Code Completion with Recurrent Neural Networks

Erik van Scharrenburg

Spring 2018, 37 pages

Academic supervisor: E. Gavves
Host organisation: Info Support B.V.
Host supervisor: W. Meints

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Code completion is a feature that offers suggestions for what a software developer may want to type. Correct suggestions can help developers when programming. There are several forms of code completion and we study the problem of predicting source code tokens based on previous tokens. There is not enough knowledge about using recurrent neural networks to make code completion suggestions. Statistical methods and neural networks have achieved great results on natural language processing tasks. Programming language is also, like natural language, repetitive and this repetitiveness can be modeled using statistical methods or neural networks. We trained models with different architectures and hyperparameters on an open-source dataset with Java code to answer the question: How can recurrent neural networks be used for code completion? In our experiments, the highest accuracy we achieved was 61.0% correctly predicted tokens. This accuracy was achieved with a model with a single hidden layer, 512 LSTM units, a lookup table embedding and 10 words as input. We found that models with 512 hidden units achieved a significantly higher mean accuracy in our experiments than models with 128 hidden units. Furthermore, we found that in our experiments using 10 words as input led to a significantly higher accuracy than using five words and using five words as input led to a significantly higher accuracy than using one word. Our preliminary results indicate that more hidden layers can increase the generalization, that the accuracy of word-based models is higher than that of character-based models, and that different types of recurrent units and embedding types achieve a similar accuracy. We present recommendations for using recurrent neural networks for code completion based on our findings and preliminary results. We report the results and implementation details for all models in our experiments to allow these models and results to be compared to other approaches.

Contents

Chapter 1

Introduction

In § 1.1 we describe the code completion problem that we study, with examples of input and expected output and we describe why it is a problem. In § 1.2 we explain that we use techniques from the natural language processing field because there are similarities between code and natural language. We list several usage scenarios in addition to code completion for approaches that address the problem in § 1.3. In § 1.4 we describe several challenges related to the problem and machine learning in general. We list the research question and subquestions in § 1.5. At the end of the chapter, we provide a summary in § 1.6.

1.1 The problem

Code completion is a feature that predicts what a software developer may want to type and offers these predictions as suggestions to the user. Figure 1.1 illustrates the problem with four examples. The text at the top is a line of example code split in tokens. Below the example code are four examples with the input and desired output. There are several forms of code completion and we focus on the form where several words have already been typed and a new word should be suggested.

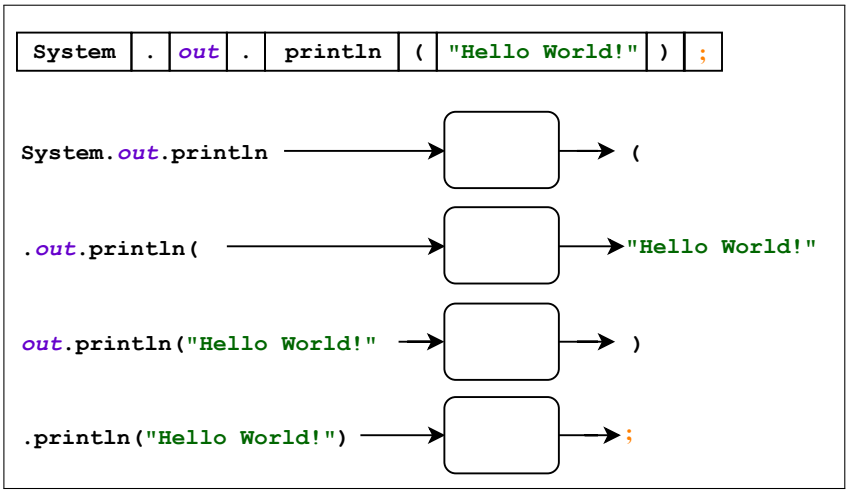


Figure 1.1: Example of the problem with inputs on the left and expected outputs on the right

Code completion is very useful, but often limited to the generation of single elements (e.g. method calls and properties) and the usage of templates. Furthermore, too many recommendations can decrease the usefulness[44]. Additionally, there is a lot of room for improvement of code assistants, in an experiment by Arrebola and Aquino Junior [2] over half of the interactions with the assistant was dismissed, interrupted or the proposed completions did not contribute to the task at hand. Participants reported an overall satisfaction rate of 77.78% for a more advanced completion system that sorted recommendations using statistical usage models and supported method chaining compared to 41.67% for the standard alphabetical sorting without

support for method chaining. Furthermore, while the state of the art in program synthesis and code generation is very promising, there are still many obstacles such as the difficulty of debugging synthesized programs and the infeasibility of specifying the intention of the user in a formal way for complex problems. Generating code from previous lines of code removes the need for a formal specification of the user intention and generated code could be easier to use in practice as an incorrect program that is difficult to debug has little use while readable code could offer hints or new insights and nearly correct code could be corrected manually.

1.2 Programming vs. natural language

Statistical methods and neural networks have achieved great results on natural language processing tasks mainly because in practice natural language is often repetitive and predictable. The repetitiveness can be modeled using statistical or neural language modeling. Language models such as N-gram models can be used to estimate the likeliness that a certain word will follow a given sequence and determine which word is most likely to follow a given sequence.

Hindle et al. [23] show that source code is also, like natural language, repetitive and present the claim that *"though software, in theory, can be very complex, in practice, it appears that even a fairly simple statistical model can capture a surprising amount of regularity in "natural" software"*[23, p. 131]. Additionally, the authors show that the default code completion of Eclipse[16] based on type information can be improved by using N-gram models to generate short suggestions (with a length below 7).

We will aim our research towards neural language models instead of statistical models such as N-grams, **for the following reasons**: Neural language models can outperform carefully tuned N-gram models when modeling natural language[26]. For source code Hellendoorn and Devanbu [21] introduce a "dynamically updatable, nested scope, unlimited vocabulary count-based N-gram model"[21] that outperforms several deep learning models on the task of code completion. Researching how neural language models can be used for code completion can help to gain insight in the differences between natural language and source code. Additionally, there is a large amount of open-source code available and this is beneficial for machine learning.

1.3 Use cases

Models that predict code and provide a measure for the likeliness that a certain token will appear could be used for several other tasks.

Error detection

Measuring the likeliness for tokens to appear in certain places can be used to detect possible errors in code. Suppose a model can reliably determine the likeliness of tokens and it is trained on correct code. If a single token is found that is highly unlikely to appear in that position according to the model, that token is highly unlikely to appear in correct code and thus there is an increased probability that that code is incorrect compared to the rest of the code.

Touchscreen input

Writing on a touchscreen keyboard without autocomplete, autocorrect and similar features is often slow and error-prone. For natural languages, these functions are available and widely used but this is not the case for programming languages. A model that provides an ordering based on how likely it is that a certain token will appear can be used to build such functionality for programming languages. This could make programming using a touch screen keyboard faster and help reduce the number of errors.

Optical character recognition

OCR techniques for code could be improved with a model that determines the probability for tokens to appear. If an OCR system assigns similar probabilities for multiple characters while these characters are parts of different tokens with different probabilities the combination of these techniques could reduce the number of errors that are made and improve the accuracy of OCR systems.

Compression

A model that predicts tokens could be used to compress source code. Suppose a relatively simple system stores the first n tokens for a source code file, one bit for each token to determine if it should be predicted and all tokens that are predicted incorrectly. Each correctly predicted token will then be compressed to a single bit. While much more efficient algorithms are possible even this simple one illustrates that code completion models could be used for the compression of software code given that the accuracy is high enough to outweigh the introduced overhead.

1.4 Challenges

A challenge with the prediction of source code is that methods and variables can be defined by developers and that many of these will not be frequent or contained at all in the training set. The problem of words that are not contained in the training set is known as the Out-Of-Vocabulary (OOV) or rare word problem. Additionally because of this freedom to define new words a very large vocabulary is required to minimize the number of OOV words. The main challenge with large vocabularies is that calculating the last step of the process, a softmax over all words, will become very time consuming.

A problem often encountered in machine learning is that models start *overfitting*[50] the training data when training for multiple epochs. A model that overfits the training data will have a much higher accuracy on the training data than on the validation or test data. In that case, the model does not generalize and a better description of the process is that the model memorized the training data instead of learning the patterns contained in the training data. This problem is a result of repeatedly adjusting the model on the same data which causes it to learn properties that are specific to the dataset and are not useful for the problem in general, the model learns to memorize the data it is trained on.

The main differences between the task of predicting single tokens and the task of predicting sequences of tokens are that it is more difficult to achieve an higher accuracy and that predictions other than the first prediction have less value in the context of code completion. The average accuracy of repeatedly using an approach that predicts single tokens with an accuracy of a_1 to generate a sequence of n tokens is $a_n = {a_1}^n$. To illustrate the difficulty in achieving a higher accuracy suppose a prediction system has a top-1 accuracy of 75%, when this system is used to generate three tokens the accuracy drops to $0,75^3 = 0,42$. Additionally, predictions other than the top-1 prediction have less value in the context of code completion because the number of possible suggestions increases exponentially when the number of tokens in the sequence is increased and too many recommendations can decrease the usefulness of a code completion system.[43]

1.5 Research question

How can recurrent neural networks be used for code completion?

Sub-questions

SQ1: How does the type of recurrent unit influence the prediction performance?

SQ2: How does the type of embedding influence the prediction performance?

SQ3: How does the number of hidden units in a network influence the prediction performance?

SQ4: How does stacking multiple layers in a network influence the prediction performance?

SQ5: How does the length of the input for a network influence the prediction performance?

SQ6: How does using words or characters as input for a network influence the prediction performance?

1.6 Summary of findings

We study the problem of predicting source code tokens from previous tokens. This can be used to make suggestions for tokens that are likely to appear next to improve code completion systems. Our approach for

this problem is to train an artificial neural network based on long short-term memory units on a collection of open-source software code. The thesis is organized as follows: In § 2 we describe related work and the materials that we use in our research. In § 3 we describe the models that we use in our experiments and the implementation details needed to reproduce the results. We also describe how we prepare the data before training the neural networks and how we evaluate the different models. We report that the best performing model achieves an accuracy of 61.0% and an MRR of 69.7% in § 4. We also show that in our experiments models with 512 hidden units achieve a higher accuracy than models with 128 hidden units. Furthermore, we show that models achieve a higher accuracy with 10 input words than with five and a higher accuracy with five words than with one. In § 5 we give suggestions based on our findings for designing neural network models to make source code token predictions based on previous tokens. We present our conclusions and directions for future work in § 7. Finally, we describe threats to the validity of our results in § 6.

Chapter 2

Background

In § 2.1 we describe work that is similar and relevant to our work. We describe what the differences are with our work and why the work is relevant for our research. In § 2.2 we describe the material that we use for our research and experiments.

2.1 Related work

Comparison of statistical language modeling techniques

Hellendoorn and Devanbu [21] compare N-gram, RNN and LSTM language models and introduce a "dynamically updatable, nested scope, unlimited vocabulary count-based N-gram model".[21] The authors evaluate the models on a corpus of open-source Java code.[1] The code is tokenized and tokens are compared for an exact match. Code suggestion performance is measured using the Mean Reciprocal Rank (MRR) and top-k accuracy, the number of times the first k suggestions contains the correct result. The results show that for the task of code completion the N-gram model outperforms the deep learning models. The main differences in our research are that we focus on neural networks and experiment with different types of networks and different hyperparameters for the networks.

