types of words. These user studies could also investigate different ways of showing the scores of the results to the user since for most queries the scores NeuralNetspeak calculates for the top results are very close to each other. The scores could be shown on a logarithmic scale or the scores could be mapped to a value between zero and 100, with the former being the score of the lowest-ranked result and the latter being the score for the best result. Also, our query pre-processing could be extended by an additional step to generate sentences from short queries, e.g. with a generative language model such as OpenAI's GPT (Radford [2018]), which then could be used to provide BERT with more context for more accurate results. Finally, a hybrid variant of Netspeak incorporating results from both the n-gram and the language model based retrieval strategies could be built and tested. For such a hybrid system, heuristics would be needed to decide which retrieval strategy to use, e.g. processing all queries longer than five words with the language model-based retrieval strategy while using the n-gram based strategy for short queries including the synonym operator, or even how the results of both strategies could be combined or used to improve each other.

Appendix A

Appendix

A.1 Multi Mask Scoring Strategies

Table A.1: Experiment results of three multi mask prediction strategies. The tests were conducted on the same 4000 queries containing the multi whole-word wildcard operator that were generated from the BNC dataset.

Approach	Relative Recall Short / Long	Average Rank Short / Long	Seconds per 1,000 queries
Combination Scoring	17.00 % / 31.73 %	25.93 / 10.06	845.5
Sequential Scoring	20.27 % / 38.73 %	15.59 / 6.53	2722.5
Batch-Rescore	20.30 % / 38.33 %	16.58 / 6.51	904.5

A.2 Synonym Retrieval Strategies

Table A.2: Experiment results of three synonym retrieval strategies. The tests were conducted on the same 4000 queries containing the synonym operator that were generated from the BNC dataset.

Approach	Relative Recall Short / Long	Average Rank Short / Long	Seconds per 1,000 queries
Individual Scoring	17.8 % / 13.0 %	1.72 / 1.29	355
Batch-Rescore	17.8 % / 12.9 %	2.41 / 1.82	212
Mask Prediction	13.8 % / 12.0 %	1.21 / 1.12	262

A.3 Query Examples

Table A.3: Examples for auto-generated queries from our dataset. We randomly selected two and three queries for each in-word and whole-word operator respectively. The queries for different operators are separated by a horizontal line.

Source Sentence	Short Query	Long Query	Expected Result
Hitchin is celebrating its 2Oth anniversary.	is celebrating i?s 2Oth anniversary	Hitchin is celebrating i?s 2Oth anniversary	is celebrating its 2Oth anniversary
I talked non-stop one day for twenty-three hours.	non-stop o?e day for	I talked non-stop o?e day for twenty hours	non-stop one day for
Please note that the performance (including two intervals) last for $3\frac{1}{2}$ hours.	the performance incl...g two intervals	Please note that the performance incl...g two intervals last for $3\frac{1}{2}$ hours	the performance including two intervals
You can re-read a book — but a lecture is a unique event whose emotional impact can never be successfully reproduced, even by video-recording.	re-read a b...k but a	You can re-read a b...k but a lecture is a unique event whose emotional impact can never be successfully reproduced even by video	re-read a book but a
'A bullet fired at close range into the back of the neck,' Wycliffe said.	fired at ?	A bullet fired at ? range into the back of the neck Wycliffe said	fired at close
Not surprising considering government's fifty billion pound public borrowing requirement.	pound ? borrowing	Not surprising considering government's fifty billion pound ? borrowing requirement	pound public borrowing
Sometimes he thought how easy it would be to let go, to drift.	how ? it would	Sometimes he thought how ? it would be to let go to drift	how easy it would
This placed the employee at a disadvantage.	employee ... disadvantage	This placed the employee ... disadvantage	employee at a disadvantage
Notice how central is his concern for public reputation.	his concern ... reputation	Notice how central is his concern ... reputation	his concern for public reputation
Yes, it looks like it.	Yes it ...	Yes it ...	Yes it looks like it
A symbol is never a mere object.	#symbolisation is never	A #symbolisation is never a mere object	symbol is never
We didn't like the food, we preferred Mum's.	like the #nutrient we preferred	We didn't like the #nutrient we preferred Mum's	like the food we preferred
I'd hate to be responsible for organising an event billed as 'our last chance to save the Earth'.	to be responsible for #coordinate	I'd hate to be responsible for #coordinate an event billed as our last chance to save the Earth	to be responsible for organising
The Ediacaran fossils, however, provide only a brief isolated glimpse of the progress of the invertebrates.	[glimpse glance] of the progress of	The Ediacaran fossils however provide only a brief isolated [glimpse glance] of the progress of the invertebrates	glimpse of the progress of
Nor, I realize now, was she exactly what one would call a liar:.	would [call name] a liar	Nor I realize now was she exactly what one would [call name] a liar	would call a liar
Havel asked the legislature to grant him broader powers to defuse the constitutional crisis.	defuse the [constitutional constituent]	Havel asked the legislature to grant him broader powers to defuse the [constitutional constituent] crisis	defuse the constitutional
Maintain the pre-eminent position of British nursing in the world.	position of { nursing British }	Maintain the pre position of { nursing British } in the world	position of British nursing
Woe betide anyone who was untidy or out of step on one of the marches.	on { one of } the marches	Woe betide anyone who was untidy or out of step on { one of } the marches	on one of the marches
The coroner recorded a verdict of death by misadventure.	{ a recorded } verdict	The coroner { a recorded } verdict of death by misadventure	recorded a verdict

