Utilizing Lingual Structures to Enhance Transformer Performance in Source Code Completions

Version of August 8, 2022

Jonathan Bernhard Katzy

Utilizing Lingual Structures to Enhance Transformer Performance in Source Code Completions

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jonathan Bernhard Katzy
born in St. Gallen, Switzerland

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Utilizing Lingual Structures to Enhance Transformer Performance in Source Code Completions

Author: Jonathan Bernhard Katzy
Student id: 4460235
Email: J.B.Katzy@student.tudelft.nl

Abstract

We explored the effect of augmenting a standard language model's architecture (BERT) with a structural component based on the Abstract Syntax Trees (ASTs) of the source code. We created a universal abstract syntax tree structure that can be applied to multiple languages to enable the model to work in a multilingual setting. We adapted the general graph transformer architecture [20] to function as the structural component of the transformer. Furthermore, we extended the Embeddings from Language Models (ELMo) [47] style embeddings to work in a multilingual setting when working with incomplete source code. The final results showed that the multilingual setting was beneficial to achieving higher quality embeddings for the embedding model, however, monolingual models performed better in most cases for the transformer model. The addition of ASTs resulted in increased performance in the best performing models on all languages, while also reducing the need for a pretraining task to achieve the best performance. The largest increase in performance for a Java model compared to its baseline counterpart was 3.0% on average on the test set, the largest increase in performance for a Julia model compared to its baseline counterpart was 1.1% on average on the test set, and the largest increase in performance of a CPP model compared to its baseline counterpart was 5.7% on average on the test set.

Thesis Committee:

Chair:	Professor Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Assistant Professor Dr. M. Aniche, Faculty EEMCS, TU Delft
Committee Member:	Assistant Professor Dr. G. Migut, Faculty EEMCS, TU Delft

Preface

This thesis was written for the completion of the degree of Master of Science in Computer Science at the Delft University of Technology. I would like to thank the people who have made it possible for this thesis to be completed by giving guidance during the process.

Maurício Aniche and Amir Mir, I would like to thank you for the guidance during the process of creating this thesis, the assistance in using the correct tools to create the model, the input in designing a setup to fairly evaluate the model and the input in creating a completed thesis report.

Jonathan Bernhard Katzy
Delft, The Netherlands
August 8, 2022

Contents

List of Figures

List of Tables

Table of abbreviations

Abbreviation	Meaning
NLP	Natural Language Processing
ML	Machine Learning
GPU	Graphics Processing Unit
TPU	Tensor Processing Unit
MRR	Mean Reciprocal Rank
BOS	Beginning Of String
EOS	End Of String
EOP	End Of Prediction
FFN	Feed Forward Network
MLP	MultiLayer Perceptron
LSTM	Long Short-Term Memory (Network)
Bi-LSTM	Bi-directional Long Short-Term Memory(Network)
GRU	Gated Recurrent Unit
RNN	Recurrent Neural Network
Bi-RNN	Bi-directional Recurrent Neural Network
GNN	Graph Neural Network
GGNN	Gated Graph Neural Network
GAT	Graph Attention Transformer
GPT	General Purpose Transformer [48, 49, 12]
BERT	Bidirectional Encoder Representations from Transformers [18]
AST	Abstract Syntax Tree

Table 1: Table of common abbreviations used throughout this thesis.

Chapter 1

Introduction

The use of IDEs (integrated development environment) when developing source code is ubiquitous within the developer community. This is due to the amount of time they can save a developer through executing tasks such as code completion, testing, and debugging. The problem of code completion can be defined as predicting the next sequence of characters the developer will type given the provided input. Currently, the main method used by many IDEs is intelligent code completion, which uses an in-memory database of attributes and methods that can be searched to give a suggestion [13]. This method however can mostly only predict one method name at a time and has problems when suggesting rarely used methods. Furthermore, each separate programming language needs to have its specific implementation for it to work. There have been attempts at solving the problem of predicting longer snippets of text through the use of LSTMs and transformers that have shown promise, however, these models will Largely only work for one programming language. Furthermore, many IDEs support automatic code highlighting based on their abstract syntax trees (ASTs). The presence of ASTs in IDEs as well as a need for a more intelligent code completion system raises the goal for this thesis.

The goal of this thesis is to create a unified multilingual unsupervised code completion model capable of automatically completing code in various programming languages, we attempt to incorporate cross-language learning through using multilingual embeddings, and maximizing available information to be fed to the model through the inclusion of the abstract syntax trees.

We propose a 2-step approach to solve this issue. The first step of the approach is to learn a high-quality multilingual embedding for source code. This multilingual embedding will then be used to boost the performance of a multilingual transformer model trained to use both the information present in the ASTs of each language and the code itself. The ASTs are transformed into universal ASTs meant to help the model abstract away from any specific language, instead, it learns to reason over the general structure of source code.

For the models in this thesis, we strive to require as little supervision as possible, to this end the models are trained on a dataset that is scraped from GitHub[1] containing the top 100 projects for Julia, Java, and CPP. The models trained on these 100 projects will finally be

[1] github.com

evaluated on both a validation set containing 10% of the files from the initial dataset, as well as a final dataset containing the next 10 most starred projects from Github. This is done to see the performance on truly unseen data.

This report will contain an overview of similar models and attempts to predict source code in the background theory, chapter 2. We then move on to define the model being used as well as its scaling capabilities compared to other models in chapter 3, after which we define the setup of the experiments in chapter 4. We wrap up this investigation by discussing the results of the experiments in chapter 5, and conclude with recommendations for future work in chapter 6.

1.1 Research Questions

To give a clear overview of what we aim to achieve in this work we will lay out the research questions in this section. As there is a clear divide in the work between the embedding and final prediction, we choose to split the research into 2 parts. First we will list what we want to test for the multilingual embedding of source code, and then we will list the questions we aim to answer regarding the code completion.

Embedding Initially we want to know if there is any knowledge to be learned from different languages when learning an embedding. As we have chosen to work with two popular languages, CPP and Java, and one relatively new language, Julia, we want to know in what scenarios Julia can benefit from training together with CPP and Java. The exact questions we want to answer are summarized in the questions below.

RQ1. Does training on multiple languages help to find higher quality embeddings?

RQ2. Is pre-training on a set of languages beneficial when fine-tuning on a final target language?

Code prediction Once we are aware of the effect of multiple language on learning embeddings, we will attempt to see in what scenarios these embeddings are useful for the final model, as well as analysing how the resource scarce language, Julia, can benefit from any multilingual information. The questions below summarize what we are investigating in this paper.

RQ3. Does adding a structural component to the BERT [18] architecture benefit the performance?

RQ4. Does training on multiple languages benefit the model?

Chapter 2

Background Research

To gain a good understanding of what areas have been researched when it comes to the areas of machine learning for software engineering, multilingual embeddings, and transformer models, we will split the research into three main sections. First, we will focus on the first part of the project, we will look at how embeddings are usually calculated and how to apply this to a multilingual setting. Then we will review the state-of-the-art graph-based transformers that have been developed for any task as well as looking at the methodology behind the state-of-the-art transformer models. Finally, we will take a closer look at different papers in the area of machine learning for software engineering, focusing on identifying promising trends and the nature of problems being solved using machine learning so far. We start with a quick overview of the papers studied in-depth, given in table 2.1.

Title	Year	Area
Efficient Estimation of Word Representations in Vector Space [44]	2013	Embeddings
context2vec: Learning generic context embedding with bidirectional lstm [43]	2016	Embeddings
Deep contextualized word representations [47]	2018	Embeddings
Unsupervised Cross-lingual Word Embedding by Multilingual Neural Language Models [62]	2018	Multilingual embeddings
Emerging Cross-lingual Structure in Pre-trained Language Models [16]	2019	Multilingual embeddings
SCELMo: Source Code Embeddings from Language Models [32]	2020	Embeddings
BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding [18]	2019	Pre-training transformers
Learning and Evaluating Contextual Embedding of Source Code [31]	2019	Pre-trained transformers
Improving Language Understanding by Generative Pre-Training [48]	2018	Pre-training transformers
Language Models are Unsupervised Multitask Learners [49]	2019	Pre-trained Transformers
Language Models are Few-Shot Learners [12]	2020	Pre-trained transformers
IntelliCode Compose: Code Generation Using Transformer [57]	2020	Code Completion
Learning to Represent Programs with Graphs [4]	2017	Graph based approaches
Global Relational Models of Source Code [24]	2019	Graph based approaches
code2vec: Learning Distributed Representations of Code [8]	2018	Graph based approaches
code2seq: Generating Sequences from Structured Representations of Code [7]	2018	Graph based approaches
A general path-based representation for predicting program properties [6]	2018	AST representations
Code Prediction by Feeding Trees to Transformers [33]	2020	Code completion
Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs [63]	2021	Code completion
Language Modelling for Source Code with Transformer-XL [19]	2020	Code Completion
Graph Attention Networks [61]	2017	Graph transformer
A Generalization of Transformer Networks to Graphs [20]	2020	Graph transformer
Graph Transformer Networks [66]	2019	Graph transformer
Pythia: AI-assisted Code Completion System [56]	2019	Method naming
CodeFill: Multi-token Code Completion by Jointly Learning from Structure and Naming Sequences [28]	2022	Code Completion
InCoder: A Generative Model for Code Infilling and Synthesis [22]	2022	Code Completion

Table 2.1: Papers studied in depth for the background research.

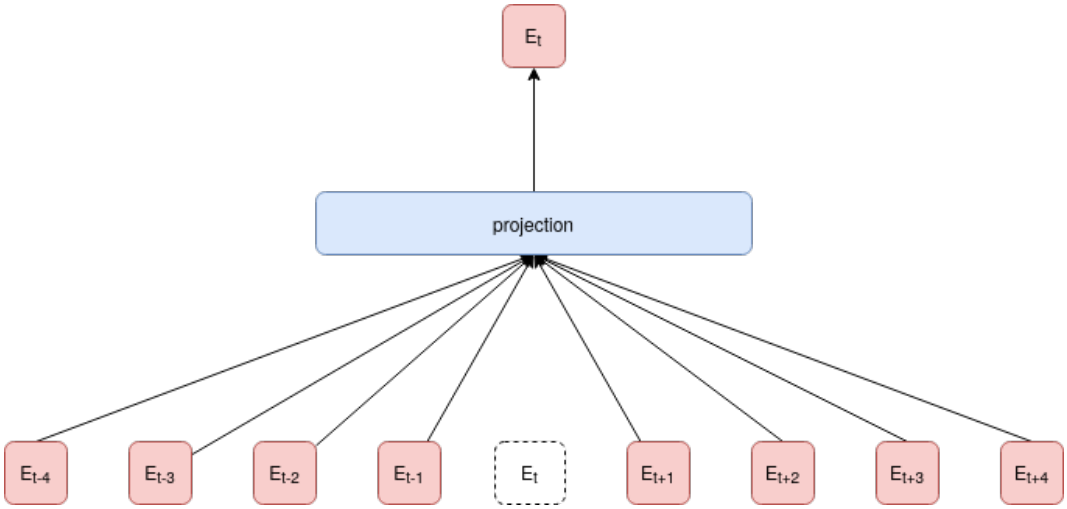


Figure 2.1: CBOW takes the surrounding tokens to predict the target token.

2.1 Multilingual Embeddings

In the area of Natural language processing the best performing models seem to be contextual models that embed the words according to their neighbours [44, 43]. The original paper published on this subject is now best known as Word2Vec [44]. However, there have been many different attempts at improving the Word2Vec model. We will begin by giving a quick overview of the Word2Vec model. After this, we will explain how the idea of contextual embeddings in one language can be expanded into a multilingual setting. We then show the methods that have been developed since the publishing of the Word2Vec model and how they are of interest when it comes to working with source code.

In the Word2Vec paper, a continuous bag of words (CBOW) model is proposed that uses a fixed window around each word to predict the embedding of the word (an overview of the model is given in figure 2.1). The benefits of learning the embeddings of a word by its context have been shown to exhibit relationships within the data [44]. Furthermore, to increase the accuracy of the model a negative sampling objective function is used. This objective function was chosen to boost performance [44] and works by giving the model negative samples, which are samples for which the distance needs to be maximized to achieve a better result.

The idea of contextual embeddings has been extended past the use of a feed-forward layer to also include language models based on LSTMs [43, 47, 16]. The general idea in these models is largely the same, in most models, the target token is predicted based upon the preceding and succeeding context in the sequence. However, other than in the CBOW model most models do not have a fixed window size but work on the entire sentence surrounding the word [43, 62, 47]. The predictive task of the LSTM models can vary slightly. Whilst some models may predict a probability distribution over the tokens in the vocabulary [62, 47], other models may attempt to map the embedding of a word to its context [43].

A basic contextual embedding that maps the embedding of a word to the embedding

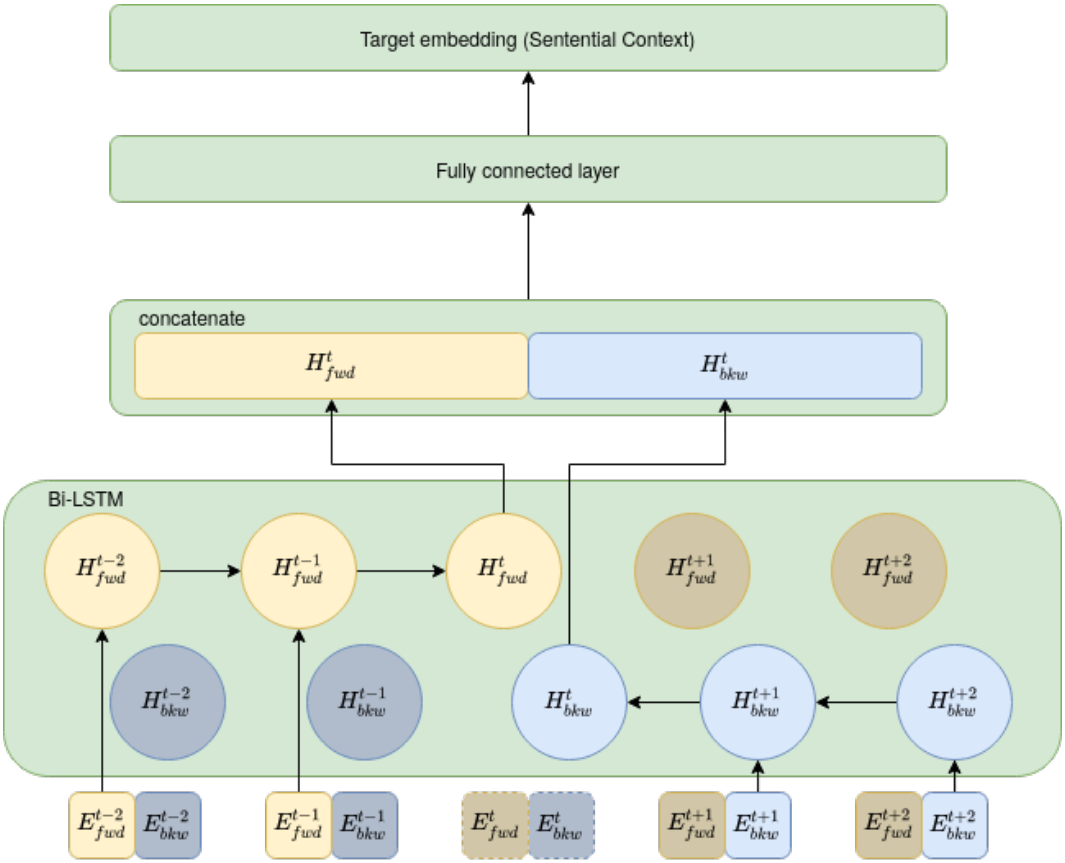


Figure 2.2: Context2vec model and training setup.

of the context is the context2vec model [43]. In this model, each token has a forward and backward embedding that are used as inputs to a Bi-LSTM. This Bi-LSTM then concatenates its forward LSTM and backward LSTM output and feeds it to a combining MLP to give a prediction of the embedding of the token (see figure 2.2). This model is very similar to the Word2Vec model as it also uses a negative sample objective function to maximize the distance to wrong predictions as well as minimize the distance to correct predictions. However other than the Word2Vec model this model uses the entire sentence to predict the word embedding rather than a fixed window. In the paper, they refer to this as the sentential context of the word [43].

When moving into the multilingual settings there has been some attempt to make use of 'anchor points' [16]. These anchor points are tokens in the dataset that according to [16] are the same across languages. Applying this idea to the Wikipedia corpus for cross-lingual embeddings we can use trans-lingual words such as "DNA" and "Paris" as anchor points. These are words that have the same meaning in multiple languages [16]. This idea is also used in other models that are built for multilingual rather than cross-lingual embeddings. When wanting to build a proper multilingual embedding that is fully unsupervised it can not be relied upon that all languages use the same alphabet or characters. For some works,

the only common structure that can be found is the beginning and ending of a sentence [62]. In their model Wada et al. [62] found significant benefits for resource-scarce languages when training embeddings relative to the EOS (end of sentence) and BOS (beginning of sentence) tokens [62].

The use of Bi-LSTMs has also shown promise when using Embeddings from Language Models (ELMo) [47]. The ELMo model has been shown to be an effective embedding for source code in the domain of bug detection [32] where the embeddings are used to improve performance in a preexisting bug detection model. We will however begin by explaining how the ELMo architecture works. ELMo learns its embedding by training a model to predict the next token in a sequence using multiple Bi-LSTMs (the training task is given in figure 2.3) this training task learns a set of forward (fwd) and backward (bkw) language models for a corpus that maximizes equation 2.1.

$$\sum_{k=1}^{N}(logp(t_k|t_1,...,t_{k1};\theta_x,\overrightarrow{\theta_{LSTM}},\theta_s)+logp(t_k|t_{k+1},...,t_N;\theta_x,\overleftarrow{\theta_{LSTM}},\theta_s)) \tag{2.1}$$

Given the parameters learned by this model ELMo concatenates all fwd and bkw language models and the embedding parameters into one vector which is the embedding of token k:

$$ELMo_k = E(R_k;\theta_e) \tag{2.2a}$$

where:

$$R_k = \{x_k^{LM},\overrightarrow{h_{k,j}^{LM}},\overleftarrow{h_{k,j}^{LM}}|j=1,...,L\} \tag{2.2b}$$

These embeddings are later optimized per task using the scaling factor γ and a softmax function s per task. This all gives the final ELMo model for a task given in 2.3. The initial Soft-Max function used for pre-training the embeddings is discarded after the pre-training [47].

$$ELMo_k^{task} = E(R_k;\theta^{task}) = \gamma^{task}\sum_{j=0}^{L}s_j^{task}h_{k,j}^{LM} \tag{2.3}$$

To evaluate the performance of ELMo style embeddings an ELMo model (called SCELMo) was trained on 150,000 JavaScript files. These embeddings were then compared to Word2Vec [44] and FastText [10] embeddings when training a model on bug detection. The SCELMo embeddings were able to detect 92.11% of all Swapped Argument bugs compared to 87.38% and 89.55% with Word2Vec and FastText embeddings respectively, also the SCELMo embeddings helped boost performance to 100% when detecting wrong binary operator bugs compared to 91.05% with Word2Vec and 91.11% with FastText, finally, the SCELMo embeddings increased the performance of detecting wrong binary operands to 84.25% compared to 77.06% with Word2Vec and 79.74% with FastText. This hints that using high-quality embeddings in source code tasks can lead to a significant performance boost.

2.2 Code representations using ASTs

Finally, we will lay out a few methods used to create representations of source code ASTs that will be used in further papers mentioned in this chapter. Few papers deal exclusively

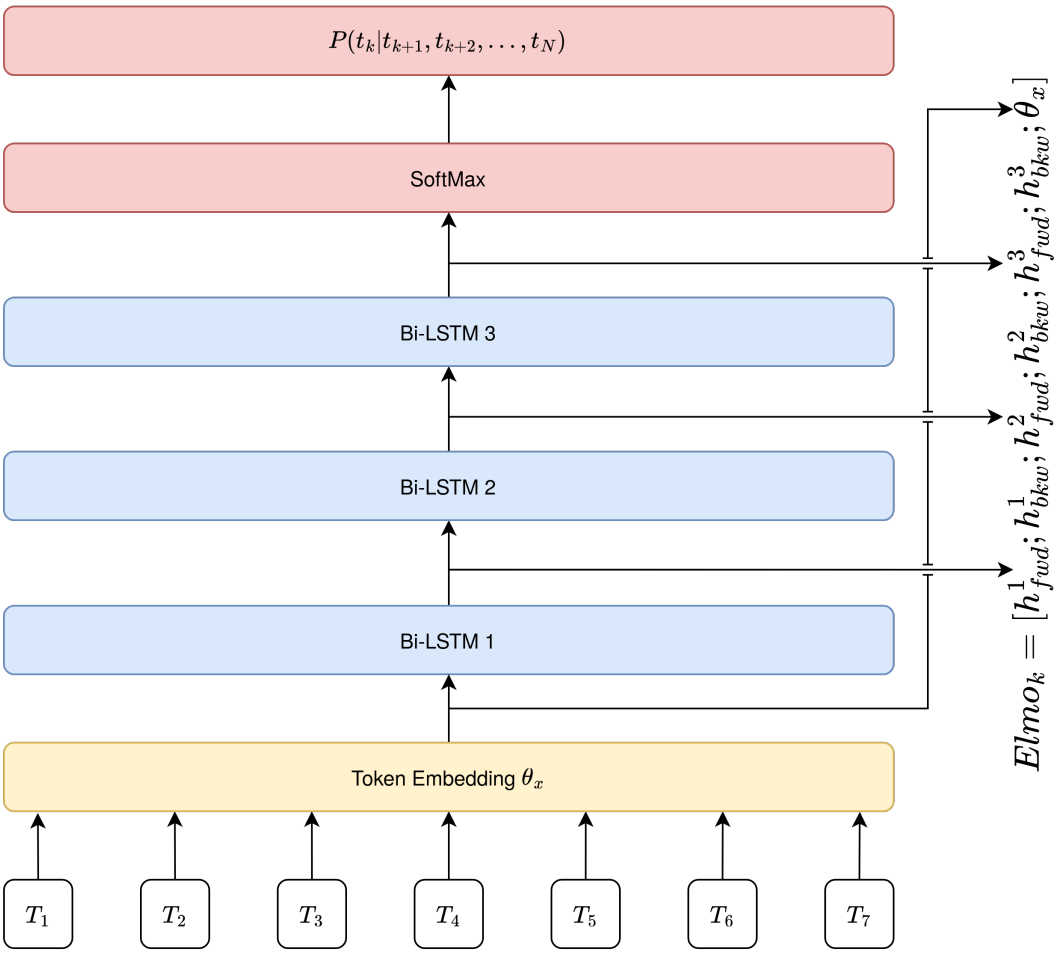


Figure 2.3: ELMo concatenates the outputs of multiple LSTMs to create an embedding.

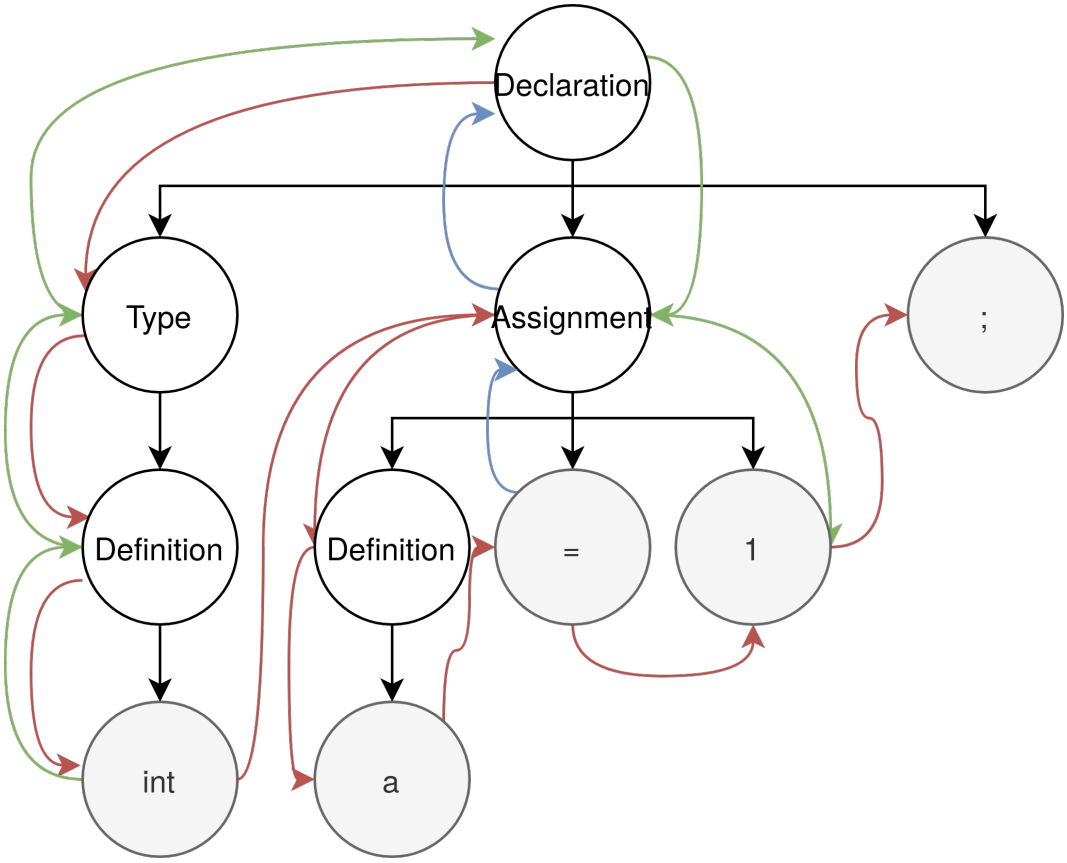


Figure 2.4: The different ways to represent an AST as a sequence. Red: Pre-Order Traversal, Blue: RootPath, green: Leaf2Leaf.