Dam, Tran, and Pham [13] evaluate several LSTM models with dimensions between 20 and 200 and input lengths between 10 and 500 on the task of language modeling. The size of the vocabulary is limited to 1000. The results suggest that an LSTM can be used to build a good language model for source code. The main differences in our research are that we experiment with more types of models, with a larger dataset and vocabulary size and the embedding dimensionality is fixed in our models.

Pointer mixture network

Li et al. [35] propose a pointer mixture network that consists of an RNN component and a pointer component. The authors compare the performance of the network with an LSTM network and an LSTM network with an attention mechanism. The results show that the mixture network outperforms both the LSTM and attention-enhanced LSTM network on the task of completing Python and JavaScript code. The authors attribute the increase in performance to the prediction of out of vocabulary words by the pointer component. The main differences in our research are that we do not experiment with networks with an attention or pointer mechanism, we experiment with different types of recurrent units and we evaluate the networks on the task of Java code completion.

Hindle et al. [23] use an n-gram model to show that code is even more regular than natural languages. The authors show that *"language models capture a significant level of local regularity that is not an artifact of the programming language syntax, but rather arising from "naturalness," or regularity, specific to each project."* Additionally, the authors show that there is a lot of regularity within application domains and much less across application domains.

Feed-forward model with attention

Das and Shah [14] explore an approach for code completion using a feed-forward neural network model with soft attention. The authors evaluate this approach on the code for the Linux kernel[51] and the Twisted library[32]. The authors briefly experimented with a GRU based recurrent model but did not experiment further because they achieved a better initial performance with a feed-forward model. "Recurrent neural network models based on cells like LSTM and GRU have recently been shown to achieve state-of-the-art performance in language models. Inspired by these results, we also attempted to use a GRU based recurrent model for our prediction task…As we did not achieve competitive performance with this model as compared to NL-1, we did not experiment further with deeper GRU models."[14] The authors note several modifications to their approach such as combining language rules to remove syntactically invalid predictions. The authors also suggest the joint processing of .h and .c files and note that this may not scale well with their feed-forward model while recurrent models are known to perform well with large inputs. The main difference in our research is that the data is split per project and a complete project can be in either the train, validation or test set whereas Das and Shah [14] train on one part of a project and test on another part.

Comparison of RNN units

Chung et al. [12] empirically evaluated RNNs with basic, LSTM and GRU units on the tasks of polyphonic music modeling and speech signal modeling. The authors show that LSTM and GRU outperform the basic RNN on this task in terms of accuracy when each model has approximately the same number of parameters. The main difference in our research that we evaluate the networks on the task of Java code completion.

Comparison of CNN and RNN for NLP tasks

Yin et al. [55] performed a systematic comparison of CNN, GRU, and LSTM on several tasks among which Relation Classification (RC), Textual Entailment (TE) and Answer Selection (AS). The authors found that RNNs perform well in a broad range of tasks with the exception of key-phrase recognition and question-answer matching tasks.

Comparison of existing approaches

Many publications on code assistance tools introduce new evaluation strategies or use custom datasets and regularly implementation details or datasets are not published. This makes comparing different evaluations or evaluating approaches with different datasets difficult or impossible.[43]

	Clones[22]	VCC[46]	SLANG[45]	PBN[44]	N-gram-LSTM[21]
Dataset available	✓*	✓	✘	✓	✓
Implementation details	✘	✓	✓	✓	✓
Source code published	✘	✘	✘	✓†	✓
Suggests sequences	✓	✓	✓	✘	✓
Context	Atomic clones	Project	APIs	Framework	General

* The authors use open-source projects but do not mention specific versions.
† The source code for the analysis step is not published.

Table 2.1: Comparison of existing approaches.

Table 2.1 contains several aspects of the different approaches that can be compared. As mentioned it is often difficult or impossible to compare different evaluations. This is also the case for the approaches in the table with the exception of the mixed N-gram-LSTM approach[21]. To compare the approaches they have to be re-evaluated such that the results can be compared. For the approach that uses clone detection[22] this is impossible because while the authors use code from open-source projects they do not specify the specific versions and not enough details are provided to create a new implementation. This prevents using the dataset with other approaches and using different datasets with this approach. Re-evaluating VCC[46] and SLANG[45] should be possible because while the source code is not available the authors provide enough details to create a new implementation of the approach. For PBN[44] the authors provide not only implementation details for the approach but also the source code for most steps. Creating a new implementation for the

analysis step should make it possible to re-evaluate the approach. Comparing a new approach to the mixed N-gram-LSTM approach[21] or only the N-gram part of this approach should be possible as the authors provide implementation details, source code and evaluated the approach using code from open-source projects.

Gehring et al. [17] introduce a sequence-to-sequence network architecture based on CNNs instead of RNNs. The authors evaluate the model on machine translation tasks. The results show that the model outperforms the previous best model on the WMT' 16 task, this was an attention-based sequence to sequence model. On the WMT' 14 task the model outperforms an LSTM model with attention, a character based encoder-decoder model based on CNNs without attention and a word-based LSTM encoder-decoder model in terms of accuracy and computational cost.

Kaiser, Gomez, and Chollet [27] studied how depthwise separable convolutions can be applied to neural machine translation. The authors introduce and evaluate an architecture based on depthwise separable convolution layers with residual connections. The results show that the architecture outperforms an architecture based on non-depthwise separable convolutions in terms of accuracy with a similar parameter count.

Copying mechanism in sequence to sequence learning

Gu et al. [19] present a model called CopyNet where a copying mechanism is incorporated in the decoder. The copying mechanism can put subsequences from the input sequence in the output sequence. The authors compare their approach to RNNsearch, an RNN Encoder-Decoder with attention. The results show that CopyNet performs better than RNNsearch on text summarization and single-turn dialogue tasks, especially regarding the handling of words not contained in the vocabulary.

Code clones

Hill and Rideout [22] present an approach for automatic method completion based on suggesting previously detected code clones. The approach helps a programmer to write atomic clones, clones with a size of five to 10 lines and demonstrate the viability of automatic method completion. The authors note that the accuracy is poor when the programmer is writing arbitrary methods. The accuracy was measured on only 10 partial methods written by the authors and ranked relatively to three manually written completions. The authors use open-source code from three projects for the evaluation but do not mention the exact version. Furthermore, the implementation details and source code of the approach are not published. This makes it impossible to compare the approach to different approaches.

Sequential pattern mining

Silva Junior, Plastino, and Murta [46] present an approach for code completion with sequential pattern mining. The approach analyses the source code to discover recurring sequential patterns. Then during coding when a programmer writes code that matches the beginning of a sequence in the previously obtained patterns the rest of the sequence is suggested. A difference between the approach from Silva Junior, Plastino, and Murta [46] named Vertical Code Completion (VCC) and the project is that VCC uses the PLWAP algorithm to find frequent sequences and suggests those sequences while the project uses a sequence to sequence model to generate suggestions. The authors use open-source code from four projects for the evaluation but do not mention the exact version. Instead of splitting the source code into two sets the authors use different revisions of the projects to simulate the expected usage. Because methods that are added in the newer revisions are used for the validation there is no overlap between the test and validation data. Replicating the experiment or evaluating the approach on a different dataset would require creating a new implementation of the approach because neither source code or compiled code is available.

Statistical language models

Raychev, Vechev, and Yahav [45] present an approach called SLANG for code completion with statistical language models for programs that use APIs. The approach extracts sequences of API calls from source code, trains a combination of an N-gram model and a recurrent neural network on the extracted data. The statistical language model is then used to generate candidate sequences ordered from most to least probable and suggest the sequences with the highest probability. A trained model, most of the validation data and a compiled version of the code are available. An exact replication of the experiment is impossible because of the missing

information and would have little value because the code cannot be verified. Evaluating the approach with different validation data should be possible. Because the training data is unknown there is a certain chance that the validation data is contained in the training data which is a threat to the validity of the experiment. Training the model on a different dataset might be possible but will certainly be difficult because there is no documentation relating to this available.

Bayesian network

Proksch, Lerch, and Mezini [44] present an approach for more intelligent code completion called Pattern-based Bayesian Network (PBN). PBN identifies how likely specific method calls are using a Bayesian network and suggests the most likely calls. The experiment can be replicated with the source code and data that is available. For the analysis step, the result and implementation details are published, but not the source code for performing the analysis. This makes evaluating the approach with different data possible, but it would require creating an implementation of the analysis step. Such an implementation could then also be used to replicate the analysis performed by the authors.

Co-creative natural language generation system

Manjavacas et al. [37] describe a co-creative natural language generation system to be used by a novelist. The system offers a text editor and generates sentences based on (a part of) the previous text that can be added by the writer.

Evaluation of assistance tools

Proksch, Amann, and Mezini [43] present a framework for the standardized evaluation of assistance tools including code completion systems (the authors use the term code recommenders). The framework includes requirements for datasets to ensure that they are reusable. A general evaluation metric is not included because metrics should be defined per recommender type and finding a general metric is impossible.

2.2 Material

We use three types of recurrent units. A **RNN** is a neural network with neurons that have cyclic connections. An example of an RNN with a single hidden layer consisting of a single neuron is illustrated in Figure 2.1. U, V, and W are the parameters that are learned during training. The RNN receives a sequence as input in multiple steps and at each step, the hidden state is calculated from the input at that step multiplied with weight U and the hidden state from the previous step multiplied with weight V.

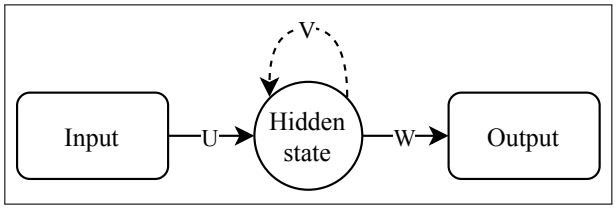


Figure 2.1: RNN with a single hidden layer of one neuron

Long-Short Term Memory(LSTM)[25] units include input, forget and output gates. These gates control what new information should be added to the state, what should be forgotten and what should be in the output based on parameters that are trained. Gated Recurrent Units(**GRU**)[9] include update and reset gates. These gates control how much of the previous information should be kept and how much should be forgotten. Convolutional Neural Networks(**CNN**)[34] are feed-forward networks that apply a convolutional operation to the input.

2.3 Dataset

We will use the source code collected by Allamanis and Sutton [1]. The corpus contains Java code from 14.785 open-source projects hosted on GitHub that were forked at least once. The authors manually inspected projects with duplicate commit hashes because these are likely forks that are not marked as such on GitHub. For those projects, the authors picked the projects that seemed to be the original and removed the forks from the corpus to prevent duplicate data in the test and training set. For the experiments, we will use a subset of the corpus with 1% of the data. There are two reasons for this, the first reason is that training the models on the full dataset is infeasible. To illustrate this we can estimate the time it would take to train on the full dataset by multiplying the time it takes to train on the 1% subset with 100. The relatively small LSTM model with 128 units requires around 33 hours of training per epoch on the 1% subset on a quad-core Intel i7-7700 CPU. The evaluation takes around 5 hours. Training the model for one epoch on the full dataset and evaluating it would thus take around 160 days. In our experiments training a model on a Tesla K80 GPU takes approximately half the time it takes on a CPU which means training on the full dataset is still infeasible. The second reason is that Hellendoorn and Devanbu [21] use a 1% subset of the full corpus for their evaluation and report the list of projects in each dataset (training, validation, and testing) which makes it possible to compare different models using the same dataset. For the evaluation on the test set, we use a larger subset of approximately 2% of the full corpus. This subset is constructed by taking the 1% subset and adding randomly selected projects from the training dataset until the subset is approximately twice the size of the 1% subset. The size is approximately 2% and not exactly because it is extended by adding a complete project at a time. To illustrate what tokens are contained in the 1% dataset we list the 15 most frequently occurring words and characters below:

Words: `'.', '(', ')', ';', ',', '', '', '=', '+', 'this', 'if', 'public', '[', ']', 'return'`

Characters: `'e', 't', 'n', 'i', 'r', 'a', 'o', 's', 'l', 'c', 'p', '.', 'u', 'd', '('`

Chapter 3

Method

In § 3.1 we describe the null- and our research hypotheses for the research question and the subquestions. We describe the different models we experiment with in § 3.2 and in § 3.3 we describe the details required to implement the models used in our experiments. In § 3.4 we describe that we lex the source code, map words to integers and for models with a lookup table we use a one-hot encoding to represent the data. In § 3.5 we describe that we use the accuracy and MRR metrics to evaluate the models.

