Master Thesis

May 29, 2020

Generating Code Documentation from Context Information Using Deep Learning

A StackOverflow Case Study

Philipp Mockenhaupt

of Bern, Switzerland (13-761-135)

supervised by

Prof. Dr. Harald C. Gall

Dr. Pasquale Salza

Adelina Ciurumelea

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

Generating Code Documentation from Context Information Using Deep Learning

A StackOverflow Case Study

Philipp Mockenhaupt

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Philipp Mockenhaupt, philipp.mockenhaupt@uzh.ch

Project period: 01.12.2019 - 31.05.2020

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

A big thank you to Pasquale and Adelina for their support, the meetings were very inspiring and your guidance helped a lot for staying focused on the research goal.

Abstract

The Transformer model structure is state-of-the-art in sequence generation, and BERT-like (Bidirectional Encoder Representations from Transformers) general language models are among the best-performing language models for general natural language processing tasks. We investigate whether these state-of-the-art deep learning model architectures can be used to generate code documentation from context information. We do this by defining StackOverflow content as our context information case study and we create the code documentation oracle from documentation that includes a link to StackOverflow found on GitHub. To understand if the structure of the context information affects the results we use two datasets: one is a simple concatenation of different StackOverflow page parts, whereas the other, in addition, indicates the different parts by a keyword. We compare different deep learning approaches and our results indicate that the state-of-the-art in deep learning in connection with natural language processing can be used to generate code documentation from context information, as we outperform the model structure created for the general natural language processing task. The best-performing model is a Transformer with a BERT encoder additionally trained on the StackOverflow data and equipped with additional vocabulary. This model structure outperforms our baseline model Transformer with classical BERT encoder, producing a 14% higher BLEU (Bilingual Evaluation Understudy) score, 8% higher accuracy and a 1.4% decrease in the word mover distance. An improved performance of the raw Transformer structure in connection with a more structured context information indicates the Transformer can gather context information in a flexible way by focusing on different context information parts based on keyword indication.

Zusammenfassung

Die Transformer Model Struktur ist die aktuell führende Struktur um geschriebene Sätze zu generieren und BERT (Bidirectional Encoder Representations from Transformers) ähnliche Modelle sind unter den besten Modellen für allgemeine, auf Sprachverständnis basierende Aufgaben. Wir untersuchen, ob diese Modelle benützt werden können, um Code Dokumentation auf Basis von Kontext Information zu erstellen. Für diese Aufgabe fokussieren wir uns in einer Fallstudie auf Kontext Information zusammengetragen von StackOverflow und als Ziel Dokumentation benützen wir Code Dokumentation von GitHub Repositories, welche einen Link zu StackOverflow beinhalten. Um zu verstehen, ob die Struktur der Kontext Information die Resultate beeinflusst, benützen wir zwei Datasets: eines besteht aus einer einfachen Aneinanderreihung von Inhalten verschiedener StackOverflow Seitenbereiche, das andere Dataset markiert diese unterschiedlichen Teile mit einem Stichwort, welches den Typ des Inhalts beschreibt. Wir vergleichen verschiedene Deep Learning Ansätze und die Ergebnisse deuten an, dass die aktuell führenden Deep Learning Modelle Code Dokumentation mit Hilfe von Kontext Information erstellen können, da das Modell entwickelt für den allgemeinen Fall in Performance übertroffen werden kann. Das beste Modell ist ein Transformer mit einem BERT ähnlichen Encoder, welcher zusätzlich mit den StackOverflow Daten trainiert wurde und mit zusätzlichem Vokabular ausgestattet ist. Diese Modell Struktur erzielt im Vergleich zum BERT Modell, trainiert für den allgemeinen Fall, einen BLEU (Bilingual Evaluation Understudy) Score welcher 14% höher ist, eine 8% höhere Genauigkeit und eine um 1.4% niedrigere Word Mover Distance. Das auf dem Original basierende Transformer Modell erzielt auf dem strukturierten Dataset eine höhere Performance, dies ist ein Indiz die Transformer Struktur kann Kontext Information flexibel verarbeiten, indem sich das Modell mit Hilfe von Stichworten auf unterschiedliche Kontext Information fokussieren kann.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In recent years, there has been a constant flow of deep learning models that achieved new high scores in multiple natural language processing benchmarks, from the Transformer model structure in 2017 [1], which is heavily based on an attention mechanism that allows it to extract information from different parts of an input sentence in different intensities, to large and heavily-trained general language models that provide contextual representation of words for the general English language case. Depending on the given task, these general language models even outperform the results from human participants [2].

In the software engineering literature, it is an established opinion that code documentation should include context information such as how a code fragment is related to the rest of the system, which problem it tries to solve and in which situation the code fragment is used [3] [4]. Code documentation should not include technical details or simply restate the code in natural language; it should be abstract and focused on context information.

As deep learning in connection with natural language has progressed in recent years and documentation should be based on context information, we investigate whether we can use deep learning to generate code documentation from textual context information. Because context information is very general as a term and the exact context information pieces are very situation-dependent, we want to focus on one specific context information case, StackOverflow [5]. We define StackOverflow content as our context information base and investigate in our StackOverflow case study whether we can use deep learning to generate code documentation from StackOverflow page content. We want to investigate how this problem could be solved with deep learning natural language processing and how different approaches like training a model from scratch, transfer learning in which a general language model provides the understanding of an input sequence, fine tuning existing models and changes in model structures compare to each other. To the best of our knowledge, we are the first to use deep learning to generate code documentation from StackOverflow context information. In the absence of an established reference approach, we compare the results with the state-of-the-art general language model BERT [2]. If we can improve on the results it is an indication that a custom deep learning model can be created for the specific task of code documentation generation from context information. In addition to the research on the model level, we investigate whether a possible aid such as a more structured context information representation affects the results.

The main contributions of this thesis are:

- A dataset created through data mining from StackOverflow and GitHub that serves as training and evaluation data for the models
- A Transformer implementation that can be easily used with different pretrained language models and datasets

- A study of feasibility

We first start with an overview of related work to give a better understanding of how our work is related to the current literature. A subsequent chapter that explains the state-of-the-art deep learning model structures used provides information for a better understanding of the experiment and its results. We then explain the design of our experiment and the concrete research questions. Chapter 5 and 6 explain which steps were necessary to perform the experiment with the available data. The experimental results and our corresponding evaluation complete the experimental part. In the discussion section we summarize the findings and provide a first analysis of the code documentation quality. Finally, we conclude whether we can use deep learning to generate code documentation from context information.

Chapter 2

Related work

In this section we give an overview of the literature in code documentation generation and deep learning in connection with natural language processing. The purpose is to provide information about the development in these fields, as it explains how this thesis is related to current innovations in natural language processing and creates a broader understanding of how the task of automatic code documentation based on context information could be approached.

2.1 Code documentation

As we investigate the generation of code documentation with deep learning models, it is important to understand what code documentation is and what could be the definition of good code documentation. Selic [3] defines the purpose of code documentation as an instruction about how a system is structured, how it works and the design rationale that led to this system. The target audience of these instructions are those who are unfamiliar with the system. Selic [3] describes documentation of software on a code level as foolish, as it only results in keeping duplicated information consistent. Instead, documentation should be made on a design level with a higher level of abstraction, no unnecessary technological details, and a focus on application concepts and requirements. Spinellis [4] describes restating the code's function in English as a disaster, because when the code changes the comment is likely to be left behind, and characterizes useless comments as worse than missing ones. Completeness and consistency are the most important attributes of good code documentation. The results of a survey study among software engineers [6] reveal that out-of-date software documentation still remains useful in many tasks and that software engineers do not maintain documentation. Fluri et al. [7] conclude that newly added code is rarely commented. When new code is commented, it is in most cases on the class and method declaration level. In contrast to Lethbridge et al. [6], the results of Fluri et al. [7] demonstrate that new code is commented in a timely fashion. Kramer [8] mentions that API specification documentation ideally has to fulfill a broad range of requirements from a highly diversified audience. It is impractical to try to satisfy every potential reader as the documentation would be too long compared with the code. Kramer [8] describes trust between API documentation writers and programmers as essential. The research of Parnin et al. [9] indicates that API methods are to a large extent described in blog posts and on question and answer pages like StackOverflow. They propose a new way of carrying out code documentation, not by a central authority but by the crowd of API users who post their knowledge on the web. These mentioned sources indicate that code documentation should be abstract and focused on context information. Code documentation should be at least on the method or class level, not lower.

In our experimental setup we follow these insights by focusing on method and class documentation generation only; we exclude inline comments, which forces the models to adopt a style

based on method and class code documentation.

2.2 Automatic code documentation generation

The task of automatic code documentation generation, as in this thesis, is an active area of research. There are mainly two approaches for this task: either take the code as the input and create the documentation based on the code itself or create the documentation based on natural language from an external information source. Many recent works also focus on generation of artefacts like method or class names as a code summarization proxy [10] rather than generating a complete documentation. Allamanis et al. [10] take as input a code snippet and ask a CNN (Convolutional Neural Network) with an attention mechanism to create a short and descriptive name. Allamanis et al. [10] have the opinion that source code is unambiguous and structured compared to natural language, which is not. They also propose a copy mechanism, which identifies important tokens even if they are not in the training-set vocabulary. The model of McBurney et al. [11] creates a summary of a method by looking at the most important methods in the context of the target method; afterwards, keywords are extracted, which represent the actions these methods have evoked, and these keywords are then used to generate English sentences that describe the usage of the target method.

Guerrouj et al. [12] create a summary of a method or class that is mentioned in a StackOverflow post based on the local context found around it. For this purpose, StackOverflow text near the target identifier is transformed to a n-gram language model, and the most used n-grams are used to create the summary. Haiduc et al. [13] use text summarization techniques to create a method or class summary based on the most frequent word terms used in the comments and identifier names of the target method or class. The authors of [14] use a RNN (Recurrent Neural Network) with an attention mechanism to generate method name sequences based on functional descriptions of the methods. Because method names like getByteList are split to 'get Byte List', the resulting optimization problem is a sequence to sequence. Their model can learn difficult naming conventions that are hard to solve with a rule-based mechanism. In general some of the predictions are different from the ground truth but the created method names are still reasonable. Peng et al. [15] use a knowledge graph to gather and connect information about API elements like methods from different sources. Sources can be the method source code itself, StackOverflow or comments, for example. Text information is converted to keywords, and these keywords, in combination with the plain method source code, represent the input for a sequence to sequence neural network. The target output is existing functional descriptions of the API elements. Iyer et al. [16] use an RNN combined with an attention and memory mechanism to create short descriptions based on source code. In the training phase titles of StackOverflow pages are used as description targets and a code snippet from the same StackOverflow page serves as input. Iyer et al. [16] only consider code snippets from answers that are accepted by the question asker.

To the best of our knowledge, we are the first to link to StackOverflow context information in text form directly with real code documentation for training a deep learning model. In addition our experimental setup is relatively untouched, which means we do not restrict the length of the output code documentation or use a specific documentation style like method names as code documentations.

2.3 Sequence to sequence

As we transform an input sequence that consists of textual context information into a code documentation sequence, the general task is called a sequence to sequence problem. This task can be

executed by various deep learning models specializing in natural language processing. Young et al. [17] conducted a survey about different natural language processing deep learning techniques. They conclude that CNNs are basically n-gram feature detectors but have shortcomings in modeling long distance dependencies. According to Young et al. [17], CNNs are data-heavy models with a large number of parameters, which limits their application if the dataset size rises. They propose that RNNs are based on the idea of processing sequential information and therefore have the ability to capture the sequential nature of language. Young et al. [17] summarize that RNNs can understand long-distance dependencies because of a carried hidden state that represents contextual information; in addition, this hidden state can be optimize with the usage of specific gate unites that decide when and to which amount the hidden state is updated. Young et al. [17] conclude that RNNs are better suited for sequence to sequence than CNNs but RNNs have the problem that the model is sometimes forced to encode information that is not important or the model cannot encode information in a selective way if the input is very long or information-rich. To address this problem, different versions of attention mechanism are added to RNNs, which allow the model to focus on different parts of the input. Young et al. [17] mention the further evolution of the attention mechanism that finally led to the Transformer model, which is mainly based on attention and tackles the problem of sequential processing in RNNs. They conclude that through the Transformer the attention architecture becomes more parallelizable, which leads to reduced training time along with better results in different natural language tasks, including sequence to sequence.