Method	Representation
Leaf2Leaf	[int $\uparrow$ *Definition* $\uparrow$ *Type* $\uparrow$ *Declaration* $\downarrow$ *assignment* $\downarrow$ 1]
RootPath	[= Assignment Declaration]
pre-order Visiting	[Declaration Type Definition int Assignment Definition a = 1 ;]

Table 2.2: Sequence representation of the AST shown in figure 2.4, the arrows are also included in the Leaf2Leaf representation.

with the generation of high-quality embeddings of code using ASTs. In many conventional cases, the structure of the AST is learned through tracing paths through the AST and using this as an input to a model [7, 8, 6, 33].

These four papers use two similar ways to represent a program with AST paths and one more novel method. The first is to trace a set of paths between 2 leaf nodes [7, 8, 6] and the second is to trace paths from the leaf nodes to the root [33]. The third method of ordering the nodes into a sequence is using a pre-order traversal from the root. All three methods are depicted for a generic AST in figure 2.4.

The representations from figure 2.4 are given in table 2.2, in this table for the Leaf2Leaf

and RootPath representation we only give an example for 1 node/leaf pair, however in the paper multiple of the representations are concatenated. These inputs are then fed to a fully connected network [8], an LSTM [7], or a transformer [33].

Of the four papers exploring this option of code representations, only one [33] works with code completion. However, we will also look at the performance of the other papers to get an idea of how the models benefited from the structural information on other source-code-related tasks in later sections.

2.3 Transformer models

To explore what transformer models are used in state-of-the-art applications, we will first discuss the architecture and benefits of the widely popular GPT and BERT architectures. Following up on the pre-trained language models we explain the use of transformer models that have been adapted to work on graph based datasets.

2.3.1 Pre-trained Language models

Although contextual embeddings have become very popular due to the work of Mikolov et al. [44] and its ability to increase the performance of models on later tasks. However, more recent works have been published on pre-training models on large datasets which can be later fine-tuned on any task. More specifically for source-code using the BERT [18] and GPT[48, 49, 12] architecture in the code understanding BERT (CuBERT) [31] model and GPT-C [57] architecture respectively. The BERT architecture consists of a bidirectional encoder transformer [60] which is pre-trained on unsupervised tasks before being retrained on a downstream task. The GPT architecture on the other hand is a more traditional language model as it works in the direction of the sequence, not using any information ahead of the current token, which BERT and Bi-LSTMs do use. We will begin by describing the model and methodology behind the BERT and GPT papers, the results when adapted to source-code tasks will be discussed in later sections.

BERT The very popular Bidirectional Encoder Representations from Transformers (BERT) [18] is a transformer model that works by learning a general language model using pre-training tasks on a large corpus of data which can later be fine-tuned to work on downstream tasks that may have less data, or needs to be learned in less time.

The architecture of the model, shown in figure 2.5 can be seen to be a basic encoder already described in the original transformer paper [60]. The most notable difference to a standard transformer is the use of an attention mask only for padding tokens and pre-training tasks when training.

The attention mask is what lets the transformer know what information it is allowed to see. Most models will have a self-attention mask and an encoder-decoder attention mask, however, as the BERT model only has an encoder there is only a self-attention mask. The self-attention mask can be used to select what part of the sequence a transformer should be able to ”see”. When working in a language setting and trying to predict the next token, the expected way to train a model would be to mask all tokens after a point and start predicting.

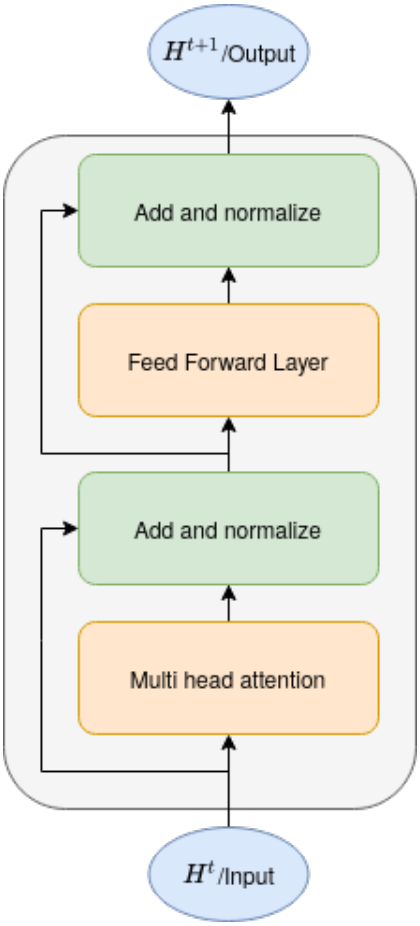


Figure 2.5: Basic components of the BERT transformer model.

This would give the model the same information as it would have in one-directional LSTM models. BERT However is a bi-directional model, this means it can also get information from tokens ahead. In the case of BERT the self-attention mask is only used to mask the padding tokens. Figure 2.6 shows a visual representation.

As can be seen, when working on learning bidirectional representations, the BERT model can look ahead as well as backward. This does raise questions, how can the model learn representations of words if it can see ahead and behind? Intuitively, it would seem that the model can see the word it is predicting. This is where the pre-training tasks come into play.

To learn a high-quality representation of the language the BERT model uses multi-task learning on 2 tasks. The first task is the Masked Language Model (MLM) task, and the second is the Next Sentence Prediction (NSP) task.

The MLM task is the most similar to the way (Bi)-LSTMs and unidirectional transformers learn a language. Which is to select 15% of the tokens in the input and cover them with a <MASK> token, this is also often called the 'cloze' task (of these 15% of tokens

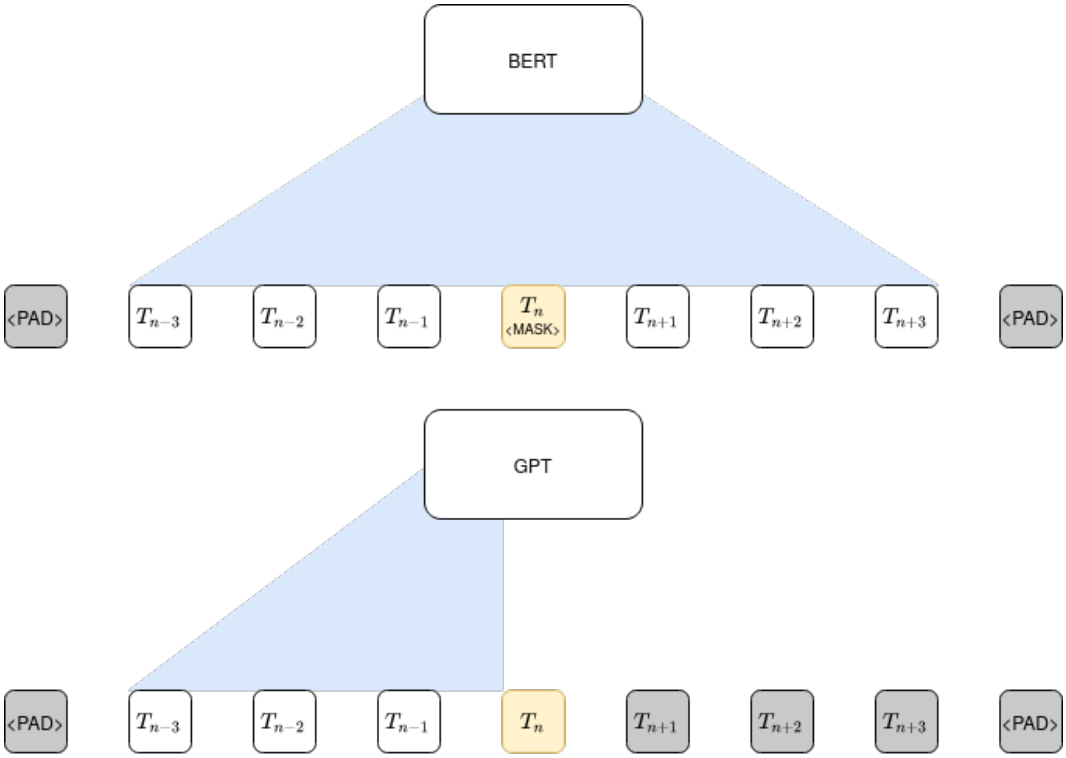


Figure 2.6: Visualization of different attention masks.

10% will be changed to a random token instead of the mask token, and 10% will remain unchanged) [58]. The model will then learn to reconstruct the missing tokens. From this description, we can see that although the BERT model is capable of using future information when learning representations, each sample the model uses will only give 15% of the updates a unidirectional model will be able to use due to the nature of the task.

The NSP task is designed to help the model learn relationships between 2 sentences which are often not captured by language modeling. This is expected to help with downstream tasks such as Question answering on natural language inference. In this task, the model is given 2 sentences separated by a token. In 50% of the cases, the sentences follow after one another in the corpus, and in 50% of the cases, they do not. The model is then asked to tell the 2 apart. The input to the model and training predictions are given in figure 2.7. After the pre-training is completed the authors claim that all state-of-the-art results they produced can be replicated in a few hours on a GPU or a single hour on a TPU.

General Purpose Transformer (GPT-1,2,3) The General Purpose Transformers (GPT) are a set of transformer models that were created by OpenAI [1]. The goal of these transformers is to have a single transformer model that can perform a wide variety of tasks.

[1] https://openai.com/

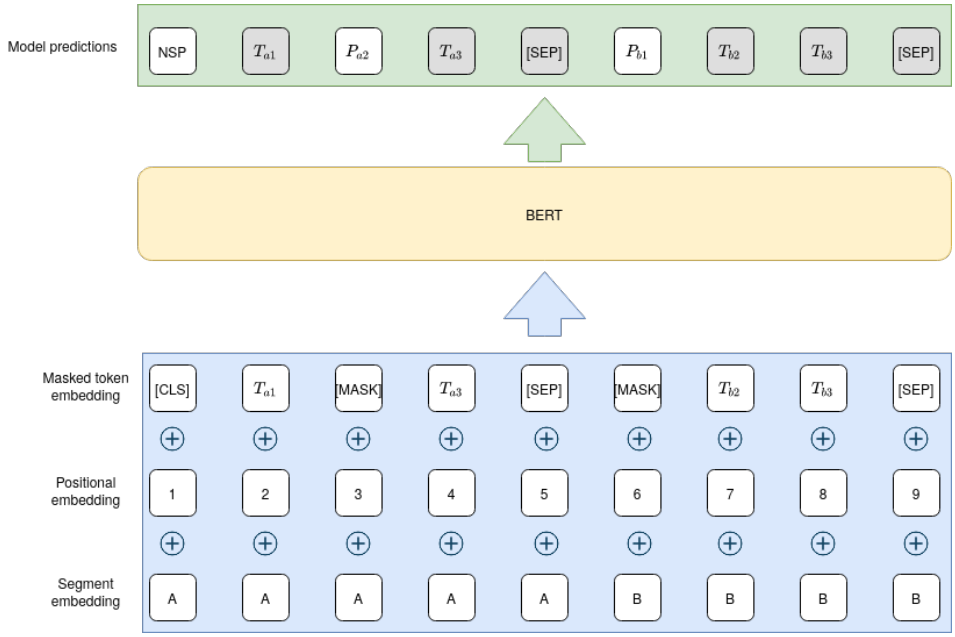


Figure 2.7: BERT Training procedure, only the masked and [cls] tokens are learned on in the output.

The initial transformer GPT-1 was similar in approach to the BERT model in that it was pre-trained on a set of data and then later fine-tuned on downstream tasks [48]. However, other than BERT, GPT-1 uses a decoder transformer [40] to predict a given token given the previous tokens.

To follow up on the GPT-1 model openAI created the GPT-2 model. In this model they moved more in the direction of meta-learning, the original goal of training a language model on maximizing $p(output|input)$ is replaced to put it into a more general setting, maximizing $p(output|input, task)$, This means that the output should be conditioned on the task it is given not only on the input [49]. The authors continue to expand the network to the general case by evaluating the network in a zero-shot setting. In this setting the network is given a few examples of a task, the authors use an example of an English →French translation, this is called the context. The input the networks would get is a few examples of English →French, English →French and then one input of English →, from this the model is sampled to get the output. It must be noted that no weights are updated at this point in the zero-shot setting, the performance is what the model was able to learn from the language model from the original dataset (WebText [2]).

[2]This is an unreleased dataset maintained by OpenAI (https://openai.com/)

The most recent GPT model OpenAI has published when writing is the GPT-3 model [12], this model is architecturally similar to the GPT-1 and GPT-2 models, however, the amount of training data and parameters is increased by an order of magnitude, and the language models performance is measured in the zero-shot, one-shot, and few-shot settings. The largest original GPT-1 model had in the order of 100M parameters, the next model GPT-2 had 1542M parameters and the GPT-3 model had 175B parameters. This increase in parameters was also matched by an increase in the size of the training set. Where GPT-1 trained on only a single corpus for pre-training, this was increased to a larger, more varied dataset for GPT-2. GPT-3 was trained on five different corpora that were sampled by a given distribution until 300 billion tokens were trained on [48, 49, 12]. During experimentation, it was investigated whether using smaller models was feasible. Unfortunately, They found a strong positive correlation between the number of parameters a model has and its performance.

Although the GPT-3 model did not reach any state-of-the-art performance, they did mention that it is expected for the GPT-3 transformer to reach state-of-the-art performance if it were to be fine-tuned on any specific task. This, however, was not in their scope of research.

2.3.2 Graph transformers

The non-sequential nature of graphs combined with the bottlenecks experienced by many Gated Graph Neural Network (GGNN) architectures offer a opportunity to adapt transformers to learn to reason on graphs. This has been done in many different ways. In some cases, a new architecture is developed [61, 20, 66] or in some other cases, an existing architecture is adapted to graphs [67].

Graph Attention Transformer The Graph Attention Transformer (GAT) [61] is one of the earlier models to incorporate attention mechanisms into learning about graphs in a generalized setting. The GAT works by ignoring edges in the attention calculation. Then for each node, calculating a new representation based on all other available nodes using the calculation 2.4 and also depicted graphically in figure 2.8.

$$h_i' = \sigma\left(\sum_{j \in N} \alpha W h_j\right) \tag{2.4}$$

The available nodes mentioned previously are the nodes in the neighborhood of the current node. This neighborhood is specified as some neighborhood in the original paper and is the only time where the edges between nodes are taken into account. In the paper they used only first-degree neighbors for their experiments but as the neighbors are filtered out using a masked attention mechanism any rule can be used to specify which node is in the neighborhood and which is not.

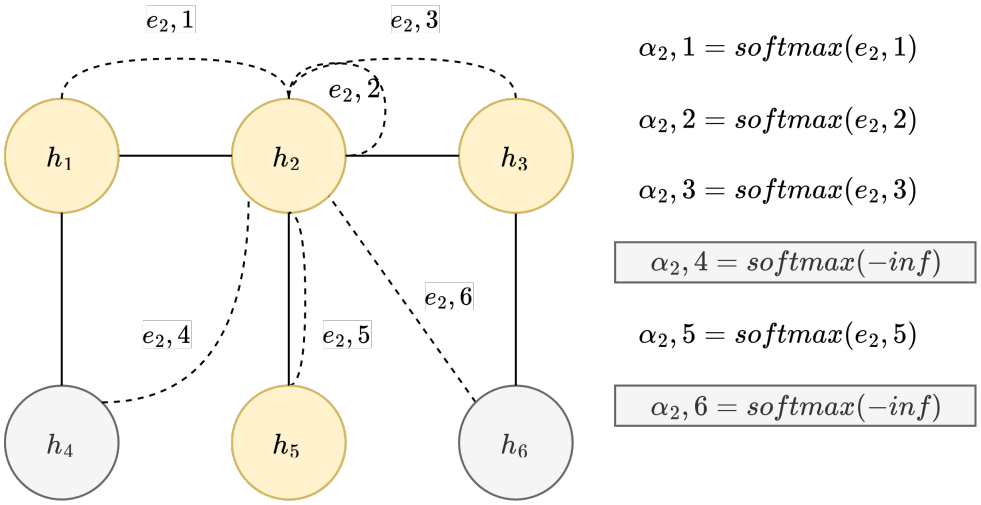


Figure 2.8: Visualization of the Graph Attention Transformer attention calculation.

Graph-BERT A comparative network that is based upon existing architectures is Graph-BERT. The name is taken from the Bert Transformer [18] which has a similar setup. The similarities between the models are predominantly in the setup of the learning tasks. Graph-BERT focuses on creating a graph transformer that benefits from pre-training, as well as focusing on high-quality positional embeddings for the nodes. The model itself consists of five parts. The first three parts are, linkless subgraph sampling (sampling nodes from the larger graph), node input vector embeddings, and the graph transformer layers, followed by the representation fusion and functional component. The first three components are most important to the current research, whereas the last two are more important to the specific tasks being solved in the paper.

An essential feature of this model is that it doesn't require the full graph to be used during training/predicting. The model works by sampling many subgraphs from the main graph using an intimacy matrix. In the case of the paper, this intimacy matrix is based upon the pagerank [46] score of each node. However, this intimacy matrix seems to be replaceable with any metric that will give a neighborhood from which to sample nodes. Similar to the GAT [61]. Furthermore, the information presented to the network regarding edges is only present in the choice of nodes for the link-less subgraph and later on in the positional encodings.

Once the subgraphs have been sampled each node is encoded using an embedding of the value of the node, and 3 positional embeddings. The first positional embedding is the hop-based positional embedding. Similar to the GAT [61]. The second is the Intimacy based Relative Positional Embedding, described previously. The final positional encoding is the Weisfeiler-Lehman positional encoding (WL-encoding). The WL-encoding combines the encoding of graphs in the Weisfeiler-Lehman isomorphism test [55] with the positional encoding function from the original Transformer paper [60]. Given the Weisfeiler-Lehman encoding of each node (a natural number summarizing the node types of its neighbors up

to a depth of h), the cosine and sine are taken of this code to come up with a positional embedding [67]. This is similar to the original transformer [60] using the index of the position in a sequence rather than the WL-encoding.

Once the embeddings have been calculated, the input vectors to the transformers have to be constructed. The Graph-BERT authors use an aggregation function which can be any function that combines all the different embeddings into one vector per node. The Graph-BERT paper summed all vectors by entry to come up with an embedding per node. These vectors are then combined into a matrix H which is fed to a normal transformer model. The final output of Graph-BERT is then fused to give a final output z_i which is fed to the final task-specific output layer.

Generalization of Transformer networks to graphs Further attempts at making a general architecture for graph transformers have been made in "A Generalization of Transformers Networks to Graphs" [20]. Here both the node structure as well as the contribution of the edges have been taken into account.

In this paper, two related architectures are brought forward, in the first only nodes are taken into account. In the second architecture, the model is augmented with knowledge from the different edge types.

The first architecture is very similar to the original Transformer architecture. As input, it takes a 2D tensor where one dimension is the number of Nodes and the other dimension is the dimension of the embedding of that node. The eigenvalues of the graph Laplacian corresponding to the N (hyper-parameter to be set) lowest nonzero eigenvalues are used as a positional embedding.

After summing the positional encodings, the Query (Q) and Key (K) matrices are multiplied, passed through a softmax function, and multiplied by a Value (V) vector. Similar to the original Transformer. Residual connections and normalization operations are added as can be seen in figure 2.9.

The main difference in this work from the original transformer appears when graph edges get included. The authors implemented the use of different edge types by creating an edge embedding. Similar to the embedding of a node. This edge embedding gets multiplied by the product of the Query and Key vector as shown in figure: 2.9. This computes the pairwise importance of 2 nodes to one another, which calculates the importance of each edge. This vector is passed to the next layer as the next edge embedding vector.

Furthermore, the authors claim that transformers see NLP sentences as fully connected graphs [20], only that the interactions between words are not always understood. This new architecture attempts to solve this issue in settings where the interactions are known. Molecular classifications is given as an example. It is however uncertain from the paper how the model will function in a setting where there are unconnected components to a graph which may result in a very sparse matrix (if a tensor of 0s is used to embed no interaction).

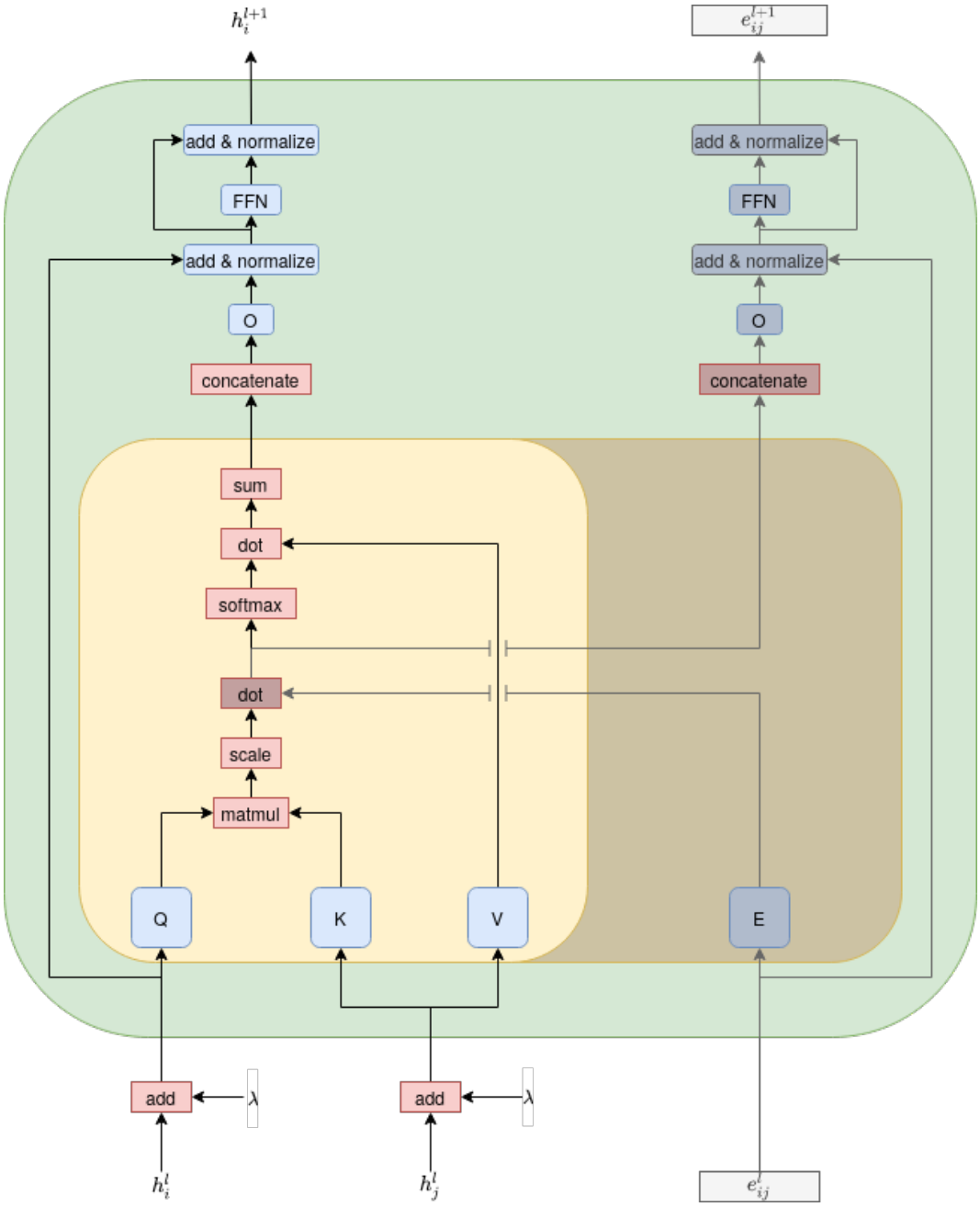


Figure 2.9: Architecture of the generalized graph transformer, the modules affected by the presence of edge features are shaded.

Graph Transformer Network The final type of transformer network we examine is the Graph Transformer Networks (GTN). Contrary to the reliance of the previous networks on a form of positional encoding, Graph Transformer networks work by transforming graphs into a new graph structure using meta-paths. This new graph can be used in an attention calculation to come up with a prediction (node classification in the paper) [66].

To understand GTNs we must first understand what a meta path is. The authors describe a meta path as a path that can take any edge type between any node types in a heterogeneous graph. To find these meta paths of an unknown length/type combination, the GTN has a feature extraction step where a set of adjacency matrices is kept to denote each type of edge A. From this set, two adjacency matrices are softly selected using a 1×1 convolution, with non-negative weights. Once two adjacency matrices have been selected, they are multiplied by each other to get a new adjacency matrix. This adjacency matrix corresponds to a new meta-path. As the multiplication of two adjacency matrices will only allow the generation of $l+1$ (number of layers) length meta-paths, the authors include the identity matrix in the set of adjacency matrices. This allows the network to select any number of meta-paths up to length $l+1$.