3.1 Hypotheses

For the research question, the hypothesis is that neural networks can be used to predict source code tokens based on previous tokens. The hypothesis is based on the theory that programmers often write code that is repetitive and that neural networks can learn regularity in source code and then predict source code tokens. This theory is supported by the results of an experiment by Hindle et al. [23] that show that there is a high degree of local regularity present in source code. Additionally, Hellendoorn and Devanbu [21] have shown that combining an LSTM with an N-gram model decreases the entropy, but does not improve the predictions. Their approach uses the LSTM to make a prediction and in the case where the LSTM predicts an unknown token the prediction of the N-gram is used instead. Furthermore, Dam, Tran, and Pham [13] and White et al. [52] both achieve promising results with neural language models.

SQ1: How does the type of recurrent unit influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of the different recurrent units. The **research hypothesis** states that the performance of the gated LSTM and GRU is higher than that of the non-gated RNN. This hypothesis is based on the preliminary results presented by Chung et al. [12] where LSTM and GRU models outperformed RNN models on several different tasks and the results presented by Dam, Tran, and Pham [13] where LSTM models outperformed RNN models on the task of modeling software code.

SQ2: How does the type of embedding influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of the different embedding types. The **research hypothesis** states that the performance of a CNN based embedding is higher than that of a lookup table embedding. This hypothesis is based on the results presented by Kim et al. [30] where a character-based CNN outperformed word-based models with a lookup table embedding.

SQ3: How does the number of hidden units in a network influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of networks with different numbers of hidden units. The **research hypothesis** states that the performance of networks with more hidden units is higher than that of networks with less hidden units. This hypothesis is based on the theory that networks with more hidden units have more representational power and if early stopping is used there is little loss in generalization performance for networks with excess capacity[8].

SQ4: How does stacking multiple layers in a network influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of networks with different numbers of stacked layers. The **research hypothesis** states that the performance of networks with more stacked layers is higher than that of networks with less stacked layers or a single layer. This hypothesis is based on the results from the experiments by Pascanu et al. [40] where several variants of deep RNN models outperformed more shallow RNN models.

SQ5: How does the length of the input for a network influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of networks with inputs of different lengths. The **research hypothesis** states that the performance of networks with longer input lengths is higher than that of networks with shorter input lengths. This hypothesis is based on the results from the experiments by Dam, Tran, and Pham [13] and Yin et al. [55] where higher performance scores were achieved with longer input lengths.

SQ6: How does using words or characters as input for a network influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of networks with characters or words as input. The **research hypothesis** states that the performance of networks with characters as input is higher than that of networks with words as input. This hypothesis is based on the research from Kim et al. [30] where character-based model outperformed word-based models. Additionally, the influence of the OOV word problem is much greater with word-based models as there are much less unique characters which makes it possible to use a relatively larger vocabulary while the absolute size of the vocabulary stays smaller.

3.2 Models

We experiment with recurrent models because recurrent networks outperform feed-forward networks in terms of accuracy at several other tasks such as time series forecasting[5] and speech recognition[48].

We focus mainly on LSTM based models over basic RNN and GRU based models because we aim to study among others the influence of the length of the input on the performance, and it is difficult for RNN models to learn long-term dependencies because of the vanishing and exploding gradient problems[4, 39]. Additionally, in the evaluation by Chung et al. [12] both LSTM and GRU based models outperformed RNN based models and in the experiments by Das and Shah [14] a GRU based model did not achieve competitive results. We experiment with RNN and GRU based models because the task and dataset in their experiments are different from ours and Chung et al. [12] note that "the choice of the type of gated recurrent unit may depend heavily on the dataset and corresponding task". Chung et al. [12] use different numbers of units for the RNN, GRU and LSTM models so that the different models have approximately the same number of parameters. The reason the authors offer for this choice is that their aim is to compare the models fairly. We use the same number of units for the different types of recurrent units because the relation between the number of units and the number of parameters is a property of the type of unit. For a fair comparison, we will report not only the performance but also the number of parameters.

We experiment with models with different hidden sizes because optimization of the hidden size and batch size is crucial to good performance of RNNs[55]. We train all models with a batch size of 128 because time constraints limit the number of models we are able to train and a batch size of 128 should provide a good trade-off between the training speed and the accuracy of the estimation of the complete set. Additionally this batch size is still relatively small compared to the batch sizes in the experiments by Keskar et al. [29] who find that the models in their experiments achieve lower accuracies with larger batch sizes starting from a size of approximately 5000.

We experiment with models where a character-level convolutional network is used instead of word embeddings because the input vocabulary can be much smaller, subword information is available and these models have much fewer parameters.

3.2.1 RNN

We train several models based on recurrent units where we vary the type of unit and the size of the network. The input for these models is a fixed length sequence of tokens on which we train word embeddings. These word embeddings are passed to the RNN layer and the output of the RNN layer is passed to a densely connected layer with a dimensionality of the size of the vocabulary. These models are illustrated in Figure 3.1.

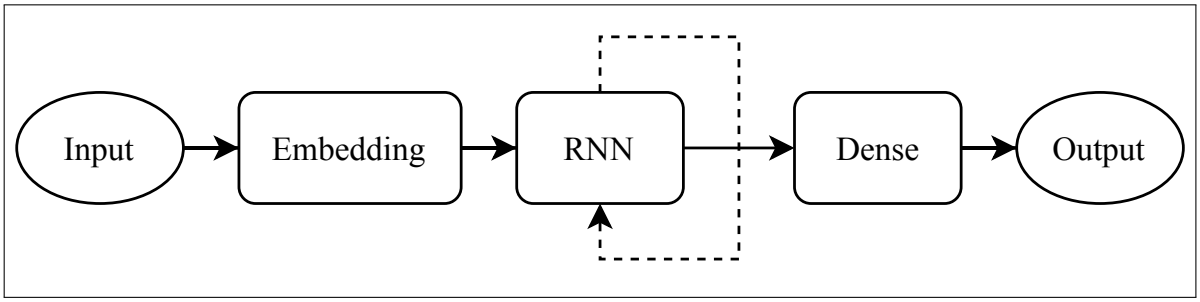


Figure 3.1: RNN model

3.2.2 GRU

We train one model based on gated recurrent units with a single layer of 128 units using five words as input. The design of this model is the same as that of the non-gated RNN model illustrated in Figure 3.1 except that the type of RNN unit is different.

3.2.3 LSTM

We train several models based on long short-term memory units. The design of these models is the same as that of the non-gated RNN and GRU models illustrated in Figure 3.1 except that the type of RNN unit is different. To evaluate the influence of the size of the network on the prediction performance we train four single layer models with 128, 256, 512 and 1024 units using five words as input. To evaluate the influence of the number of words used as input we train single layer models with 128 and 512 units on 5, 10 and 20 words and additionally the model with 512 units on 1, 2 and 15 words.

Multiple layers

In addition to the single layer models, we train two models, with two and three layers of 128 hidden units on five input words and two models with three layers of 512 hidden units on 10 and 15 input words. We use these models to evaluate the influence of stacking multiple layers on the performance. The two-layer model is illustrated in Figure 3.2, the design of the three-layer is similar with one extra layer.

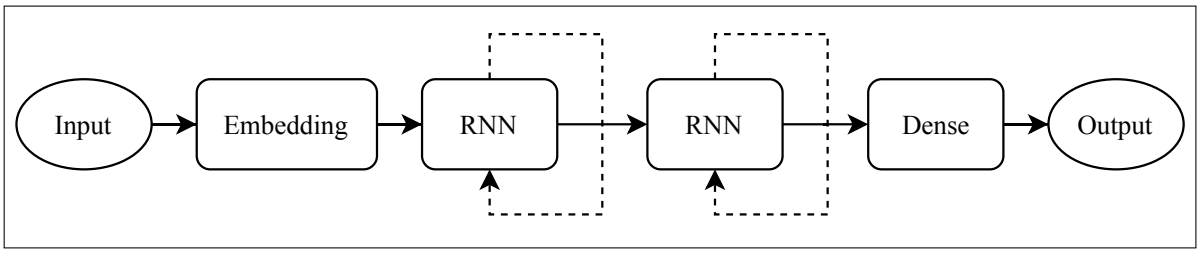


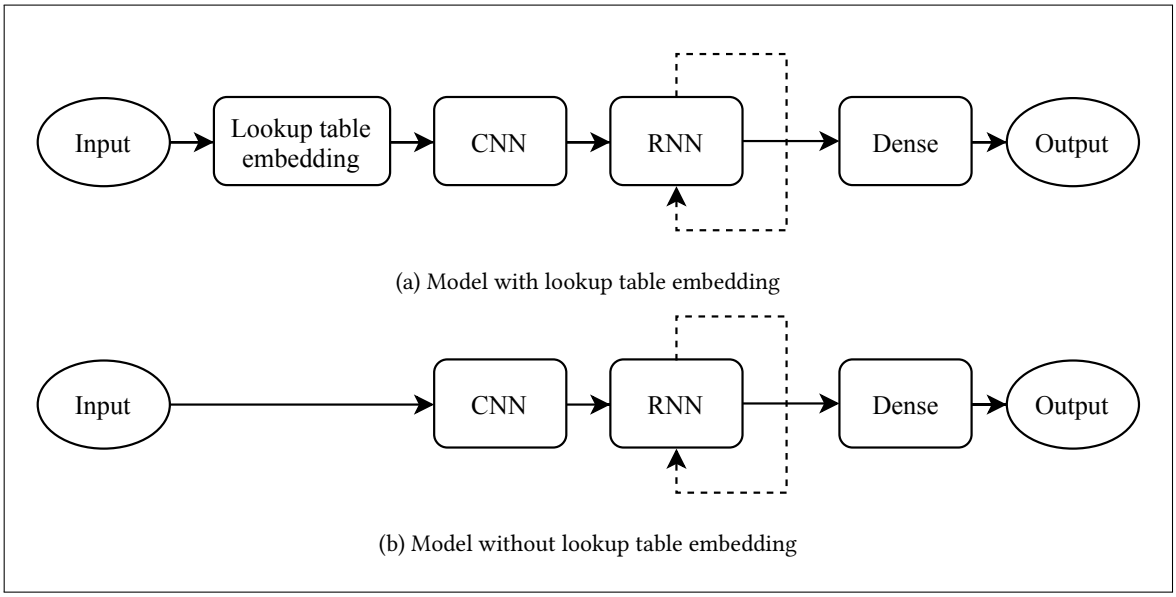
Figure 3.2: Two-layer RNN model

Character input

We experiment with two models based on single layers of 128 and 512 units using 10 characters as input.