Table A.4: Examples for queries NeuralNetspeak ranked higher than Netspeak. The source sentence is shown with a grey background, with the short query in italics. Both Netspeak and NeuralNetspeak (here BERT-short) were tested with the short query only. For each model, the first five results and the rank of the expected result are displayed. The expected result is printed in bold if it is included in the first five results.

Model	The 5 highest ranked results					Rank
The painting is *based on the* `?` *[story]* recounted in Brother Thomas of Celano's Second Life of St Francis completed in 1247						
Netspeak	geographical	following	assumption	number	results	93
BERT-short	film	movie	novel	book	script	12
His flat was gutted *by the* `?` *[fire]* which burst through windows and roof						
Netspeak	time	end	way	same	state	94
BERT-short	sun	stars	sea	band	authors	15
In one corner a `?` *[blue] computer screen* blips out the latest scores for anyone with good enough eyesight to read the small print						
Netspeak	the	your	a	my	his	77
BERT-short	white	black	view	**blue**	red	4
You've `?` *[got] to be* kidding pal						
Netspeak	is	need	have	seems	going	65
BERT-short	want	wanted	**got**	proud	trying	3
The enclosure movement gave us much of what is said *to* `?` *[be] the* traditional English landscape						
Netspeak	be	make	see	use	get	81
BERT-short	hear	reach	stop	have	watch	27
Olivine for instance is the most unstable mineral in the weathering series and crystallizes at *the highest* `?` *[temperature]* in the reaction series						
Netspeak	quality	level	degree	levels	standards	57
BERT-short	grade	place	**temperature**	rank	altitude	3
Unconcerned that it took him more than a year to prepare for he points out the record `?` *[wait] for a* first speech is 40 years and even Margaret Thatcher took 18 months before she made hers						
Netspeak	looking	,	search	here	up	44
BERT-short	a	**wait**	something	looking	waiting	2
Culture turned completely into commodity must *also* `?` *[turn] into* the star commodity of the spectacular society						
Netspeak	take	taken	entered	takes	come	57
BERT-short	entered	going	went	checked	folded	17
A 4m `?` *[plan] to improve* the Tyneside Metro system including upgrading the trains was approved by councillors yesterday						
Netspeak	like	how	used	order	ways	48
BERT-short	strive	learning	motivation	work	determination	14
Last week I *went to* `?` *[dinner]* at Philippa's						
Netspeak	the	a	see	bed	work	45
BERT-short	sleep	church	school	market	heaven	12
And *all the time* `?` *[it]* was behind the wall						
Netspeak	,	and	!	in	i	26
BERT-short	now	with	**it**	i	you	3
The conventional view of his time was that all species were immutable and *that each* `?` *[had]* been individually and separately created by God						
Netspeak	of	person	one	individual	member	23
BERT-short	person	has	**had**	man	would	3
All sorts are on offer handles in conventional or T form with or `?` a ratchet						
Netspeak	buy	as	in	even	to	21
BERT-short	simply	even	just	**without**	less	4

Table A.5: Examples for queries Netspeak ranked higher than NeuralNetspeak. The source sentence is shown with a grey background, with the short query in italics. Both Netspeak and NeuralNetspeak (here BERT-short) were tested with the short query only. For each model, the first five results and the rank of the expected result are displayed. The expected result is printed in bold if it is included in the first five results.