After generating these meta-path adjacency matrices, the set of meta-path adjacency matrices is passed to a Graph Convolutional Network. This module performs the final step of the overall network. To raise performance, multiple different channels can be concatenated that all generate meta-paths from a given heterogeneous graph.

This approach seems similar to the W-L positional encoding used by the Graph Attention Transformer. Although this approach doesn't rely on a positional encoding, it removes some of the work from the programmer by selecting meta-paths automatically. Furthermore, it lifts the constraint of each edge being of the same edge type, allowing for more complex features. The final benefit of this model is the network's ability to ignore meta paths that could harm performance, which is labor-intensive for a human to do [66].

2.4 Machine Learning for Software Engineering

The adaptation of NLP models to source code has been a logical step in the development of machine learning tools for software engineering. However, there are yet to be many papers published that all work on improving the same problem. Many times the same problem has been changed slightly from paper to paper. Due to this, in the next sections, we will give an overview of what problems are being solved using machine learning and what models seem to perform the best. Then we will review code completion tasks, although many of the papers will have slight variations in their approach.

General software engineering tasks When looking at the nature of problems being solved in software engineering there is a number of closely related tasks that have been investigated in recent years. The tasks that were covered in the papers we are investigating are: var-naming [4, 24], var-misuse [4, 24, 31], method naming [56], code labelling [8, 7, 6], code summarization [7, 31], wrong operator [31], wrong operand [31], matching docstring [31] and Exception Type [31].

The first paper we look at introduces the var-naming and var-misuse tasks [4]. In these tasks naming errors are detected in a code base. Specifically the var-naming task is predicting the name of a given variable in the code, and the var-misuse is deciding whether a variable in the code was used appropriately. These naming errors are usually introduced when copying and pasting parts of code, or using the wrong iterator variable [4]. Allamanis et al. [4] used a Gated Graph Neural network [37] that received an augmented AST as input to determine whether the incorrect variable was used or to predict the name of the variable.

The mentioned augmented AST is a normal AST with extra edges. These edges give further structural information about the code. The edges used in this work are LastUse, ComputedFrom, NextToken, LastRead, LastWrite, ReturnsTo, FormalArgName, GuardedBy, and GuardedByNegation. All these edges are meant to give the GGNN more information on the structure of the code. They also aid the model in looking toward important parts of the code. These important parts may be obscured due to the knowledge being lost in the recurrent node of the GGNN [5].

The performance of this setup returned an 85.5% accuracy when looking at the var-misuse task on a project seen during training. This improves the baseline of a BiRNN of 73.7%. When looking at projects that were not seen during training the GGNN has a slight drop in accuracy to 78.2% whereas the BiRNN drops to 60.2%. For the var-naming task, performance is slightly worse, which may be due to the increased difficulty of the task. On a project that was seen during training, the GGNN had an accuracy of 53.6%. This is better than the the 42.9% of the BiRNN. When looking at an unseen project the GGNN had an accuracy of 44.0% compared to the 23.4% of the BiRNN. This shows that there is a benefit to adding structural information to the AST when learning language tasks.

In the paper Global Relational Models of Source Code [24], the authors expand on the previously mentioned work in 2 ways. Primarily, for the input graphs, the authors include the edges used in the previous paper. However, they also add information to pass to the networks such as control flow edges. Furthermore, the authors question whether a full AST is necessary for the performance of a model and choose to push all edges between internal nodes of the AST to the leaf nodes. They then remove all internal nodes. This gives a more condensed representation of the graph (using fewer nodes) than with the AST nodes.

The second change is the models used in the paper. The authors created two model architectures to capture the global structure of the graph. The first method is to use a sandwich model. The basis for the sandwich model is that the leaf nodes are part of the AST. This enables an RNN or transformer network to calculate a representation of the node to feed to the GGNN. The authors denote such a network as [RNN, GGNN(3), RNN]. Here first an RNN is used to calculate the representation, then 3 message-passing steps are run. Finally, an RNN is used to come up with the final prediction. These intermediate networks can both be run after every GGNN step, or only at the start and end as given in the example. The second proposed model is the Global Relational Embedding Attention Transformer (GREAT), this transformer embeds the structure of the graph in the attention calculation using the same methods as when calculating relative-positional attention [54].

The GREAT and sandwich models [24] are evaluated in their capability of deciding if a file contains a bug, where the bug is, and how to repair the bug [59]. For the file classification task, the RNN sandwich model performed the best with an accuracy of 82.5%. Compared to the GREAT model which achieved an accuracy of 80.1% and the best baseline model (vanilla transformer) which achieved an accuracy of 81.4%. For the localization and repair task, the GREAT model performed best on short (<250 Tokens) files with an accuracy of 76.4% compared to the 75.8% of the RNN sandwich model 67.7% of the best baseline model (vanilla transformer). However, for longer files, the RNN Sandwich model performed best with an accuracy of 73.8% compared to the GREAT model's accuracy of 73.1% and the best baselines accuracy of 63% (vanilla transformer).

Closely related to the problem of var-naming and var-misuse is method naming. Pythia [56] is an LSTM-based model that tries to predict method names given the previous nodes in an AST. The input is the AST visited in depth-first order up to the node corresponding to the method. This representation is fed to an LSTM model which was trained to predict the name of the method. Unlike the previous papers, this paper did not augment the AST with any edges as such information can not be represented in a simple vector.

The performance of this model is measured using the MRR and accuracy of the predictions at the top-1 and top-5 levels. To get an idea of the performance the model is compared to several other approaches, the best-performing ones being a Markov chain based model. Pythia managed to get a top1 accuracy of 71% on the test set, compared to a top-1 accuracy of 58% for the Markov chain. The top-5 accuracy saw an improvement up to 92% accuracy for the Pythia model and 83% for the Markov chain. The MRR was also higher for the Pythia model at 0.814 compared to the 0.704 of the Markov chain.

Next, we look at the paper code2vec [8] which works on the task of code labeling, more precisely put as generating semantic labels for code. This is closely related to method naming as a good method name should summarize the code it calls [8]. The Code2Vec model works similarly to the Word2Vec model [44] mentioned at the beginning of this section. This model traces paths between two terminal nodes and uses these paths to create a context that is used to predict a label. The authors' main goal of this paper is to come up with an embedding of source code much like that of the Word2Vec model. This is a very early model when looking at the field of machine learning for software engineering and is improved upon in more recent papers. The evaluation of this model is done by comparing its precision, recall, and F1 score on predicting a label given a snippet of code. The Code2vec model is evaluated against a Conditional Random Field (CRF) model [6]. The Code2Vec model scored a precision score of 63.1, a recall of 54.4, and an F1 score of 58.4. This is an improvement over the CRF that had a precision score of 53.6, a recall of 46.6, and an F1 score of 49.9.

The improvement of using paths traced through ASTs was further expanded upon in the similarly named paper Code2Seq [7]. Code2Seq takes the approach of Code2Vec and uses it in a seq2seq-based task. This allows for the use of the high-quality source code embeddings developed in Code2Vec to work with more complex sequence base tasks such as comment generation and code summarizing. The approach used in this paper is to create embeddings from snippets of source code which are fed to an LSTM model. This LSTM model is then used to auto-regressively predict a caption for the given code. As this is a seq2seq task it is

difficult to evaluate it using the usual accuracy scores, so the model is evaluated using the BLEU score. When compared to the best baseline of CodeNN [27], which is an LSTM-based model with an attention mechanism, the Code2Seq model achieves a BLEU score of 23.04 compared to the 20.53 of codeNN. This shows that a similar model (both Code2Seq and CodeNN are LSTM models with an attention mechanism) that does not use any form of AST embedding in the input performs slightly worse than a model that does include the AST.

Finally, we look at a BERT-based model named Code Understanding BERT (CuBERT) [31]. CuBERT follows the same training procedure as BERT above, only on a Python corpus. The sentences used in the classic BERT model are defined as logical code lines, which are defined by the python standard [31], for the pre-training tasks. Everything else is the same as for the BERT model. Once the model has been pre-trained it is fine-tuned on a multitude of tasks.

We begin with the var-misuse task, this task is also implemented for the CuBERT model and is compared to the BiLSTM and transformer model. The BiLSTM achieves an accuracy of 80.53% and the transformer model an accuracy of only 78.28%. Both are improved by the CuBERT model which achieves an accuracy of 95.21%. Following the var-misuse task is the wrong binary operator task. In this task, the models need to find whether a binary operator is incorrect. This task proved to be more complex for all three models. The best BiLSTM model achieved an accuracy of 86.82%, the best Transformer achieved an accuracy of 76.55%. Both are outperformed by the CuBERT model with an accuracy of 92.46%. The next task is the swapped operand task. This task asks the models to identify whether there was an error with the ordering of binary operands in a non-commutative expression. For this task, the CuBERT model had the best performance with an accuracy of 93.36% compared to the accuracy of 90.14% and 87.83% of the BiLSTM and transformer models respectively. The next task is the matching docstring task. In this task, the models decide whether a given docstring belongs to the code. This seems to have been the least complex task for the transformer-based models, yet harder for the BiLSTM-based models. The CuBERT and transformer model achieved 98.09% and 92.02% accuracy respectively while the BiLSTM only managed an accuracy of 89.08%, again the CuBERT model outperformed all others. The final task is the exception type task, were given a code snippet that handles an exception the type of the exception had to be predicted. The CuBERT model managed an accuracy of 79.12% compared to 67.01% of the BiLSTM and 49.56% of the transformer model. Given all these tasks we see that some tasks are more complex than others. Also, it shows that the difficulty of the task is dependent on the architecture of the model. Finally, it shows that a pre-trained transformer architecture such as BERT results in increased performance across the board.

Similar to the GPT-C [57] paper the InCoder model [22] attempts to use a similar training method to the GPT transformers [48, 49, 12], mixed with elements from the BERT [18] training approach. The InCoder paper learns a language model to use the left-to-right context that is at the heart of the GPT papers, combined with the ability to use future context such as in the BERT papers. The general idea behind this is to mask a span of code for the model to predict rather than masking 15% of the tokens randomly. They propose the training objective as the Causally-Masked Multimodal Model (CM3) [2]. For this training

objective, a span is removed from the code file, replaced with a <MASK> token, and appended to the end of the input. This allows the use of a left-to-right decoder transformer model such as the GPT model to autoregressively generate any span of text. The equation of the objective is given in equation 2.5.

$$logP([Leftcontext; <MASK>; Rightcontext; <MASK>; Span; <EndofMASK>]) \tag{2.5}$$

The InCoder model was evaluated in a zero-shot environment against a series of pre-trained BERT models (RoBERTa [41], codeBERT [21], PLBART [3] and CodeT5 [65]) on the docstring generation task. The authors found that although the InCoder model had never seen the task before, it was almost able to compete with the BERT model scoring a BLEU-4 score of 18.27 compared to the best comparative model (CodeT5) of 20.36, while also improving the performance of the worst comparative model (RoBERTa) which only scored 18.14. The model was also compared to CodeBERT on the cloze task, only predicting max or min, in multiple languages. Here the InCoder model outperformed CodeBERT on the languages Python, Javascript, Go, Java, and PHP, only under-performing on the Ruby programming language.

Code completion Finally, we look at papers published that focus on code completion. The importance on ASTs varies between papers. We will be giving a quick overview of the methodology used in each paper and an evaluation of the results the authors found. We start with papers that do not use any information from ASTs as in the TransformerXL [19] and GPT-C [57] papers, followed by papers that attempt to incorporate the ASTs into a sequence representation to use them with NLP models. As shown in the papers "Code Prediction by Feeding Trees to Transformers" [33], and "Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs" [63]. Finally, we look at a new approach of replacing tokens with AST representations presented in the "CodeFill: Multi-token Code Completion by Jointly Learning from Structure and Naming Sequences" paper [28].

The TransformerXL [19, 17] architecture showed that it was able to outperform an LSTM and GRU model when working with source code. The authors used both Byte Pair Encoding (BPE) and subwords when running the experiments with a vocabulary size of 1000. Furthermore, the authors noted that the TransformerXL architecture both outperformed the LSTM and GRU models, and had preferable scaling capacities. Showcasing that going from a 4-layer TransformerXL architecture to an 8-layer TransformerXL architecture resulted in a 1.38x training time. Using a 4-layer GRU took 3.58 times longer to train, and using an LSTM took 4.2 times longer.

One of the few papers that looked at multilingual language modeling was the IntelliCode compose paper (GPT-C) [57]. The paper initially tested the GPT-2 architecture and pre-training method on its ability to predict the next token given an incomplete snippet of code, followed by a beam search algorithm to improve performance. The authors found that using the GPT-C architecture in a multilingual setting was plausible, but did not always result in an improvement of per language performance. When training on all four languages for C# the perplexity rose from 1.91 to 2.01, stayed effectively the same for Python, 1.82 monolingual vs 1.83 multilingual, and dropped slightly for Typescript and Javascript, from

1.40 monolingual to 1.36 multilingual. A similar trend is seen in both the Precision and Recall score on all test sets.

One of the criticisms of transformer models is the large number of parameters needed to achieve good performance. Which led to the use of knowledge distillation in many models [52, 57]. This is important as the previously mentioned performance is by a model containing 350 million - 375 million parameters. After training a smaller model with 96 million parameters the performance dropped from a precision score of 0.58 to 0.53 and the recall dropped substantially from 0.72 to 0.58 [57].

Continuing with the trend of using transformers in code completion "Code Prediction by Feeding Trees to Transformers" [33] uses the pre-order traversal as well as path-to-root representations of ASTs when working with code predictions to predict the next token. They do this by creating input vectors described in section 2.2 which are passed to LSTM models. These embeddings are then fed to a modified GPT-2 [49] model [33]. It must be noted that in this paper contrary to common practice there are no positional encodings due to an expected reduction in performance from early trials [33].

In this paper, the authors predict the next token and not the node types as some papers do. The final results are compared to deep3 [50] an RNN [29] model, and the code2seq [7] model on the py150 [3] dataset, using the MRR (shown in equation 2.6) as a metric. When predicting the next token the TravTrans (the model from the paper [33]) achieves an MRR of 54.9%. This is an improvement on the 36.6% of the RNN model. When comparing to the code2seq and deep3 model, the TravTrans achieves an MRR score of 58% compared to 43.7% and 43.9% respectively.

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i} \tag{2.6}$$

Similar to the previous paper "Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs" uses the above methods of representing a graph as a sequence using a pre-order traversal but adds extra attention mechanisms that may be more suited to the graph structure [63].

In this paper, the authors present the code completion problem as predicting the nodes following a 'rightmost node'. In an incomplete AST, the last 'rightmost node' is the final node visited in a depth-first traversal before the next node starts. The authors mention that one shortcoming of using a pre-order traversal of the AST in a sequence on its own loses a large amount of the structural information of the AST. To solve this they introduce an AST Graph Attention Block (ASTGab) that combines the calculation of the local attention on the first-degree neighbourhood, as with the GAT [61], with a more global attention mechanism and a parent-child attention mechanism to come up with a final prediction.

The model, Code Completion method by modeling flattened ASTs as Graphs (CCAG), was evaluated on the PY150 [4] and a JavaScript dataset with 150,000 files. The results were compared to runs by a vanilla LSTM [38], a Parent LSTM [36], a Pointer mixture network [9], a Transformer [33], and a TransformerXL [1] network. In all configurations

[3] https://www.sri.inf.ethz.ch/py150

[4] https://www.sri.inf.ethz.ch/py150

```
if a is None:
    print(<STRING>)
```
(a) Tokens

```
IF LOCAL_VARIABLE IS NONE : <EOS>
INDENT FUNCTION_NAME(<STRING>) <EOS>
```
(b) Token Types

Snippet 2.1: Example input to the CodeFill model.

run during the experiments, the CCAG model outperformed the best baseline by between 2.68% and 12.32%.

The final approach that has shown promise in recent times is the CodeFill model [28]. This model is a GPT-based model that encodes the AST structure of the code into the sequence by learning from two different input representations. The first representation is only the code, used for tasks such as Token Value Prediction (TVP) and Statement Completion (SC). These tasks are closely related as TVP only predicts the next token, and SC predicts tokens until the <EOS> token. The second representation (shown in snippet 2.1) utilizes the AST types of the corresponding tokens to help learn the structure of the source code by predicting the type of the token in the Token Type Prediction (TTP) task.

The model is initially pre-trained on all three tasks, after further fine-tuning it on only the TVP and SC. The experiments were run in Python with two separate datasets. The CodeFill dataset was used for their best performing models. This dataset is all Python projects from the GHTorrent [23] project with more than 20 stars combined with the PY150 dataset. For pre-training, the CodeFill dataset was used and it was fine-tuned on the PY150 dataset (after deduplication), all other models were trained on both datasets combined.

The final evaluation of the model was done on multiple tasks, we will focus on the TVP task for this paper as it is the closest to the task we are investigating. The CodeFill model was compared to the GPT-C [57] and TravTrans+ [33] models which had the best performance of the baselines. For the TVP task, the GPT-C model had a performance of 79.8% and the TravTrans+ model had an accuracy of 78.9% these were both outperformed by the CodeFill model which achieved an accuracy of 80.6%. From these results, we can see a slight increase in the performance of the GPT transformer architecture when focusing on adding the structural data as in the TTP task for the CodeFill model.

Chapter 3

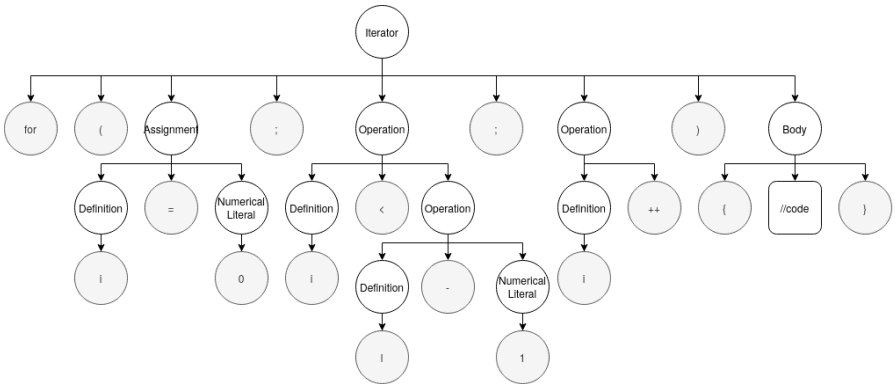
Method

The approach we present can consists of three separate steps. At the start of the pipeline, we have the construction of the universal ASTs when working on the cloze task and the generation of incomplete universal ASTs for the final prediction task. Parallel to this, we need to train a multilingual embedding for the source code tokens that embed the leaf nodes of both AST types. Both features are fed to the transformer model to generate the final prediction. To give a detailed explanation of the pipeline, we will describing the universal AST structure and how the incomplete ASTs are generated. Then we will show the design of the multilingual embedding model and the reasoning behind it. Finally, we present the architecture of the transformer model and the design choices that led to it.

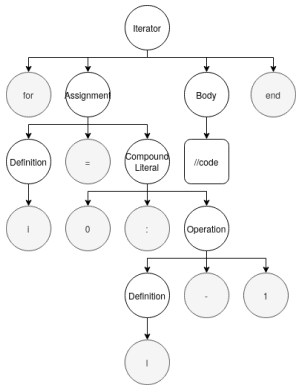
3.1 Universal ASTs

As a large portion of the proposed network relies on transfer learning between different programming languages, it is essential that the ASTs that the network learns on utilize the same nodes. We start by mentioning that we are not mapping different ASTs from different languages to a universal AST structure, but we are defining a set of common AST node types that we use to create an AST for each language. This means that while two translated code snippets may have differing ASTs, the node types these ASTs are made of are the same. To give an overview of all the node types used in the universal AST we will give a quick explanation and example for each.

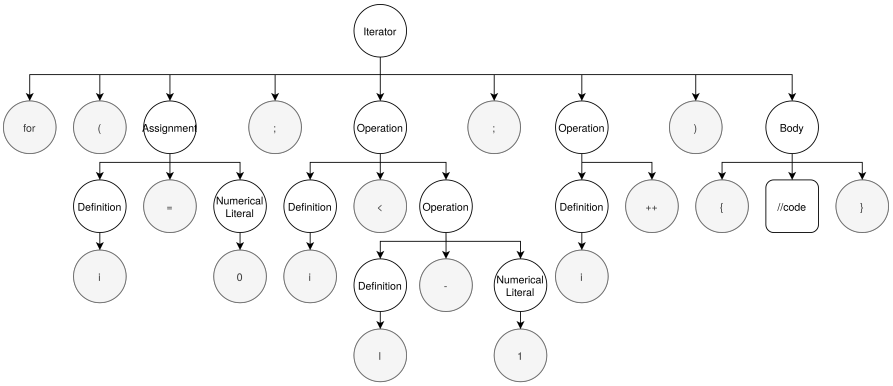
Iterator nodes One of the more common constructs in a programming language are Iterators. Iterators are blocks of code that can be executed any number of times according to a condition. Across all languages in this work, this can be summed up as a node that handles `for`, `do...while`, and `while` statements. What these statements all have in common is that there is a condition and a body. In the Universal AST, these are also implemented as such. Figure 3.1 shows an example of how these ASTs look like for a generic loop in Julia, Java, and CPP.



(a) Java



(b) Julia



(c) CPP

Figure 3.1: Example usage of an Iterator node.

```java
for(i=0; i<l-1; i++)
↪ {
    //code
}
```

(a) Java

```julia
for i = 1:l-1
    #code
end
```

(b) Julia

```cpp
for(i=0; i<l-1; i++)
↪ {
    //code
}
```

(c) CPP

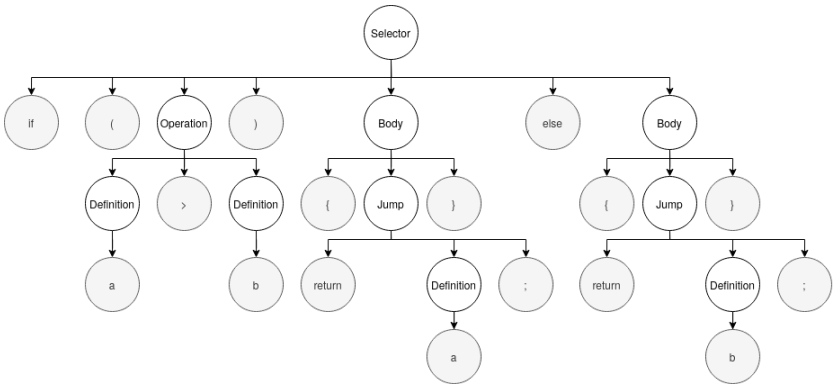
Snippet 3.1: Examples of iterator statements in all 3 languages.

Selector nodes Another common construct found in programming languages is a Selector. The most common form of a Selector is the `if` statement found across all selected languages. However, in some languages such as CPP and Java, the `switch` statements also fall under the Selector node. The Selector nodes all have in common that depending on a condition a code block will be executed. This is demonstrated in figure 3.2, for generic selector statements in Julia, Java, and CPP.

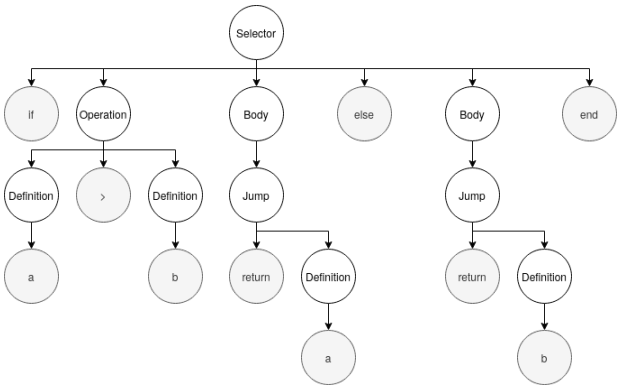
Ternary operator As a special case of an `if` statement the Ternary operator needs its own node. Whilst it can be implemented as a set of `if` statements when used in combination with an assignment as is possible in some programming languages this would require either extra nodes or tokens to be inserted into the AST which would not correspond to the code that was being completed. To solve this and keep implementations consistent the Ternary operator is given its own node.

Jump nodes An important feature of all programming languages is the ability to move from one part of code to another. This ability is captured in Jump nodes. The most common Jump node in high-level programming languages would correspond to the `return` statement. This node represents that the code is returning a value and will continue at a different location. However, other statements can correlate to a jump node. One example of this is the, more common in lower-level languages, statement which also falls under the Jump nodes, as well as statements that control iterators such as `continue` and `break` statements. On a special note, for languages such as Julia that have an `end` statement at the end of each code block. The `end` statements are not taken as Jump nodes but implemented as tokens, similar to how a closing bracket (}) would be treated in Java. An example of a Jump node can be seen in figure 3.2 in the return statement.