Vaswani et al. [1] originally propose the Transformer model structure, which is based only on the attention mechanism. The model achieves the best results in two language translation tasks, and to show the model can perform well on other tasks, experiments on English constituency parsing are done. The model consists, like an RNN sequence to sequence model, of an encoder and decoder side. The model uses two different attention mechanisms: one is classical self-attention, where the model in each layer can focus on different already-known parts of a sequence that is being processed at the moment, and the other allows the decoder side to focus on specific parts in the encoder results. Vaswani et al. [1] focus on this structure because they want a small computational complexity per layer, a high computational parallelization and a possible short path an information signal has to traverse in the network in the case of a long-range dependency. Dai et al. [18] mention the problem that the Transformer structure works with a fixed-length input, which does not allow it to process large input sequences because of limited computing resources. They propose the Transformer-XL structure, which splits the input sequence into multiple segments processed in a recurrent fashion with a small adjusted classical Transformer model structure. Dai et al. [18] mention a much lower computing time than the classical Transformer and the possibility to capture much longer context information.

Because our input sequences are not significantly long and the classical Transformer model structure is still state-of-the-art, we choose the classical Transformer structure for generating code documentation from StackOverflow context information.

2.4 Transfer learning

Pretrained language models that understand the context inherent in natural language offer many advantages, like reduced training times in downstream tasks compared to a model trained from scratch, and in most cases offer a better language understanding if the dataset in the downstream task is small. Devlin et al. [2] present BERT, a language representation model that is based on the Transformer architecture. The model's token context embeddings are trained in a bidirectional way by guessing masked tokens in sequences. According to Devlin et al. [2], this bidirectional training is one of the main advantages of BERT. The pre-trained BERT model can be

used for many downstream tasks with just one additional output layer and scores state-of-the-art results in eleven natural language processing tasks. The pre-training of BERT is carried out on unlabeled data and the fine tuning is accomplished using labeled data from corresponding downstream tasks. They conclude that extensive unsupervised pre-training is an integral part of many language understanding systems. The masked language model training objective of BERT is confirmed as competitive by Liu et al. [19], but they mention possible improvements. They find BERT is undertrained and propose different changes like longer training times, training on longer sequences and a dynamically changing masking pattern. Their model is called RoBERTa (A Robustly Optimized BERT Pretraining Approach) and achieves state-of-the-art results even compared to all other models published after BERT.

Raffel et al. [20] develop a survey-like framework that compares the performance of different transfer learning setups in sequence to sequence problems. They confirm the Transformer architecture is used in a wide range of natural language processing settings, and they use the classical encoder-decoder Transformer implementation as a base model in their framework. Their main results are that the original Transformer works very well in their text-to-text framework, in fine tuning it is advantageous to update all pre-trained parameters rather than only a fraction of them, the results are better if a larger model is trained for fewer steps than a smaller model on more data, and an ensemble of models outperforms a single model. In general, Raffel et al. [20] conclude that larger models tend to perform better. Radford et al. [21] use a semi-supervised approach for language-understanding tasks. They use the Transformer architecture in a first phase to learn a language model based on a large text corpus; afterwards, for fine tuning, they use a manually labeled dataset where the model has to predict a label for a set of input tokens. Dependent on the task, the method used in the fine tuning step can be modified. Radford et al. [21] observe that each layer in a pre-trained multilayer model contains useful information for solving a target task. Yang et al. [22] use the Transformer-XL model structure to create an autoregressive language model that outperforms BERT on 20 natural language processing tasks. According to Yang et al. [22], the main advantages of their model, called XLNet, compared to BERT are that they are not using the predict masked tokens learning target, which suffers a pretrain-finetune discrepancy and wrongly assumes predicted masked tokens are independent of each other, and their model XLNet can learn bidirectional context through a permutation of the input sequence. This negates one of the main advantages of an autoencoding model like BERT compared to an autoregressive language model, which originally only learns unidirectional content.

The problem of GPU memory limitations in connection with always-increasing language models is investigated by Lan et al. [23]. They propose a light version of BERT called ALBERT (A Lite BERT), which uses matrix factorization and sharing of model parameters across model layers to reduce the number of parameters. ALBERT, compared to the large version of BERT, has 18 times fewer parameters and trains 1.7 times faster. For next-sentence classification tasks a new coherence loss for successive sentences is introduced. Lan et al. [23] propose different ALBERT versions, and their best model achieves state-of-the-art results in multiple benchmarks.

As BERT and its subsequent versions are the actual state-of-the-art for general natural language processing tasks, we choose BERT as the benchmark for evaluating the effect of fine tuning and comparing its results with a model trained from scratch. In addition, we seek to discover how advanced BERT models that achieve higher scores on multiple general natural language processing tasks perform compared to BERT.

Chapter 3

Deep learning for text generation

We use current state-of-the-art deep learning models to generate code documentation from context information. This section explains these models in more detail and thus provides the necessary information for understanding our experiment, which is based on these models.

3.1 General Transformer for sequence to sequence

The Transformer model structure consists of two parts. On the one hand, there is the encoding side, which is traversed first and puts out a vector of float values for each input token from the input sequence. On the other hand, the decoding side creates the output sequence by looking at all created tokens and the vector representation of the input sequence that was created by the encoding side. In the classical Transformer model structure [1], the encoder and decoder consist of 6 layers each.

3.1.1 Encoder side

At the beginning of the first encoder layer an embedding layer converts each token of the input sequence to a vector representation of float values. The number of floats each token is converted to equals the number of float values the encoder will finally output, i.e. the dimension of the model. The embedding layer is basically a lookup table where for every token in the vocabulary of the model a numeric representation is provided; these representations are trained in the regular training phase of the model. The model chooses the corresponding float values according to the input tokens; afterwards, values are added to these floats dependent on the position of the source token in the input sequence. This provides the possibility for the Transformer model structure to understand positional effects in sequences. In the original paper [1] they use a sine and cosine function to determine the positional values based on the position in the sequence. As depicted in Figure 3.1, the positional adjusted embedding values are then fed into three different linear layers, which represent queries, keys and values. In the training phase, the model learns the parameters of these linear layers; in this way, the model learns to ask the right questions (queries), how to answer them (keys) and how to transform potential answers to contextual information (values). In the Transformer model structure, these queries, keys and values are split into eight different sets called heads, which allows the model to jointly attend to different information representations. General speaking, one set of queries, keys and values could focus on a specific set of keywords and another could focus on the general context of the sentence, for example. In the scaled dot

product step, the query and key matrices are multiplied, divided by the square root of the key dimension (the scaling factor) to counter possible small softmax gradients if the key dimensionality is high. These values are then fed into a softmax function, which is multiplied with the value vector. General speaking, the model learns to ask queries; these are then compared to learned key transformations and the result is mapped to learned value transformations. At the end of the scaled dot product step, a linear layer is attached, which carries out a final transformation and converts the calculated values to the overall model dimensionality. This output of the attention mechanism is then summed with the original positional adjusted embedding representation and followed by layer normalization. In layer normalization the mean and standard deviation are computed over the values of an input sample independently of other samples in a batch [24]. Generally speaking, the float representation calculated so far is normalized on its own values. In the next step, the values are fed into a feed forward network with one intermediate layer. In the original paper [1], the authors used 2048 hidden neurons and a ReLU (Rectified Linear Unit) activation function. The output of this feed forward network again corresponds to the overall model dimensionality. This output of the feed forward network is then summed with the values before the feed forward network and followed by layer normalization. The resulting values are the outputs of the encoder. In the original paper [1] the authors stacked this combination of linear layers, scaled dot product and feed forward network six times. The output of the first encoder layer was fed again into query, key and value layers and so on; only the embedding layer with the positional adjustment was omitted.

3.1.2 Decoder side

Compared to the encoder side, the decoder has three additional mechanisms implemented. The first addition is the masking in the first decoder attention block. Compared to the encoder attention mechanism, the output matrix of the query key multiplication is multiplied by a negative mask value if required; as a consequence, the model cannot look at these masked values because of the subsequent softmax function, which turns the negative mask values into zero probabilities. This masking process is mainly used in the training phase where the model is trained on complete output sentences and the masking simulates text generation where the model cannot see future tokens. After the first self-attention block, a second attention procedure takes into account the output of the encoder. The encoder output is put in the key and value linear layers of this second attention block, as depicted in Figure 3.2. Generally speaking, the model learns to transform the encoder output into possible answers it can query on. The calculations are the same as in the encoder-scaled dot product; only the source of the key and value linear layer inputs changes. The third addition compared to the encoder side is the use of an additional linear layer and a softmax function at the end of the last decoder layer. This final linear layer, on the one hand, can transform the output values, and on the other, the dimensionality of the model output is changed to the size of the vocabulary. The softmax function at the end converts the model values per vocabulary token into percentages for this vocabulary token.

3.1.3 Training

In the training phase, the optimization is carried out over the combined structure. The encoder layer transforms the input sequence to a useful information representation for the decoder and the decoder layer uses this information to generate a new sequence. The target of the optimization is to guess the same tokens as in the target output sequence. To avoid overfitting, the original Transformer model structure uses a dropout rate of 0.1 before every summarization and layer normalization step. A dropout rate of 0.1 means 10% of the newly calculated values in the attention and feed forward sub models are ignored in the subsequent summarization step, which

means their values are the same as before the attention mechanism or feed forward neural network passthrough [1].