3.2.4 CNN-LSTM

We experiment with several CNN based models with two different architectures and various combinations of hyperparameters. This design is based on the approach from Kim et al. [30] of which we use only the character-CNN technique because while the authors also propose other techniques such as a highway layer we aim to evaluate the influence of the type of embedding and the authors note that in their experiments the model where only the embedding is replaced performs well. In one variant, the vectors from a lookup table embedding are used as input for the CNN. In the other variant the input for the model is directly passed to the CNN without a lookup table embedding. The designs of these variants are illustrated in Figure 3.3.



(a) Model with lookup table embedding

(b) Model without lookup table embedding

Figure 3.3: CNN RNN models

Character input

We experiment with four models with characters as input. Two models with a lookup table embedding, 128 hidden units and input lengths of 10 and 20. The other two models have no lookup table embedding, both input lengths of 10 and 128 and 512 hidden units.

3.3 Implementation details

3.3.1 Vocabulary size

We limit the vocabulary to 74,064 words so that tokens that occur less than five times in the training set are replaced by a special *unknown* token. This is the same limit that Hellendoorn and Devanbu [21] used in their work. We chose to use the same limit to be able to compare and combine the approaches as we also use the same dataset. The limit is similar to the limits used by White et al. [52] (71,293) and Dam, Tran, and Pham [13] (81,213). Some examples of words that are replaced by the *unknown* token are: `'getGroovyProblemTypes'`, `'testCustomLayoutSubgraphFilteredAccess'`, `'testGridLayoutAlgorithmEmptyGraph'`.

3.3.2 Overcoming the OOV word problem

One approach to address the OOV word problem is to limit the scope to a specific domain. This approach is common in code completion research, i.e. VCC[46] is limited to single projects, SLANG[45] is limited to API usage and PBN[44] is limited to framework usage. Allamanis and Sutton [1] analyzed a large Java source code corpus and note that "API calls are significantly more predictable compared to types and variable names."[1,

p. 6] A different approach is to combine the model with an N-gram model. In case the sequence-to-sequence model predicts an unknown token the prediction from the N-gram model is used instead. This approach has the potential to allow a more general application of the code recommendations, and Hellendoorn and Devanbu [21] found that the combination of an LSTM and an N-gram model achieved a lower entropy than either model separately. Another approach is to replace the embedding layer that is commonly used in combination with an LSTM with a convolutional neural network (CNN). Kim et al. [30] describe this approach and evaluate it on several natural languages. The results show that the approach achieves state-of-the-art accuracy with fewer parameters. Furthermore, Józefowicz et al. [26] compare several models and combinations of models on the task of natural language modeling and find that the model based on this approach achieves the lowest perplexity of the compared single models. We experiment with the combination of a CNN with an LSTM to attempt to overcome the OOV problem in the input of a model. In the output we let the models suggest the *unknown* token for OOV words as our research is aimed at a general approach for code completion not limited to a specific domain. Predicting the unknown token makes it possible to combine multiple models as done by Hellendoorn and Devanbu [21] who make suggestions with an N-gram in case an LSTM predicts an unknown token.

3.3.3 Overfitting

There are several approaches to prevent models from overfitting the training data. An approach that is commonly used is the *dropout* technique as described by Hinton et al. [24] where random units of the network are dropped during training. L1 or L2 regularization[38] and weight decay[36], which is equivalent to L2 regularization depending on the optimizer and parameters, are also frequently used. Furthermore, there is max-out regularization[18], and multiple techniques can also be combined, e.g. max-out is often combined with dropout. A common factor with all of these approaches is that they are generally used when training for multiple epochs and thus repeatedly adjusting the model to the same data. Training for multiple epochs is common because especially larger models require many adjustments before they fit the training data well and there is often not enough data available to do this in one epoch. For the task of modeling source code, the availability of data is not a problem. As mentioned in § 2.3 we only use 1% of a Java corpus[1] and more data can relatively easily be collected by mining open-source code repositories. In contrast, many other machine learning tasks require data that is manually labeled by humans making it very resource intensive to collect more data. To prevent the networks from overfitting the training data we train the models only for a single epoch. This approach can be seen as a form of early stopping[42].

3.3.4 Optimizer

We use the Adam optimizer[31] for all models. The research from Keskar and Socher [28] and Wilson et al. [53] suggests that the generalization performance may be increased when using stochastic gradient descent (SGD) with well-tuned parameters. Because we test many different models, tuning the parameters for every model would be very time consuming, and, more importantly, the parameters would be extra variables that influence the results. This would make it more difficult to evaluate the influence of other variables such as the type of recurrent unit. Using the same parameters for all models allows for a better comparison between the variables of which we aim to evaluate the influence. Additionally, the differences in the performance of SGD and Adam in the experiments in both papers are differences in the generalization of the model i.e. the models trained with the Adam optimizer were overfitting more quickly in the experiments and our approach to prevent overfitting is different from the approaches from Keskar and Socher [28] and Wilson et al. [53].

3.4 Data preparation

3.4.1 Collection

As mentioned in § 2.3 we use a part of the dataset that Allamanis and Sutton [1] collected from GitHub repositories. The Java source code files can be downloaded in compressed form and are split up in separate folders for each project.
Hellendoorn and Devanbu [21] made lists available with the projects in the training, validation, and testing sets that they used in their research. To get the same dataset and split the data into training, validation, and

testing sets we extracted the folders from the lists from the full archive.

3.4.2 Tokenization

We lex the source code to split the code in separate tokens and remove whitespace outside of strings. Depending on whether words or characters are used as input for the model sequences of tokens or characters are created. For both models, all input tokens or characters and output tokens are mapped to integers so that these can be easily converted to vectors.

3.4.3 Encoding

For models with a lookup table embedding, the integers representing words or characters are used as the index for a table containing the vector representations.

For the models without a lookup table, the integers are one-hot encoded. This is a more accurate representation because the data is not ordinal and an order was introduced by mapping the words or characters to integers.

3.5 Evaluation

We evaluate all models on the validation dataset. On the test set, we evaluate only the model that achieves the best performance on the validation set. Evaluating only a single model on the test set is necessary because evaluating multiple models on the test set would mean that a parameter, which of the models is used, is trained specifically on the test set and that would harm the validity of the results.

3.5.1 Accuracy

We mainly report the accuracy metric for the different models. The accuracy is the ratio of the number of correct predictions compared to the total number of predictions. The accuracy is especially useful when the aim is to suggest sequences based on multiple single token predictions as described in § 1.4. We report the achieved accuracy for each of the trained models. The accuracy of a trained model can be used as an indication for the performance of the design of the model on the task of code completion when only a single suggestion is made instead of a list of suggestions.

3.5.2 Mean reciprocal rank

The MRR is a measure for the probability of correctness for multiple results. The rank is calculated by dividing the probability for a correct result by the rank of the result. To illustrate, suppose a model is used three times to make five suggestions at a time. If the first time the first suggestion is correct, the second time the fourth and the last time none of the suggestions are correct the MRR would be $({}^1/_1 + {}^1/_4 + 0)/3 \approx 0.42$. We calculate the MRR using the top-10 suggestions. We report the MRR for the model that we evaluate on the test set. The MRR is relevant for the code completion task because the metric takes into account that the rank of correct suggestions influences the usefulness of the suggestions e.g. the score is higher when the first prediction is correct than when the 10th prediction is correct. The accuracy, on the other hand, is based solely on the first suggestion. Additionally calculating the MRR allows us to compare our results to those from Hellendoorn and Devanbu [21] because the authors mainly report the MRR scores for their experiments.

3.5.3 Significance

To test the significance of our results we train and evaluate several model designs two times. Training specific designs multiple times is necessary because parameters that influence the performance are assigned random values (i.e. initial weights). Lawrence et al. [33] show that different starting conditions can lead to different results and moreover that the distribution of the results can vary widely depending on the architecture. We argue that to test the significance it is insufficient to evaluate a specific trained model on multiple sentences in a dataset. In our opinion to evaluate a design instead of a specific model one prediction should not be considered as a sample. Instead, the evaluation of one trained model on the complete validation set should

be considered as one sample. A drawback of considering one complete evaluation to be one sample is that it makes it infeasible to acquire many different samples because of the increased computational cost per sample. A drawback of considering one prediction as one sample is that small differences caused by random parameters in a specific model can be viewed as being significant. We believe that marking differences caused by randomly assigned parameters as significant is a far greater problem than testing the significance with a small number of samples because the small number of samples is reported and can thus be considered together with the result of the significance test. We use the randomization model[15] to calculate the p-value. With this model, the inferences are limited to the models in the study. We compare the difference in the mean accuracy of two groups with two trained models per group. Because the inferences and the number of samples are limited we use a confidence level of $a = 0.01$.

Chapter 4

Analysis

In § 4.1 we show that for the task of code completion the performance difference between different types of recurrent units is small. In § 4.2 we show that in our experiments the combination of a CNN with an LSTM achieves a performance similar to the combination of a lookup table embedding with an LSTM and that the combination of a lookup table embedding with a CNN and an LSTM performs slightly worse. In § 4.3 we show how different hyperparameters influence the performance of the models. Specifically in § 4.3.1 we show that in our experiments models with 512 hidden units perform better than models with 128 hidden units. In § 4.3.2 we show that the preliminary results regarding the number of layers in a model indicate that more layers can lead to a higher accuracy and that these models are more difficult to train. In § 4.3.3 we show that an LSTM model with 512 hidden units achieves a higher performance with five input words than with one input word and a higher performance with 10 input words than with five input words. In § 4.3.4 we show that in our experiments the performance per average number of tokens is higher with a word based model than with character based models. In § 4.3.5 we show that one of the most simple models we experimented with starts to overfit after the first epoch. In § 4.3.7 we report that the best performing model achieves an MRR of 69.7% and we compare this to the models and results from Hellendoorn and Devanbu [21]. Finally in § 4.4 we summarize the main results that determined the different aspects of the best performing model. Table 4.1 shows an overview with the hyperparameters and validation accuracy of all models that we trained and evaluated.

Embedding	Input	Input length	Type	Units (per layer)	Total parameters	Accuracy
Lookup	Words	1	1 layer LSTM	512	49M	0.315*
		2	1 layer LSTM	512	49M	0.434
		5	1 layer RNN	128	19M	0.512
			1 layer LSTM	128	19M	0.511
				256	29M	0.513
				512	49M	0.515*
				1024	90M	0.519
			2 layer LSTM	128	19M	0.512
			3 layer LSTM	128	19M	0.521
				512	53M	0.514
			1 layer GRU	128	19M	0.508
			CNN + LSTM	128	19M	0.495
		10	1 layer LSTM	128	19M	0.543*
				512	49M	0.553*
						0.586†
			3 layer LSTM	512	53M	0.092
		15	1 layer LSTM	512	49M	0.084
			3 layer LSTM	512	53M	0.092
		20	1 layer LSTM	128	19M	0.092
				512	49M	0.084
	Characters	10	1 layer LSTM	128	10M	0.390
				512	39M	0.407
			CNN + LSTM	128	10M	0.379
		20	CNN + LSTM	128	10M	0.085
CNN	Characters	10	1 layer LSTM	128	10M	0.394
				512	39M	0.390

* These models are trained two times and the accuracy is the mean of two evaluations.
† This model is trained on a dataset that is approximately twice the size of the other models.