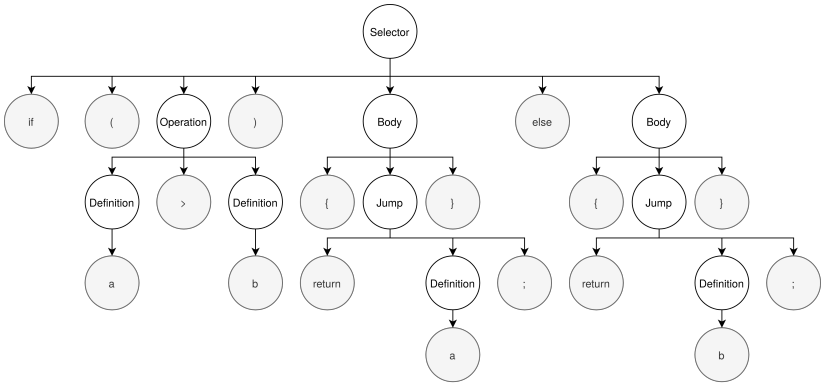
Body nodes Body nodes are wrapping nodes that are used to give the scope of variables in a program. These are for example wrapped around the list of statements that can be in a constructor or function definition as the body, the statements within an iterator or selector statement as well as special scopes that can be used in some programming languages such as Julia that implements their `using` statement. A Body node is also inserted at the base of every AST as the root node. Looking at figures 3.1 and 3.2, we can see that a Body node is inserted in the iterator and selector statements.



(a) Java



(b) Julia



(c) CPP

Figure 3.2: Example usage of a Selection node.

```java
if (a > b) {
    return a;
} else {
    return b;
}
```

(a) Java

```julia
if a > b
    return a
else
    return b
end
```

(b) Julia

```cpp
if (a > b) {
    return a;
} else {
    return b;
}
```

(c) CPP

Snippet 3.2: Examples of selector statements in all 3 languages.

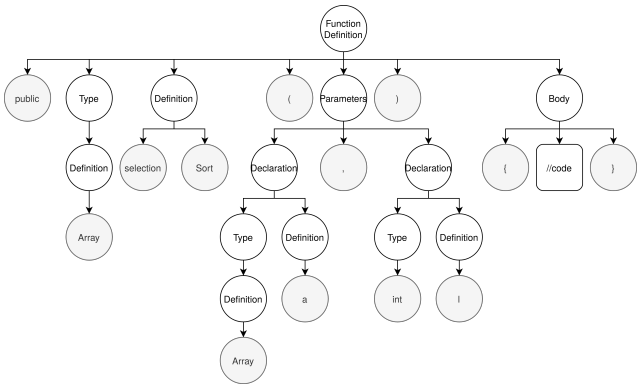
Definition nodes When working with user-defined names, this can be anything from variable names, to function/class names. Each token that is not a token reserved by the language will be the child of a Definition node. After tokenization, this will give the network information about the user-defined word. This is similar to how spaces delimit the gap between words in a 'normal' language. The last child of a Definition node will also be an 'end of node' (<EON>) token.

Function and class Definition nodes To facilitate the definition of functions and classes we employ a Function Definition node for function definitions and a Class Definition node for class definitions. These nodes will have children containing any modifiers the languages use, such as the `private` and `static` keywords from java, the name of the function or class, and the body. An example of a function definition can be seen in figure 3.3.

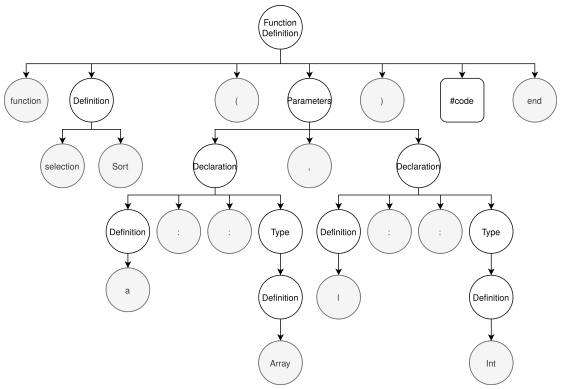
Function call nodes The function call nodes are used to show when a function is called. This is done by giving the name and a list of parameters for the function. Given that in some languages it is not possible to tell apart a function call from an object instantiation we map both function calls and object instantiations to the same node. We demonstrate this in figure 3.4.

Operation nodes To add operations to the Universal AST the Operation node is used. In an Operation node, there is any number of unary or binary operators that affect statements. These statements are anything within a language that an operator can operate on. This means it could be objects, arrays, strings, numbers, or booleans, all depending on the original language. An important note to make is that the assignment operators (including special assignments such as +=, -=, etc..) are not included in the Operation nodes but in the Assignment node.

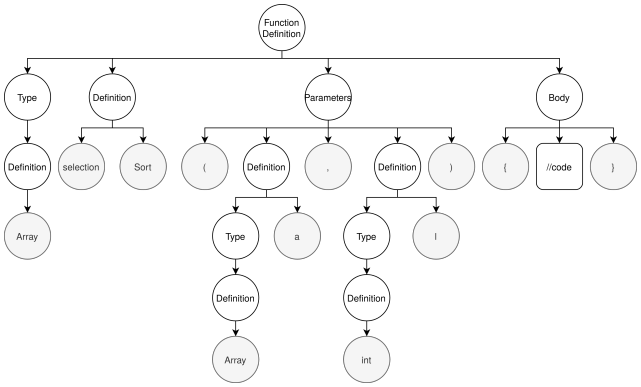
Literal nodes When values are used in source code, it adds a large amount of complexity to the problem. For normal numbers, many values can be predicted, and for strings, the length of the string adds a new dimension to the problem. As we are more interested in the performance of the model when writing code instead of predicting values we use a Literal node to abstract from this issue. All numeric values will be masked with a <NUM> token, and all string will be masked with a <STRING> token. For booleans we make an exception as we know there is only the values `true` or `false`.



(a) Java



(b) Julia



(c) CPP

Figure 3.3: Example usage of a Function Definition node.

```java
public Array
↪ selectionSort(Array
↪ a, int l) {
    //body
}
```

(a) Java

```julia
function
↪ selectionSort(a::Array,
↪ l::Int)
    #code
end
```

(b) Julia

```cpp
Array
↪ selectionSort(Array
↪ a, int l) {
    //body
}
```

(c) CPP

Snippet 3.3: Examples of function definitions in all 3 languages.

Compound Literals To facilitate the many different data structures such as arrays, dictionaries, lists, sets, and others that are built into many different languages the compound literal node is used. no distinction is made between a list or array in the node itself. However, the tokens that are children of the Compound Literal node make this distinction. These Compound Literals are only for data structures that are built into the grammar. Data structures such as lists in Java will still be handled as objects.

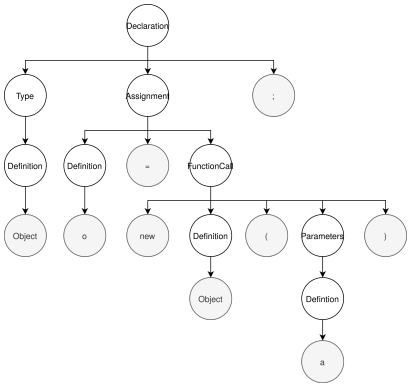
Declaration nodes As the name of Declaration nodes suggests they are used to declare a variable in languages where this is possible. A Declaration node is only used when the type is given in the declaration. To clarify `int a = 0;` would use a declaration node, however `a = 0;` would not.

Assignment nodes Assignment nodes are used when a variable is assigned to something assignable, this will in most cases be a literal or object, however, in many languages it is also possible to pass functions so these may also be assigned to a variable. What an assignment node has in common across all languages is that there is a user-defined name, with potentially a type annotation, an assignment operator, and then something assignable.

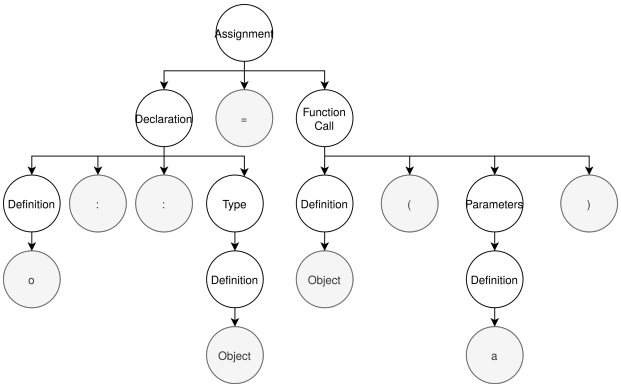
Import nodes One common feature in programming languages is being able to import code from different modules. This allows programmers to reuse code and keep their files more readable. This also lets them import common libraries. This is important to note as there may be a set of common libraries in a programming language used frequently. To add this feature to the Universal AST Import nodes are used. Each line of code that imports a library corresponds to one import node.

Type nodes In many languages, types are important. To convey the information in types they receive a special node type, Type nodes. This node is used whenever a type is used in situations such as return value declarations in method signatures, casts, or type specifications in areas such as template methods and generic constructors. An example of the use in a method signature can be seen in figure 3.3.

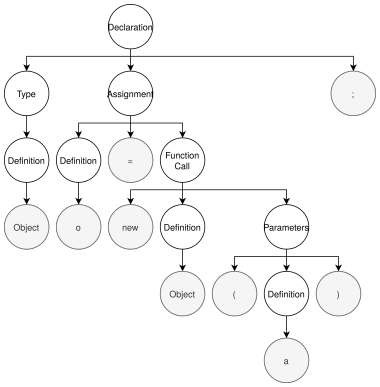
Error handling nodes To add error handling to the Universal AST four nodes are added. Primarily, the Try node is added this corresponds to the `try` statement in the programming



(a) Java



(b) Julia



(c) CPP

Figure 3.4: Example of a Function Call node being used during object creation.

```
Object o = new
↪  Object(a);
```

(a) Java

```
o = Object(a);
```

(b) Julia

```
Object o = new
↪  Object(a);
```

(c) CPP

Snippet 3.4: Examples of object creation in all 3 languages.

languages, each Try node also has a body node that contains all the code within the `try` statement. Secondly, the Catch node is used in cases where an error is caught in the `catch` statement. Thirdly, the Final node corresponds to the `finally` statement that runs after an error has been handled. Lastly, the Throw node is added in the error handling node as mentioned in the Jump node section as it is used to handle errors.

It must be noted that for languages such as Java when there is a `throws` statement in the method/constructor specifier a Throw node is not added to the Universal AST, the `throws` is added as a token to the corresponding node.

Enum nodes A special kind of value used in some programming languages is the Enum. As they behave differently syntactically from most other constructs in a language, we use an Enum node. The Enum node behaves similar to a class wrapper as it gives the name and Enum annotation to the code and contains a Body node that describes the rest of the behavior.

In figure 3.5 we give an example of how an Enum looks in each separate language.

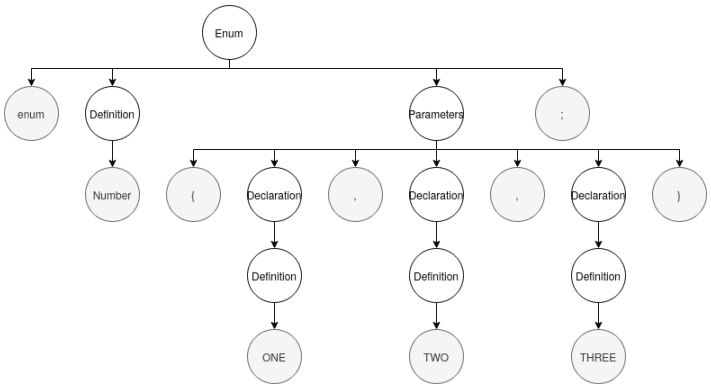
Tokens The Token node is a special node that is the only node that can be a leaf node. It is also the only node that can not have children. So it is forced to be a leaf node. When traversing the AST in depth-first order, returning only the tokens will return the original source code. The tokens in the tree correspond to the tokens returned by the lexer. These tokens are later also used when creating the embedding for the network to use. Depending on the tokenization being used the tokens may be split up into multiple nodes.

Throughout this work the token nodes have been split according to the rules described in section 3.4, they have also been shaded in to make the code easier to distinguish.

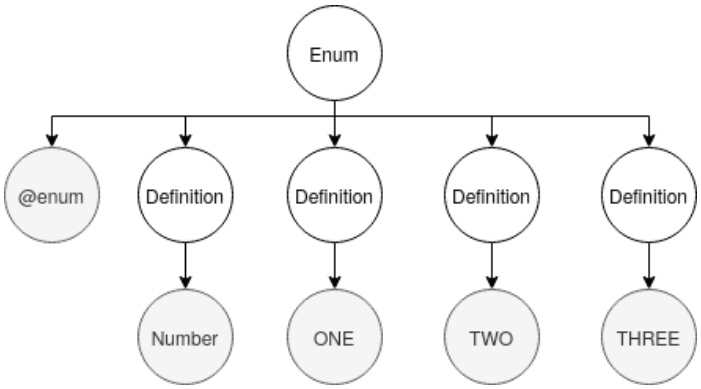
3.2 Incomplete ASTs

When generating ASTs one of the main issues with available Context-Free Grammars is that they require a valid (following all the rules of the grammar) input. While this is desirable when working with running code, code that does not follow the rules should crash to remain predictable, creating ASTs that are incomplete will not work. This is an issue when using ASTs for code completion as any AST can only be generated when it is known what the code already is.

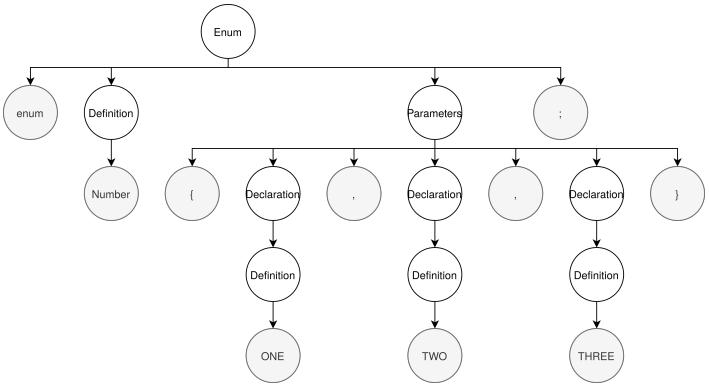
Due to the nature of the problem (code completion) we can benefit from the fact that we know where we want to be generating a completion. This lets us insert a location token which tells the parser where we know that the AST will be incomplete. This allows us to give the location token as a possible alternative to the rule at all locations where we would like to be able to run a prediction. An example of this is given in code fragment 3.6 where a rule is adapted to accept the location token. An example of an incomplete AST is given in figure 3.6.



(a) Java



(b) Julia



(c) CPP

Figure 3.5: Example usage of an Enum node.

```java
enum Number {
    ONE,
    TWO,
    THREE
}
```

(a) Java

```julia
@enum Number ONE TWO
↪  THREE
```

(b) Julia

```cpp
enum Number {
    ONE,
    TWO,
    THREE
};
```

(c) CPP

Snippet 3.5: Examples of an Enum definition in all 3 languages.

3.3 Grammars

Throughout this work, we make use of the ANTLR 4 [1] parser generator to create a parser from a given grammar. We use the generated parser to transform the parse tree into the Universal (incomplete) ASTs described in this section.

For the parsing of the CPP and Java files, we used the grammars for CPP14 and Java9 supplied by the ANTLR team [2]. For the incomplete AST generation, we adapted these grammars to accept the location token. For Julia however, we wrote our own parser in ANTLR 4(as it was not available at the time), and adapted this for the parsing of incomplete ASTs.

3.4 Multilingual Embedding

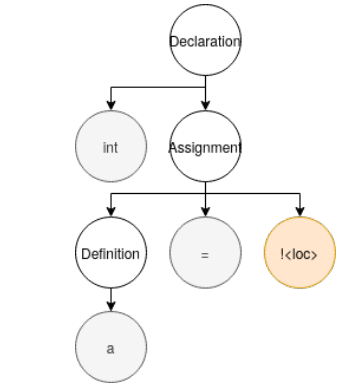
The first step when working with a model is to create a high-quality embedding of the tokens. We start this section by describing the tokenization applied to the code files, followed by the design of the embedding network. We then elaborate on the need for each separate component in the network and how the Loss is calculated. Finally, we explain how the two losses are combined to complete the description of the entire model.

Tokenization Before being able to embed the source code the text needs to be tokenized. Existing models have used a combination of Byte-Pair Encoding (BPE) and sub-word tokens [53], similar to the SentencePiece encoding used by IntelliCode [57]. Some methods as word-level tokenization are unsuited for source code as when creating variable names there are no specific rules for what is allowed. Furthermore, variable names are often combinations of different descriptive words concatenated making it prohibitively expensive to create a dictionary containing all possible variable names. It is better to split them according to the conventions they follow. Conventions such as camelCase, PascalCase, snake_case, or kebab-case are all ways of merging words but keeping them readable to a programmer.

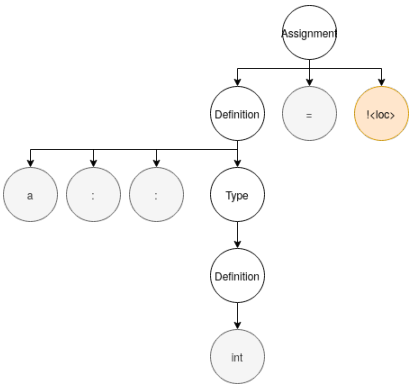
When handling the tokens of all programming languages we split the tokens into two groups. The first group is the group of keywords and tokens used by a language. We refer to these tokens as language-defined tokens. These will be keywords such as `if`, `for`, `else`,

[1] https://www.antlr.org/

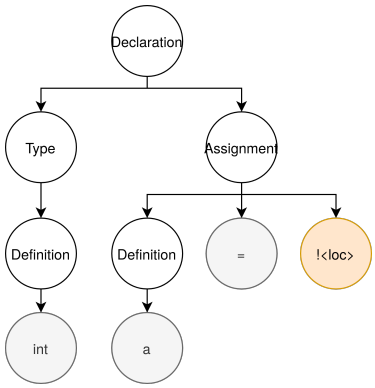
[2] https://github.com/antlr/grammars-v4



(a) Java



(b) Julia



(c) CPP

Figure 3.6: Examples of incomplete ASTs in all 3 languages.

```
int a = !<loc>
```

(a) Java

```
a::int = !<loc>
```

(b) Julia

```
int a = !<loc>
```

(c) CPP

Snippet 3.6: Example of incomplete code snippets in all 3 languages.

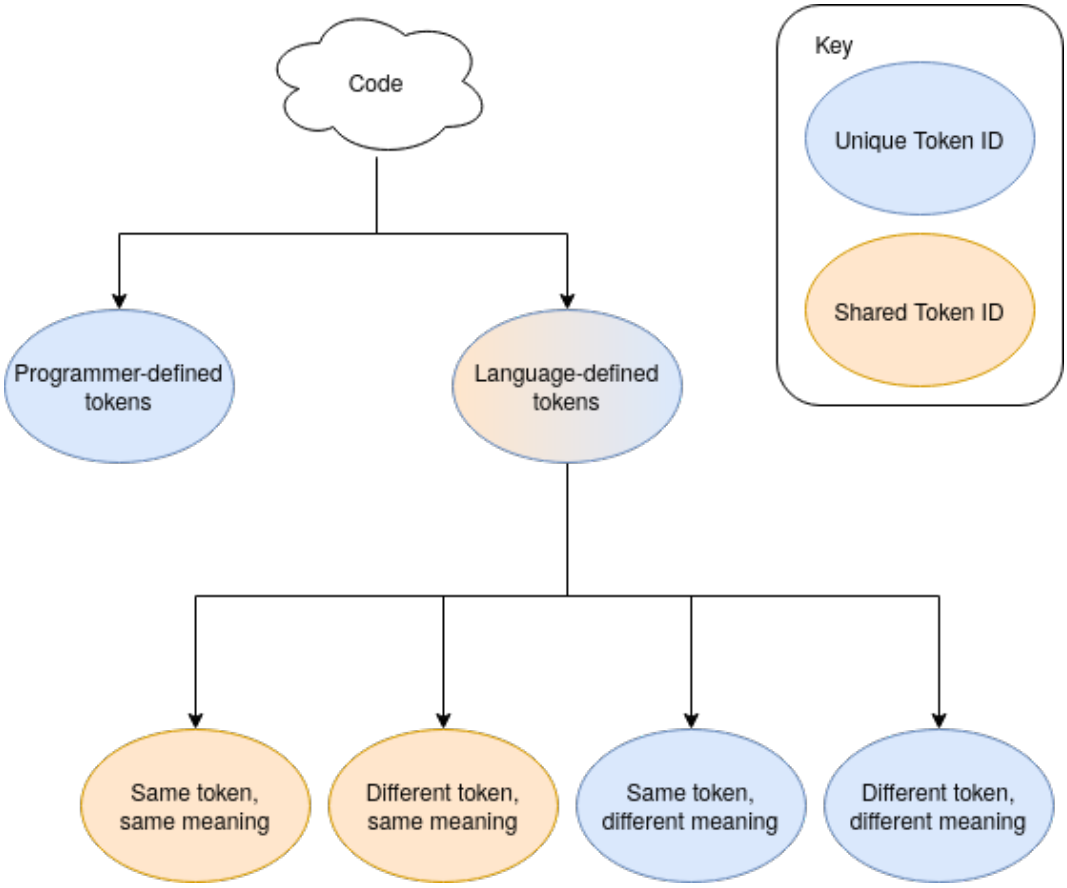


Figure 3.7: Visual representation of the taxonomy of different tokens.

but also symbols that may be defined in the language such as `print, bigInt, set_time`. The second set of tokens is all tokens that could be supplied by the programmer. These would be method names, variable names, or anything that is up to the discretion of the programmer. We will refer to this set of tokens as the programmer-defined tokens.

We begin with the first group of tokens, the language-defined tokens. As we are working in a multi-lingual setting and want to use trans-lingual anchor points, we will be using these language-defined tokens as the anchor points. When tokenizing, we want the same token in different languages to map to the same token ID. In practice that means that you can map the same tokens across languages, such as **`if`**, **`for`**, **`while`**, ',' which are shared between CPP, Java, and Julia, to one token. However, as not every language is the same, not all language-defined tokens are the same. When working with keywords that are different in each language yet have the same meaning, we also want to map these to the same token ID. An overview of the taxonomy is given in figure 3.7.

Once the language-defined tokens we work on the programmer-defined tokens. When looking at the way methods and variables are named during programming, we can see that they are often many different words connected in some way. Which lets the programmer

read the sub-words making up the variable/method name. The most common conventions for writing variable/method names are camelCase, PascalCase, snake_case, or kebab-case. As shown in the names of these conventions they can each be split into two words either by looking at capital letters or the charaters '_' and '-'. When tokenizing the programmer-defined tokens we begin by splitting compound method/variable names according to these rules. We also convert everything to lower case, and remove the connecting characters '_' and '-'. Once the model is deployed these tokens can be reinserted according to the desired convention.

After extracting the subwords unigrams [34], Byte-Pair Encoding (BPE) [53] or word piece [64] can be used to divide them further. The overarching thought behind these methods is to split the subwords further into smaller parts. Which reduces the size of the required vocabulary and helps with the Out Of Vocabulary problem (OOV) [64, 53, 34]. However, we must note that as each token is inserted as its own node in the graph, splitting subwords into smaller parts would skew the ratio of token nodes to structural nodes far in the favor of token nodes. Due to the limitation on available hardware, we choose not to split the sub-words further to save memory when training the model. Evidence has shown that word-piece or BPE is generally advantageous and would be a lucrative avenue to exploit in future work.

Finally, we insert some extra tokens into the AST. These will help determine when to stop predicting, or when to continue to the next node. As we are splitting each leaf node into its subwords, we insert an end-of-node token, <EON>, to the end of each leaf node so we know when to move on to the next node. Also, as there is not always a logical moment to end predictions, such as a full stop after a sentence, we add the end-of-prediction, <EOP>, token to the target vocabulary to help the model learn when it no longer needs to predict the next token.

We must note that we omit all non-syntactic whitespace and line endings. Due to stylistic choices, each developer may have their own preferences. Non-syntactic whitespace can be reintroduced after predicting according to each developer's wishes.

3.5 Embedding LSTM

The proposed model is an adaptation of the multilingual embedding network by Takashi Wada et al. [62] To adapt it to source code we introduce some changes. The idea behind the network is similar to word2vec, the embedding of each token relies on the neighboring tokens. A forward LSTM uses the preceding n tokens and a backward LSTM uses the succeeding n tokens. This Bi-LSTM is followed by a projection layer to generate a prediction for each token. The final projection layer that maps the output of the Bi-LSTM back to the languages is split into two parts. The first part of the layer is the neurons that predict the shared tokens between the languages. This part is shared. The other part of the final layer is the neurons that predict the tokens that are specific for each language. This last part of the projection layer is not shared. Each language trains its own weights. We implement the model and increase the number of anchor points from 2 (end of sentence and beginning of sentence tokens) to any number of tokens from the shared token ID groups described ear-

lier. We believe these shared tokens help map each language to the same embedding space. However, not sharing all but a few tokens prevents any issues from ordering as described previously. Furthermore, we change the training goal of maximizing the probability of a token given the surrounding tokens into a multi-task learning setting.