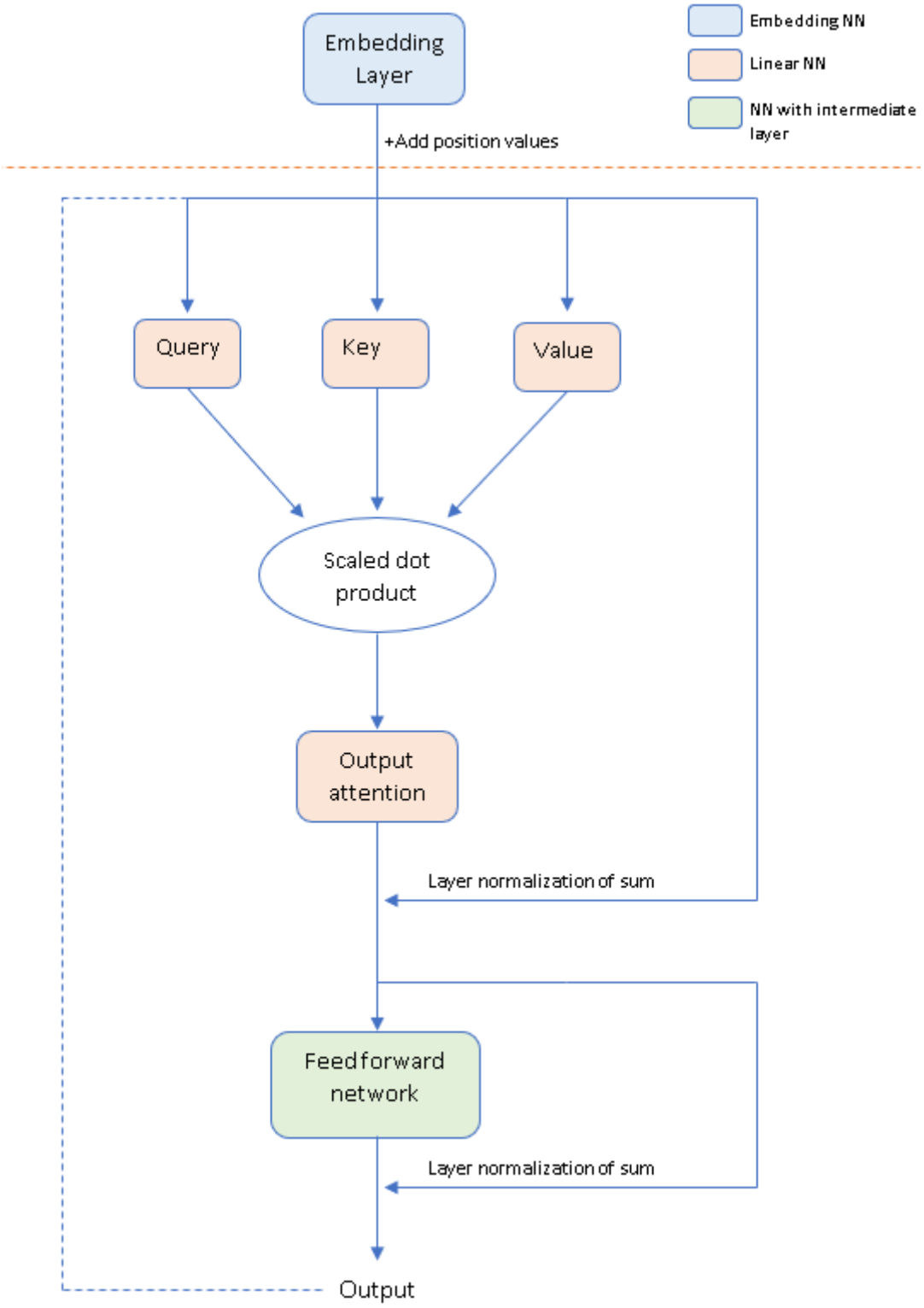

Figure 3.1: Transformer encoder

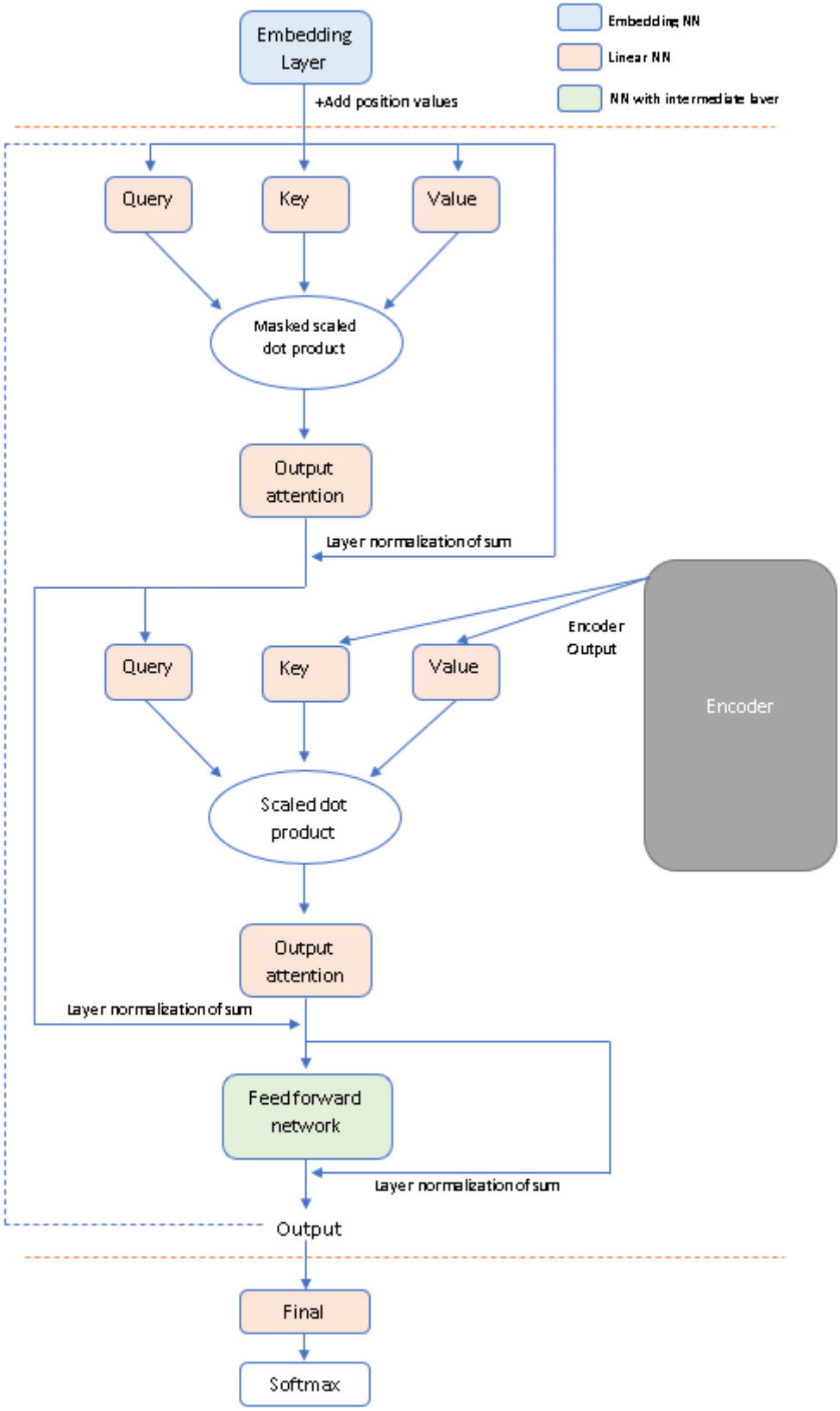

Figure 3.2: Transformer decoder

3.2 BERT language model

The basic architecture of BERT [2] corresponds to a Transformer encoder. Compared to the classical Transformer encoder, BERT has a higher number of layers, a higher context representation dimensionality and more attention heads. In contrast to the Transformer architecture, BERT is designed as general language model that can be used as a base model for a variety of tasks. In 2019 it was added to the Google search engine algorithm [25]. The main idea of this model is to generate contextual representations of tokens based on a large dataset, allowing it to use these values with only one additional layer on top to fulfill tasks like language inference or question and answering.

The pretraining phase consists of two trainings, which are independent of each other but trained at the same time [2]. In masked language modeling training 15% of the input tokens are picked as tokens the model has to predict. Of these 15%, 80% are replaced with a [MASKED] token, 10% are replaced with a random token and 10% keep their original token value. To predict the tokens, an output softmax over the vocabulary is added on top of the Transformer encoder model architecture. As depicted in Figure 3.3, the model has to predict these tokens by only using their hidden state representation, consisting of n float values where n equals the model dimension, i.e. the same output as in the normal Transformer encoder. To achieve this, the model will optimize in the direction where the hidden states represent the context of a token, otherwise there is no possibility to create the link between hidden states and token to predict. The reason only 80% of the tokens to predict are replaced with a masked token is that after training in applications the input sequence will never have the token [MASKED] included; using the real token values in 10% of the cases and random token values in 10%, forces the model in a direction where it does not focus on the [MASKED] string value itself. Because in 20% of the prediction cases it has to predict a token that does not have the [MASKED] string value, it does not know exactly in advance when it has to predict, and the model has to focus on context representations of tokens only. The key advantage of this masked language modeling approach is that BERT can use bidirectional context information. Context information can be gathered from the left and right sequence side. The second training objective of BERT is a sentence classification task, which can be useful for sentence classification or question answering. The input sequence is replaced by a concatenation of two distinct sequences, and a classification token indicates if these two sentences have a contextual relationship. In this thesis, only the masked language modeling results are used. BERT is pretrained on a dataset that consists of the BooksCorpus, which contains the content of over 10000 books (800M words) [26] and English Wikipedia (2500M words). The pretraining lasts 40 runs over the complete 3.3 billion-word corpus, with a batch size of 128000 tokens.

Fine tuning for a specific task can be done in two ways. In the classical way, a task-specific output layer is added on top of the pretrained BERT with a corresponding optimization goal, and as a consequence the BERT parameters are fine tuned in the optimization process. The second possibility is the feature-based approach, where the hidden states calculated by BERT are fed into another model. For this approach, the pretrained BERT can first be further trained on the input data, and later the hidden states from this fine tuned BERT version can be used in the desired model.

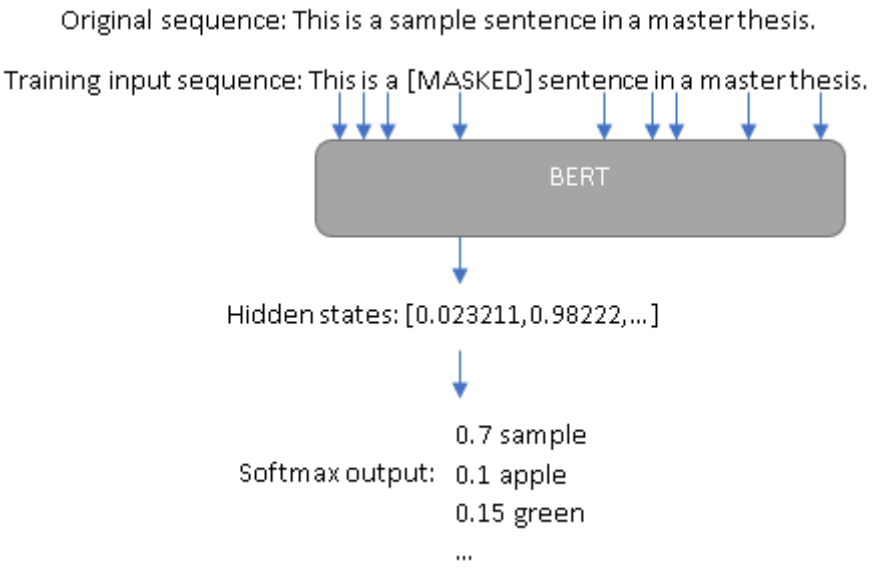

Figure 3.3: BERT language model

3.3 RoBERTa

RoBERTa [19] is the result of a BERT replication study. RoBERTa is trained longer, with bigger batches and more learning data. In addition, RoBERTa only uses masked-language modeling training as the next sentence prediction task is removed; it is trained on longer sequences and uses a changing masking pattern [19]. The overall model structure is the same as in BERT; basically, it is a retraining of the BERT model. The dataset RoBERTa is trained on is 10 times bigger than the dataset used in the training of BERT. The newly-added data comes from news articles, webpages and a dataset in a story-telling style. BERT uses a static masking procedure where, in a preprocessing step, the mask positions are calculated randomly, but the resulting masking positions are used in multiple training epochs without changing them. RoBERTa uses dynamic masking, where masking positions are calculated individually for every sample and every epoch [19].

3.4 ALBERT

ALBERT [23] is a BERT-like model with two main differences. In ALBERT, the initial token embeddings and the hidden states, i.e. context representations, have different dimensions. This is achieved by first projecting the embeddings to a representation with a lower dimension and afterwards projecting them to the same dimension as the hidden states. The intention behind this dimension distinction is that embeddings should be a context-independent representation of a token and the hidden states should represent the context-dependent representation; they have a different representation target. As a result, in the ALBERT base version the dimensionality of the embeddings is 128 compared to 786 in BERT. The second difference in ALBERT compared to BERT is the parameter sharing across layers. In ALBERT every parameter has the same value in each model layer; the overall model structure is basically a repetition of one layer. The base version of ALBERT uses the same dataset for training as BERT [23]. The main advantage of ALBERT is its reduced parameter number and therefore the ability to train longer, on more input data and an overall bigger model size [23]. There exist multiple bigger versions of ALBERT and its top performances are indeed achieved with the biggest models, where ALBERT can play out its optimized structure. In the thesis we use the base version of ALBERT, which only differs in the dimensionality of the embeddings and the parameter sharing across layers compared to BERT. We use the base versions as we run out of memory limits errors in the bigger ALBERT versions even with its optimized structure.

Chapter 4

Experimental design

The thesis answers two distinct research questions with an increased focus on the first one:

RQ1: *"Can we use deep learning to generate code documentation from context information?"*

RQ2: *"Does the structure of the context information influence the deep learning models and their predicted code documentations?"*

RQ1 is interesting for several reasons. First, in recent years, natural language processing in connection with deep learning has gone through a rapid development process, with many new published models and increased performances on broad natural language processing benchmarks, including context understanding [2] [19] [22] [23]. It would be interesting to see how these new published models perform on such a specific downstream task. In addition, the task of creating code documentation can be described as difficult, because depending on the dataset used there could be many interfering forces like different writing styles of software engineers, documentation style dependencies based on the programming language used, and the problem of outdated and incomplete documentation in the training set [6] [7]. It would be interesting to see how a state-of-the-art language model can handle these obstacles. Second, as mentioned in the literature [3] [4] [7], code documentation creation is time-consuming and therefore comes with a cost. A possible automatization would free up time, which could be used for other tasks like refactoring, or in general, human resources needed for producing good documented code could be reduced. Third, as mentioned in several papers [3] [8] [9], good documentation contains context information, which improves the understanding of the purpose and connections to other code entities, and not a repetition of information that is already visible through the code itself. A good working deep neural network that relies on context information only could improve the general documentation quality, because the created code documentation must include context information as this is the only input the model has. Lastly, as the amount of available code snippets in the web is constantly increasing [9] and in many cases these snippets solve urgent problems of a broad base of programmers; these code snippets could be used to generate useful methods on a large scale, and an automatic code snippet documentation tool that creates the documentation based on information surrounding the snippet could provide additional value. The reason for RQ2 is as follows.

The state-of-the-art model structure in sequence generation is the Transformer model structure; although the model can capture context information and corresponding dependencies in the input sequence [1], it is not exactly clear if the model can prefer different parts of an input sequence based on keywords that indicate and differentiate distinct input parts. RQ2 is interesting as the answer could have implications for how model inputs should be structured.

4.1 StackOverflow as context information case study

We use context information from StackOverflow pages as a case study for documentation generation based on context. We choose this setup as StackOverflow pages provide context information in a diversified and comprehensive way. Programmers who post a question on StackOverflow have a specific problem, which they describe with the chosen title and the question body. This description is valuable because it describes what problem a code snippet or method solves, and in this way describes the situation a method is used in. Per definition, a situation description equals context information [27] [28]. Possible answers on the StackOverflow page should provide valuable additional context information about how the problem can be solved.