Table 4.1: Overview of all models

4.1 Choice of recurrent unit

Table 4.2 shows that there is a small difference in the validation accuracy between the models based on different RNN types. The models were trained with 5 words as input and a word embedding with 128 dimensions. The RNN model achieves the highest accuracy while it has the lowest number of parameters.

Unit type	Parameters	Accuracy
RNN	19,067,601	0.5115
LSTM	19,166,289	0.5111
GRU	19,133,393	0.5077

Table 4.2: Validation accuracy of models with different types of recurrent units

Figure 4.1 shows the training accuracy over one epoch for the RNN, LSTM and GRU models.

SQ1: How does the type of recurrent unit influence the prediction performance?

The null hypothesis states that there is no difference in the performance of the different recurrent units. The research hypothesis states that the performance of the gated LSTM and GRU is higher than that of the non-gated RNN. Because the differences in the accuracies of the different models are very small we trained

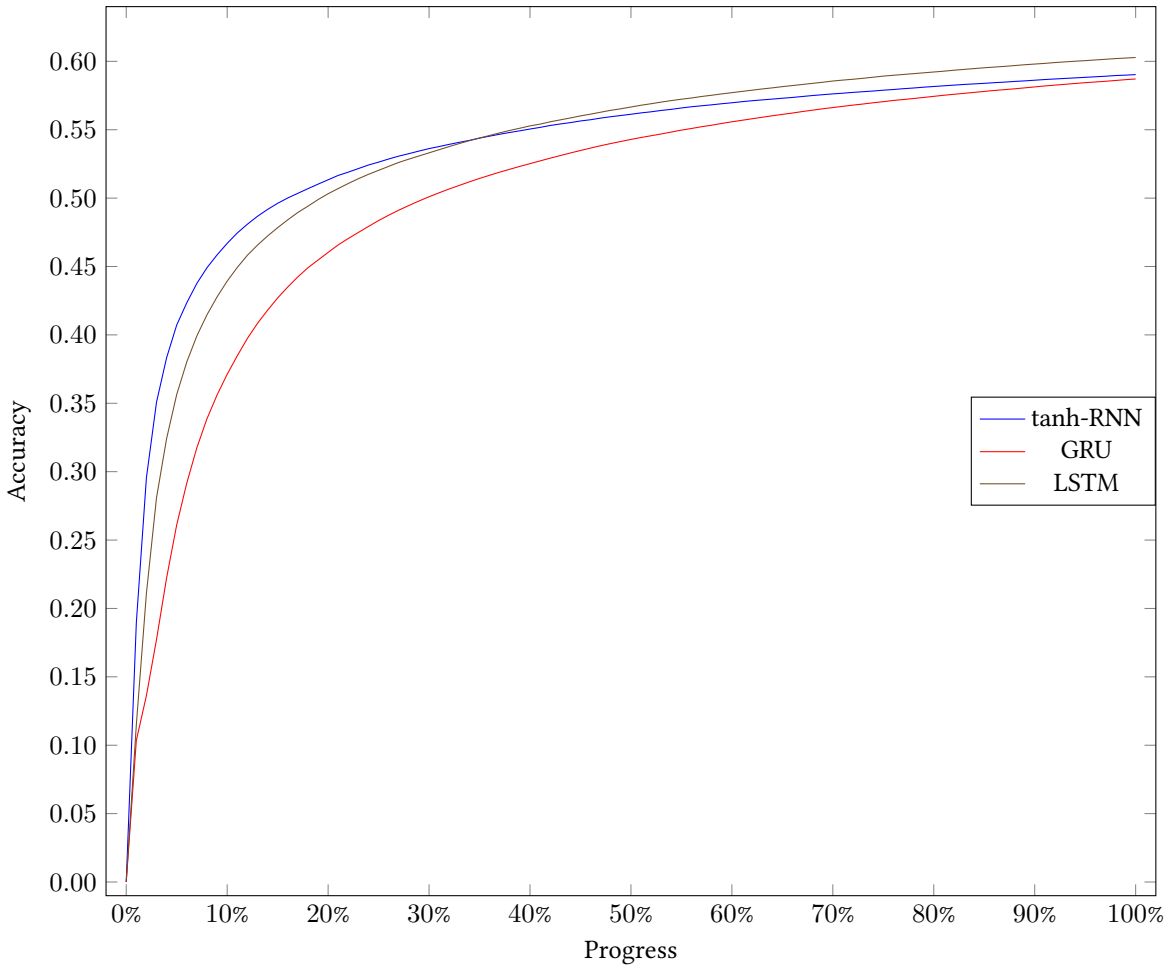


Figure 4.1: Training accuracy over one epoch of models with different recurrent unit types

each of the models only one time. The results are preliminary, we report the accuracy for each of the trained models and we do not reject the null hypothesis.

Takeaway

The preliminary results show that in our experiments there are only slight differences in the performance of different types of recurrent units in models with a single hidden layer of 128 units and an input length of 5 tokens on the task of code completion.

4.2 Choice of embedding

Table 4.3 shows that using either a lookup table or a CNN to transform character tokens into vectors leads to similar performance. Applying a CNN on the vectors from a lookup table leads to a lower performance.

Lookup table embedding	CNN	LSTM size	Total parameters	Accuracy
✓	✓	128	9.78M	0.3793
✓	✘	128	9.70M	0.3902
✘	✓	128	9.75M	0.3937
✓	✘	512	39.32M	0.4067
✘	✓	512	39.37M	0.3902

Table 4.3: Validation accuracy of LSTM models with different embedding types with 10 characters as input

SQ2: How does the type of embedding influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of the different embedding types. The **research hypothesis** states that the performance of a CNN based embedding will be higher than that of a lookup table embedding. We trained each of the models a single time because these models have characters as input and in our experiments, these are outperformed by the models with words as input. The models have characters as input because the design is based on the approach from Kim et al. [30]. We report the results for all trained models and we do not reject the null hypothesis.

Takeaway

The preliminary results indicate that the combination of a lookup table embedding and a CNN achieves a lower performance than either of these separately. In our experiments, the separate embedding types achieved a similar accuracy.

4.3 Hyperparameter search

4.3.1 Number of hidden units

Table 4.4 shows the accuracies of LSTM models with different numbers of hidden units and input lengths and types. When using the mean accuracy for models that were trained multiple times in all cases a higher number of hidden units leads to a higher performance.

Figure 4.2 shows the training accuracy over one epoch for the four models with 5 words as input. LSTM 512 appears twice because we trained this design two times. The models with more hidden units achieve a higher training accuracy after training for the same number of batches than the models with less hidden units.

SQ3: How does the number of hidden units in a network influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of networks with different numbers of hidden units. The **research hypothesis** states that the performance of networks with more hidden units will be higher than that of networks with less hidden units. The mean accuracy of the models with 128 hidden

Units	Input length and type		
	5 word	10 word	10 character
128	0.5111	0.5401 0.5457	0.3902
256	0.5127	-	-
512	0.5073 0.5221	0.5516 0.5535	0.4067
1024	0.5185	-	-

Table 4.4: Validation accuracy of models with different numbers of hidden units and input lengths

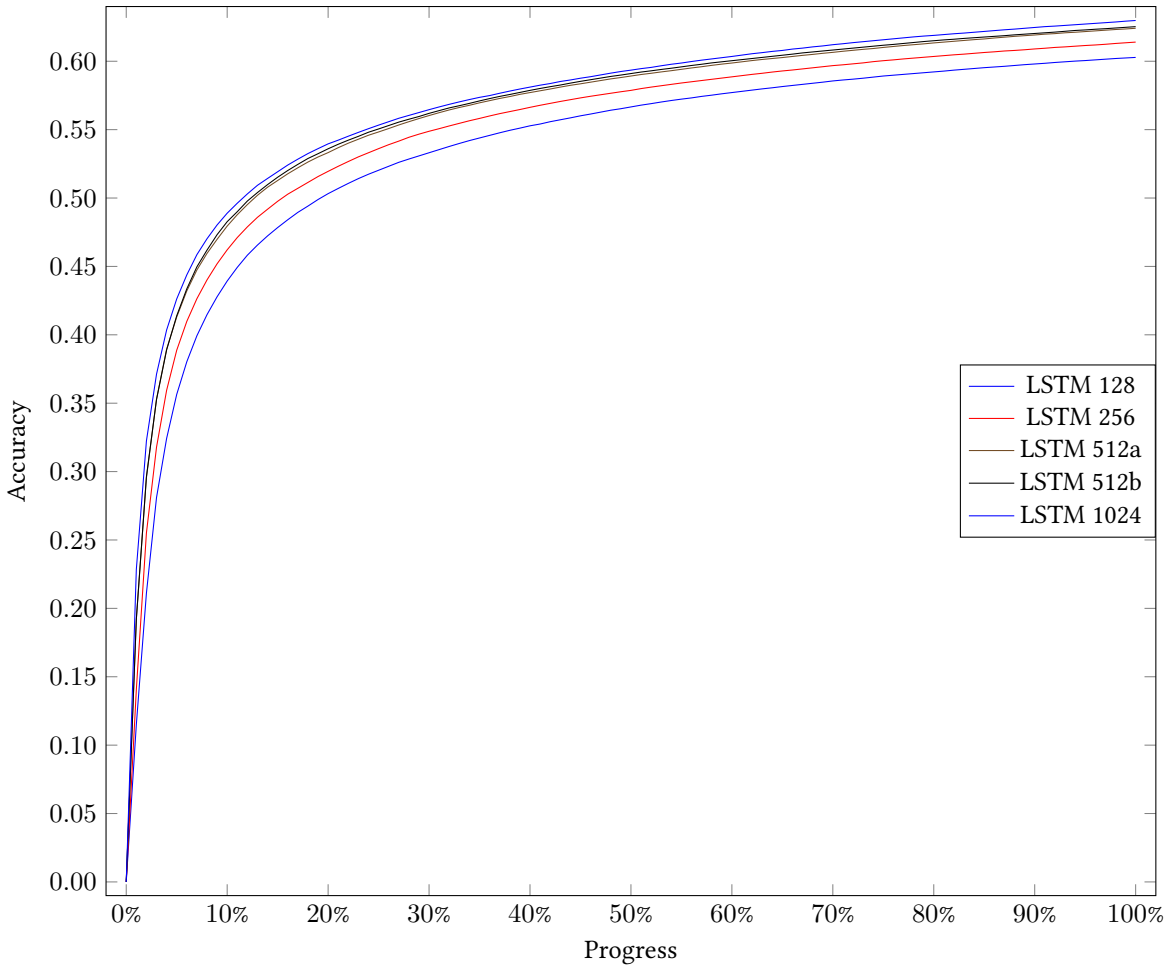


Figure 4.2: Training accuracy over one epoch of LSTM models with different numbers of hidden units

units and 10 input words is 0.5429 and the mean accuracy of the models with 512 hidden units is 0.5526. The mean accuracy of the models with 512 hidden units is thus 0.0097 higher. We trained the models with 128 and 512 hidden units and 10 words as input two times and calculate the p-value to determine the statistical significance of our results. As described in § 3.5.3 we use the randomization model to test whether the observed differences are significant. For the difference between 128 and 512 hidden units $p = 0.00$. With a significance level of $a = 0.01$ the difference is statistically significant thus we reject the null hypothesis and accept the research hypothesis for the models with 128 and 512 hidden units and 10 words as input. We trained the other models in the table only a single time and thus do not test the significance of the differences. The preliminary results show that in the experiments for each input length and type the mean accuracy is higher for models with more hidden units.