An intuitive way to visualize the idea behind the network is that it maps tokens to an embedding space that uses the neighborhood of each token. Running this method without sharing weights would give one mapping to a unique language embedding space per language. Once the weights are shared between languages, the shared parameters will be updated while training each language. Here each language updates the weights to suit itself. Due to all languages updating these weights the only way to get a good result is if the network changes the embedding of language-specific tokens around the shared tokens. This ensures the separate unique language embedding spaces mentioned earlier are now tied together at each shared token. Consequently, all other tokens will also be in a multilingual embedding space. These points are also named ”anchor points” [16]. In previous works these anchor points have always been instances of loan words in natural language.

As mentioned earlier, we extend the problem of finding high-quality embeddings to a multi-task learning setting. Looking at figure 3.8 we can see why this is necessary. The conventional method of training embeddings [62, 47] is to maximize the probability of the given token as is done in Task 1. However, when working with incomplete code, we would not possess the information of the forward context (the input to the backward LSTM) which would make the network infeasible. To remedy this we extend the problem definition to include Task 2. Task 2 aims to reduce the difference in embeddings given when concatenating the forward and backward state of the LSTM model to the embedding of the actual token. This lets us benefit from high-quality embeddings similar to the ones used successfully in the ELMo model [47, 32], without needing to run the entire Bi-LSTM model for each prediction.

Training objectives The first task used to train the network is predicting the correct token given the surrounding tokens. This is the usual setting when learning any language model. We attempt to maximize the probability of each token by maximizing the log-likelihood as given in equation 3.1.

$$\sum^{L} log(P(T_n^L | T_n^L - l, ..., T_n^L - 1, T_n^L + 1, ..., T_n^L + l, \overrightarrow{\theta}, \overleftarrow{\theta}), \theta^L, \theta^S, \theta^E) \tag{3.1}$$

In equation 3.1 T_n^L refers to the n^{th} token T from language L. l is the length to the rest of the file, and $\overrightarrow{\theta}$, $\overleftarrow{\theta}$ are the parameters for the forward and backward LSTM models respectively, θ^L, θ^S, θ^E are the parameters of the language-specific layer, the shared layer and the embedding layers respectively.

For the second task, we do not maximize the log-likelihood. However, we minimize the Mean Squared Error (MSE) as given in equation 3.2.

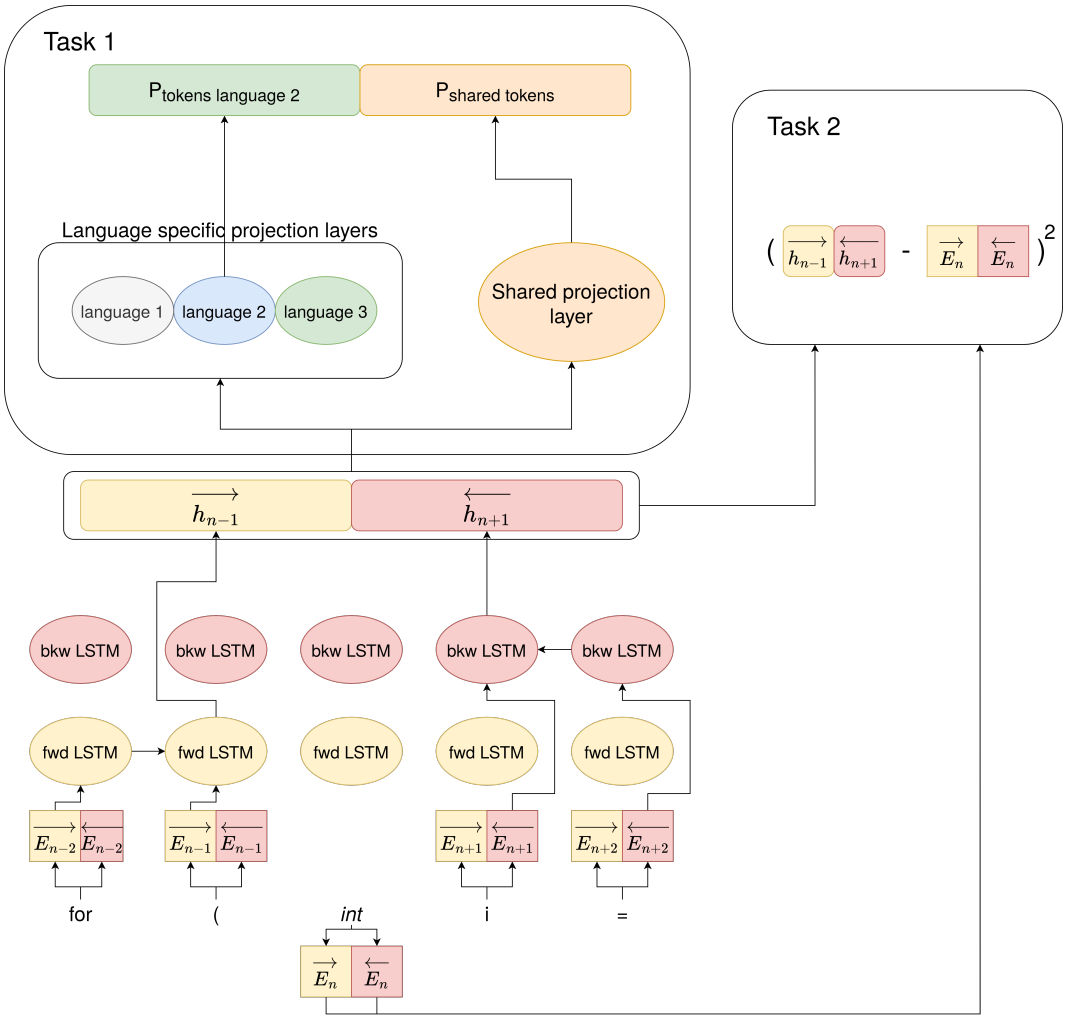


Figure 3.8: The architecture of the embedding model. As can be seen the model takes in a file of code, and for each token tries to predict the correct token based on the preceding and succeeding hidden state of the forward and backward LSTM respectively. Task 1 minimizes the cross entropy loss of predicting the correct token, Task 2 minimizes the Mean Squared Error (MSE) between the token embedding and concatenation of the hidden states.

$$MSE = (P_n - E_n)^2 \tag{3.2a}$$
$$MSE = ([\overrightarrow{h_n}; \overleftarrow{h_n}] - [\overrightarrow{E_n}; \overleftarrow{E_n}])^2 \tag{3.2b}$$
$$\overrightarrow{h_n} = \overrightarrow{LSTM}(T^L_{n-1}, ..., T^L_{n-l}, \overrightarrow{\theta}) \tag{3.2c}$$
$$\overleftarrow{h_n} = \overleftarrow{LSTM}(T^L_{n+1}, ..., T^L_{n+l}, \overleftarrow{\theta}) \tag{3.2d}$$
$$\overrightarrow{E_n} = E(T_n, \overrightarrow{\theta^E}) \tag{3.2e}$$
$$\overleftarrow{E_n} = E(T_n, \overleftarrow{\theta^E}) \tag{3.2f}$$

In equation 3.2, $\overrightarrow{h_n}$ represents the hidden state at time n of the forward LSTM model, $\overleftarrow{h_n}$ represents the hidden state at time n of the backward LSTM model, $\overrightarrow{E_n}$ is the embedding of Token n for the forward LSTM model, $\overleftarrow{E_n}$ is the embedding of token n for the backward LSTM model.

From the above descriptions of the training tasks, we see that most of the functions are parameterized with the parameters belonging to the model. These are important to discern as some of them are shared between languages while others are not. For clarity we will list the parameters and group them into shared and not shared.

Shared parameters

- $\overrightarrow{\theta}$ the weights of the forward LSTM
- $\overleftarrow{\theta}$ the weights of the backward LSTM
- $\overrightarrow{\theta^E}$ the weights of the forward embedding layer
- $\overleftarrow{\theta^E}$ the weights of the backward embedding layer
- θ^S the weights of the projection layer from the output of the Bi-LSTM to the shared probability distribution

Language-specific parameters

- θ^L the weights of the projection layer from the output of the Bi-LSTM to the language-specific probability distribution

3.5.1 Multi-Task Learning

After defining the task, we describe how we combine the losses. Although running the model and updating the weights over each task independently would work, we see some issues that may arise in this setup. Looking at the two loss functions, we see that the range of the output of the loss function varies. If the loss for one task is always higher than for the

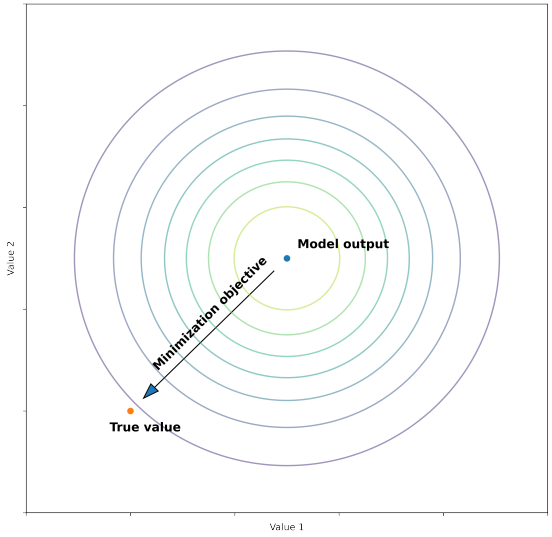


Figure 3.9: Graphical representation of the homoscedastic uncertainty.

other task, the model will learn more in the favor of the task with the higher loss. To solve this we will need to find a way to balance the losses to combine them optimally.

This problem of combining loss functions with varying magnitude is not new. It has shown promise in many different areas, such as computer vision [15]. The main idea behind Multi-Task Learning (MTL) is that a network predicts multiple outputs to multiple tasks, then a joint loss function is calculated over all tasks, and the losses are used to update the weights of the model.

Although an optimal mixture of losses can be found through a basic grid search, trying out different weights for each combination of losses. This is prohibitively expensive (The model would need to be fully trained for each weight combination). The weights for the losses can however be learned at the same time as training the model [15].

For our model, we employ a method that minimizes the homoscedastic uncertainty of each task. Homoscedastic uncertainty is the uncertainty that is specific to the task. Keeping all data the same homoscedastic uncertainty is the only uncertainty that changes between tasks.

To understand this better, we model the output of the regression task, Task 2, as a Gaussian distribution around the predicted value. The distribution, prediction and true value are shown graphically in figure 3.9 and mathematically in equation 3.3.

$$p(y|ENC(x,\hat{\theta})) = \mathcal{N}(ENC(x,\hat{\theta}),\sigma) \tag{3.3a}$$

$$\hat{\theta} = \cup(\overrightarrow{\theta}, \overleftarrow{\theta}, \overrightarrow{\theta^E}, \overleftarrow{\theta^E}, \theta^L, \theta^S) \tag{3.3b}$$

$$\tag{3.3c}$$

From equation 3.3, we can see that the uncertainty of the model around the true value is governed by the variable σ. The other terms in the equation are $ENC()$ which is the encoding model being described, x the input to the model, y the true value, and $\hat{\theta}$ the union of all parameters used by the encoding model.

Similar to Task 2, Task 1 can also be written as a probability distribution. For discrete training goals, we use a Boltzmann distribution. This cannot be drawn clearly so we only show the mathematical definition in equation 3.4.

$$p(y = c|ENC(x,\hat{\theta}),\sigma) = Softmax(\frac{1}{\sigma^2}ENC(x,\hat{\theta})) \tag{3.4a}$$

The Boltzmann distribution gives the probability of a system being in a certain state (the probability that the token is correct). We notice that the parameter σ used before in equation 3.3 is now similarly used in equation 3.4 to flatten the discrete probabilities (the temperature of a Boltzmann distribution).

Now that we know the distributions we will be using to train the model, we need to calculate the log-likelihood used for training. The log-likelihood for Task 1 is given in equation 3.5a and for Task 2 in equation 3.5b.

$$log(p(y = c|ENC(x,\hat{\theta}),\sigma)) = \frac{1}{\sigma^2}ENC_c(x,\hat{\theta}) - log(\sum_{c'} exp(ENC_{c'}(x,\hat{\theta}))) \tag{3.5a}$$

$$log(p(y|ENC(x,\hat{\theta}))) \propto -\frac{1}{2\sigma^2}\|y - ENC(x,\hat{\theta})\|^2 - log(\sigma) \tag{3.5b}$$

Now that we know both log-likelihoods we can combine them as is done in equation 3.5.1.

$$\mathcal{L}(\hat{\theta}, y_1 = c, y_2) = -log(p(y_1 = c, y_2|ENC(x,\hat{\theta}))) \tag{3.6}$$

$$\mathcal{L}(\hat{\theta}, y_1 = c, y_2) = -log(Softmax(\frac{1}{\sigma_1^2}ENC(x,\hat{\theta})) \cdot \mathcal{N}(ENC(x,\hat{\theta}),\sigma_2)) \tag{3.7}$$

$$\mathcal{L}(\hat{\theta}, y_1 = c, y_2) = \frac{1}{\sigma_1^2}\mathcal{L}_1(\hat{\theta}) + \frac{1}{2\sigma_2^2}\mathcal{L}_2(\hat{\theta}) + log(\sigma_1) + log(\sigma_2) \tag{3.8}$$

This derivation shows that for an optimal mixture of losses, we need to scale both losses. We use the variables σ_1 and σ_2 for scaling, the terms $log(\sigma_1)$ and $log(\sigma_2)$ can be seen as regularizers that prevent the model from setting a large value for either σ in order to get a low loss.

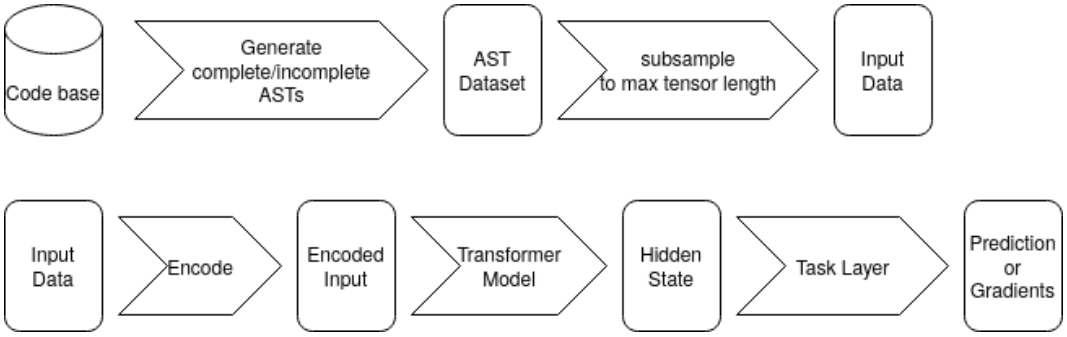


Figure 3.10: The data pipeline from source code file to prediction.

3.6 Transformer Model

The use of transformer models has been quite widespread in NLP as shown with the state of the art models BERT and GPT 1,2,3 mentioned in chapter 2. This makes them a strong candidate to be used when learning cross-lingual structures across multiple languages. To clearly describe the model that is being proposed, we divide the explanation into three sections. First, we give a general overview of the model, then the shape of the input tensors will be described, followed by a more detailed explanation of the model.

3.6.1 Model Overview

To give a precursory overview of all aspects covered in the coming section, we will give an overview of components of the model. This overview will focus on the general architecture rather than implementation which follows in later sections.

The pipeline used in the model is summarized in figure 3.10. As can be seen, there are several distinct steps. Firstly, the source code is taken and converted into an augmented Universal AST. The exact shape of this AST is dependent on the task as the pre-training and final tasks use slightly different grammars. These ASTs are then sub-sampled and transformed into one input tensor per sub-graph (for this work we limit it to 2 subgraphs, AST and source code). These tensors are then fed to the model. The model then pre-trains by learning on the MLM task, or the fine-tuning task of expr-gen. The embedding LSTM is not shown as it is trained separately. The embedding LSTM embedding layer is copied into the final transformer network.

3.6.2 The inputs

To start the detailed explanation of the model, we will begin with the inputs. Several key features have been implemented for this model. We begin by describing how we transition from an AST to an input vector. Then we will explain how the tokens are embedded. We finish by describing the choices of positional encoding.

First, we show how we transform a graph into a tensor that we can pass to the network. In other works, shown in chapter 2 tracing paths through the AST or visiting each node in a

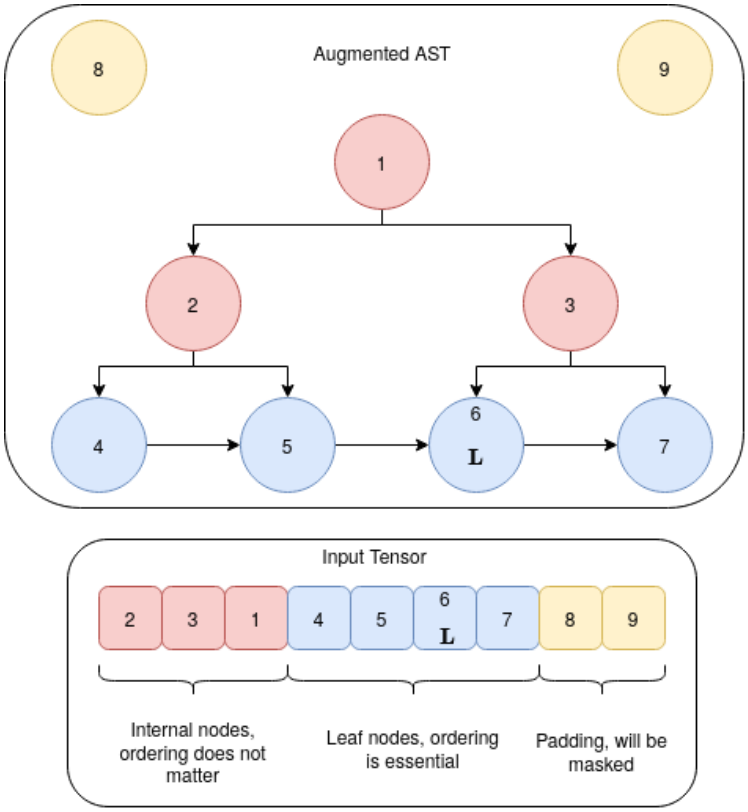


Figure 3.11: Visualization of the transformation of an AST to an input tensor.

pre-order ordering are both valid ways to represent an AST as a sequence. We take a slightly different approach to this. We recognize that three types of entries will be part of the input. These types are internal nodes, leaf nodes, and padding nodes. The ordering of the internal nodes is not important. Because positional information is added using positional encodings there is no mathematical purpose to imposing an ordering on the vector representation of the internal nodes. For the leaf nodes, we will be using relative positional attention [54]. This requires that the leaf nodes be ordered according to the order in the source code. There may not be any non-leaf node between any 2 pairs of consecutive leaf nodes. Finally, as some ASTs may be smaller than the limit of the maximum number of nodes we must include padding nodes. Padding nodes may also be added anywhere (except between leaf nodes) as they will be removed from the input using attention masks. This gives us the final input tensors that will be used in the model. We include a graphical representation of this in figure 3.11, this basic input tensor will be augmented with positional encodings depending on which encodings are used for the given sub-graph.

After the input has been transformed into a tensor we must embed the information contained in the nodes to make it usable for the network. As mentioned for this work there will be two different input vectors used. One contains the structural information from the AST nodes, and the other the information of the code tokens. For the structural information, we

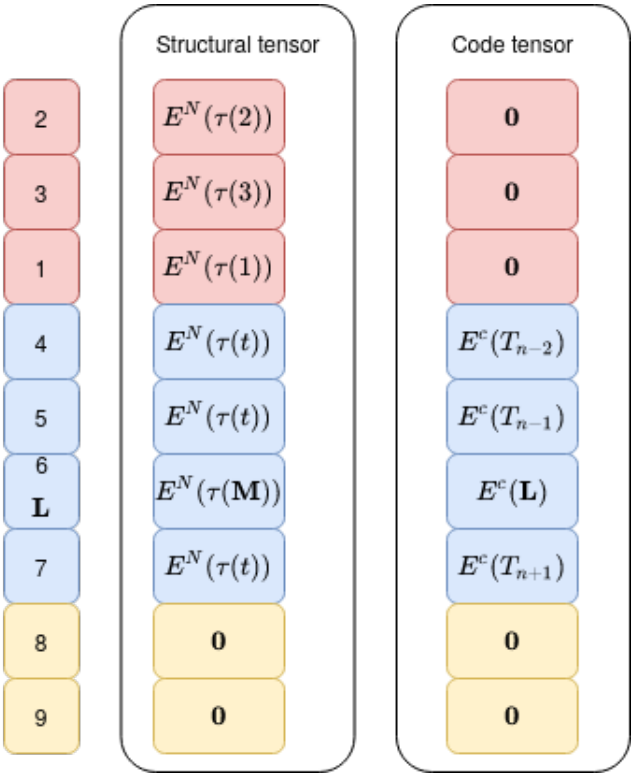


Figure 3.12: Visual representation of how a tensor is embedded.

learn an embedding from scratch. This will embed the information of the type of each node in an appropriately sized embedding space. For the source code information, we will use the embedding layers trained with the embedding LSTM, in combination with a projection layer to project them to the correct dimension. We give a graphical representation using the same tensor as in figure 3.11 in figure 3.12. In figure 3.12 we define the set of all node types as τ, and use t to denote the type of token. Furthermore, E^N gives the node type embedding, while E^c gives the code embedding. The special token **L** is the location token, which is also included in the vocabulary of E^c, similarly to how the prediction location is masked with a <MASK> token in the code, we introduce a special mask type (M) for the structural component of the input. The structural embedding corresponding to the location token is $E^N(\tau(M))$.

Now that we know how the input tensor is created we look at the different positional encodings. One of the main differences between a transformer and a traditional model such as an LSTM used for language tasks is that a transformer, with the original attention mechanism, does not need the tokens supplied to be in a specific order because the Q, K, V vectors that it calculates attend over the entirety of the input that it can see. This is useful as vanilla LSTMs have issues modeling long-range dependencies. As an LSTM updates a hidden state at every token, older tokens are quickly forgotten as only information from the last state can be added or removed. This makes it more and more likely that information

about an older token is forgotten the further away it is. Transformers do not have this problem as they can choose what part of the sequence to use. However, transformers without positional encodings do not know what order data is in. To illustrate this we give a basic problem of a transformer that is trained to predict whether a set of tokens is a question. Given the two inputs ”this is a dog” and ”is this a dog” a transformer will have difficulties telling which of the two is a question as the order is vital. To address this problem, positional encodings are added to transformers. Positional encodings are a way to tell the transformer in what order the words appear. Positional encodings would turn the input from ”this is a dog” and ”is this a dog” into ”(this,0) (is,1) (a,2) (dog,3)” and ”(is,0) (this,1) (a,2) (dog,3)” which now contains the information of the order of the words.

Positional encodings for nodes in a graph are more complex than ordering tokens in a sequence. There have been attempts at feeding the positional information to transformers. Other papers use a form of hop-distance, or an encoding such as the W-L encoding [67], however, these papers tend to calculate the attention on a per-node basis, which allows for the different values in hop distance encoding. As we are using one tensor for the entire graph we would like to encode the position of the nodes globally. For this purpose, we use a spectral approach to positional encoding. This has been shown to be advantageous when working with molecular tasks [20].

In the spectral approach, we use methods that have in the past been used effectively in feature reduction and clustering [45]. These methods are similar to finding a positional encoding of a node in an abstract space as, especially for clustering, the nodes with similar values will be close (same cluster) and nodes with dissimilar values will be far away (different cluster). The method we use for this is taking the m eigenvectors that correspond to the m smallest non-zero eigenvalues of the normalized Laplacian matrix of the graph. The equation for the Laplacian is given in equation 3.9 and the equation for the normalized Laplacian is given in equation 3.10. In both equations, 3.9 and 3.10, D is the degree matrix of the graph, A is the adjacency matrix of the graph, L is the Laplacian of the graph, and Δ is the normalized Laplacian of the graph.

$$L = D - A \tag{3.9}$$

$$\Delta = I - D^{\frac{-1}{2}} A D^{\frac{-1}{2}} \tag{3.10}$$

Looking at the Laplacian matrix and its construction there are a few things we can deduce. Firstly, given a connected graph (in this case all graphs will be connected) the Laplacian will be positive semi-definite. As a consequence all the eigenvalues of the matrix will be non-negative [30]. Also, given that the graph is fully connected, the smallest eigenvalue will be 0, this is an issue as the trivial solution will map everything onto the same point [30]. Therefore we start with the eigenvectors corresponding to the second smallest eigenvalue.

Furthermore, While a graph embedding may be a good idea for the AST portion of the input, the sub-graph corresponding to the source code nodes in the augmented AST needs a different embedding. The sub-graph will always contain two nodes, at the start and end of the code, with two edges. In the middle there will be $n-2$ nodes, with four edges each.

Taking the eigenvectors of such a graph will not give much information about the order of the source code. Furthermore, from chapter 2 we know that successful models such as BERT [18], and GPT [12] make use of the previous mentioned positional encodings. The distinction between the setup of the BERT and GPT models is that there is not a very well-defined concept of a sentence in source code. There have been attempts at modeling the source code as pairs of lines of code [31] with some success, however as many languages let you write all code on one line (if you wanted to) we choose to not try and impose the idea of sentences onto source code. Instead, we will be taking a random sub-sequence of the source code which could start and stop at any point.