4.2 Code documentation oracle

We use code documentations from GitHub [29] repositories code files that include a web link to a corresponding StackOverflow page to form the oracle. These code documentations show us how the context information on the StackOverflow page gets converted into an actual code documentation as we assume programmers include a link to StackOverflow in their code documentation, if their code and as a consequence their code documentation is related to the StackOverflow page. This context information into code documentation transformation serves us as the necessary function where we can train a deep learning model on.

4.3 Experimental setup

We create six different Transformer model structures, in each structure a different language model provides the encoding of the StackOverflow context information and a decoder, which is trained individually for every model strucutre, should use this information to generate code documentations similar to the GitHub code documentations in the oracle. The code documentation predictions made by the models are compared to the ground truth in the oracle based on numerical metrics. We train and evaluate the models on self created datasets. According to RQ2 the experimental runs are performed on a structured version of the input dataset and on an unstructured version. As we are investigating deep neural networks for natural language we are only interested in textual representation of context.

4.3.1 Sequence to sequence framework

A Transformer like model structure creates the code documentation in our experiments. We choose this structure because it is state-of-the-art and many lately published language models are based on it in some way. BERT-like models that achieve top positions in well-known language modeling benchmark leaderboards [30] [31] use the Transformer encoder architecture, and even recent architecture improvements like Transformer-XL are not far away from the classical Transformer and are only really preferable in specific situations like large model input sequences in the case of Transformer-XL. We implement the Transfer model structure in Python with the library TensorFlow [32]. The implementation is mainly based on [33]. TensorFlow is a well-known Python library for machine learning and provides support for the CUDA (Compute Unified Device Architecture) Toolkit, which allows users to run Python code on NVIDIA graphic cards [34].

4.3.2 Language models

In five of the six model structures, we use a general pretrained language model as encoder for providing the context representations of the input sequence. The encoder supplies the decoder with the hidden states from the last layer of the corresponding language model. We choose the last layer as there is no general understanding of the layer from which possible context representations should be chosen. In [2], the authors investigate the effect of the chosen layer for one entity recognition task. The last layer achieves the worst result of five possible alternatives, but the difference from the top solution, which is the concatenation of the last four layers is only 1.2 F1 score. For the ease of the solution and in accordance with the general language model idea that the values at the last layer should represent the context of a token, we chose the last layer. In the experiments, the following 5 pretrained language models are used:

- BERT base
- BERT fine tuned on the StackOverflow input data
- BERT fine tuned on the StackOverflow input data and additional 1000 vocabulary tokens
- RoBERTa base
- ALBERT base

We choose BERT as the base model of our experiment as it can be still considered as state-of-the-art, because subsequent models that actually lead the benchmarks [30] [31] still use its basic architecture. We fine tune BERT on the StackOverflow input to see how this additional learning improves the context understanding in the area of programming and software engineering. The fine tuning is executed in line with the feature-based approach by carrying out an additional regular BERT training on our input data. The additional vocabulary is provided to give BERT the possibility to transfer often-used terms in programming that the model does not know from the pretraining to distinct tokens that will improve the ability of BERT to assign these terms a more precise context representation. We create the additional vocabulary by adding the most-used 1000 words in the StackOverflow inputdata training set, which are not represented in the original vocabulary list of BERT, to the vocabulary list. RoBERTa and ALBERT are advanced BERT models with improved scores compared to BERT; ALBERT leads on two important natural language benchmarks [30] [31]. These two models are added because of their improved performance in several benchmarks compared to BERT and potential insights about the effect of BERT model changes on code documentation generation. We use the base versions for all models as the processing times and hardware requirements limit the usage of the larger versions. The models and the fine tuning script are provided by the Hugging Face transformers library [35]. BERT base, RoBERTa base and ALBERT base are implemented with the TensorFlow library. BERT fine tuned uses the PyTorch library [36] and the results are converted to fit the Transformer TensorFlow implementation. In one of the six model structures, a classical Transformer encoder trained from scratch acts as the encoder. To summarize we use six model configurations in our experiments:

Encoder	Decoder
BERT base	Custom decoder
BERT fine tuned	Custom decoder
BERT fine tuned and additional vocabulary	Custom decoder
RoBERTa base	Custom decoder
ALBERT base	Custom decoder
Custom encoder	Custom decoder

Table 4.1: Model configurations

In all six model structures we use word piece tokenizers based on the BERT version provided by the Hugging Face transformers library for initial encoding of the model inputs and the final decoding of the predicted tokens [35]. We use the BERT wordpiece tokenizer as it is using a common word piece tokenization technique, and ultimately the tokenization is not important for the experimental results as long as the sentences are tokenized reliably and without the insertion of additional special tokens. The pretrained language models use their own tokenizers.

4.3.3 Hyperparameters

The focus of the experiment is not to find the best model parameters. For the Transformer decoder, we start with model parameters mentioned in the literature [1] and then reasonable changes are tested with regular experiment runs. We mention only the best parameter setting in the thesis.

4.3.4 Input data structure

According to RQ2, we want to investigate if the structure of the input can help the models to extract context information in a better way. Therefore, we use two different StackOverflow context information input datasets. In one, the StackOverflow question title, the question body and, if a direct link to a StackOverflow answer is available, the linked answer are simply concatenated. In a second dataset, these parts are also concatenated, but the different parts are marked with a corresponding keyword followed by a colon. We insert at the beginning of the question title "Title:", at the beginning of the question body "Question:" and at the beginning of the answer "Answer:". The following StackOverflow input data example illustrates how the StackOverflow context information looks like:

Concatenated *Convert.ChangeType() fails on Nullable Types I want to convert a string to an object property value, whose name I have as a string. I am trying to do this like so: The problem is this is failing and throwing an Invalid Cast Exception when the property type is a nullable type. This is not the case of the values being unable to be Converted - they will work if I do this manually (e.g.) I've seen some similiar questions but still can't get it to work. This is a little bit long-ish for an example, but this is a relatively robust approach, and separates the task of casting from unknown value to unknown type I have a TryCast method that does something similar, and takes nullable types into account. Of course TryCast is a Method with a Type Parameter, so to call it dynamically you have to construct the MethodInfo yourself: Then to set the actual property value: And the extension methods to deal with*

Structured *Title: Convert.ChangeType() fails on Nullable Types Question: I want to convert a string to an object property value, whose name I have as a string. I am trying to do this like so: The problem is this is failing and throwing an Invalid Cast Exception when the property type is a nullable type. This is not the case of the values being unable to be Converted - they will work if I do this manually (e.g.) I've seen some similiar questions but still can't get it to work. Answer: This is a little bit long-ish for an example, but this is a relatively robust approach, and separates the task of casting from unknown value to unknown type I have a TryCast method that does something similar, and takes nullable types into account. Of course TryCast is a Method with a Type Parameter, so to call it dynamically you have to construct the MethodInfo yourself: Then to set the actual property value: And the extension methods to deal with*

We use this additional structured context information version with the intention that different parts could contain different forms of context information. The page title, in many cases, is a precise and short summary of the problem, whereas the question body and potential answers can include more diversified information.

4.3.5 Hardware

We run the individual training phases on a NVIDIA Tesla V100 with 32 GB GPU RAM. To be able to read the datasets and their context float representations completely into regular system memory, we require 50 GB regular system RAM for training the models. To be on the safe side, we reserve a computing time of 48 hours for every training run. For evaluation of the trained models we use a NVIDIA Tesla K80 with access to 12 GB GPU RAM, 4 GB regular system RAM and a reserved computing time of 10 hours.

4.4 Evaluation

As we cannot compare the created documentation from every model configuration in a qualitative way, it would be too time consuming and require an advanced experimental setup, we compare the model configurations and their created code documentations with quantitative metrics. We use three metrics:

- Accuracy
- BLEU score
- Word mover distance

Accuracy is the number of words guessed correctly by the model divided by the total number of words guessed by the model. The correct word is given by the target code documentation, and words are compared on a one to one basis; therefore, the metric is word position-sensitive. The best accuracy value is 1, which means every guessed word appeared at the same position with the same string value in the target code documentation; the worst value is 0, which means no predicted word appeared at the same position with the same string value in the target code documentation.

The BLEU score [37] is a well-known metric for translation evaluation. The model sentence prediction is converted to an n-gram structure and then a ratio is calculated which represents how many of the n-grams are represented in the target code documentation. A n-gram is a sentence slice with n subsequent words, if a sentence is converted to a n-gram structure all n-grams of this sentence are calculated with all n's up to defined n. In our experiments we use an BLEU implementation from the NLTK library [38] and n=4 .The best BLEU score is 1 which means every

n-gram of the predicted code documentation appears in the target code documentation, the worst value is 0 which means no n-gram of the predicted code documentation appears in the target code documentation.

The word mover distance [39] uses word embeddings, which represent semantical meaning, and calculates the distance these word embeddings have to travel in a multidimensional space to equal the word embeddings of the target code documentation. We use a normalized implementation from the Gensim library [40] with a pretrained word embedding model [41]. The best normalized word mover distance is 0, which means the words in the predicted and target documentations have the same word embeddings given the specific word embedding space; the worst value cannot be defined in advance as it is dependent on the word embedding model used, but because of the normalization we expect bad values to be around 1. To develop an understanding of the meaning of distance values in our word embedding model, we compare them with values from sample sentences [42]:

Completely unrelated semantical meaning:
Obama speaks to the media in Illinois Oranges are my favorite fruit Word mover distance: 1.03933
Convert an rgb tuple in an hex string Zero fills a number warning Word mover distance: 0.93504
Check the string is null or empty string Recursive traversal of the target directory Word mover distance: 0.91091

Loosely related semantical meaning:
Computes the factorial of a number Calculates a distance between two points Word mover distance: 0.72264
Check if path exists in object Given an absolute path to a file, returns the filename Word mover distance: 0.69192
Return the number of cores The algorithm will attempt to group cores for same process and not share if not needed Word mover distance: 0.65429

Related semantical meaning:
Merge one dict in another Merge dict :b: into dict :a: Word mover distance: 0.66604
Obama speaks to the media in Illinois The president greets the press in Chicago Word mover distance: 0.56286
Build a delegate to get a property value of a class Build a delegate to set a property value of a class Word mover distance: 0.07135

These sample sentences indeed indicate that a bad word mover distance value in the word embedding model used is around 1.0.

We choose these three mentioned metrics for the following reasons. The accuracy shows in a strict way if the position of predicted words is correct and in general if the correct word is created. The BLEU score points out the similarity of the overall sentence by comparing bigger sentence slices than the accuracy. The word mover distance compares the semantic meaning of words and not the raw string value, which could offer more flexibility in the comparisons of code documentations.

4.5 Possible insights

As BERT-like models are state-of-the-art for language modeling tasks, and there exists no reference approach that uses deep learning for code documentation based on textual context information like the content of StackOverflow pages, the quantitative comparison with BERT as baseline reference can show what is possible with advanced state-of-the-art models, fine tuning a general language model on a specific task or training a model from scratch. If we can improve on the performance of BERT, it is an indication that we can use deep learning for code documentation creation based on context information, as it demonstrates that we can create a specialized deep learning model for this specific task and it outperforms the model created for the general natural language processing case. The results will offer insights about the suitability of transfer-learning for a specific downstream task like code documentation generation. Transfer-learning is an important topic in the area of deep learning and offers multiple advantages such as reduced training costs and reduced dataset size requirements. General language models are created with the idea that they can be easily used for a diversified set of tasks. The experiment will indicate whether the pretrained general language models improve the performance compared to the Transformer trained from scratch, and in addition the fine tuning will reveal whether these extensively pretrained models can still benefit from a relatively small fine tuning procedure. If the results demonstrate a positive effect of transfer-learning, the diversity of possible deep learning models which create code documentation will be increased, and the application of deep learning for a specific and detailed downstream task like code documentation creation will be simplified.

The quantitative metrics can be an indication of the general quality of the created code documentation, although the individual absolute values have to be watched with caution. Especially, the BLEU score is expected to be low, as its design favors translation tasks where one sentence can be compared with a set of possible valid translations, and the BLEU formula penalizes n-grams that do not appear in any of these valid translations in a rough way by multiplying with zero in the geometric mean which sets the BLEU score to zero. We counteract this effect by using a

smoothing function provided by the NLTK library [38] which multiplies in these cases with 0.1 instead of 0.

The second research question can be answered by comparing the results of the same model but with the different input structure. A high difference in the results could indicate that the structure of the input influences the ability of the Transformer model structure to catch context information from different places in the input sequence. The results could indicate that a more structured input can help the model if context information is available in different forms and densities.