Model	The 5 highest ranked results					Rank
As for the Brooke sketch as its author I recall that *? [the] person most* upset by it was Tony Benn						
Netspeak	**the**	a	one	of	this	1
BERT-short	strongest	richest	oldest	second	last	30
? [You] can have it back said Lee						
Netspeak	**you**					1
BERT-short	anybody	adam	i	she	everybody	30
Well he's got *to be in ? [a] position* to complete by the time the notice runs out						
Netspeak	**a**	the	that	this		1
BERT-short	emotional	uncomfortable	dangerous	high	comfortable	30
The defect need not be the sole cause *of ? [the] damage*						
Netspeak	**the**	such	or	brain	dna	1
BERT-short	weather	storm	soil	blast	psychological	30
A version of *? [the] problem is* shown in Figure 5.1						
Netspeak	**the**	this	only	my	a	1
BERT-short	his	one	general	proof	a	30
There were big crowds on *the station and ? [the]* Feldwebel and the mousy man kept very close to me						
Netspeak	**the**					1
BERT-short	cemetery	grounds	platform	surrounds	reservoir	30
I feel very strongly that there are a lot of people who don't have the money and have no way *of getting ? [in] touch* with their husband no way of forcing them to pay up						
Netspeak	**in**					1
BERT-short	our	some	thy	that	proper	30
But *they ? [have] strong incentives* to try						
Netspeak	**have**					1
BERT-short	expect	want	gave	provided	introduced	30
As for possible Christian Democratic rivals to Mr Kohl there is only one Wolfgang Schäuble the interior minister and a former *head of ? [the]* chancellery						
Netspeak	**the**	a	state	his	department	1
BERT-short	administration	cabinet	protocol	parliament	finance	29
Pauline was a few years older than Chris who was 31 but they had found mutual support and love and together created the opportunity *? [to] live on their own*						
Netspeak	**to**	who	and	they	not	1
BERT-short	slaves	women	others	wolves	animals	29
Verbal abuse on the streets was commonplace a brick through *? [the] window was* not unusual						
Netspeak	**the**	a	this	every	my	1
BERT-short	whose	which	this	front	another	29
I'm doing *? [a] secretarial course* and want to get in that way						
Netspeak	**a**	the	year	legal	medical	1
BERT-short	financial	undergraduate	professional	student	executive	28
She peered at *? screen again*						
Netspeak	**the**	big	on	login	this	1
BERT-short	white	empty	same	full	off	28
He turned to *go out of ? [the] room*						
Netspeak	**the**					1
BERT-short	this	that	one	another	any	23

Table A.6: Examples for queries NeuralNetspeak ranked higher with the additional context from the long query. The source sentence is shown with a grey background, with the short query in italics. For each model, the first five results and the rank of the expected result are displayed. The expected result is printed in bold if it is included in the first five results.

Model	The 5 highest ranked results					Rank
The water polymers thus have a pattern imposed upon them *a pattern ? [which] is* determined by the substance which is dissolved the solute						
Netspeak	that	**which**	2			
BERT-short	matrix	puzzle	diagram	map	function	29
BERT-long	**which**	it	that	what	this	1
But while the sobriquet of Canaanite might have meant something some two thousand years before in Old Testament times it makes no sense *whatever in ? [the] context* of the New Testament						
Netspeak	**the**	1				
BERT-short	one	every	your	his	this	30
BERT-long	any	**the**	current	its	his	2
During the first year *if you encounter a ? [problem]* with your CompuAdd 200 300 or 400 Series desktop or deskside system which can't be resolved by our Telephone Technical Support Staff we provide On Service in most parts of the country within the next business day						
Netspeak	**problem**	bug	1			
BERT-short	wolf	spider	snake	dinosaur	storm	29
BERT-long	**problem**	problems	difficulty	conflict	issue	1
Especially *? [in] a situation* like this						
Netspeak	**in**	such	of	to	is	1
BERT-short	defining	as	define	describe	describing	29
BERT-long	like	**in**	given	assuming	at	2
The supply of such *information comes from ? [a] variety* of sources within the organization						
Netspeak	**a**	1				
BERT-short	varying	varied	one	another	various	29
BERT-long	any	**a**	an	all	every	2
Leaving home should be a normal part *of ? [young] adult* development but running away or leaving prematurely is an indication that life at home is no longer acceptable or bearable						
Netspeak	the	an	his	my	**young**	5
BERT-short	mature	her	an	woman	all	29
BERT-long	early	**young**	mature	all	the	2
Such a *right should ? [be] made* exercisable by notice given before the earliest date upon which an appointment may be made						
Netspeak	**be**	1				
BERT-short	do	seem	say	she	stay	27
BERT-long	**be**	only	been	not	usually	1
The accounts are third told years *? [after] the events*						
Netspeak	of	to	in	and	about	9
BERT-short	before	between	describing	describes	through	30
BERT-long	before	preceding	during	**after**	between	4
The first stage would be to visit the school and then return to *discuss ? [the] matter again*						
Netspeak	**the**	this	1			
BERT-short	this	our	a	their	each	28
BERT-long	this	said	that	**the**	a	4
Dulé found himself longing this carnage this bloodshed must come *to ? [an] end* we must call a truce make a new treaty						
Netspeak	the	**an**	this	that	its	2
BERT-short	whatever	that	this	any	its	17
BERT-long	**an**	its	our	no	the	1