The randomly samples subsequence could be anywhere in the code. This hinders absolute positional encodings as only the relative distance between tokens is important, rather than the position in the sequence. There have also been other works that apply transformer models to source code that have found the absolute positional encodings to be detrimental during early testing [33] which correlate to our early results. Due to this, we will not be using an absolute positional encoding for any inputs. Instead, we replace the attention head with one that uses a relative positional attention mechanism [54].

To conclude this section, there are two different input vectors to the model. One for each sub-graph type. The structural tensor containing the node types uses the Laplacian positional encodings, whereas the code tensor does not have any positional encoding as it will use a relative positional attention head.

3.6.3 The Model

Now that we understand the inputs to the model, we will explain how the model works. We will start by describing the layout of the model, followed by the calculations being used in each section. The model being proposed is an extension of the model for molecular predictions [20] where each transformer block has its own sub-blocks that are responsible for each different type of edge. We choose to not share weights between the different transformer blocks.

To define the model we recall that there is one input tensor per sub-graph. Each of these inputs is fed to a transformer block which attempts to reason about the input data. Each transformer block also returns a hidden state at the end. Before passing the output onto the task layers, a weighted sum (where the weights are learned during training) is employed to combine the outputs for all subgraph types to give an encoding of the entire graph. In figure 3.13 we show an overview of the model including the differing attention calculations for the different sub-graphs.

From figure 3.13 we can see that there are multiple steps to calculate. To make it more manageable we will break it down into smaller parts. We will start by explaining the structural transformer block, then the source code transformer block, and finally we will finish with the task-specific layers.

As mentioned before each transformer block in the larger transformer layer gets its own input. We denote this as h_j^l where l counts the layers, and j denotes the transformer block. In this case $j \in \{\text{structural}, \text{code}\}$ and $l \in \{0, 1, ..., N\}$. The input to the structural transformer block is calculated as in equation 3.11.

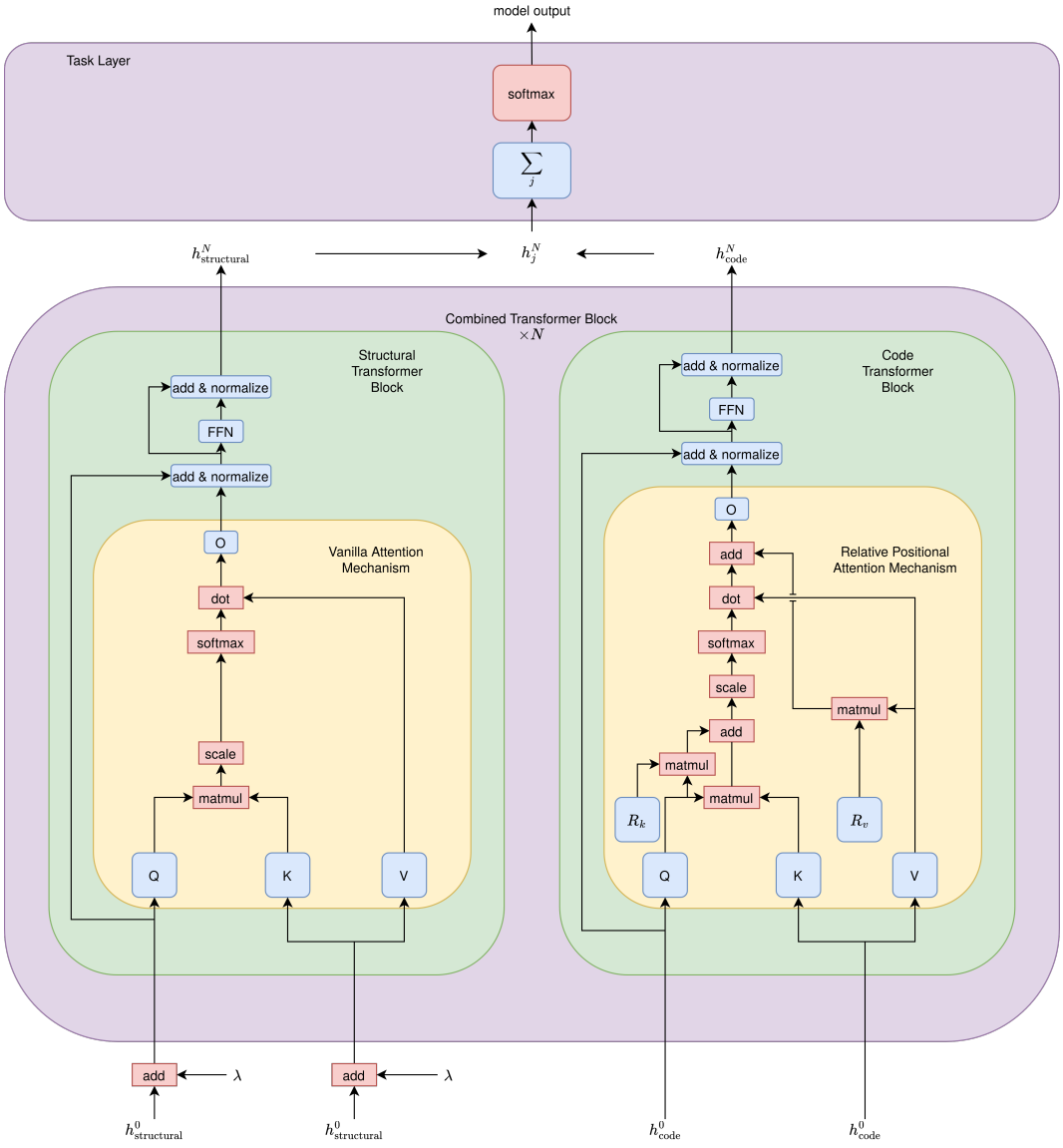


Figure 3.13: Diagram of the proposed model.

$$h_j^l = \begin{cases} h_j^l + \lambda & \text{if } j = \text{structural } \& \ l = 0 \\ h_j^l & \text{otherwise} \end{cases} \tag{3.11}$$

In equation 3.11 the Laplacian positional encodings (λ) are only added for the very first layer.

Once the input is known we can use it to calculate the original attention [60] as given in figure 3.6.3.

$$\alpha = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3.12}$$

$$Q = h_j^l W_j^Q \tag{3.13}$$

$$K = h_j^l W_j^K \tag{3.14}$$

$$V = h_j^l W_j^V \tag{3.15}$$

In equation 3.6.3 the learned parameters for the Q, K, V vectors are given by W_j^Q, W_j^K, W_j^V respectively.

As we are using multi-headed attention, we will extend equation 3.6.3 with the concatenation and projection step from the original paper [60].

$$\hat{\alpha} = \text{concat}(\alpha_1, ..., \alpha_n) W_j^O \tag{3.16}$$

In equation 3.16 the output of multiple attention heads α are concatenated and passed to a linear projection layer to project them into the dimensionality of the model, the parameters of which are contained in W_j^O.

Knowing the output of the attention mechanism we can continue to use this to calculate the output of the transformer block.

$$h_j^{l+1} = \mathbf{N_2}(\text{FFN}(\mathbf{N_1}(\hat{\alpha} + r_1)) + r_2) \tag{3.17}$$

$$FFN(x) = \text{GeLU}(xW_1 + b_1)W_2 + b_2 \tag{3.18}$$

$$r_1 = h_j^l \tag{3.19}$$

$$r_2 = \text{FFN}(\mathbf{N_1}(\hat{\alpha} + r_1) \tag{3.20}$$

$$\mathbf{N_1}(x) = \mathbf{N_2}(x) = \frac{x - \bar{x}}{\sqrt{var[x] + \varepsilon}}\gamma + \beta \tag{3.21}$$

$$\bar{x} = \text{mean}(x) \tag{3.22}$$

$$\varepsilon = 1 \times 10^{-6} \tag{3.23}$$

This gives us the output of the layer that can be passed to all later layers. In equation 3.6.3 the functions $\mathbf{N_1}$ and $\mathbf{N_2}$ are the batch norms [26], r_1 and r_2 are the residual connections also seen in figure 3.13. γ and β are the learnable parameters of the batch norm. FFN

is a feed forward network using the GeLU [25] activation function. The weight parameters of the layers are saved in W_1 and W_2 and the bias parameters are saved in b_1 and b_2. Similar to the original paper the feed forward network first projects the input into a 2048 dimensional space before condensing it back to d_{model} dimensions [60].

Akin to the structural transformer block, the code transformer block also works by calculating the attention and then passing it to the a feed forward network, however, the code transformer block does not add any positional encodings as in equation 3.11, instead the attention calculation from equation 3.16 and equation 3.6.3 have been altered slightly to give the relative positional attention [54] [3].

The new attention calculation, using relative positions is given in equation 3.6.3.

$$e = \text{softmax}\big(\frac{Q(K+\mathbf{R}^K)^T}{\sqrt{d_k}}\big) \tag{3.24}$$

$$\alpha^r = e(V+\mathbf{R}^V) \tag{3.25}$$

Equation 3.6.3 shows us that the calculations are very similar. The only difference is that the terms K and V have been replaced with $K+\mathbf{R}^K$ and $V+\mathbf{R}^V$ respectively. The matrices $\mathbf{R}^K$ and $\mathbf{R}^V$ are learned parameters that give the attention score between two tokens in the sequence up to a maximum distance of k [54]. The values Q, K, V are the same as described in 3.6.3.

Similarly to the structural transformer block, we split the relative attention score α^r into a multi-headed attention mechanism, exactly like in equation 3.16.

This multi-headed relative attention score is then passed to a FeedForward Network (FFN) as shown in equation 3.6.3.

Task layer As we want to benefit from pre-training the model on the MLM task as in BERT [18], as well as the potential to solve any tasks in the future, we define the input to the task layer to be the weighted sum of the outputs of each transformer block. The combined representation can be passed to any final layer.

For both the MLM, as the next token prediction task, we use the same architecture for the task layer. It is important to note only the architecture is the same, no weights of the task layers will be shared between tasks.

Given the output of the transformer block, we pass the weighted sum of representations for each subgraph type to a final projection layer. The projection layer will project each node into the correct output space as is shown in equation 3.6.3.

$$P = \text{GeLU}(xW^T + b^T) \tag{3.26}$$

$$x = \sum_j h_j^N \tag{3.27}$$

[3] https://github.com/evelinehong/Transformer_Relative_Position_PyTorch

For the final prediction, we take W^T as the weights of the task layer and b^T as the bias of the task layer. Furthermore, N denotes the output of the last layer of the transformer blocks.

As we are using the cross-entropy loss from the PyTorch library as the criterion, we do not take a softmax over the final layer as this is implemented in the library [4].

3.7 Training

From looking at the available data and the state of the art models [18, 31], we can see that the problem of predicting source code in a low resource environment is well suited for a pre-training setup. In this setup, a model is first trained on a similar task with more available data and can be fine-tuned to a task that may not have as much data available.

Fine-tuning is a method where the weights learned during pre-training task(s) are saved after training and loaded as the starting weights for any downstream tasks. In this case, we use a task similar to the masked language modeling or cloze task used in the original BERT paper [18]. We do this by masking a set percentage (15%) of tokens in the source code with a masking token and letting the model learn to predict the tokens given the rest of the file.

Originally, in the BERT model, the masking token was a token exclusively used during pre-training. However, as we know that the downstream task that this model is designed to solve is code completion, we choose to use the same token (!<loc>) for the pre-training masking token as the token that will tell the network at which location to predict the source code in later tasks. Furthermore, when masking the tokens, there was a 10% chance to insert a random token rather than the location token, there was a 10% chance that the token was not masked or replaced, and an 80% chance that the selected token was masked with the location token.

We summarize the MLM task as shown in equation 3.28.

$$MLM = \max(P(y = t | t \in T, C, \Theta) \tag{3.28}$$

Equation 3.28 maximizes the probability that the prediction, y, of the model equals a token, t. Given that said token is in the set of masked tokens T. The surrounding context (the input to the model) is given as C, and the parameters of the model are contained in Θ.

Once the model finishes training on the MLM task, we switch out the task layer for the final task layer and train the model to maximize the final task the Next Token Prediction task (NTP). We summarize this task in equation 4.1. This is equation gives uses the same variables as 3.28 however, instead of a set of targets, we only have the target, t, and the context is replaced by the incomplete AST, AST'. For the downstream training, we update all weights in the model not only the weights in the final layer.

$$NTP = max(P(y = t | AST', \theta)) \tag{3.29}$$

[4] https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

3.8 Baseline

To assess the benefits of the Graph-Based Transformer model over conventional methods we need to compare it to a baseline that is as similar as possible and trained on the same data. The closest model that is commonly used and has shown state-of-the-art performance is a BERT-style encoder transformer [18, 31].

We will feed the baseline the same data, the only difference is that it does not have a module to handle the positional data. The baseline is essentially the proposed model with the structural transformer block removed.

We limit ourselves to comparing the model to a vanilla BERT implementation due to several reasons. Primarily, there are not many models that have been trained on the languages CPP, Java, and Julia, so retraining and evaluating all the other models would take a large amount of time. Furthermore, as many of the other models we have looked at also work using ASTs, it would require a custom implementation of the ASTs of each specific language to retrain the models on Java, Julia, and CPP.

Chapter 4

Experimental Setup

To ensure that the experiments run to evaluate our model are reproducible we will dedicate this chapter to explaining each step we took conducting the experiments. We begin by defining the task clearly. This should remove any ambiguities of how the model is trained. We then move on to the data we use. We explain how we collected the data and what steps we took when processing it. We follow this up with the hyper parameters we used during the experiments before ending on a detailed setup for each run used hen answering the research questions.

4.1 The Task

Due to the relative novelty and specificity of using machine learning models in software engineering, almost every paper, although closely related, has its own highly specific task. This makes it essential to delineate the scope and limit of the task the models are being trained for.

There have been attempts to use ASTs to predict method and variable names [7, 56, 33], or to generate expressions [11], as well as using incomplete ASTs to predict the next token in code [63], using structural information for next token prediction [28] or only the source code [57].

For this thesis, we aim to model code completion similar to a combination of the exprgen task [11] and the training procedure of the InCoder model [22]. We define our task as a next token prediction task using incomplete ASTs. We select the locations where we can predict a statement in the source code and mask the expression from each token up to the end with a <MASK> token. An expression consisting of N tokens will generate $N+1$ incomplete code snippets, including the <End of Prediction> token (<EOP>). These incomplete code snippets are then parsed into Incomplete Augmented ASTs and given to the model. Given an incomplete AST, T' we want to maximize the probability of the token missing at the given location, as shown in equation 4.1.

$$P(y = t|T', \Theta) \tag{4.1}$$

```
function returnType f(a,b) extraAnnotations:
    int c = <num>;
    String s = <str>;
    Bool d = <bool>;

    while (a > <num>)
        if(a OR <bool>)
            api.call(a,b);
        EndIf
    EndWhile

    return a + b;
endFunction
```

Snippet 4.1: Examples of valid prediction locations, anything highlighted can be masked with a location token and predicted.

```
int c = <num>;
<MASK>
Bool d = <bool>;
```
(a) String

```
int c = <num>;
String <MASK>
Bool d = <bool>;
```
(b) s

```
int c = <num>;
String s <MASK>
Bool d = <bool>;
```
(c) =

```
int c = <num>;
String s = <MASK>
Bool d = <bool>;
```
(d) <str>

```
int c = <num>;
String s = <str> <MASK>
Bool d = <bool>;
```
(e) ;

```
int c = <num>;
String s = <str>; <MASK>
Bool d = <bool>;
```
(f) <EOP>

Snippet 4.2: Example of all incomplete code snippets that would be generated from a single statement.

In equation 4.1, t is the target token, y is the prediction, Θ are the parameters of the network, and T' is the incomplete AST. An example of valid prediction locations for the network can be found in 4.1.

We take a section of the second prediction location from listing 4.1 to show all possible (T', token) pairs shown in listing 4.2.

4.2 Dataset

Equally as important as the task, is the data. To give an accurate representation of the data we collected we will describe all steps and data-distributions in the following section.

Dataset	Number of files	Lines of code	Number of tokens	Number of unique tokens
Julia	4,094	546,334	8,092,371	17,251
Java	202,747	21,821,469	316,373,986	10,328
CPP	36,992	11,047,576	125,076,720	19,897

Table 4.1: Statistics of the dataset before splitting into train, test and validation sets.

Data Collection Often, the data collected for model training is from a curated source. The goal of this is to ensure that the data is of high quality and ensures the results of the network are accurate. As we strive to create code completions for a programmer we aim to have high-quality source code for the model to learn how to code. We aim to implement this by downloading code from GitHub where we use only the top 100 projects by amount of stars from each language. Due to the possibility that there may be duplicates of code between these libraries we removed all exact copies of code in the dataset. We give the statistics for each dataset in table 4.1.

We subsample the larger datasets when training the embedding model as the long training times for LSTMs would make it impossible to use all files. Therefore we take 10,000 files for CPP and Java. For Julia, we have less than 10,000 files. Therefore, we sample a sub-sequence from each file more than once, to ensure 10,000 samples are used for training in each epoch.

To train our models effectively there are several operations that need to be performed on the dataset. This in combination with the multiple evaluations along the training of the models means we need to pay close attention to ensure the purity of the train, test and, validation split.

Often, when working with deep learning models, the data is split 80% in the train set, 10% in the test set, and 10% in the validation set. We aim to approximate this split as closely as possible. However, as we split the files based on predictions we want to make sure that samples from the train set for pre-training, do not end up in the test set during fine-tuning.

We avoid these issues by splitting the dataset according to files. After splitting the files into incomplete AST, prediction pairs from all files that were in the test, train, and validation set during pre-training will be in the same test, train, or validation set during fine-tuning. This will result in slight deviations from the previous split percentages due to different file lengths. However, this gives the fairest evaluation of the models.

To further evaluate the performance of the model on unseen data, we download the next ten most starred projects from Github for each language. We call this dataset the unseen dataset. The main difference between this dataset and the test dataset is that the dictionary was created on the entire corpus (including the test set). This results in the occurrences of tokens in the test set being taken into account when generating the dictionary. Adding a final dataset with ten more projects gives a more fair view of real-world performance.

To speed up training we also acknowledge that due to the max number of nodes being 100, we cannot read data from large distances in the files. We split the input files by constructor and method, keeping all relevant information such as local variables and class names should they be applicable. There are some cases where the version of code is not known so the file cannot be parsed. Also, some files maybe lost due to the parser timing

Language	Dataset	Samples
Julia	Train	10701
	Validation	1268
	Test	965
	Unseen	749
Java	Train	146166
	Validation	20842
	Test	16258
	Unseen	10176
CPP	Train	87837
	Validation	11007
	Test	10605
	Unseen	15420

Table 4.2: Dataset statistics for the pre-training task.

out during the generation of the dataset. This leaves us with a slightly smaller dataset than the dataset used for the LSTM model. The statistics for the dataset used to train the masked language model are given in table 4.2.

Once the pre-training dataset has been created we use the same split of data to create the incomplete AST, prediction pairs that we will be using for the final task. To create the input pairs the ASTs are subsampled to a max of 100 nodes with a max tree height of 6 from the bottom up. The statistics for this dataset are given in table 4.3

Finally, we will give a quick insight into the distribution of the tokens in the dataset. As we are working with a language we expect the data to follow a Zipfian distribution [69]. To visualize this we plot a histogram of the 6th $-$ 106$^{\text{th}}$ most common tokens in each language with their probability in figure 4.1. Furthermore, we plot the index of the token against the number of occurrences on a logarithmic scale and verify that it is largely linear, as can be seen in figure 4.2.

4.3 AST subsampling

Due to the nature of transformers, the input to the model always needs to be the same length, this can be achieved by either padding small ASTs to have the same number of nodes as the biggest AST, or we can subsample the biggest ASTs to have the same number of nodes as the smallest. In this thesis, we choose to do a combination of both. We pad input that is shorter than 100 nodes to a length of 100 nodes, and we subsample any ASTs that are larger than 100 nodes to at most 100 nodes.

The subsampling method we propose is not a simple truncation of the AST to have 100 nodes. From past research, we have seen that there is a diminishing return for paths through an AST that are larger than 12 jumps [6]. We use this information when working on the subsampling algorithm to ensure that the longest path that may be traced through the

Language	Dataset	Samples
Julia	Train	183,228
	Validation	22,643
	Test	43,689
	Unseen	13,519
Java	Train	1,622,715
	Validation	242,672
	Test	424,162
	Unseen	259,589
CPP	Train	3,350,872
	Validation	438,772
	Test	827,808
	Unseen	719,224

Table 4.3: Number of (incomplete AST, token) pairs present in the final dataset.

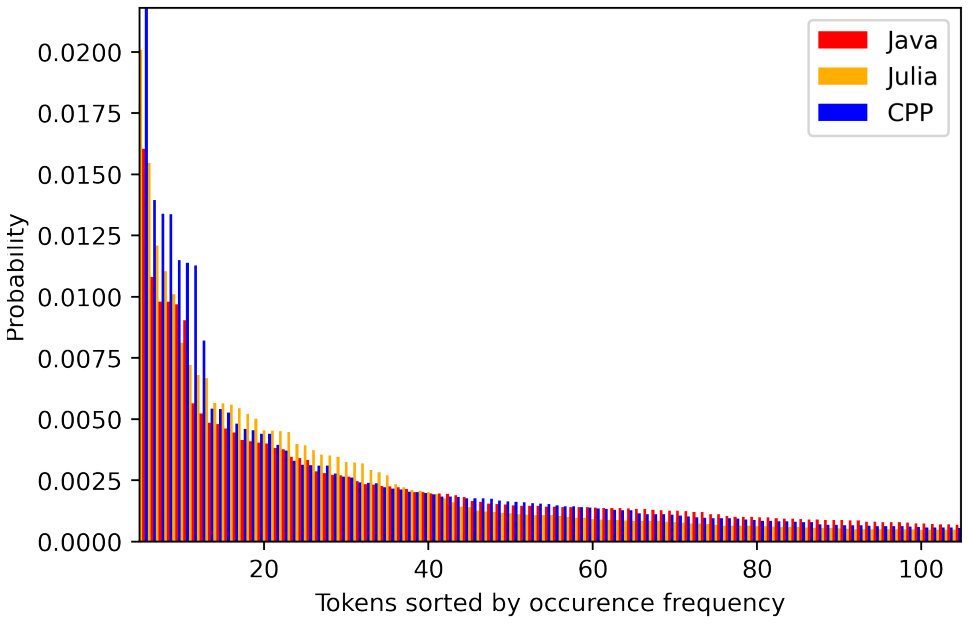


Figure 4.1: Histogram of the frequency of tokens, we look at tokens 6 - 106 in order to make it readable.

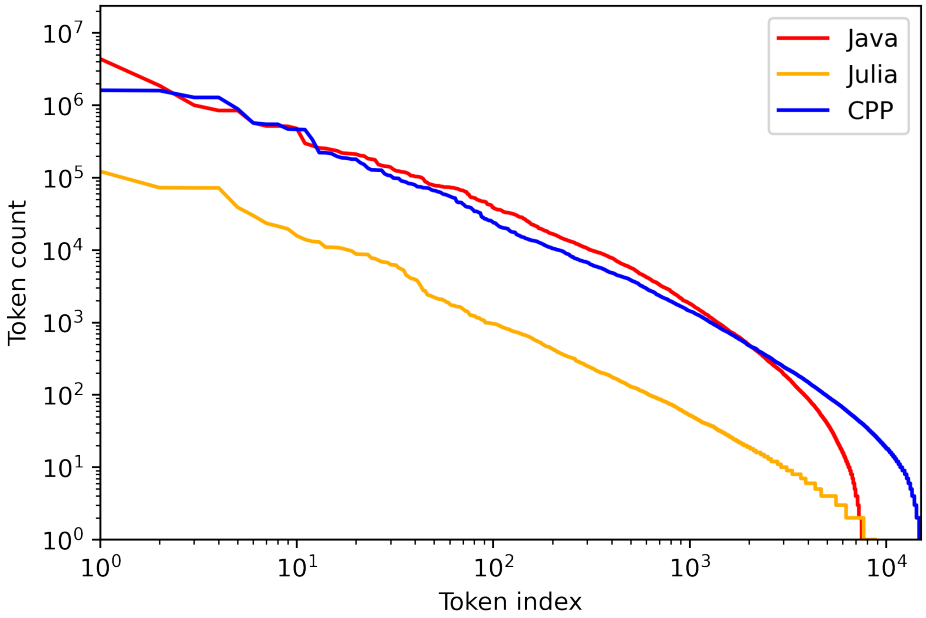


Figure 4.2: Verifying the distribution of the language tokens by plotting the occurrence vs token number on a logarithmic scale.

subsampled AST is around 12 jumps long.

The general idea behind the algorithm is to start at the target location and add parent nodes from the AST until you reach a height of $n/2$ (where n is the max hop distance). We then move one token to the left and right in the source code and add all parents that are not already in the subsampled AST. This procedure continues iteratively until the subsampled AST contains enough nodes. The pseudo-code for this algorithm is given in snippet 1.

In the case that this algorithm returns a set of disconnected subtrees, the eigenvalues of the Laplacian would have more zero value eigenvalues. This would effect the calculation of the positional encodings [30]. To fix this issue, after subsampling we connect all roots of subtrees to one another with an 'artificial' edge. This ensures the quality of the positional encoding.