Chapter 5

Data preparation

This section provides an overview of how we gathered the data in our StackOverflow context information case study, which preprocessing steps were necessary to improve the quality of the used datasets and which additional preparation steps were required in order to use the datasets in the defined experimental design.

5.1 Dataset source Bigquery

The original StackOverflow pages and GitHub code documentations were provided by the product Bigquery from Google Cloud Services [43]. Bigquery is a database-like service where clients can create tables and query them with different SQL dialects. For testing and research purposes, Bigquery offers different premade datasets, including the content of 2.9 million public, open-sourced licensed repositories on GitHub and a copy of the StackOverflow data dump, [44] based on the version available on 20.03.2019. The two datasets offer multiple subtables, but for our experiment we were mainly interested in:

GitHub contents	
Includes the source code of the corresponding files and repositories.	
Number of rows:	265 million
Size of the entire table:	2.26 Terabyte
Interesting columns:	contents,id

GitHub files	
Contains the file metadata.	
Number of rows:	2.3 billion
Size of the entire table:	319.65 Gigabyte
Interesting columns:	path,id

StackOverflow questions	
Contains the question body of all StackOverflow questions available in the data dump.	
Number of rows:	19 million
Size of the entire table:	29.66 Gigabyte
Interesting columns:	title,body,id

StackOverflow answers	
Contains the answer body of all answers available in the data dump.	
Number of rows:	28.81 million
Size of the entire table:	23.37 Gigabyte
Interesting columns:	title,body,parent_id

The tables GitHub contents and GitHub files can be linked with a file id. Every row in the GitHub contents table contains an id column with the id of the file used, which can be joined with the id in GitHub files. StackOverflow answers can be linked with corresponding StackOverflow questions by the parent_id, which equals the id column in StackOverflow questions. It is worth mentioning that in the GitHub files table, the file id is not a primary key. The same file can occur multiple times but on different repositories because of repository cloning. We can join the two GitHub tables on the file id as we are not interested in the name of the repository, only in the last string values in the path column that corresponds to the programming language used in this file. As this path ending is the same independently of the repository used, we can join the source code of a file with an arbitrary metadata entry for the same file id. The Bigquery product is meant to be used in a big data setting with queries over large tables, although we recognized some limitations. First, the CPU computation time available for a query is correlated with the size of the tables queried on; probably the reason for this is that Google Cloud charges the user based on the table sizes used in a query. Therefore, it is not possible to create a very complex query with many standard or custom SQL functions on a dataset. Second, there is a memory limit that complicates the usage of custom SQL functions on large datasets. Nevertheless, we chose Bigquery as our original data source as the GitHub dataset was created in coorporation with GitHub [45], which is an indication of correct and reliable data; both datasets offer a sufficient size and are provided free of charge.

5.2 Downloading and converting Bigquery data

First we extracted all GitHub source codes which include a possible StackOverflow link with the following SQL query:

```
select distinct c.content, reverse(substr(reverse(f.path),0,
strpos(reverse(f.path),".")))
from `bigquery-public-data.github\_repos.contents` c,
`bigquery-public-data.github\_repos.files` f
where content like "\%stackoverflow.com\%" and c.id=f.id
```

Listing 5.1: SQL query extracting GitHub source codes with StackOverflow link

We chose this simple query because the file id is no primary key in the files table, so we could not query for other columns from the file table as this would have led to too many distinct return

tuples, which finally would have triggered the memory or CPU limit of Bigquery. Even if we had split the original GitHub tables into smaller ones the limits would have been triggered because the limits are reduced in the ratio in which the tables are reduced. The filtering for unique contents or file ids with custom SQL function was also affected by the Bigquery limitations; therefore, we used the simple distinct keyword. The chosen solution allowed us to extract unique source codes and their corresponding programming language via the ending of their file path. For extracting the data from Bigquery the returning table was stored in Google Cloud. The resulting GitHub documentation dataset had a size of 9.42 Gigabytes, which corresponds to 571326 source code bodies. We downloaded the data from Google Cloud and ran it through a method documentation extraction script. We extracted the methods for five different programming languages:

Programming language	Method documentations extracted
Python	6909
Java	3385
Javascript	1641
C#	1749
PHP	699

Table 5.1: Programming language of extracted methods

We chose these languages as they together constitute 85% of the code documentations that were clearly assignable to common programming languages. The number of 14838 extracted method documentations, compared to the original number of 571326 source code bodies, seems low, but there are several reasons for this. First, we extracted only API-like method documentations with a Stackoverflow link that followed a specific pattern and not inline comments or method documentations which were generated with inline comment signs. To find method documentations, we first search for API documentation signs:

Programming language	Signs
Python	""" """, ''' '''
Java, Javascript, PHP	/**,*/
C#	///

Table 5.2: API documentation signs

These signs have to appear with keywords in the direct neighborhood:

Programming language	Keywords
Python	def
Java	access modifiers and (),{}
Javascript	function and (),{}
C#	access modifiers and (),{}
PHP	function and (),{}

Table 5.3: Method keywords

With these heuristics, we only extract method documentation in the direct neighborhood of a method declaration, and the documentation indication corresponds to the ones intended for method documentation. As a consequence, the number of extracted documentations is lower because many StackOverflow links in the GitHub dataset are posted in inline comments or in a more unofficial method documentation style. We only extract method documentations as we recognized a much lower informational value in connection with inline comments, and in general inline comments per se cannot be described as code documentation [3]. Second, in some source code bodies the StackOverflow links are simple html links and not embedded in a program documentation at all. Third, there are many source code bodies which do not have a path ending of a well-known program language, and therefore we exclude them from the investigations. Fourth, we only extract unique method documentations. If a documentation is again used in a different file but, for example, in the same project, we do not extract it. In the next step, we iterated over the extracted documentations and saved the ids of the mentioned StackOverflow questions and answers. In this way we obtained ids of 9827 StackOverflow questions and 4556 StackOverflow answers. These were stored in two separated json files, one for the question ids and one for the answer ids. With these two jsons we created two tables in Bigquery. We then joined the corresponding StackOverflow tables with the id tables on these ids and selected the question title, the question body in the case of a linked question and the question title, question body, answer body in the case of a linked answer. The final step in this download and converting phase was to create two text files that were matched line by line, with the GitHub method documentation in one text file and the StackOverflow input in the other.

5.3 Documentation quality before preprocessing

Before the preprocessing step, the quality of the extracted documentation was not sufficient for our experiments. Some regular encounters were the following.

The StackOverflow link replaced the entire method documentation, as in these full documentations:

"""http://stackoverflow.com/questions/22676/how-do-i-download-a-file-over-http-using-python"

"""See also here: http://stackoverflow.com/questions/34954608/parsing-gxl-in-python/34960625"

"""Adapted from StackExchange: http://stackoverflow.com/questions/377017"

*"/** * #source http://stackoverflow.com/a/383245/805649"*

Unnecessary repetition of signs, as in these cutouts:

*"/********** * Converts an HSL color value to RGB*

*"/**** Implementation Notes for non-IE browsers * ———————————- * Assigning a URL to the href property*

*"/** * Private Methods * ———— * These methods remain accessible only to the ScrollReveal instance,*

*/** * 2015-02-27 * 0412043e043704320440043004490430043504420434043e*

Code inside the method documentations, as in these cutouts:

""" list dict_productdict number=[1,2],character='ab'[{'character':'a','number':1}]

*/**reloadOnSearch: false routeProvider.otherwise{template:' <h2>Whoops!<h2>*

mb_ereg_replace('X',"(xe2x23xa1)";Output:**X)*

regex=True, regex_flags=re.IGNORECASE{'max': 910, 'mean': 287.1212121212121, 'min': 21}"}

For our experiment useless informations, like in these cutouts:

""" That's all I have for now.If you have any questions I'll be happy to answer them :)

** TODO: follow the recipe from http://stackoverflow.com/questions/22639336/#22645327 **

*"/** * Thanks to Jason Bunting via StackOverflow.com * * http://stackoverflow.com/questions/359788/how-to-execute-a-javascript-function-when-i-have-its-name-as-a-string#answer-359910*

Thanks elliotlarson for the solution: http://stackoverflow.com/questions/2345784"*

We think this noisiness compared to classical API documentation styles like Javadoc guidelines [46] comes from the fact many of the source code bodies are from smaller and open source GitHub repositories. They are not necessarily made with the idea in mind that a bigger user base could use them, and therefore less effort is allocated to documentation. In addition, the source codes do not have to be of a finished production quality as the GitHub projects can be still in development status. An interesting observation is that in many cases the StackOverflow link replaces the method documentation completely, with the intention in mind that if the method user wants additional information about the method he/she finds it on the corresponding StackOverflow page but not in the method documentation itself.

5.4 Preprocessing

To allow the models to find the context connection between the StackOverflow context information input data and the method documentation without getting distracted by noise, we had to ensure the dataset was as clean as possible and all unnecessary information was removed. As the StackOverflow pages already had a relatively good quality, we focused mainly on the extracted

method documentations. These are the preprocessing steps we did mainly for the method documentations and their corresponding implication on the dataset size:

Preprocessing step	Method documentations after step
Original Bigquery quality	14383
Filtering code,StackOverflow link,html tags	14289
Filtering sign repetitions and inline comment signs	13104
Filtering meaningless sentences	12074
Cutout meaningless words	12070
Filtering English language recognition	10675
Filtering not code like	9862
Filtering length	9663
Filtering number of words	9463
Filtering duplicates	8348

Table 5.4: Number of method documentations after preprocessing step

As we want only text as input and output, we removed in a first step any parts which were between the <code> and </code> tag. These tags are used in StackOverflow pages to indicate code parts, and in GitHub comments they can also occur in connection with C# documentation. In addition, we removed any parts which appeared in the method documentations after a specific keyword. The keywords were the following:

- @
- :param
- :returns
- :Parameters
- Args:
- args:
- Arguments:
- arguments:
- name=”

We did this because we recognized an increase in code-like writing and in the usage of incoherent sentences after these keywords. This is mainly because after these keywords, method parameters are explained and information is provided in a noisy enumeration-like style. Our observations are in line with general method documentation style guidance, [46] where the first parts of the documentation should be a summary followed by parameter descriptions. We removed StackOverflow links and html tags as they do not provide any information for the model. After these steps, if a method description became empty, it was removed from the dataset.

We removed sign repetitions and words that were attached to such sign repetitions because they do not offer any informational value for the model and would distract the model. In addition, we removed inline comment signs for the same reason. After these steps, if a method description became empty, it was removed from the dataset.

In the "Filtering meaningless sentences" step, we first searched for the 500 most used n-grams for n=1,2,3,4 in the method documentations. We then extracted the ones we thought were meaningless for the model task as they did not provide any informational value. Some examples are:

- Thanks to
- Based on
- Adapted from
- Inspired by
- Found at

Afterwards we calculated the ratio of these useless substrings appearing in every method documentation string and deleted the documentations with a ratio over 0.3. This number provided some room for such substrings as they could be still removed in a later preprocessing step without hurting the structure of the overall sentence too much.

In the "Cutout meaningless words" step, we simply removed any substring from the documentations that was part of the mentioned meaningless n-grams list. After this step, if a method description became empty, it was removed from the dataset.

In analogy to [20] we used the langdetect library [47] to filter out non-English documentations. A method documentation only passed this filtering step if the probability the documentation was based on the English language was greater than or equal to 99%. This filtering had two advantages. First, documentations written in languages other than English were filtered successfully; this is important as the pretrained models we use in the experiments are based on the English language. Second, documentations which still included a code-like or noisy style were filtered out as the langdetect library didn't recognize them as English sentences. The langdetect library uses an algorithm that searches for common language-specific patterns in n-grams of sentences.

To fully ensure the method documentations were similar to common sentences and did not correspond to a noisy code-like style, we deleted method documentations which had a ratio of alphabetical characters, commas and dots lower than 90%.

We filtered out documentations with a string length below 15 as they were, in many cases, leftovers of previous preprocessing steps or meaningless reference texts to StackOverflow links, and we thought a string with such a short length would not contain enough information for the model to create a connection from the StackOverflow input to the corresponding method documentation.

The method documentation should contain at least 4 words. This was an additional length filter step as in the previous filtering individual words with a length higher than 15 characters could still pass. We have chosen 4 words as we thought that was the minimum for expressing a very simple method documentation; we want to avoid cases with just an article, an adjective and a noun for example.

Through this whole preprocessing step, different method documentations could be assigned the same string value, as in the original dataset some documentations only differ in some subwords. We did not want the models to focus on some samples more than others; we therefore removed any duplicates.