Takeaway

The mean accuracy of models with 512 hidden units is significantly higher than with 128 units using 10 words as input. For the other combinations of the number of hidden units, input length and type the preliminary results indicate that more hidden units lead to a higher performance and training speed per batch.

4.3.2 Number of layers

In Table 4.5 we show both the training accuracy and the validation accuracy for LSTM models with one layer or a two or three stacked layers. We show the training accuracy in the table for these models because the models with a higher validation accuracy have a lower training accuracy while in the comparisons of the other models nearly all of the models with a higher validation accuracy also have a higher training accuracy. Figure 4.3 shows that the training accuracy during one epoch of training is lower for the models with more stacked layers.

Layers	Training accuracy	Validation accuracy
1	0.6028	0.5111
2	0.5921	0.5121
3	0.5734	0.5207

Table 4.5: Training and validation accuracy of LSTM models with different numbers of layers of 128 hidden units and 5 words as input

Layers	5 input words		10 input words		15 input words	
	Training acc.	Validation acc.	Training acc.	Validation acc.	Training acc.	Validation acc.
1	0.6241	0.5221	0.6194	0.5516	0.0821	0.0841
3	0.5766	0.5135	0.0822	0.0924	0.0828	0.0924

Table 4.6: Training and validation accuracy of LSTM models with different numbers of layers of 512 hidden units and 5, 10 or 15 words as input

SQ4: How does stacking multiple layers in a network influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of networks with different numbers of stacked layers. The **research hypothesis** states that the performance of networks with more stacked layers will be higher than that of networks with less stacked layer or a single layer. We train the different models only a single time and report all accuracies because the accuracy differences for models with 128 hidden units are small and of the four models with 10 and 15 input words three failed to learn during the training. We can thus not test the significance of these differences.

Takeaway

The preliminary results indicate that three layers lead to better generalization than one layer for the LSTM models with 128 hidden units. For the LSTM models with 512 hidden units and 10 input words, the variant with

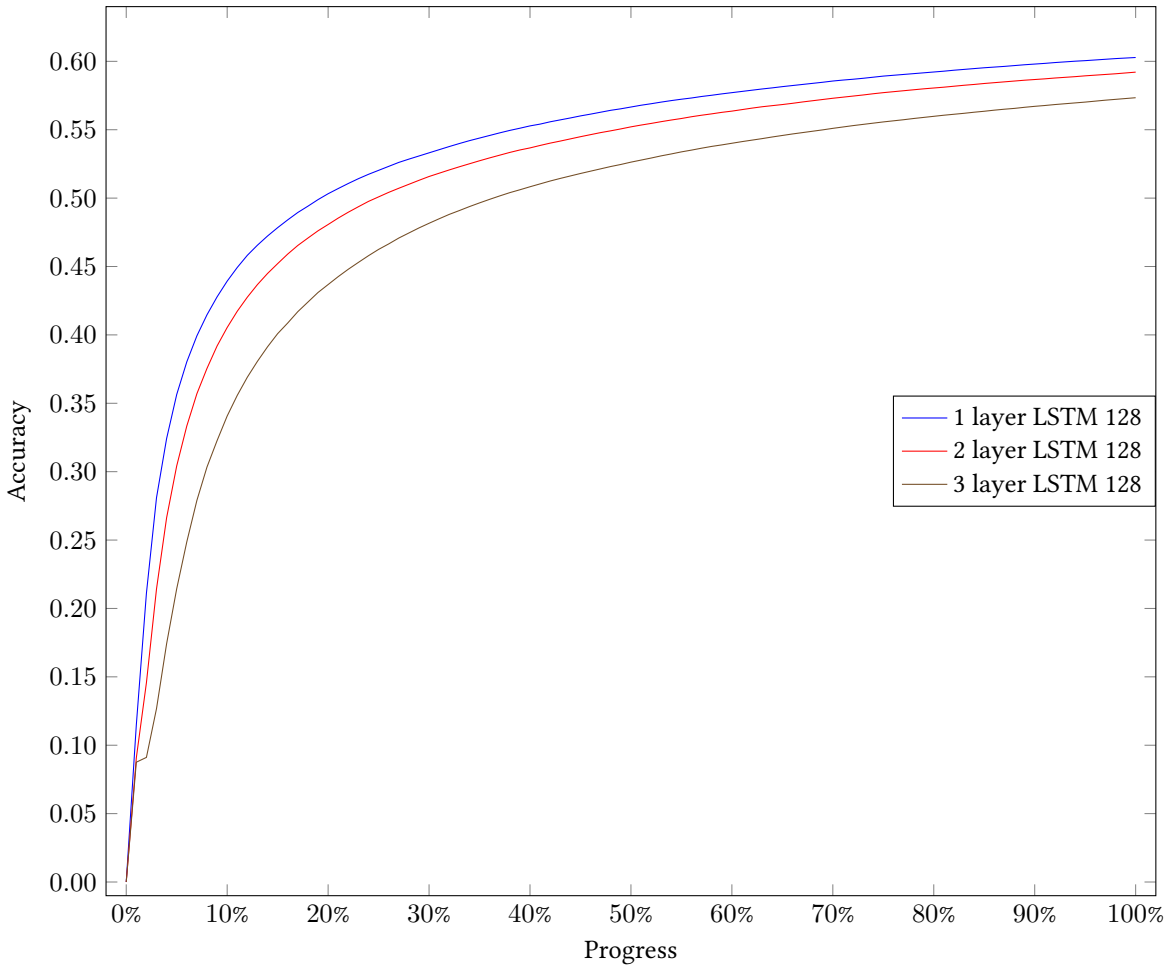


Figure 4.3: Training accuracy over one epoch of LSTM models with different numbers of layers

three layers fails to learn during training while the variant with a single layer is one of the best performing models in our experiments.

4.3.3 Input length

Table 4.7 shows that the accuracy of LSTM models with 128 and 512 hidden units increases when more word tokens are used as input up to 10 tokens. With 15 or 20 tokens as input, the models fail to learn and the accuracy does not improve during training. We observe a similar result with the character based models, where the CNN-LSTM with 128 hidden units achieves an accuracy of 0.3793 with an input length of 10 and an accuracy of 0.0850 with an input length of 20.

Input length	Hidden units	
	128	512
1	-	0.3141 0.3157
2	-	0.4341
5	0.5111	0.5073 0.5221
10	0.5401 0.5457	0.5516 0.5535
15	-	0.0841
20	0.0924	0.0841

Table 4.7: Validation accuracy of LSTM models with different numbers of hidden units and input lengths

SQ5: How does the length of the input for a network influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of networks with inputs of different lengths. The **research hypothesis** states that the performance of networks with longer input lengths will be higher than that of networks with shorter input lengths. We trained the models with 512 hidden units and one, five and 10 words as input two times and calculate the p-value to determine the statistical significance of our results. As described in § 3.5.3 we use the randomization model to test whether the observed differences are significant. For the difference between both one and five input words and five and 10 input words $p = 0.00$. With a significance level of $a = 0.01$ both differences are statistically significant thus we reject the null hypothesis and accept the research hypothesis for input lengths of one, five and 10. With input lengths of 15 and 20, the models fail to learn and the accuracy on the validation set is below 10%.

Takeaway

For the LSTM models with 512 hidden units, an input length of 10 words leads to a significantly higher accuracy than an input length of five words. An input length of five words leads to significantly higher accuracy than an input length of one word. The models in our experiments fail to learn with input lengths of 10 and 15 words.

4.3.4 Word vs. character input

Table 4.8 shows the validation accuracy for the different models with characters as input instead of words. The average length of the word input tokens is approximately 3.9 characters. When adding one character to separate the words 10 characters as input would thus contain on average approximately two word tokens.

SQ6: How does using words or characters as input for a network influence the prediction performance?

The **null hypothesis** states that there is no difference in the performance of networks with characters or words as input. The **research hypothesis** states that the performance of networks with characters as input

Embedding	Input length	Type	Hidden units	Accuracy
Lookup	2 words	LSTM	512	0.4341
	10 characters	LSTM	128	0.3902
			512	0.4067
		CNN + LSTM	128	0.3793
	20 characters	CNN + LSTM	128	0.0850
CNN	10 characters	LSTM	128	0.3937
			512	0.3902

Table 4.8: Validation accuracy of the models with character inputs

will be higher than that of networks with words as input. We compare the models with 10 characters as input with a word-based model with two tokens as input because as mentioned in § 4.3.4 the average token length is 3.9 characters. The word based LSTM model with 512 hidden units and two input words achieves a higher accuracy than all of the character based models. We trained these models a single time and report all results. Although the preliminary results indicate that the performance is lower with characters as input than with words as input without multiple results per model we do not reject the null hypothesis.

Takeaway

The preliminary results indicate that the accuracy is higher when using words as input than when using characters as input.

4.3.5 Number of epochs

Figure 4.4 shows the training and validation accuracy and loss of the single layer LSTM model with 128 hidden units and an input of five words. After the first epoch, the training accuracy keeps increasing and the loss decreasing while the opposite is the case for the validation accuracy and loss. The model is thus overfitting the training data from the second epoch on.

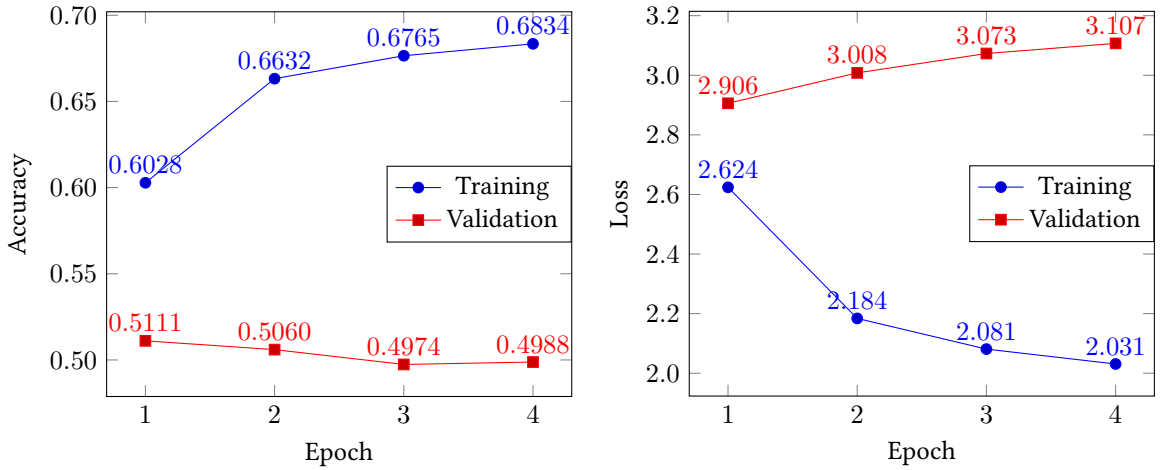


Figure 4.4: Accuracy and loss for the single Layer LSTM model with 128 hidden units and 5 words as input

4.3.6 Amount of training data

Table 4.9 shows that the single layer LSTM model with 512 hidden units achieves a higher accuracy when trained on more data. The accuracy of this model on the test set is 61.0%. The final training accuracy of the model trained on 2% of the data is 0.6468 while the training accuracies of the models trained on 1% of the data

Training tokens	16M	33M
Accuracy	0.5526*	0.5862

* This model is trained two times and the accuracy is the mean of two evaluations.