4.4 Model settings

Due to the many interworking elements of this model, many hyperparameters need to be optimized to ensure good performance. However, the goal of this research is to determine the effect of using structural information in a multilingual setting, so if there is indeed a correlation then it should be evident for all (reasonable) parameters. Conducting a hyperparameter search by ourselves is prohibitively expensive, so we will base our choice of hyperparameters on previous research rather than conducting an expensive search ourselves.

Algorithm 1 Pseudo-code for sub-sampling the ASTs

```
n_t ← sub-tree with root of node with target token
T ← empty tree
T_in ← too large tree
N ← max number of nodes
n ← n_t
while h < H do
    predecessor ← n.ASTParent
    h ← h + 1
    if |predecessor| < N then
        T ← predecessor
    else
        continue
    end if
end while
n_l ← n_t sub-tree with root of node to the left (order edge)
n_r ← n_t sub-tree with root of node to the right (order edge)
while |T| < N do
    n_l ← n_l.getLeft
    n_r ← n_r.getRight
    T_p
    h ← 0
    if n_l exists then
        T_t ← n_l
        while h < H do
            T_p ← T_t
            T_t ← T_t.ASTparent
            h+ = 1
            if |T_t| + |T| > N then
                T.add(T_p)
            end if
        end while
    end if
    if n_r exists then
        T_t ← n_l
        while h < H do
            T_p ← T_t
            T_t ← T_t.ASTparent
            h+ = 1
            if |T_t| + |T| > N then
                T.add(T_p)
            end if
        end while
    end if
end while
```

Name	Value
Optimizer	RAdam [39]
weight decay	0.001
max learning rate	2×10^{-4}
Scheduler	Cosine
Laplacian positional encodings dimension	7
Masking fraction	15%
Max graph size	100

Table 4.4: General hyper parameters of all transformers on all tasks.

First, we will look at the more well-established LSTM, and give the hyper parameters for the embedding model, next we will look at the more novel graph transformer and give all hyper-parameters that were used in the experiments.

Data Processing hyper-parameters As previously mentioned, the transformer model requires all inputs to be the same size. We take the size of at most 100 possible nodes in the AST and a max height of 6 for the subsampling algorithms. The maximum height is based upon the findings of Uri Alon et al. [6] who found that a maximum length of 12 jumps in a path traced through the AST was beneficial for method naming of Javascript, while a length of up to 6 was beneficial for method naming of Java, as there was not a detrimental impact to performance, merely a diminishing return, we choose to set the max height of an AST to 6 for our experiments, this would give a max path of 12 in most cases.

LSTM hyper-parameters For an LSTM model, there are a few hyper-parameters that are needed. These hyper-parameters are the number of layers, the learning rate, and how many epochs the models are trained. As the LSTM model had very few problems overfitting/underfitting, we were able to run all experiments with a 3-layer LSTM and a learning rate of 0.01. The runs were all run for 75 epochs before being killed with a maximum sequence length of 100 tokens.

Transformer setup To train the transformer model we set the following hyper-parameters. We split the hyper parameters into three groups. The first group is the hyper-parameters used for the training setup that are shared across all tasks and languages, see table 4.4. The second group is the hyper-parameters that vary per task, see table 4.5. The final group of hyper-parameters is the hyper-parameters that describe the model architecture, see table 4.6.

It must be noted that these hyper-parameters are all used in the final model. It may be the case that some hyper-parameters are not used in the baseline, however, the ones mentioned are the same for the baseline and final model.

Task	Name	Value
Cloze (MLM)	Epochs	30
Cloze (MLM)	Batch size	32
Cloze (MLM)	Masking fraction	15%
Cloze (MLM)	Samples per epoch per language	10,000
Token prediction	Epochs	10
Token prediction	Batch size	32
Token prediction	Samples per epoch per language	30,000

Table 4.5: Task specific hyper-parameter setups used throughout all runs.

Name	Value
Layers	6
Attention heads	8
Hidden dimensions	512
Edges	2

Table 4.6: Model specific hyper-parameters used throughout all runs.

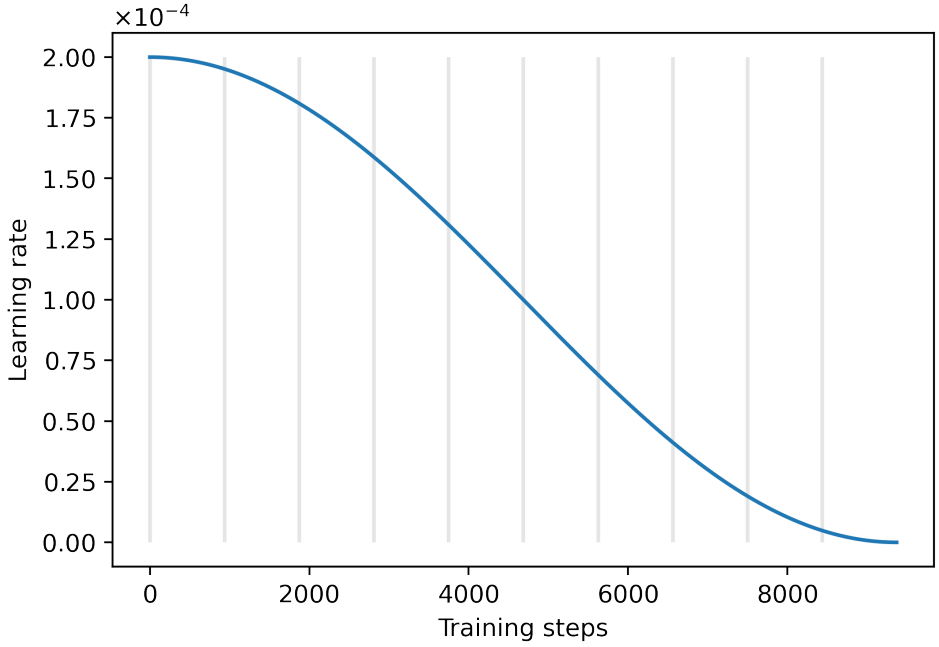


Figure 4.3: Progression of the Learning rate per update step for the setup given for the next token prediction task, the start of each epoch is given by the grey line.

Research Question	Languages		Training samples	
	Pre-training	fine-tuning	Pre-training	fine-tuning
RQ1	N/A	Java Julia CPP	N/A	Java: 10,000 Julia: 10,000 CPP: 10,000
	N/A	Java	N/A	Java: 10,000
	N/A	Julia	N/A	Julia: 10,000
	N/A	CPP	N/A	CPP: 10,000
RQ2	Java CPP	Julia	Java: 10,000 CPP: 10,000	Julia: 10,000
	Julia Java	CPP	Julia: 10,000 Java: 10,000	CPP: 10,000
	Julia CPP	Java	Julia: 10,000 CPP: 10,000	Java: 10,000

Table 4.7: Experimental setup to answer research questions RQ1, RQ2.

4.5 Runs

Finally, to answer our research questions from chapter 1.1 we will describe the setup of the runs and what we are trying to measure with it. We begin with the runs that we set up for the LSTM embeddings and link the runs to each specific research question. Then we will show the runs we conduct for the pre-training and fine-tuning of the transformer model.

To answer RQ1 we want to know whether the addition of other languages is at all beneficial to the training of the LSTM. To answer this we must run four experiments, the first experiment will be training the LSTM model with all languages consecutively, and the other three will be training the LSTM model separately on each language. To evaluate the performance of the models, we will only take the loss of Task 1, as this is the loss that shows how well the language model can model the language. For RQ2 we will be looking at the use of this model in the case where there exists a pre-trained model that has been fully trained on a set of languages, and we then fine-tune this model on a new language. To answer this question we will first train the model on each pair of languages made up of Java, Julia, and CPP, after training we will fine-tune the model on the one language left out of the pair. We will give an overview of all experimental runs and which research questions they answer in table 4.7.

Following the runs for the embedding model, we will explain the experiments we run for the transformer models. The main question we want to know the answer to is RQ3. To answer this question we need to train a set of models both with and without the addition of ASTs. We do this for all languages separately and all languages trained concurrently. To decide whether it is beneficial to train in the multilingual setting versus the monolingual setting we conduct the same runs but vary which languages we use. To answer both RQ3 and RQ4 we can use the same runs, with an overview of all runs given in 4.8. To ensure the reliability of the results we will run each three times and use the average results when evaluating the models.

Research Question	Task		Training samples		Augmented with ASTs
	Pre-training	Fine-tuning	Pre-training	Fine-tuning	
RQ3, RQ4	Cloze	Next token prediction	Java: 10,000 Julia: 10,000 CPP: 10,000	Java: 30,000 Julia: 30,000 CPP: 30,000	yes
	Cloze	Next token prediction	Java: 10,000 Julia: 10,000 CPP: 10,000	Java: 30,000 Julia: 30,000 CPP: 30,000	no
	Cloze	Next token prediction	Java: 10,000 Julia: 10,000 CPP: 10,000	Java: 30,000	yes
	Cloze	Next token prediction	Java: 10,000 Julia: 10,000 CPP: 10,000	Julia: 30,000	yes
	Cloze	Next token prediction	Java: 10,000 Julia: 10,000 CPP: 10,000	CPP: 30,000	yes
	Cloze	Next token prediction	Java: 10,000 Julia: 10,000 CPP: 10,000	Java: 30,000	no
	Cloze	Next token prediction	Java: 10,000 Julia: 10,000 CPP: 10,000	Julia: 30,000	no
	Cloze	Next token prediction	Java: 10,000 Julia: 10,000 CPP: 10,000	CPP: 30,000	no
	Cloze	Next token prediction	Java: 10,000	Java: 30,000	yes
	Cloze	Next token prediction	Julia: 10,000	Julia: 30,000	yes
	Cloze	Next token prediction	CPP: 10,000	CPP: 30,000	yes
	Cloze	Next token prediction	Java: 10,000	Java: 30,000	no
	Cloze	Next token prediction	Julia: 10,000	Julia: 30,000	no
	Cloze	Next token prediction	CPP: 10,000	CPP: 30,000	no

Table 4.8: Experimental setup to answer research questions RQ3, RQ4.

Use	Library	Version
Grammar generation and AST parsing	ANTLR	4
Model training	PyTorch	1.10.1+cu113
GPU communication	CUDA	11.3
AST generation	Java	11.0.14
General programming	Python	3.8
AST subsampling	NetworkX	2.5.1
Eigenvalue calculation (wrapper)	SciPy	0.19.1
Eigenvalue calculation (functions)	ARPACK	(via SciPy)

Table 4.9: Important libraries, programming languages and toolboxes used with version numbers.

4.6 Implementation

To make our results easier to reproduce, we will use this section to describe the hardware setup and libraries used to generate every step of this thesis. We will begin by describing the server used to run all code, followed by the libraries used for each step of the pipeline.

To begin, the server used to run all code used in this thesis was running Ubuntu 20.04.4 on an AMD Ryzen Threadripper 3990X with 128GB of RAM. The GPU available was an NVIDIA GeForce RTX 3080 with 10 GB of memory.

In table 4.9 we give an overview of the programming languages, libraries, and toolboxes used for the entire pipeline.

Chapter 5

Results

Now that the model has been described, we want to know how well it performs for each research question. We start by answering questions that refer to only the Embedding LSTM and then move on to questions that include the Graph Transformer. After answering the research questions, we wish to know what predictions may be expected during use. To this end, we will give several prediction examples for each language. Then, to understand the nature of the different performances we will look at the performances of the models on a per token basis, we will look at what types of tokens increase performance most by comparing the monolingual to the multilingual setting, and the baseline to the augmented setting. Finally, to ensure that the performance increases may indeed be attributed to the features evaluated when looking at the results we perform an ablation study. We look at the performance of the model with and without pre-training and with and without the pre-trained embeddings.

Throughout the discussion of the results, we will be using two metrics. To compare the performance of the LSTM language models and the Masked Language Models, we will be using the Cross-Entropy loss. For the Cross-Entropy loss we associate a lower value with a better model. For comparison of the final transformer models, we use the accuracy. The accuracy is a metric where a higher value is better. However, we use a slightly modified version. We use a metric of "Accuracy@5" this is the percentage of times that the correct value is in the top-5 most likely tokens predicted by the model. Furthermore, to ensure that the models are not predicting the most common labels for each prediction, we remove the 5 most commonly occurring tokens from the calculation.

5.1 Embedding LSTM

To understand the effects of multilingual training on the ability of the embedding LSTM to find a high-quality embedding we will be answering the research questions stated in section 1.1.

RQ1: Does training on multiple languages help to find higher quality embeddings? To answer this question we trained the model on each language separately, as well as on all

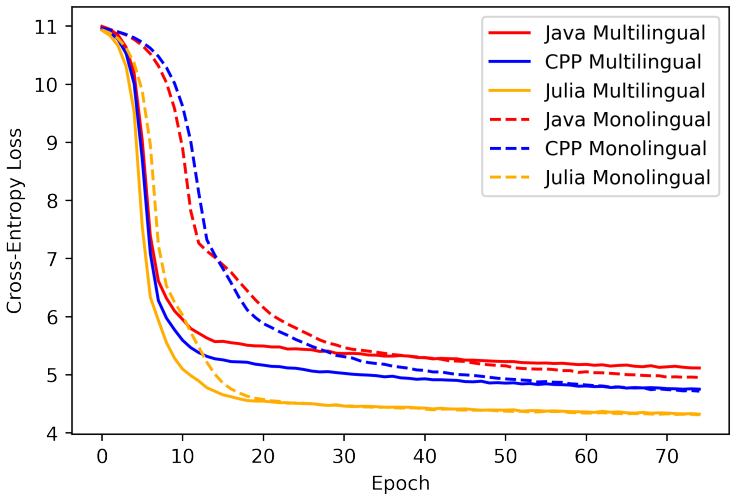
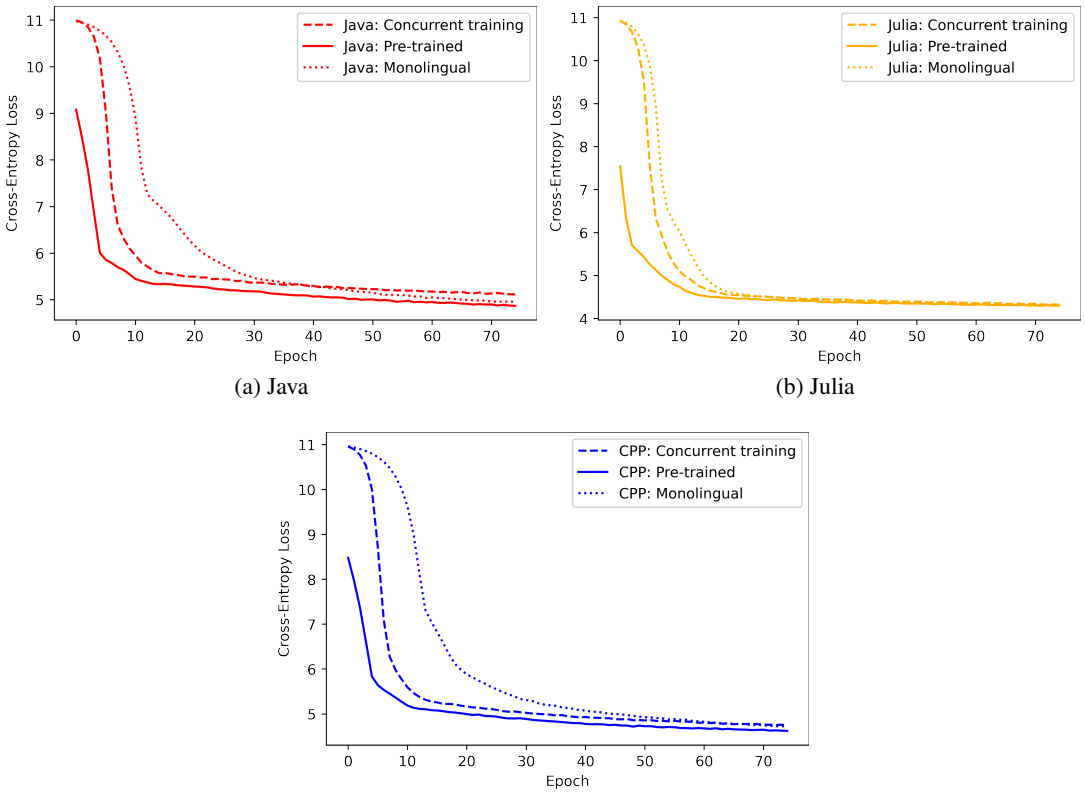


Figure 5.1: Concurrent and independent training of the LSTM model.

languages concurrently. The results of these runs are given in figure 5.1.

Looking at figure 5.1, we can notice two main trends from the data. Primarily we see a difference in the performance of the LSTM model in the different languages. The worst language is Java, followed by CPP and then Julia. This may seem counter-intuitive as it is the inverse order of the amount of data available to the languages. We attribute this difference to two potential causes. First, we notice a difference in the size of the vocabularies. This influences the score in the cross-entropy loss. Secondly, we look at the nature of the two languages, as LSTMs have difficulties with longer-range dependencies between languages, we notice that the language which has the least syntactic sugar, Julia, as well as being less verbose, is the best performing despite not having as many training samples. Finally, we also notice the trend that when the languages are trained together they require fewer passes over the data to achieve a similar result early on in training. However, towards the end, there is a slight performance benefit in the monolingual model compared to the multilingual method. We believe this may be due to the model needing to make trade-offs to create a good solution for three different languages.

To answer this research question we summarize it in three findings. First, the multilingual model benefits from all three languages at the start of training. As there are more weight updates in total, yet the number of weight updates per language remains the same, the performance benefit at the start of training can be attributed to knowledge gained by the network from all three languages at once. Secondly, in the long run, there is no performance benefit from learning in a multilingual setting when a good prediction is needed in each separate language. Finally, for LSTM models we see a trend that the verbosity of the language affects the overall performance, which backs up the papers that claim a major downside of LSTMs is their inability to model long-range dependencies [68].



(a) Java

(b) Julia

(c) CPP

Figure 5.2: Language model performance, comparison between concurrently trained and pre-trained models.

RQ2: Is pre-training on a set of languages beneficial when fine-tuning on a final target language? To investigate the scenario where a language model is learned for a new language when there already exists a language model for other languages, we pre-train an LSTM model on two languages initially. These pairs are Julia and CPP, Julia and Java, and Java and CPP. Then we fine-tune the model on the missing language, Java, CPP, and Julia respectively. We show the results of this training in figure 5.2. We also plotted the line of the concurrently trained language model for comparison sake.

From figure 5.2, we can see that there is an advantage to using a pre-trained model at two points in the training process. Primarily, the initial error of all three languages is far lower when pre-training versus training concurrently on multiple languages. This also hints that there is some transfer of knowledge between the languages as the early performance of the weights was better than the random initialization. Furthermore, towards the end of training, we can see that the overall performance of all models is better when fine-tuning on a final language.

5.2 Transformer Model

Now that we know what we need to create a good embedding, we can move on to using this embedding to help train the transformer model. The evaluation of the transformer model will be done in two parts, we are both interested in the performance of each language on the pre-training task, as well as how this will finally affect performance on the code prediction task. The final results of all experiments can be seen in table 5.1. We start by answering each research question about the pre-training task, where applicable, and then move on to the code prediction task.

RQ3: Does adding a structural component to the BERT architecture benefit the performance? Due to the imbalanced nature of the dataset, which follows a Zipfian distribution, we will evaluate the performance of the next token prediction task by removing the five most common tokens. This prevents the model from showing good performance while only predicting the most common tokens which gives a skewed view of the model's actual performance. For RQ3, we look at the effect of adding the structural information to the model. We compare the results of an identical model without a structural component to the performance of a model with a structural component. We do this in the monolingual setting, where the model is pre-trained on one language and fine-tuned on the same language, in the multilingual setting, where the model is pre-trained on all languages and fine-tuned on all languages concurrently, and finally in a mixed setting, where the model is pre-trained on all languages and then fine-tuned on each language separately. The performance we measure will be using the Cross-Entropy loss for the masked language model and the accuracy@5 [1] with the five most common tokens removed from the calculation. In order to be certain that the gathered results can be reproduced consistently, we ran each experiment 3 times and used the mean value for the results and 1 standard deviation from the mean as the uncertainty.

Looking at the graphs in figure 5.3, we can see that there is a definite benefit to adding a structural component to the input for the monolingual setting. All points in figure 5.3a, shown more clearly in figure 5.3c, show better performance (lower loss) when the structural component is included. In the multilingual setting, however, the results are not as conclusive. As shown in figure 5.3 we can see that the difference between the performance of all baseline runs compared to the structurally augmented runs is insignificant. The mean loss of the baseline runs was slightly lower than the augmented runs for CPP, Java, and equal for Julia. We believe that the cause for this may be linked to the differences in AST structure. As can be seen in section 3.1, for many similar expressions in different programming languages, the generated ASTs may look very different.

Following the performance on the pre-training task, we look at the performance on the final task, next token prediction. In figure 5.4, we see the performance of the model on the final task, using the metrics described previously. From both the graphs in figure 5.4 and the results from table 5.1 we see that the best performing model are the models augmented with an AST. However, when comparing the performances in just one of the three training

[1] The prediction is correct if the correct token is among the top 5 most likely tokens predicted by the model.

	Pre-training	Fine-tuning		
	Loss	Validation Accuracy@5 (%)	Test Accuracy@5 (%)	Unseen Accuracy@5 (%)
Mixed augmented	2.82 ± 0.15	**60.1 ± 1.0**	**58.6 ± 1.4**	**52.4 ± 1.5**
Mixed baseline	**2.70 ± 0.09**	56.7 ± 3.6	55.6 ± 3.3	49.8 ± 2.8
Multilingual augmented	2.82 ± 0.15	57.5 ± 2.8	56.0 ± 2.0	49.9 ± 1.6
Multilingual baseline	**2.70 ± 0.09**	56.3 ± 2.3	57.2 ± 2.0	50.9 ± 1.2
Monolingual augmented	2.72 ± 0.16	55.8 ± 6.5	54.0 ± 6.4	49.0 ± 4.7
Monolingual baseline	2.80 ± 0.16	58.6 ± 0.4	57.3 ± 0.6	51.3 ± 0.5

(a) Java

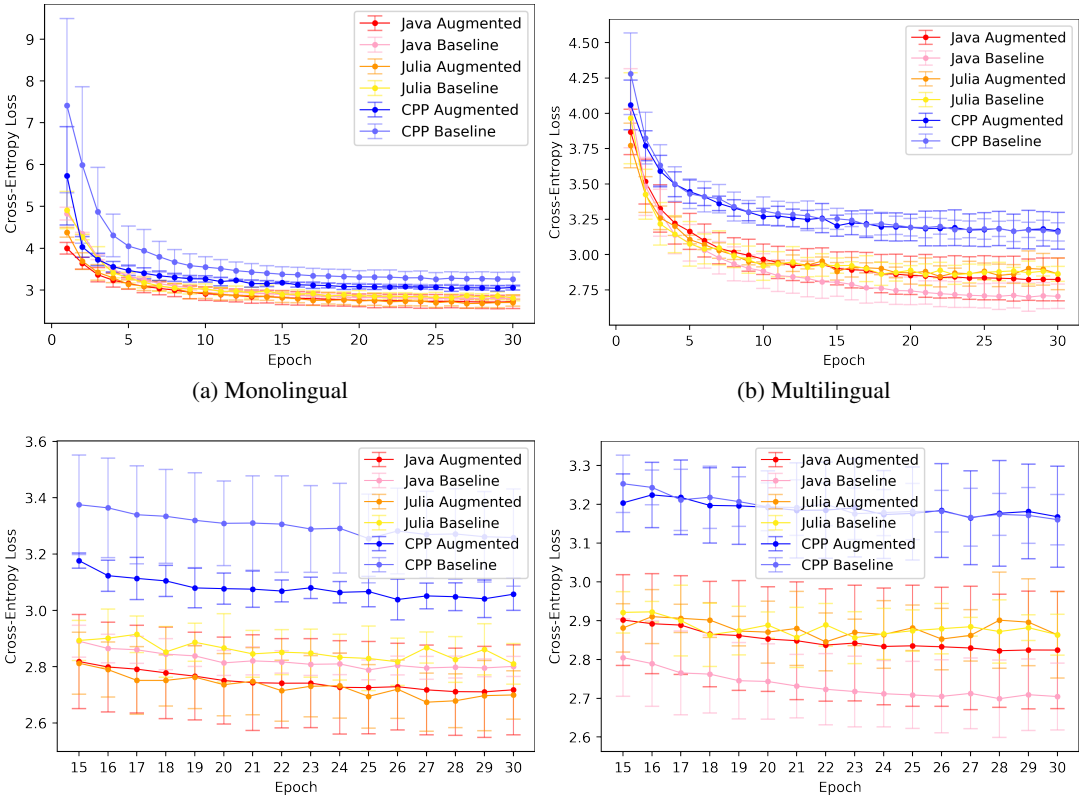
	Pre-training	Fine-tuning		
	Loss	Validation Accuracy@5 (%)	Test Accuracy@5 (%)	Unseen Accuracy@5 (%)
Mixed augmented	2.86 ± 0.11	51.2 ± 3.9	49.4 ± 2.8	44.4 ± 3.4
Mixed baseline	2.86 ± 0.05	52.3 ± 2.0	49.7 ± 2.5	44.5 ± 0.5
Multilingual augmented	2.86 ± 0.11	52.0 ± 1.9	50.1 ± 2.1	44.8 ± 2.2
Multilingual baseline	2.86 ± 0.05	49.7 ± 2.7	49.9 ± 2.8	44.9 ± 1.5
Monolingual augmented	**2.70 ± 0.09**	**55.4 ± 2.2**	**52.6 ± 2.8**	**47.5 ± 1.4**
Monolingual baseline	2.81 ± 0.07	53.0 ± 0.9	51.5 ± 1.3	43.8 ± 0.6

(b) Julia

	Pre-training	Fine-tuning		
	Loss	Validation Accuracy@5 (%)	Test Accuracy@5 (%)	Unseen Accuracy@5 (%)
Mixed augmented	3.17 ± 0.13	38.5 ± 5.4	39.2 ± 5.2	43.6 ± 4.9
Mixed baseline	3.16 ± 0.06	37.1 ± 1.0	38.9 ± 0.5	42.7 ± 1.1
Multilingual augmented	3.17 ± 0.13	**41.0 ± 3.3**	**42.1 ± 2.7**	**45.9 ± 2.5**
Multilingual baseline	3.16 ± 0.06	40.3 ± 1.5	41.8 ± 2.1	45.6 ± 1.9
Monolingual augmented	**3.06 ± 0.06**	35.8 ± 8.5	35.9 ± 7.9	39.9 ± 9.3
Monolingual baseline	3.26 ± 0.17	29.1 ± 12.4	30.2 ± 13.1	33.1 ± 14.8

(c) CPP

Table 5.1: Final results of the transformer model evaluated on a validation dataset, test dataset, and 10 unseen projects from Github.