The dataset quality after these preprocessing steps can be described as good. The method documentations still contain some substrings which will distract the models, but overall the style can

be described as a clean API summary. The first ten method documentations in the dataset are:

Documentation 1 *also: but we keep the prototype.constructor and prototype.name assignment lines too for compatibility with userland code which might access the derived class in a 'classic' way.*

Documentation 2 *generate lookup hashes for neighbours; for faster/simpler neighborhood lookup from:*

Documentation 3 *deduplicates links (-target-index*

Documentation 4 *Implementation Notes for non-IE browsers Assigning a URL to the h property of an anchor DOM node, even one attached to the DOM, results both in the normalizing and parsing of the URL. Normalizing means that a relative URL will be resolved into an absolute URL in the context of the application document. Parsing means that the anchor node's host, hostname, protocol, port, pathname and related properties are all populated to lect the normalized URL. This approach has wide compatibility - Safari 1+, Mozilla 1+, Opera 7+,e etc. Notes for IE IE >= 8 and 10 normalizes the URL when assigned to the anchor node similar to the other browsers. However, the parsed components will not be set if the URL assigned did not specify them. (e.g. if you assign a.h = "foo", then a.protocol, a.host, etc. will be empty.) We work around that by performing the parsing in a 2nd step by taking a previously normalized URL (e.g. by assigning to a.h) and assigning it a.h again. This correctly populates the properties such as protocol, hostname, port, etc. IE7 does not normalize the URL when assigned to an anchor node. (Apparently, it does, if one uses the inner HTML approach to assign the URL as part of an HTML snippet - However, setting img[src] does normalize the URL. Unfortunately, setting img[src] to something like "javascript:foo" on IE throws an exception. Since the primary usage for normalizing URLs is to sanitize such URLs, we can't use that method and IE 8 is unsupported.*

Documentation 5 *Normalize phone number so it doesn't contain invalid characters This removes all characters besides a leading +, digits and x as described here:*

Documentation 6 *Build a delegate to get a property value of a class.*

Documentation 7 *Build a delegate to set a property value of a class.*

Documentation 8 *Gets a random string of chars length*

Documentation 9 *An OrderedDict with a maximum size. Lifted from*

Documentation 10 *JavaScript function to match (and return) the video Id of any valid Youtube Url, given as input string.*

As mentioned, the sentences are very clean with only some distractions, like "(-target-index" in the third example, which is probably a leftover of a parameter description or a code part. "also: but" in the first example is probably also a leftover from a cutout. As in example two (the substring "from:"), example five ("as described here") and example nine ("Lifted from"), there are still some StackOverflow reference strings left in the overall dataset as they can be very individual and therefore they are hard to catch with an heuristic.

5.5 Final preparations

Before the start of the experiments, we carried out two additional steps. First, we random shuffled the dataset because we wanted to avoid possible learnable patterns triggered by method documentations from the same source code file, which would have appeared one after the other in the dataset or patterns based on time, as in the original Bigquery dataset the data could be ordered based on time, with earlier Stackoverflow pages at the beginning of the table, for example. Sec-

ond, we have split the dataset into a training set, which the models use for learning, and a test set for evaluation of the models. We have chosen a split of 85% - 15% as our dataset with 8348 samples is relatively small and we want to ensure we give the models enough training data. A test set of 1243 samples seems high enough for results where indicative conclusions can be drawn.

Chapter 6

Model parameter optimization

The StackOverflow context information and the GitHub method documentations we have gathered for our experiment differed significantly in size and style from the data used for the original Transformer structure and BERT-like pretrained language models. As a result, we decided to optimize the Transformer model parameters to fit our StackOverflow context information case study.

6.1 Initial model hyperparameters

For the Transformer model structure, we first used the parameters posted in [33], which were based on the original hyperparameters used in the original paper [1] but adjusted for a faster computing performance and lower hardware prerequisites. We used the following configuration at the beginning:

Hyperparameter	Value
Model layers	4
Dimension context representation	128
Hidden units in the feed forward network	512
Attention heads	8
Dropout rate	0.1

Table 6.1: Starting hyperparameters

The hyperparameters in the pretrained language models were given by the models and not changed in our experiments as otherwise the models had to be trained again from scratch. The hyperparameters of the pretrained language models are independent from the decoder side as the pretrained model context representations are only used in key and value linear neural networks as inputs on the decoder side, and these linear transformations ensure the dimension of the encoder output is always transferred to the model dimension of the decoder.

6.2 Hyperparameter fine tuning

During our fine tuning procedure we modified the parameter configurations, ran the experiment in isolation with the changed parameters and then compared the results on the test set. Our main reason for this procedure is as follows. The target of this thesis is to investigate the general applicability of deep learning for code documentation generation based on context information and not to find the best possible parameter configuration for this specific task. We change the hyperparameters with the intention in mind that the model configuration is adapted to the given task, but we still want to see how the general Transformer model architecture performs. Therefore, it is sufficient if we start with the configuration known in the literature and carry out a small hyperparameter fine tuning for the specific task of documentation generation. As we only carry out a small number of hyperparameter fine tuning steps and we start with values which are well-known and proven by the literature, the usage of the test set for this hyperparameter fine tuning procedure should not be a problem. The experimental results should not be affected in their informational value as we do not overfit the hyperparameters on the test set. An additional reason for not using a validation set is our relatively small dataset size. The usage of a reasonable validation set would come with the cost of reducing the training set, and we think the test set should not be reduced with only 1243 samples in it. But as our training set is also relatively small, we think the cost of a validation set exceeds the small value it would provide in our hyperparameter fine tuning procedure.

Intuitively, we thought the number of layers should be lower than four because the training set size of 7100 was small compared to 4.5 million sentence pairs in the English-German training set of the original Transformer [1], and with a high number of layers the model would probably have overfit on the dataset. Therefore, we did experiments with one, two and three layers, and indeed the results improved; the model configuration with three layers achieved the best results. An increase in the number of layers to six decreased the model performance.

As the context representation dimension value, 128, was already relatively low compared to the original Transformer value of 512 dimensions, we conducted experiments with the original value, 512, but the performance decreased. A decrease to 64 did not improve the results either.

We conducted experiments with 2048 hidden units, as in the original Transformer paper. We recognized an improved performance for the models with BERT and RoBERTa as encoder, but overall the tendency was not clear. Experiments with 16 attention heads and four attention heads did not improve the results.

In analogy to ALBERT v2 [48], which was released very recently on 30th December 2019, we decreased the dropout rate to 0.01, and indeed the performance improved over all models. An increase of the dropout rate to a high value like 0.4 decreased the performance.

We did experiments with 50, 100, 200 and 300 training epochs. The best results were achieved with 100 epochs, probably because with lower epoch numbers the model learning time is too low and with a higher number the model overfit on the training set. As the possible batch size is mainly defined by the available GPU RAM and we experienced out of memory problems with higher batch sizes, we chose a constant batch size of 5.

With these hyperparameter fine tuning experiences in mind, we defined the model configuration for the final experiment as follows:

Hyperparameter	Value
Decoder layers	3
Custom Transformer, encoder layers	3
Dimension context representation	128
Hidden units in the feed forward network	512
Attention heads	8
Dropout rate	0.01
Training epochs	100
Batchsize	5

Table 6.2: Final hyperparameters

By investigating the generated code documentations in this hyperparameter fine tuning process, we gained the following insights. We list them as inspiration for further research and will not go into more detail. It seemed the number of hidden units in the feed forward neural networks had a direct impact on the structural complexity of the method documentations created. A higher number of hidden units correlated with the length of the sentences and the sentence complexity. The number of heads defined the thematical complexity of the sentence. In the experiments with four attention heads, the sentences seemed to focus on only one or a few topics. Although the reduction of the dropout rate to 0.01 improved the performance, overall the dropout number seemed to be relatively unimportant for the performance metrics and the general quality of the method documentations.

Chapter 7

Experiments results

After training the models on the training set we evaluate their performance on the test set. It is the first time the final model configurations see this dataset. In Table 7.1 we include the computed metrics for the different models we trained. In terms of overall model structure, BERT fine tuned and BERT fine tuned with additional vocabulary achieve the best results in our experiment. They achieve a BLEU score of around 0.025, an accuracy of around 0.043 and a word mover distance of around 0.71. The three Transformer model structures with pretrained BERT versions BERT, RoBERTa and ALBERT as encoder perform relatively similarly, although overall the results of the ALBERT model structure are the worst of the three. The model structure with original BERT as encoder outperforms model structure RoBERTa in the BLEU and accuracy metrics, but model structure RoBERTa achieves the better word mover distance score. The Transformer structure trained from scratch has by far the worst performance across all 6 tested model structures. The BLEU is below 0.01, the accuracy is around 0.022 and the word mover distance is around 0.76. The impact of the input structure is not clear; the model trained from scratch improves but pretrained models have a lower performance in tendency. Transformer trained from scratch is the only model which improves on all metrics with the structured input. In ALBERT the BLEU and the accuracy are better, but the word mover distance is lower. BERT fine tuned has a better accuracy with the structured input, but the BLEU and word mover distance are lower. In BERT, BERT fine tuned with additional vocabulary and RoBERTa the performance is worse or equal for all three metrics with the structured dataset.

Fine tuning BERT fine tuned and BERT fine tuned with additional vocabulary perform better in every performance metric than the classical BERT. On the concatenated dataset, the BLEU increases by 14%, the accuracy increases by 8% and the word mover distance decreases by 1.4%. The fine tuned versions of BERT provide the decoder with context representations of the words used in the StackOverflow input data that are more in line with their actual meaning in the context of software engineering and programming. Words like “Python”, which in the original BERT model could have the context representation of a snake, or “list”, which has an advanced meaning in the program context, will have an adjusted encoder representation which finally allows the decoder to generate sequences that are more in line with the target method documentations in the GitHub dataset. The performance increases are in line with numbers we expected as the overall parameters in BERT should not change completely through the fine tuning. Only small adaptions to the newly seen fine tuning data should occur, as the overall model architecture, with 110 million parameters in the case of BERT base, is too large to change on a relatively small fine tuning run. The decrease in word mover distance may be slightly lower than expected, but this metric nevertheless seems to be relatively stable.

Model	BLEU	Accuracy	WMD
BERT base concat	0.02526	0.04149	0.71967
BERT base struc	0.02339	0.03904	0.71962
BERT ft concat	0.02597	0.04295	0.71383
BERT ft struc	0.02522	0.04465	0.71758
BERT ft+vo concat	**0.02887**	**0.04490**	**0.70961**
BERT ft+vo struc	0.02377	0.04086	0.70936
Cust.Transformer	0.00717	0.01813	0.76469
Cust.Transformer	0.00889	0.02619	0.75997
RoBERTa base concat	0.02468	0.03846	0.71008
RoBERTa base struc	0.02230	0.03689	0.71760
ALBERT base concat	0.01889	0.03424	0.72171
ALBERT base struc	0.02135	0.03817	0.72390

ft: fine tuned, vo: additional vocabulary, concat: concatenated dataset, struc: structured dataset

Table 7.1: Experiment results, average of 1243 method documentation predictions

Additional vocabulary BERT fine tuned with additional vocabulary outpeforms BERT fine tuned on all three metrics in the concatenated datasets, and over all model and dataset configurations BERT fine tuned with additional vocabulary achieves the best scores in every metric. The additional vocabulary helps the encoder build a more precise context representation for these common terms of the StackOverflow input as otherwise the context representation has to be combined out of subtokens' context representations, which leads to a lower context representation precision overall. This better context representation for common StackOverflow terms leads to a better context representation of the entire input sequence.

Comparison model trained from scratch with pretrained models Compared to the Transformer trained from scratch, the performance metrics increased clearly by using a pretrained language model as encoder. The five model structures with a pretrained encoder achieve on average a BLEU score on the concatenated dataset which is 3.45 times higher, an accuracy which is 2.22 times higher and a word mover distance which is 6.5% lower. These numbers indicate that transfer learning offers clear value in tasks where the dataset set size is too low to train a language encoder from scratch, and a pretrained language model can easily be plugged into the state-of-the-art Transformer sequence generation model structure as encoder.