Table 4.9: Validation accuracy of the LSTM model with 512 hidden units trained on different amounts of data

are 0.6315 and 0.6194. These differences are smaller than the differences in the validation accuracy (0.5862 with 2% data vs. 0.5535 and 0.5516 with 1% data) which indicates that the generalization of the model is better.

Figure 4.5 shows the training accuracy of the three models over one epoch. The progress is relative to the 2% dataset.

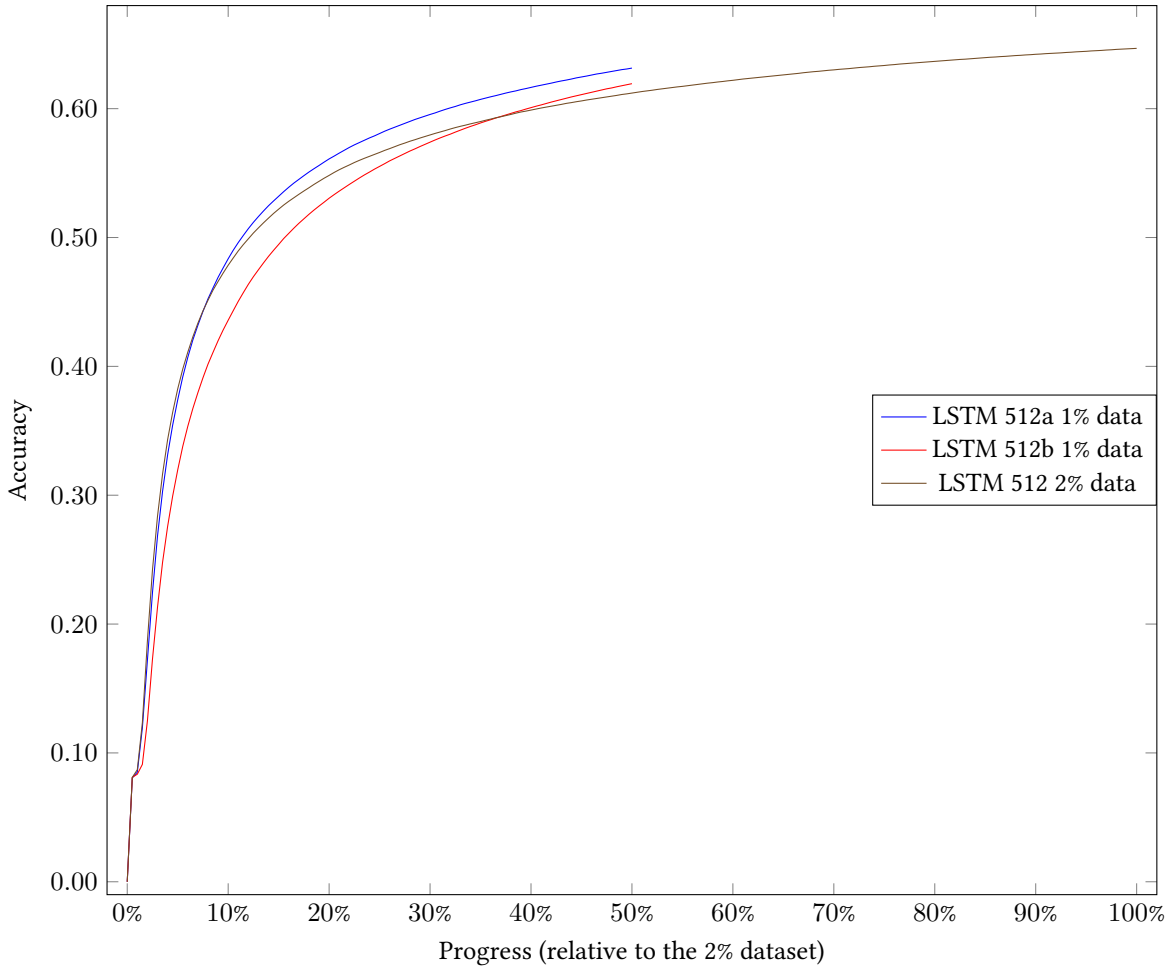


Figure 4.5: Training accuracy over one epoch of the single layer LSTM model with 512 hidden units and different amounts of data

4.3.7 Mean reciprocal rank

The single layer LSTM model with 512 hidden units trained on the 2% dataset achieves an MRR of 69.7% on the test set. To compare this result to the results from Hellendoorn and Devanbu [21] the 'static setting' with vocabulary limits is the most similar. With that setting the plain n-gram achieves an MRR of 58.0%, the LSTM/300 achieves an MRR of 66.1%, the LSTM/650 achieves an MRR of 67.9% and the n-gram with cache achieves an MRR of 75.7%. An important difference between the LSTM 512 model and the n-gram model with

cache is that the cache contains local information while the state of the LSTM is reset for every prediction and thus does not contain local information. The LSTM 512 contains only a single hidden layer with 512 units while LSTM/650 contains two hidden layers with 650 units. Furthermore, the LSTM/650 is trained for 39 epochs while the LSTM 512 is trained for a single epoch. Lastly, the LSTM 512 is trained on approximately 2% of the dataset while the other models are trained on 1% of the dataset.

4.4 Summary of the main results

To answer the research question '**How can recurrent neural networks be used for code completion?**' we summarize the results of the choices that achieved the highest accuracy in our experiments. The model with a **single LSTM layer, 512 hidden units, a lookup table embedding and 10 words as input trained on the largest of the two used datasets** achieved an accuracy of 0.5862 on the validation set which is the highest accuracy of all models in our results. The accuracy of this model on the test set is 61.0% and the MRR is 69.7%. The choice for LSTM's as recurrent units is motivated by the results achieved on natural language tasks and the intention to increase the input length further in the future. When using five words as input, the preliminary results indicate that the differences in the accuracy are very slight for the different types of units. In our experiments, the mean accuracy of models with 512 hidden units is significantly higher than that of models with 128 hidden units. The preliminary results indicate that increasing the number of hidden units further may lead to a higher performance with the drawback that models will then require more computational power to train. Currently, the choice of embedding and the input type are connected because the CNN is only used with characters as input. While preliminary results show a similar accuracy with a lookup table or a CNN we found that the word based models achieved a higher performance and are relatively less impacted by the problems we encounter with long input lengths. The input length of 10 words is motivated by the significant positive difference in the mean accuracy we found over using five words or one word as input.

Chapter 5

Discussion

5.1 Difficulty of the task

The differences in the performance of the models with different recurrent unit types in our experiments are small and the differences in the performance of the models with different numbers of hidden units are also small. While there are several differences in the setup of the experiments by Chung et al. [12] we can compare their findings to our results. The differences are described in § 3.2, the main difference is that in our experiments the number of hidden units for the different unit types is the same (128) which results in different numbers of parameters for the models while in the experiments by Chung et al. [12] the numbers of units for the RNN, LSTM, and GRU were chosen specifically to make the numbers of parameters approximately the same. Chung et al. [12] found that the LSTM and GRU models outperform the RNN models on different tasks with the exception of one of which they note that it is less challenging than the others. On the less difficult task, the difference between gated and non-gated units was smaller. Furthermore, training with longer sequences is more difficult[39] and five words is considered short. The results from the different models indicate that the task of predicting code tokens based on five previous tokens with an accuracy of approximately 51% may not be very challenging for recurrent neural networks. This is consistent with the finding from Hindle et al. [23] that there is regularity and repetitiveness in software code that can be modeled. Our results indicate that the task becomes more challenging when the aim is to achieve an accuracy over approximately 51%, when using more words as input and when building deeper models with more hidden layers.

5.2 Best architecture for code completion

Based on our results we present several suggestions to achieve the highest accuracy when designing a recurrent neural network model to predict code tokens based on previous tokens. These suggestions can be used both to build a code completion system and as a starting point or baseline for further research.

Number of hidden layers Using a single hidden layer in the network is recommended. This is mainly because training networks with more hidden layers is more difficult and we have seen only a small performance increase in the preliminary results compared to the influence of the input length.

Input length Using an input length of 10 is recommended. In our experiments, this is the maximum number of words or characters that we could train the models on. We have found that the performance with 10 words as input is significantly higher than with five words and that the performance with five words as input is significantly higher than with one word.

Number of hidden units Using 512 hidden units is recommended because in our experiments, the accuracy is significantly higher with 512 hidden units than with 128 hidden units. More hidden units may increase the performance further at a greater computational cost thus we recommend experimenting with increasing the number of hidden units when the training data is limited. Otherwise, we recommend training on more data as both will require extra computational power during training but a model with more hidden units also requires

more power when predicting while a model that is trained on more data does not. More training data should improve the generalization of the model and thus the accuracy.

Input type Using words as input is recommended because the preliminary results indicate that this leads to a higher accuracy.

Embedding type There is no recommendation for a specific embedding. In our experiments the model that achieves the highest accuracy has a lookup table embedding but this model was not compared to a variant with a CNN based embedding. The results from our experiments where we compared the accuracy of models with a lookup table embedding to the accuracy of models with a CNN based embedding are inconclusive.

Chapter 6

Threats to validity

We trained the models with similar accuracy scores in our comparisons only one time because time and computational power constraints required us to make a trade-off between training and evaluating multiple different models or evaluating the same models multiple times. Because there is not enough knowledge about the influence of different designs and hyperparameters specifically on the task of code completion and the similarity between natural language and programming language we choose to aim our research mainly at multiple different models. A sample size of one is too small to estimate the distribution of the performance of the different models and we can not assume that the distribution is normal[33] thus for the models that we trained a single time we did not test the significance of the observed differences and instead reported the results for these models as preliminary. We selected several models of which we tested whether the observed differences in the mean accuracy are significant and trained each of these models two times. This sample size is very small and because the inferences we made with the randomization model are limited to the models in the experiment we acknowledge that it would be preferable to have more trained models that could be used to test the significance of the observed differences.

The accuracy and MRR assign a score for the performance of the different models. The exact relation between this score and the usefulness of a model that makes code completion suggestions is unknown. Proksch, Lerch, and Mezini [44] have shown that too many suggestions can decrease the usefulness of a code recommender. Suggestions that appear lower in the list are less useful because they require more actions from the user to be selected. Using the MRR as an indication for code completion performance is based on the assumption that a correct suggestion is half as useful in the second position than the first, a third as useful in the third position etc. and it is not checked whether this assumption is correct.

Chapter 7

Conclusions

In this work, we presented recommendations for using recurrent neural networks for code completion. We show that in our experiments a model based on these recommendations achieved an accuracy of 61.0% and an MRR of 69.7% on the test set. We reported results to evaluate the influence of the choice of recurrent unit, the choice of embedding, the number of hidden units and layers in a network and the input length and type on the accuracy on the code completion task. We showed that in our experiments the mean accuracy of models with 512 hidden units is significantly higher than the mean accuracy of models with 128 hidden units. Furthermore, we showed that with the models in our experiments using 10 words as input lead to a significantly higher mean accuracy than using five words as input and that using five words as input lead to a significantly higher mean accuracy than using one word as input.