Bibliography

Jay Alammar. The illustrated transformer. Blog post, 2018. URL `https://jalammar.github.io/illustrated-transformer/`. 2.2, 2.1.2, 2.3, 2.4, 2.5, 2.6, 2.8

Dimitris Alikaniotis and Vipul Raheja. The unreasonable effectiveness of transformer language models in grammatical error correction. *CoRR*, abs/1906.01733, 2019. URL `http://arxiv.org/abs/1906.01733`. 3.1, 3.3

Eric Atwell and S. Elliott. Dealing with ill-formed english text. *Comput. Anal. Engl.: Corpus-Based App.*, 01 1987. 3.1

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. 2.1.3

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016. 2.1

BNC Consortium. British national corpus, XML edition, 2007. URL `http://hdl.handle.net/20.500.12024/2554`. Oxford Text Archive. 4.2

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. 1, 2.2

Karen Jensen, George E Heidorn, and Stephen D Richardson. *Natural language processing: the PLNLP approach*. Boston: Kluwer Academic Publishers, 1993. 3.1

Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans, 2020. 6

Philipp Koehn. Europarl: A parallel corpus for statistical machine translation. In *MT summit*, volume 5, pages 79–86. Citeseer, 2005. 4.2

Bruno Martins and Mário J. Silva. Spelling correction for search engine queries. In José Luis Vicedo, Patricio Martínez-Barco, Rafael Muńoz, and Maximiliano Saiz Noeda, editors, *Advances in Natural Language Processing*, pages 372–383, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30228-5. 3.1

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. volume 2, pages 1045–1048, 01 2010. 1, 2.1

Marcin Miłkowski. Automating rule generation for grammar checkers, 2012. 3.1

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018. URL `http://arxiv.org/abs/1802.05365`. 1, 2.1, 2.2

James L. Peterson. Computer programs for detecting and correcting spelling errors. *Commun. ACM*, 23(12):676–687, December 1980. ISSN 0001-0782. doi: 10.1145/359038.359041. URL `https://doi.org/10.1145/359038.359041`. 3.1

Martin Potthast, Martin Trenkmann, and Benno Stein. Using Web N-Grams to Help Second-Language Speakers. In *Web N-Gram Workshop at SIGIR 2010*, page 49, July 2010. URL `https://web.archive.org/web/20110220062356/http://research.microsoft.com/en-us/events/webngram/sigir2010web_ngram_workshop_proceedings.pdf`. 1, 3.1, 3.1

R. Quirk, R. Quirk, S. Greenbaum, D. Crystal, D.S.E.U.S. Greenbaum, G. Leech, L. Geoffrey, Pearson Education, Pearson Longman, J. Svartvik, et al. *A Comprehensive Grammar of the English Language*. A Comprehensive Grammar of the English Language. Longman, 1985. ISBN 9780582517349. URL `https://books.google.de/books?id=wj9BAQAAIAAJ`. 3.1

A. Radford. Improving language understanding by generative pre-training. 2018. 2.1, 2.2, 6

Evan Sandhaus. The New York Times Annotated Corpus. LDC2008T19. DVD. Philadelphia: Linguistic Data Consortium, 2008. 4.2

Wilson L. Taylor. "cloze procedure": A new tool for measuring readability. *Journalism Quarterly*, 30(4):415–433, 1953. doi:

10.1177/107769905303000401. URL https://doi.org/10.1177/107769905303000401. 2.2

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. 1, 2.1, 2.1, 2.7, 2.9

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016. 2.2.1

Li Zhuang, Ta Bao, Xioyan Zhu, Chunheng Wang, and S. Naoi. A chinese ocr spelling check approach based on statistical language models. volume 5, pages 4727 – 4732 vol.5, 11 2004. ISBN 0-7803-8566-7. doi: 10.1109/ICSMC.2004.1401278. 3.1