Comparison BERT-like models As all recent pretrained language models are trained on very large datasets, like the entire Wikipedia content in combination with more than 10000 books in the case of BERT [2], and therefore they should all represent the context of English words in more or less the same way, we thought there could be the possibility that different pretrained language models achieve relatively similar results, especially in the case where the language models are basically different versions of the same model, as in our case with BERT, RoBERTa and ALBERT. Although the results are not completely different, our experiment shows that the model architecture matters for a downstream task like method documentation generation, even if the pretrained language models are all trained on very large and similar datasets. ALBERT, for example, per-

forms worse than BERT in all experiments. The reason for this is mainly that the ALBERT base version is, in our setting, an inflexible version of BERT with more or less the same hyperparameters, trained on the same dataset but with the same parameter values for every layer. RoBERTa's context representations are worse for guessing the correct word compared to BERT as the BLEU score and the accuracy are lower, but overall the semantic meaning of the generated documentation is more similar to the target documentation. The reason could be that RoBERTa provides the context of a word in a more precise way as it is trained on 10 times more data. The decoder is not trained enough to work with this precision, so it cannot predict the exact right word, which leads to a lower BLEU and accuracy, but overall the more precise context representation allows the decoder to learn the context link between the StackOverflow input and the method documentations in the training set in a better way, which finally leads to a decoder that can generate method documentations with a more similar semantical meaning. The difference in the results of BERT, RoBERTa and ALBERT demonstrates that given a specific downstream task, potential pretrained language models first have to be checked on their suitability for the given task as relatively small differences in the model structure or in the training setup of these large pretrained models impact the results.

Structured dataset The Transformer model trained from scratch is the only model that improves on all 3 metrics with the structured input. In addition, the improvements are the largest across all model structures. The BLEU increases by 23.95%, the accuracy increases by 44.5% and the word mover distance decreases by 0.62%. In the cases of the other models, ALBERT has the biggest BLEU increase with 13%, ALBERT again has the biggest accuracy increase with 11.47% and BERT fine tuned with additional vocabulary has a 0.035% lower word mover distance. This result indicates that the Transformer architecture can indeed learn to gather context information in a different way from different input parts based only on simple keyword indications. The poor performance of the pretrained language models in connection with the structured input dataset supports this finding, as in these cases the encoder is decoupled from the overall model target of predicting the correct method documentation; they cannot know that an adjustment for the input structure could increase the performance, as they were trained separately as a general language model without the possibility of seeing the effect of the input structure on the method documentation. In the cases of the pretrained language models, the keywords that indicate different input parts are simply additional words where a context representation has to be calculated without any effect on the context representation of the other words. Ultimately, these keywords are unnecessary information that will distract the decoder, and as a result the performances are slightly lower. In the case of the Transformer trained from scratch, the encoder can understand the effect of the keywords on the predictions as the encoder is included in the optimization procedure.

Dataset size The results indicate that the dataset is too small for a model trained from scratch. In general, a model trained from scratch with the same amount of data as a pretrained language model should have an advantage over a model structure that uses a pretrained encoder because of the flexibility to find additional information beyond simple word context representations, which finally leads to a better performance for the whole model structure while optimizing. As the model trained from scratch performs poorly in our experiment, it is an indication the dataset is too small.

Word mover distance In our experimental setting, the word mover distance seems to be a stable and reliable performance metric. The values do not fluctuate significantly and the changes in the values across models are in line with our initial result expectations. The results slightly increase starting with BERT, the fine tuning and finally the additional vocabulary. RoBERTa, in our setup an advanced version of BERT because of the larger dataset it was trained on, has better results than BERT. ALBERT, in our setup the most restricted BERT version, has the worst performance of all three BERT models. The model trained from scratch falls off because of the limited dataset size. A reason for the higher stability of the word mover distance could be the fact that it is a

comparison of semantic meaning and not a one-to-one string comparison like accuracy and BLEU. It is therefore not so strict regarding individual guesses and therefore shows the performance in a more reliable way.

Performance metrics values Regarding the overall absolute level of the performance metrics, we recognize the BLEU score is very low, but this is in some way expected. As the connection between the StackOverflow model input data and the GitHub method documentations is not as defined and strict as in the case of learnable translations, the chance a predicted n-gram will not appear in the target method documentations is relatively high. As explained in 4.5, this automatically leads to low BLEU values. BERT fine tuned with additional vocabulary as the best model achieves an accuracy of 4.49%. This value is somewhat better, as it means that in 4.49% of the predictions the model can predict the right word, in this case out of a vocabulary with 11144 tokens, where some tokens even have to be combined to generate the correct word. The word mover distance values are also acceptable, as the predictions have a rough similarity, like these examples from chapter 4.4:

Loosely related semantical meaning:
Computes the factorial of a number Calculates a distance between two points Word mover distance: 0.72264
Check if path exists in object Given an absolute path to a file, returns the filename Word mover distance: 0.69192
Return the number of cores The algorithm will attempt to group cores for same process and not share if not needed Word mover distance: 0.65429

The achieved word mover distance values in the experiment indicate that the generated method documentations have at least a slight thematic similarity with the target method documentations.

Our experiments results indicate that RQ1 (“Can we use deep learning to generate code documentation from context information?”) can be answered yes. The state-of-the-art general language model BERT can be fine tuned on the task of code documentation generation from context information and it outperforms its version trained for the general language understanding task. The absolute values of the performance metrics indicate a reasonable similarity between the predicted method documentations and the oracle documentations. RQ2 (“Does the structure of the context information influence the deep learning models and their predicted code documentations?”) can be answered yes, the Transformer trained from scratch performance improves with the structured dataset, the model structures with a pretrained encoder have a worse performance in tendency.

Chapter 8

Discussion

The results indicate that we can use deep learning for method documentation generation from context information because in our case study, with StackOverflow content as context representation, we could use the state-of-the-art sequence generation model structure Transformer in connection with pretrained language models to generate method documentations that achieve reasonable similarity, based on quantitative metrics, to the target documentations gathered from GitHub. In addition, the results demonstrate that fine tuning of the pretrained language models improves the performance of the overall model structure, which is an indication that language models that understand the context of words in the general case can be optimized for a specific downstream task like method documentation. Although our dataset is too small to train a good performing model from scratch, the results with the structured dataset indicate that models trained from scratch could have advantages over the use of pretrained models as they can gather context information in a more flexible way by focusing on different input data parts and processing context information inherent in these different parts in a distinct way by using only keywords in the input data as indications.

To better understand the general quality of the generated method documentations and the potential implications of our results for further research, we analyze some samples generated by the best model, BERT fine tuned with additional vocabulary, on the test set. We use the model structure that was trained on the concatenated dataset structure, as this dataset is meant to be the base setup, whereas the structured dataset mainly served for answering the implication of input data structure. First, we only select generated method documentation, as we want to have an idea about the general structure of the prediction and whether there are any abnormalities. For this purpose we analyze ten randomly chosen method documentation predictions:

Prediction 1 *counts the date of at the 2 or ienumerable. or this is the following so : it returns the sum of the total time as the length in time as the number of elements in an integer*

Prediction 2 *returns the list of nodes according to the given coordinates, all items in the way to calculates the specified position of the start between the second position of the ‘ n ‘ value. [1], ‘ d * * * 2], _ 1] array of for for setting the next coordinates of these method to calculate all its rows, returns the next values to calculate a list of the distances of the next value (a list, null n - arbitrary number of elements, e. g., none, e., none is a list of the same permutations % of the same permutation can be used in for the rows _ sort of the sort to calculate all indices is a list, null (a list)„ _ s permutation’s sorting value), 8 equal to compute all end of the same permutation can be used to calculate the same permutationsig*

Prediction 3 *this code is inspired from :*

Prediction 4 *test whether the axis - aligned box with minimum corner and maximum corner intersects the sphere with the given center and square circle at :*

Prediction 5 *return the size of the terminal as ( ) - works this case i found this for debugging, and will be converted to for the terminal.*

Prediction 6 *check if a function is a function that is defined as one. can be used by default. (first) is a first headers exists in list of default request, so, any list, the first formatted list is the same url and download object that there are no self. _ 20 or node method is the same call to a first key is the attempt on type. to send. if more details, a first or a workaround is the same list. this is the same time. if no and only do some cases are used as self. ('s a loop on that they are used as self. here is used to allow!). here : func _ list is a loop on self. headers, and will be thrown!'s can be check if no scope is a loop. [x, my headers.'could be thrown!'s a lot function a loop. from! and to download is, 2, it is a lot : ' here : ' ' ' ' ' ' function will be done else, that request. import headers ' set to handle c * internet url!'could be called by headers ' check only, which is the headers ' ' ' : ' limit is used as self.'s of query to handle.'s issue is the headers ' method is the headers ' method is used to handle ' scope to handle issues and before headers ' method is used for request.'code is used to set to handle. enable _ 20 (_ none * client url then before headers ' which is used if you will check if needed as a function can be called out, we are very time to handle.'is used as self. make self. print output method is used as a first side of the tree directly exception is used instead of self. make self - delete request so to handle to handle to handle a can be changed to handle added to handle self*

Prediction 7 *populates an array with a single value. comes from*

Prediction 8 *a ' that is integer. 10 the same of column as ' and ' ' ' it ' ' it is no). * ' ' ' and ' ' : ' to index to week to parse the pixel and end of ' axis. parameters np. ndarray ' to names the points. ndarray ' ' ' characters along a) ' characters along a). axis. axis. method. method to ' characters along the pixel the pixel ' for each matrix is tuple ' ', the pixel is number of rows a) ' and ' characters. default function that was found ' and whether that is returned that function that will be passed to add a) ' ' ' ' ' is found : ' is returned from a) '. method. a) ' ' characters of select an same like np. ' ' ' ' ' ' ' characters to a) ' ' ' ' ' characters that are sent from a) ' characters that is not a) ' ' ' ' ' ' ' characters that will be passed to a) ' ' ' ' and index of images (' ('. ' and end of select range ' and parse _ number of elements of columns to only return numpy arrays that is passed to convert the pixel as a) ', ', ', the pixel is tuple to a) to a) ' and that does that is number of ' ' ', ' to only use a) ', that can be passed to only return '. '. '. ' is tuple (select index to only use ' ' (' (', the pixel is tuple x with ', the pixel is 2) ', the pixel is passed to add _ like images that is 3 :) ', optional ', this method from a) ', on in ', a) ', a) ' (x with an is left - tuples function that the, ', ', ', optional : to add a, this ' can be passed to add hash ' ' is tuple ' ' ' ' ', optional. ndarray ' ' ' ' ' is number of rows a) ' can be passed to dom. ndarray ', two rows ' ' ' ', optional, optional, ', ' (int which does 3 ' can be passed to only only use a) ' bool, optional ' characters with ', ' ' ' ' ' ' ' ' ' ' ' ' can only! ndarray ' to the vectors ' int the pixel so that can obtain _ np. method to the, a, ', the data is passed to*

Prediction 9 *check if executable exists. up*

Prediction 10 *check that an ip is a on a simple ip code.*

Overall, the quality of the sentence structure is sufficient. It seems the model could learn the general sentence structure inherent in the English language. The model does not make many grammatical or syntactical errors. There are slices where the created structure equals the one we expect in method documentation sentences. Nouns and verbs are ordered correctly and objects are used in the correct way:

returns the list of nodes according to the given coordinates

return the size of the terminal as

the pixel is passed to add_like images

In some examples, the structure of the predicted sentence is not correct:

counts the date of at the 2 or ienumerable

request so to handle to handle to handle a

is a loop on self. headers, and will be thrown!'s can be check if no

In the first example slice, the model uses "of" and "at" together; the meaning of the sentence is unclear as a result. In the second sample, it repeats the verb three times, and in the third sample slice, "thrown" is used correctly in connection with "will be," but the "!'s" is wrongly used.

Regarding the content of the method documentations, we think it is below the quality of a human writer. The sentences feel grammatically and syntactically more or less correct, but while reading we can notice that concrete content is missing and it is a vague description of an artificial created situation, as in this example from prediction 6:

check if a function is a function that is defined as one. can be used by default. (first) is a first headers exists in list of default request, so, any list, the first formatted list is the same url and download object that there are no self. _ 20 or node method is the same call to a first key is the attempt on type. to send.

Over all ten sample method documentation predictions, it seems the model has problems creating longer documentations that are coherent and precise. In the samples 6 and 8, it used a code-like writing style by using the symbol ' for some graphical formatting, and the created sentences are written in a confusing way. Probably, the model tried to generate a parameter description including possible return values or exceptions, as in sample 8 where first parameters are mentioned, followed by many occurrences of return and at the end many occurrences of the keyword optional. On the other hand, the shorter method documentations predictions 4 and 9 are very readable:

test whether the axis - aligned box with minimum corner and maximum corner intersects the sphere with the given center and square circle at :

check if executable exists. up

It is an interesting fact that nine of the ten prediction examples did not exist in this form in the dataset; this means the model is not simply copying method documentations it has seen before. Only prediction 7

populates an array with a single value. comes from ex

existed in this form in the dataset.

To reach an understanding of how well the model performs in its actual task, creating method documentation based on context information, we investigate three of the ten presented method documentation predictions in more detail. For this purpose, we choose one sample where the model did a good job, one where it failed and one sample where it is not clear if it worked well.