7.1 Future work

7.1.1 Input length

In future work the influence of different input lengths on the prediction accuracy can be explored. From one to five to 10 words the accuracy of the models in our experiments increased but with 15 words our models were unable to learn. We believe it is worth researching methods to train models on longer input lengths because there is more information contained in 15 words and intuitively this extra information would help achieve a higher accuracy if the models were able to utilize this. Furthermore, on several other tasks LSTM based networks have successfully learned dependencies with distances well over 15 words which makes it interesting to study why we observed a great increase in difficulty specifically for code completion.

7.1.2 Programming languages

The similarities and differences with code completion for other programming languages than Java can be studied. The method and models we presented can be easily adapted to experiment with other languages. We expect that the results would be similar for other static languages such as C#, but that they could be different for dynamic languages such as Python. Not only can the accuracy of different models be compared to evaluate whether the influence of specific design choices differs for different languages but with comparable models, the differences in repetitiveness and regularity of different languages could be studied.

7.1.3 Influence of stacked layers on generalization

As mentioned in § 4.3.2 we observed in our experiments that models achieved a lower training accuracy with more layers but a higher accuracy while in nearly all other comparisons a higher training accuracy corresponded to a higher validation accuracy. This is an interesting observation that can be further studied because we achieved the highest accuracy with a single model and the variant of this model with three layers failed to learn but may achieve a higher accuracy when it can be successfully trained.

Bibliography

[1] Miltiadis Allamanis and Charles Sutton. "Mining Source Code Repositories at Massive Scale using Language Modeling". In: *The 10th Working Conference on Mining Software Repositories*. IEEE. 2013, pp. 207–216. URL: https://groups.inf.ed.ac.uk/cup/javaGithub/.

[2] Fabio Villamarin Arrebola and Plinio Thomaz Aquino Junior. "On Source Code Completion Assistants and the Need of a Context-Aware Approach". In: *Human Interface and the Management of Information: Supporting Learning, Decision-Making and Collaboration*. Ed. by Sakae Yamamoto. Cham: Springer International Publishing, 2017, pp. 191–201. ISBN: 978-3-319-58524-6. URL: https://link.springer.com/chapter/10.1007/978-3-319-58524-6_17.

[3] Matej Balog et al. "DeepCoder: Learning to Write Programs". In: *CoRR* abs/1611.01989 (2016). arXiv: 1611.01989.

[4] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994), pp. 157–166. ISSN: 1045-9227. DOI: 10.1109/72.279181.

[5] D. Brezak et al. "A comparison of feed-forward and recurrent neural networks in time series forecasting". In: *2012 IEEE Conference on Computational Intelligence for Financial Engineering Economics (CIFEr)*. Mar. 2012, pp. 1–6. DOI: 10.1109/CIFEr.2012.6327793.

[6] Jason Brownlee. *How to Develop Word-Based Neural Language Models in Python with Keras*. 2018. URL: https://machinelearningmastery.com/develop-word-based-neural-language-models-python-keras/.

[7] Michael Anthony Cabot, Master Artificial Intelligence, and MW van Someren. "Automated Docstring Generation for Python Functions". MA thesis. University of Amsterdam, 2014. URL: https://esc.fnwi.uva.nl/thesis/centraal/files/f288690963.pdf.

[8] Rich Caruana, Steve Lawrence, and C. Lee Giles. "Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping". In: *Advances in Neural Information Processing Systems 13*. Ed. by T. K. Leen, T. G. Dietterich, and V. Tresp. MIT Press, 2001, pp. 402–408. URL: https://papers.nips.cc/paper/1895-overfitting-in-neural-nets-backpropagation-conjugate-gradient-and-early-stopping.pdf.

[9] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078.

[10] F. Chollet. *Deep Learning with Python*. Manning Publications Company, 2017. ISBN: 9781617294433. URL: https://www.manning.com/books/deep-learning-with-python.

[11] François Chollet et al. *Keras*. 2015. URL: https://keras.io.

[12] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555 (2014). arXiv: 1412.3555.

[13] Hoa Khanh Dam, Truyen Tran, and Trang Pham. "A deep language model for software code". In: *CoRR* abs/1608.02715 (2016). arXiv: 1608.02715.

[14] Subhasis Das and Chinmayee Shah. "Contextual Code Completion Using Machine Learning". In: (2015).

[15] Michael D. Ernst. "Permutation Methods: A Basis for Exact Inference". In: *Statist. Sci.* 19.4 (Nov. 2004), pp. 676–685. DOI: 10.1214/088342304000000396.

[16] Eclipse Foundation. *Eclipse IDE*. 2018. URL: https://www.eclipse.org.

[17] Jonas Gehring et al. “Convolutional Sequence to Sequence Learning”. In: *CoRR* abs/1705.03122 (2017). arXiv: 1705.03122.

[18] Ian Goodfellow et al. “Maxout Networks”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1319–1327. URL: http://proceedings.mlr.press/v28/goodfellow13.html.

[19] Jiatao Gu et al. “Incorporating Copying Mechanism in Sequence-to-Sequence Learning”. In: *CoRR* abs/1603.06393 (2016). arXiv: 1603.06393.

[20] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program Synthesis”. In: *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pp. 1–119. ISSN: 2325-1107. DOI: 10.1561/2500000010.

[21] Vincent J. Hellendoorn and Premkumar Devanbu. “Are Deep Neural Networks the Best Choice for Modeling Source Code?” In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 763–773. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106290.

[22] Rosco Hill and Joe Rideout. “Automatic Method Completion”. In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. ASE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 228–235. ISBN: 0-7695-2131-2. DOI: 10.1109/ASE.2004.19.

[23] Abram Hindle et al. “On the Naturalness of Software”. In: *Commun. ACM* 59.5 (Apr. 2016), pp. 122–131. ISSN: 0001-0782. DOI: 10.1145/2902362.

[24] Geoffrey E. Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *CoRR* abs/1207.0580 (2012). arXiv: 1207.0580.

[25] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[26] Rafal Józefowicz et al. “Exploring the Limits of Language Modeling”. In: *CoRR* abs/1602.02410 (2016). arXiv: 1602.02410.

[27] Lukasz Kaiser, Aidan N. Gomez, and François Chollet. “Depthwise Separable Convolutions for Neural Machine Translation”. In: *CoRR* abs/1706.03059 (2017). arXiv: 1706.03059.

[28] Nitish Shirish Keskar and Richard Socher. “Improving Generalization Performance by Switching from Adam to SGD”. In: *CoRR* abs/1712.07628 (2017). arXiv: 1712.07628.

[29] Nitish Shirish Keskar et al. “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”. In: *CoRR* abs/1609.04836 (2016). arXiv: 1609.04836.

[30] Yoon Kim et al. “Character-Aware Neural Language Models”. In: *CoRR* abs/1508.06615 (2015). arXiv: 1508.06615.

[31] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980.

[32] Twisted Matrix Laboratories. *Twisted Library Source Code*. 2017. URL: https://github.com/twisted/twisted.git.

[33] S. Lawrence et al. “On the distribution of performance from multiple neural-network trials”. In: *IEEE Transactions on Neural Networks* 8.6 (Nov. 1997), pp. 1507–1517. ISSN: 1045-9227. DOI: 10.1109/72.641472.

[34] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[35] Jian Li et al. “Code Completion with Neural Attention and Pointer Networks”. In: *CoRR* abs/1711.09573 (2017). arXiv: 1711.09573.

[36] Ilya Loshchilov and Frank Hutter. “Fixing Weight Decay Regularization in Adam”. In: *CoRR* abs/1711.05101 (2017). arXiv: 1711.05101.

[37] Enrique Manjavacas et al. “Synthetic Literature: Writing Science Fiction in a Co-Creative Process”. In: *Proceedings of the Workshop on Computational Creativity in Natural Language Generation (CC-NLG 2017)*. Santiago de Compostela, Spain: Association for Computational Linguistics, 2017, pp. 29–37. URL: https://aclweb.org/anthology/W17-3904.

[38] Andrew Y. Ng. "Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance". In: *Proceedings of the Twenty-first International Conference on Machine Learning*. ICML '04. Banff, Alberta, Canada: ACM, 2004, pp. 78–. ISBN: 1-58113-838-5. DOI: 10.1145/1015330.1015435.

[39] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "Understanding the exploding gradient problem". In: *CoRR* abs/1211.5063 (2012). arXiv: 1211.5063.

[40] Razvan Pascanu et al. "How to Construct Deep Recurrent Neural Networks". In: *CoRR* abs/1312.6026 (2013). arXiv: 1312.6026.

[41] Rohit Prabhavalkar et al. "A Comparison of Sequence-to-Sequence Models for Speech Recognition". In: 2017. URL: https://ai.google/research/pubs/pub46169.

[42] Lutz Prechelt. "Early Stopping - But When?" In: *Neural Networks: Tricks of the Trade*. Ed. by Genevieve B. Orr and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 55–69. ISBN: 978-3-540-49430-0. DOI: 10.1007/3-540-49430-8_3.

[43] Sebastian Proksch, Sven Amann, and Mira Mezini. "Towards standardized evaluation of developer-assistance tools". In: *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering*. ACM. 2014, pp. 14–18. URL: https://link.springer.com/chapter/10.1007/978-3-319-58524-6_17.

[44] Sebastian Proksch, Johannes Lerch, and Mira Mezini. "Intelligent Code Completion with Bayesian Networks". In: *ACM Trans. Softw. Eng. Methodol.* 25.1 (Dec. 2015), 3:1–3:31. ISSN: 1049-331X. DOI: 10.1145/2744200.

[45] Veselin Raychev, Martin Vechev, and Eran Yahav. "Code Completion with Statistical Language Models". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 419–428. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594321.

[46] Luiz Laerte Nunes da Silva Junior, Alexandre Plastino, and Leonardo Gresta Paulino Murta. "What Should I Code Now?" In: *Journal of Universal Computer Science* 20.5 (May 1, 2014), pp. 797–821. DOI: 10.3217/jucs-020-05-0797.

[47] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. "LSTM neural networks for language modeling". In: *Thirteenth Annual Conference of the International Speech Communication Association*. 2012. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.248.4448.

[48] M. Sundermeyer et al. "Comparison of feedforward and recurrent neural network language models". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. May 2013, pp. 8430–8434. DOI: 10.1109/ICASSP.2013.6639310.

[49] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 3104–3112. URL: https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf.

[50] Igor V. Tetko, David J. Livingstone, and Alexander I. Luik. "Neural network studies. 1. Comparison of overfitting and overtraining". In: *Journal of Chemical Information and Computer Sciences* 35.5 (1995), pp. 826–833. DOI: 10.1021/ci00027a006.

[51] Linus Torvalds et al. *Linux Kernel Source Code*. 2015. URL: https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.2.3.tar.xz.

[52] M. White et al. "Toward Deep Learning Software Repositories". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. May 2015, pp. 334–345. DOI: 10.1109/MSR.2015.38.

[53] A. C. Wilson et al. "The Marginal Value of Adaptive Gradient Methods in Machine Learning". In: *ArXiv e-prints* (May 2017). arXiv: 1705.08292.

[54] J.C.F. de Winter. "Using the Student's t-test with extremely small sample sizes." In: *Practical Assessment, Research & Evaluation* 18.10 (2013). URL: https://pareonline.net/getvn.asp?v=18&n=10.

[55] Wenpeng Yin et al. "Comparative Study of CNN and RNN for Natural Language Processing". In: *CoRR* abs/1702.01923 (2017). arXiv: 1702.01923.