Prediction 9, the positive example:

Stackoverflow input: *Test if executable exists in Python? In Python, is there a portable and simple way to test if an executable program exists? By simple I mean something like the command which would be just perfect. I don't want to search PATH manually or something involving trying to execute it with amp; al*

and see if it fails (that's what I'm doing now, but imagine it's)

Target GitHub documentation: *Find the path for a given program*

Predicted method documentation: *check if executable exists. Up*

This sample shows one problem inherent in our dataset setup. The GitHub documentation contains implicit information that the model cannot learn by mapping the StackOverflow input to the GitHub documentation. The StackOverflow input only talks about a test if an executable exists and not directly about finding the path for a given program; this information was added by the programmers as they are probably using the solution posted on the StackOverflow page in a way different from the one the StackOverflow page suggests. These different use cases, or information in the method documentation which is not connected to the StackOverflow input, cannot be predicted by the model as the model only gets the StackOverflow input data. In this way, the learning process of the function, which maps the context input to a method documentation that is based on context, gets distracted as the model gets a noisy learning output that is not based on context information only. The model cannot learn to create a method documentation based on context as the learning output is not entirely based on context information. We declare this sample as good, as in this case the model did not become distracted by possible implicit information in GitHub documentations. It simply reproduces the main context information inherent in the StackOverflow input, namely is there a test if an executable exists. In addition, it rephrases this information in a way that is suitable for a method documentation; the question-like style is removed and replaced with a typical programming term, "check".

Prediction 3, the negative example:

Stackoverflow input: *Converting datetime.date to UTC timestamp in Python I am dealing with dates in Python and I need to convert them to UTC timestamps to be used inside Javascript. The following code does not work: Converting the date object first to datetime also does not help. I tried the example at this from, but: and now either: or does work. So general question: how can I get a date converted to seconds since epoch according to UTC?*

Target GitHub documentation: *Convert a datetime object to unix UTC time (seconds since beginning). It wants 'from __future__ import division', but that caused issues in other functions, automatically covertting what used to produce integers into floats (e.g. "50/2"). It ms to be safe to not use it, but leaving this note just in*

Predicted method documentation: *this code is inspired from :*

This sample indicates that the dataset still contains many StackOverflow reference texts in the target method documentations, even after our cleaning of useless n-grams in the top 500 most used n-grams mentioned in chapter 5.4. The model ignores the context information of the StackOverflow input and simply writes this common reference term, which appears in different forms across the dataset.

Prediction 2, the neutral example:

Stackoverflow input: *How does the Euclidean Algorithm work? I just found this algorithm to compute the greatest common divisor in my lecture notes: So is the remainder when dividing into (get the mod). Then is assigned to and the remainder is assigned to and is returned. I can't for the life of my see how this*

works! And then, apparently this algorithm doesn't work for all cases, and this one must then be used: I don't understand the reasoning behind this. I generally get recursion and am good at Java but this is eluding me. Help please?

Target GitHub documentation: *Greatest common divisor(GCD) of two integers X and Y is the largest integer that divides both X and Y. Algorithm If X = 0 then GCD(X,Y) = Y, as GCD(0,Y) = Y. If Y = 0 then GCD(X,Y) = X, as GCD(X,0) = X. Write X in quotient remainder form (X = Y Q + R). Find GCD(Y,R) using the Euclidean algorithm since GCD(X,Y) = GCD(Y,R).*

Predicted method documentation: *returns the list of nodes according to the given coordinates, all items in the way to calculates the specified position of the start between the second position of the ' n ' value. - [1], ' d * * * 2], _ 1] array of for for setting the next coordinates of these method to calculate all its rows, returns the next values to calculate a list of the distances of the next value (a list, null n - arbitrary number of elements, e. g., none, e., none is a list of the same permutations % of the same permutation can be used in for the rows _ sort of the sort to calculate all indices is a list, null (a list)„ _ s permutation's sorting value), 8 equal to compute all end of the same permutation can be used to calculate the same permutationsig*

This sample shows a limitation inherent in the StackOverflow input data. On StackOverflow, users often use code to solve a specific problem or to communicate with each other. As a result, these code parts include important context information necessary for fully understanding the context, although in this sample the context can be understood by the code surroundings alone. We would describe the context as calculating greatest common divisor with the Euclidean algorithm, the algorithm is difficult to understand even with experience in programming and its concepts. The link between the input and the target method documentation can be described as sufficient; the target documentation mentions the Euclidean algorithm and greatest common divisor. The style of the target method documentation can be described as problematic, as the middle part is hard to understand without further information and it is written in a mathematical style. This could be a reason why the predicted method documentation is not closely related to the context of the StackOverflow input. The method documentation talks about a distance calculation between points where the coordinates are provided by a list and are ordered in a permutation-like style. The model probably understood Euclidean and created an association with the Euclidean distance. In general, the style of the predicted method documentation can be described as good, as the model tried to describe the return values, referred to the method and tried to describe the general procedure of the calculations.

Already, these three samples show the importance of the dataset. To improve on our results, or in general in the case of generating method documentation based on context information with deep learning, a further cleaning of the dataset should be carried out. Although we are happy with the quality of the extracted GitHub documentations compared to the poor quality before the preprocessing, we think there are two dataset obstacles to which close attention should be paid in further research. First, the target method documentation should include as little implicit project and programmer knowledge or other unnecessary information that is not connected with the context input as possible. This information does not offer any informational value for the model and prevents it from learning the function between context input and method documentation based on context only; the link between the context input and the method documentation based on context information should be as clean as possible. Second, for better results additional cleaning should be carried out; although we removed the most-mentioned reference texts to StackOverflow in our dataset, there are still many left. In addition, it would be advantageous to remove any sign except "." and "," from the model vocabulary or from the dataset. We removed samples with a high number of them, but the model still learnt to use them in a repetitious and confusing style. Besides the cleaner dataset, the implication of a larger dataset would be an interesting investigation. The poor performance of the Transformer trained from scratch demonstrates that there is

still considerable improvement potential, as its encoder structure is not different like those of the pretrained models, the only advantages the pretrained models have is the much bigger training set. The fact that we achieved the best results with three model layers instead of six in the original Transformer, [1] or 12 in the case of BERT base, [2] is another indication that the dataset used is relatively small as the dataset was not large enough to use larger model structures without overfitting. With a larger dataset, the higher number of layers should make it possible to fulfill the task of method documentation based on context information in a more comprehensive, and at the same time more precise, way.

Our quantitative results indicate that the current state-of-the-art in deep learning models for natural language processing can be fine tuned on the task of method documentation based on context information and they beat their model versions for the general case. This is an indication that deep learning is capable of fulfilling this task as obstacles like small downstream task datasets can be avoided and the pretrained models are able to adapt to the given task. Studies concerning the quality of the generated code documentation should be done, because quality in all its facets is hard to capture with relatively simple quantitative metrics.

Chapter 9

Conclusion

We investigated the use of context information to generate code documentation with deep learning models. For this, we trained and evaluated six different models based on the Transformer model structure. In five models we used a BERT-like pretrained language model as encoder and one model was trained from scratch.

Based on our case study results with StackOverflow content as context information representation and GitHub method documentations, which show how context information gets transformed in actual method documentations, we conclude that deep learning can be used to generate code documentation from context information. Pretrained language models can be plugged into the state-of-the-art model structure for sequence generation, Transformer, and the overall structure can be adapted for the method documentation task with only some small hyper parameter changes. The pretrained language models can be fine tuned on the task of method documentation generated from context information by individual additional learning on the context information base and providing specific vocabulary, which improves their context understanding in the area of software engineering. The fine tuned language models achieve better results than their classical counterparts. This is an indication that a customized deep learning model for this specific task can be created, as we already outperform the state-of-the-art model that was created for the general language understanding task. Our results indicate that the Transformer structure can be flexible in the way context information is gathered as the model trained from scratch improves by considering simple keywords that indicate different parts of context information; this flexibility is another indication that deep learning can be used for very specific downstream tasks like code documentation.

The investigated model structures achieve a reasonable similarity based on quantitative metrics compared to human-generated code documentations, although for a final conclusion regarding the quality of the generated code documentations, qualitative studies should be undertaken. To improve on our results, further improvements in dataset quality and quantity should be considered.

Bibliography

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[3] B. Selic, "Agile documentation, anyone?," *IEEE software*, vol. 26, no. 6, pp. 11–12, 2009.

[4] D. Spinellis, "Code documentation," *IEEE software*, vol. 27, no. 4, pp. 18–19, 2010.

[5] "Where developers learn, share, and build careers." https://stackoverflow.com/. [Online; accessed 2020-05-28].

[6] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE software*, vol. 20, no. 6, pp. 35–39, 2003.

[7] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, pp. 70–79, IEEE, 2007.

[8] D. Kramer, "Api documentation from source code comments: a case study of javadoc," in *Proceedings of the 17th annual international conference on Computer documentation*, pp. 147–153, 1999.

[9] C. Parnin and C. Treude, "Measuring api documentation on the web," in *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, pp. 25–30, 2011.

[10] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*, pp. 2091–2100, 2016.

[11] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 279–290, 2014.

[12] L. Guerrouj, D. Bourque, and P. C. Rigby, "Leveraging informal documentation to summarize classes and methods in context," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 639–642, IEEE, 2015.

[13] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*, pp. 35–44, IEEE, 2010.

[14] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S.-W. Lin, "A neural model for method name generation from functional description," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 414–421, IEEE, 2019.

[15] X. Peng, Y. Zhao, M. Liu, F. Zhang, Y. Liu, X. Wang, and Z. Xing, "Automatic generation of api documentations for open-source projects," in *2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)*, pp. 7–8, IEEE, 2018.

[16] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083, 2016.

[17] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *ieee Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[18] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," *arXiv preprint arXiv:1901.02860*, 2019.

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

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

[21] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

[22] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," in *Advances in neural information processing systems*, pp. 5754–5764, 2019.

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

[24] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[25] P. Nayak, "Understanding searches better than ever before." https://www.blog.google/products/search/search-language-understanding-bert/. [Online; accessed 2020-05-28].

[26] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the IEEE international conference on computer vision*, pp. 19–27, 2015.

[27] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *International symposium on handheld and ubiquitous computing*, pp. 304–307, Springer, 1999.

[28] A. Zimmermann, A. Lorenz, and R. Oppermann, "An operational definition of context," in *International and Interdisciplinary Conference on Modeling and Using Context*, pp. 558–571, Springer, 2007.

[29] "Build software better, together." https://github.com/. [Online; accessed 2020-05-28].

[30] "Glue benchmark." https://gluebenchmark.com/leaderboard. [Online; accessed 2020-05-09].

[31] "The stanford question answering dataset." https://rajpurkar.github.io/SQuAD-explorer/. [Online; accessed 2020-05-09].

[32] "Tensorflow." https://www.tensorflow.org/. [Online; accessed 2020-05-09].

[33] "Transformer model for language understanding tensorflow core." https://www.tensorflow.org/tutorials/text/transformer. [Online; accessed 2020-05-09].

[34] "Cuda toolkit." https://developer.nvidia.com/cuda-toolkit. [Online; accessed 2020-05-09].

[35] "Transformers." https://huggingface.co/transformers/index.html. [Online; accessed 2020-05-09].

[36] "Pytorch." https://pytorch.org/. [Online; accessed 2020-05-28].

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

[38] "Source code for nltk.translate.bleu_score." https://www.nltk.org/_modules/nltk/translate/bleu_score.html. [Online; accessed 2020-05-09].

[39] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger, "From word embeddings to document distances," in *International conference on machine learning*, pp. 957–966, 2015.

[40] "gensim: topic modelling for humans." https://radimrehurek.com/gensim/apiref.html. [Online; accessed 2020-05-28].

[41] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[42] "gensim: topic modelling for humans, word movers' distance." https://radimrehurek.com/gensim/auto_examples/tutorials/run_wmd.html. [Online; accessed 2020-05-28].

[43] "Bigquery: Cloud data warehouse, google cloud." https://cloud.google.com/bigquery?hl=de. [Online; accessed 2020-05-28].

[44] "Stack exchange data dump : Stack exchange, inc. : Free download, borrow, and streaming." https://archive.org/details/stackexchange. [Online; accessed 2020-05-11].

[45] F. Hoffa, "Github on bigquery: Analyze all the open source code." https://cloud.google.com/blog/products/gcp/github-on-bigquery-analyze-all-the-open-source-code. [Online; accessed 2020-05-11].

[46] "How to write doc comments for the javadoc tool." https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html. [Online; accessed 2020-05-11].

[47] "langdetect pypi." https://pypi.org/project/langdetect/. [Online; accessed 2020-05-11].

[48] "google-research/albert." https://github.com/google-research/ALBERT. [Online; accessed 2020-05-18].