(a) Monolingual

(b) Multilingual

(c) Monolingual last 15 epochs

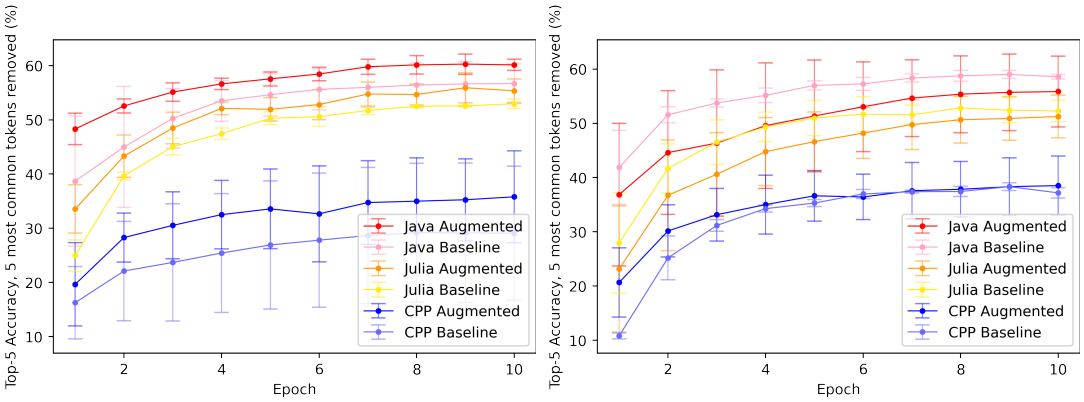
(d) Multilingual last 15 epochs

Figure 5.3: Performance of the MLM model, we show both the full training as well as the last 15 epochs for clarity.

settings, we see that in some cases the baseline outperforms the augmented model. This is shown in figure 5.4b, where the baseline Java and Julia models outperform the augmented model, and both CPP models have the same performance. Interestingly, we do note that the variance between runs in the baseline models is substantially smaller than the variance between runs of the augmented model.

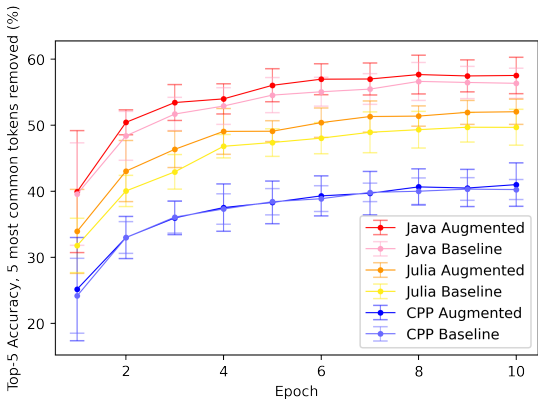
The trend of having a smaller variance between the results of different runs in the baseline models compared to the augmented models is shown in most situations, except for the monolingual CPP model. We believe this shows that the addition of ASTs to the input of the model increases the complexity of the task being solved. This hints that the augmented models would benefit more from either using a larger model or training on more data compared to the baseline models.

Finally, Looking at the performance of the monolingual CPP model, we see that the variance between runs is substantially higher than in all other models. For both the baseline as well as the augmented model the variance between runs is at least twice as high as all other variances on the unseen dataset and close to twice as high for all other data sets. In combination with this, the CPP model also had worse performance compared to the other



(a) Monolingual

(b) Multilingual pre-training, monolingual fine-tuning

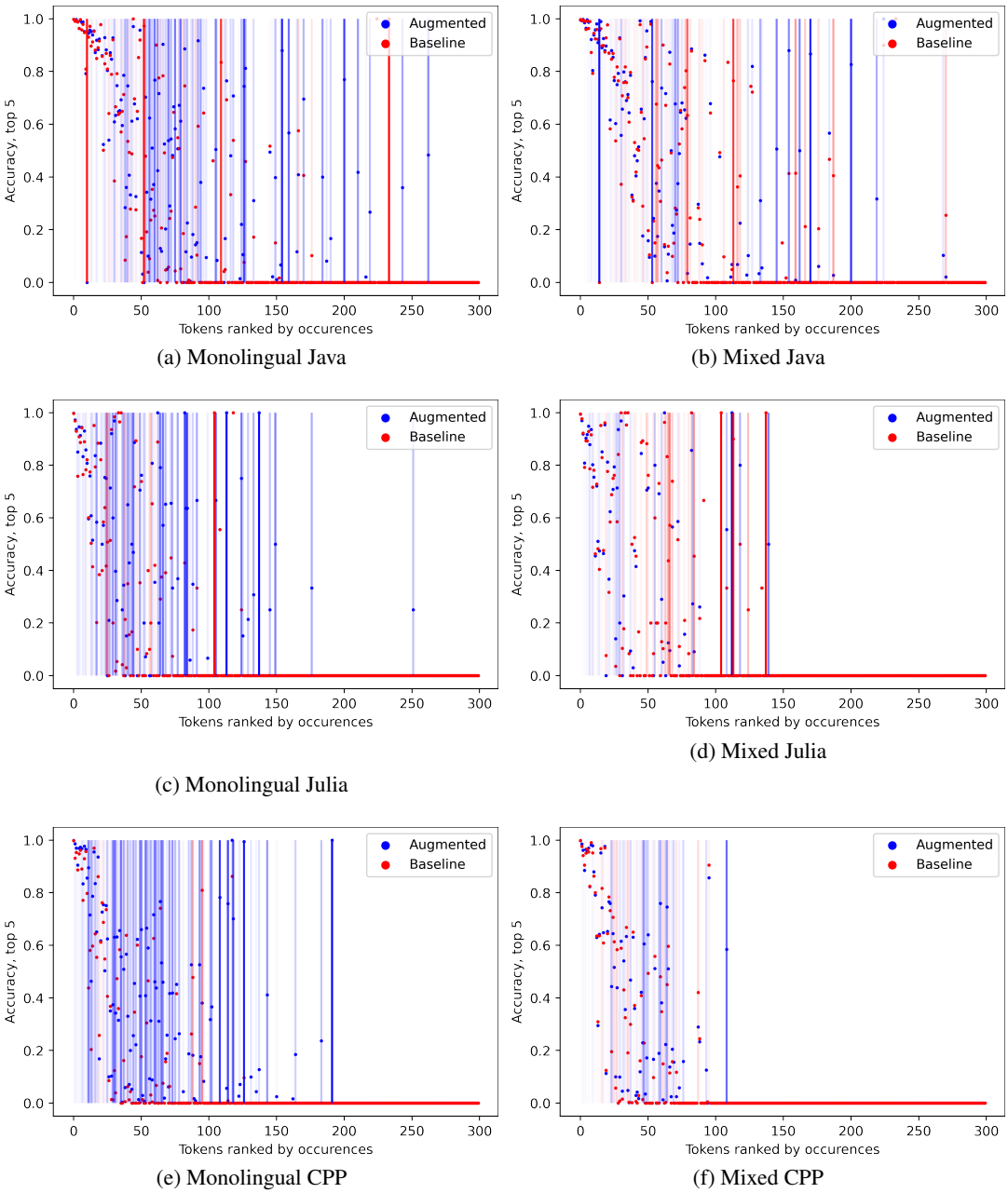


(c) Multilingual

Figure 5.4: Performance final model.

two languages. Further anomalies that we see in the data for the CPP models are that it is the only language where the accuracy of the unseen dataset is always higher than the accuracy on the test and validation dataset. We believe that this shows that the model size used during the experiments was close to the smallest model that would not underfit the data, for Java and Julia. The increase in performance on the unseen and test set, compared to the validation set also supports the theory that the model has over-generalized the data and underfit. The behavior of the CPP model, however, also shows that the use of ASTs, does help to increase performance in situations where there are not enough resources to fully train the baseline model, as in all settings the augmented model outperformed its respective baseline.

Now that we have seen a benefit to using the structural component when working on the next token prediction task, we will take a more thorough look at where the performance comes from. To give a quick overview of the performance we plot the performance on a



(a) Monolingual Java

(b) Mixed Java

(c) Monolingual Julia

(d) Mixed Julia

(e) Monolingual CPP

(f) Mixed CPP

Figure 5.5: Accuracy per token, tokens are ranked in decreasing order of occurrence, the difference in performance is given per token by shading the slice of the plot belonging to it.

per token basis comparing the performance of the two models on the first 300 tokens in figure 5.5. To clearly show the difference in performance we highlight the slice of the plot belonging to each token in its respective color, setting the opacity to how strongly each token outperforms the other. We will compare the performance of the Monolingual and the mixed models as these were the best performing types of models from table 5.1.

The first apparent feature of the plots is that most of the performance comes from the few most common data points. we see this by the distribution of dots on the graph as the most common tokens are towards the left, where we can see the accuracy of the model is highest. This also shows us that there is a correlation between the frequency a token appears and how well the model can predict this token. Furthermore, we can see, especially in figure 5.5c, that the difference in performance between the augmented models is not only from their performance on the very common tokens, however, the augmented models are able to predict less frequent tokens correctly.

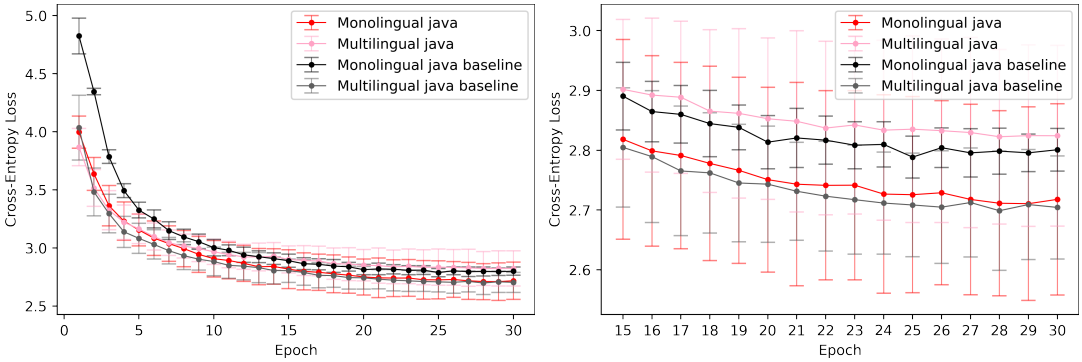
Finally, we look at all the graphs together. We see that although the smallest vocabulary of the languages used is 10,328 (see table 4.1) the best model has all of its performance in the top 300 tokens. This shows us that it would be beneficial to the performance of the models would be trained for longer, on a larger dataset, as neural models' performance on rare words has been shown to require a lot of data [14].

RQ4: Does training on multiple languages benefit the model? Similarly to RQ3, we will answer this question in two steps. First, we will look at the effect of multilingual training on the performance of the masked language model and finally, on the performance of the next token prediction task. Once again in order to be certain that the gathered results can be reproduced consistently, we ran each experiment 3 times and used the mean value for the results and 1 standard deviation from the mean as the uncertainty.

Initially, we evaluate the performance of the model by training a monolingual MLM, (only one language) and a multilingual MLM (trained on Java, Julia, and CPP) which gave us four separate models (three monolingual and one multilingual). The validation performance during the training of these models is given in figure 5.6. For final performance we look at the results in 5.1.

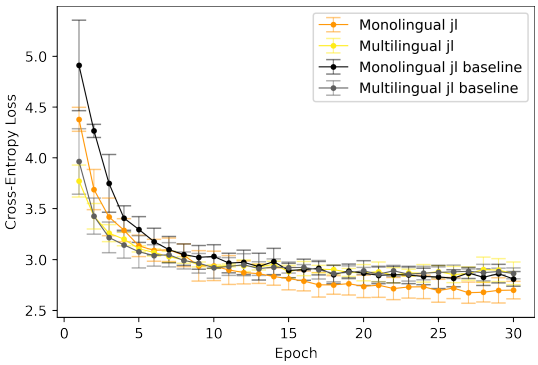
From the graphs in figure 5.6, we can see there is a strong trend to favor the monolingual training setting over the multilingual setting. In the interest of clarity, we have also plotted the baseline models in both the monolingual and multilingual settings. We can see that there is a definite drawback to training in the multilingual settings when augmenting the input with ASTs. However, for the baseline in CPP and Java, we can see that there is a benefit to training in the multilingual setting. The Julia baselines are too close to favoring one over the other. The multilingual baseline model evaluated on Java even outperforms the augmented model. This behavior leads us to believe that although the ASTs are generated using the same node types, there is a too large difference between the shapes of the ASTs between each language to benefit more from transfer learning with ASTs than without.

Following the model's performance on the pre-training task, we will now analyze the performance of the model on the fine-tuning task. For the fine-tuning task we trained eight models, the monolingual models were pre-trained on the same languages used for fine-tuning. The mixed models were fine-tuned in each language separately but pre-trained on

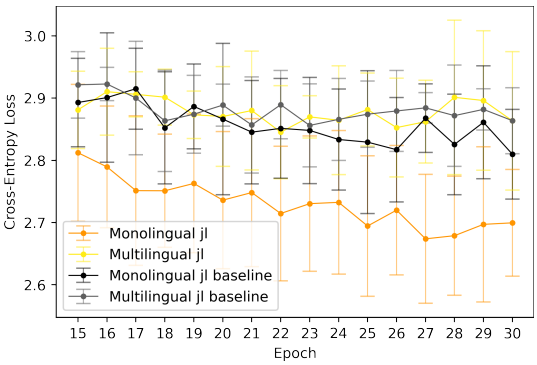


(a) Java, full pre-training

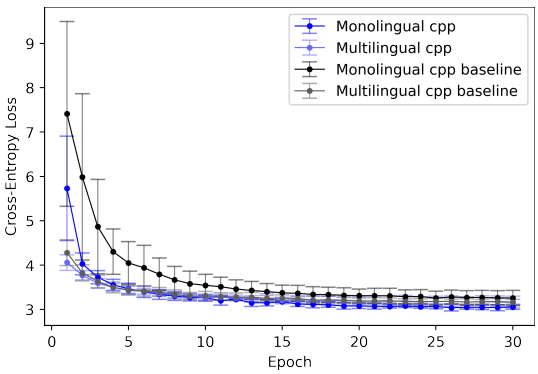
(b) Java, last 15 epochs



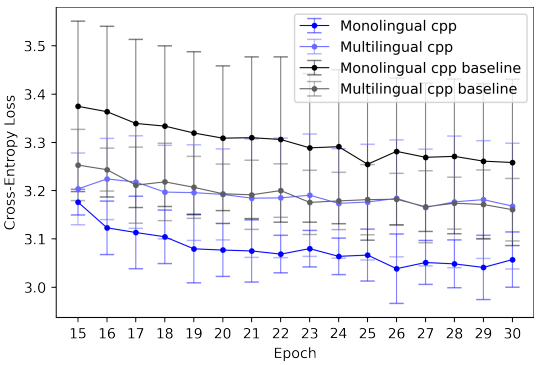
(c) Julia, full pre-training



(d) Julia, last 15 epochs

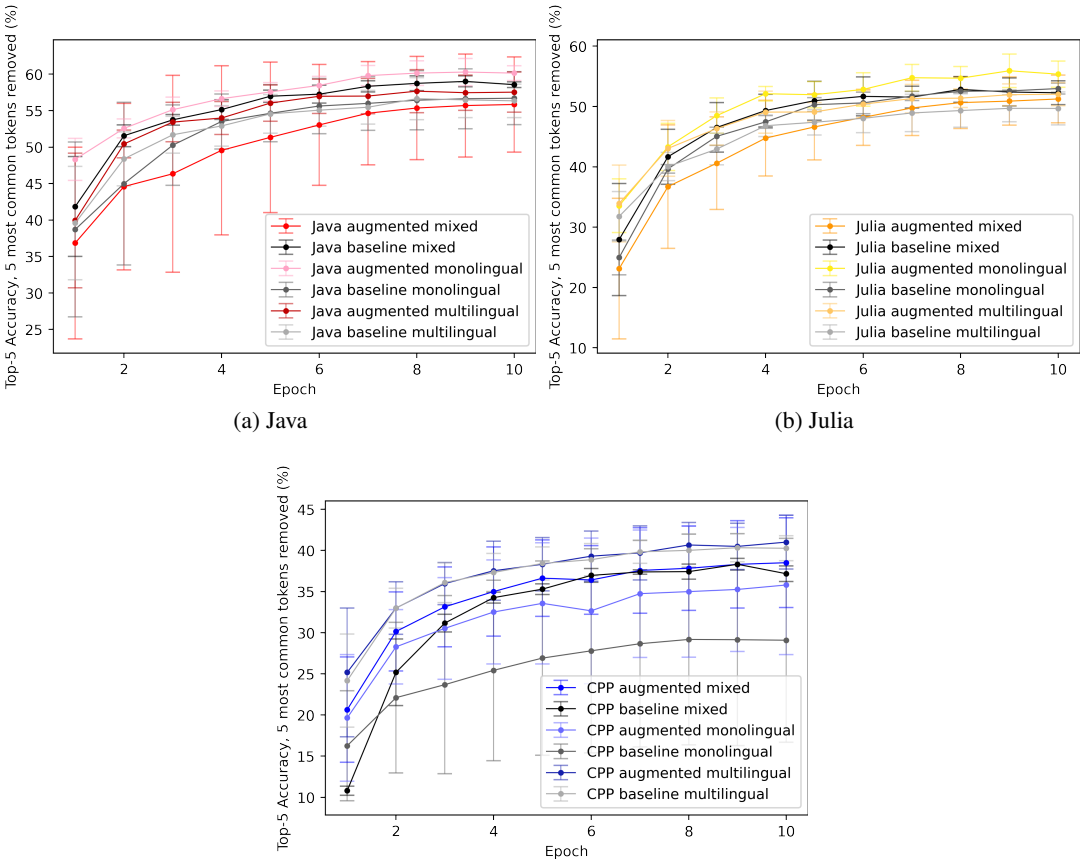


(e) CPP, full pre-training



(f) CPP, last 15 epochs

Figure 5.6: The performance of the MLM, comparing monolingual to multilingual models. We use the Cross Entropy Loss as metric, lower is better.



(a) Java

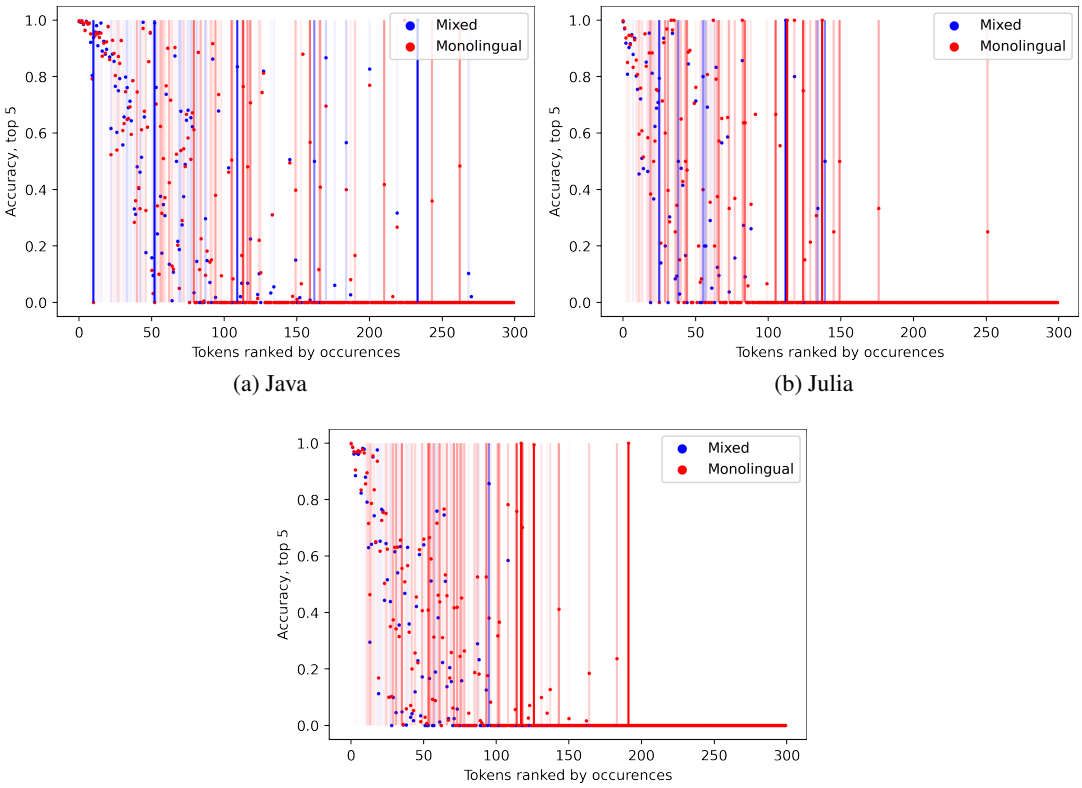
(b) Julia

(c) CPP

Figure 5.7: Top-5 accuracy of all models, comparing the monolingual and multilingual settings. The 5 most common tokens have been removed.

all three languages concurrently, and finally, the multilingual models were pre-trained on all three languages concurrently and fine-tuned on all three languages concurrently. The results of these runs for the test and unseen data can be seen in table 5.1, the performance on top-5 accuracy with the five most common tokens removed during training is given in figure 5.7.

From the results gathered in figure 5.7, we see an interesting trend. In the case of fine-tuning from a multilingual or a monolingual model, we see that Java, the language that had the worst performance in the multilingual setting during pre-training, performs best out of all Java models when fine-tuning on only java (mixed setting). This seems to show that the Loss of the pre-training model is not directly linked to the performance on downstream tasks. Furthermore, we do see that when comparing the variance between different runs for the baseline models. The monolingual models of Java and Julia have a very low variance for the accuracies on all datasets, which increases significantly when moving into the multilingual setting. This trend seems to be reversed in the augmented model where CPP and Julia monolingual models have a very high variance between runs,



(a) Java

(b) Julia

(c) CPP

Figure 5.8: Comparing token-wise performance between the monolingual and multilingual setting, the difference in performance is given by the shade of the background.

yet the multilingual models reduce this variance.

Finally, similar to RQ3, we will investigate where the difference in performance comes from by plotting the per token accuracy of the most common tokens to compare performance between the monolingual and multilingual settings. We show the plots for this in figure 5.8.

From the plots in figure 5.8, we can see that similar to the answer to RQ3, the main difference in performance is explained by an increase in performance in only a few tokens. The interesting feature of the comparison of the models in 5.8 compared to 5.5 is that for Java we see that where when comparing the augmented model to the baseline version there was a cutoff after which the baseline model could no longer predict tokens correctly. In this case, the graph shows that the difference is not in the number of tokens the model can predict, as this is quite similar for both models, however the tokens that the mixed model predicts have a higher accuracy than the monolingual model. The Julia models show the same trend but to a lesser degree, with a few well-performing tokens for the monolingual model at the end. However, the CPP models once again show that the monolingual model has a higher capacity of predicting less common tokens decently compared to the mixed model.

5.3 Ablation Study

To fully grasp which areas of the proposed model are responsible for the performance we will conduct experiments to understand contributions that are not directly addressed by the research questions. As we are running all experiments in the results chapter with a pre-training step and pre-trained embeddings, we will rerun the best-performing experiments. First, we remove the pre-trained embedding layer and then the cloze pre-training task. Similar to the results gathered to answer the research questions, we run each experiment 3 times, and take the mean results as a data-point, the uncertainty we use is 1 standard deviation from the mean.

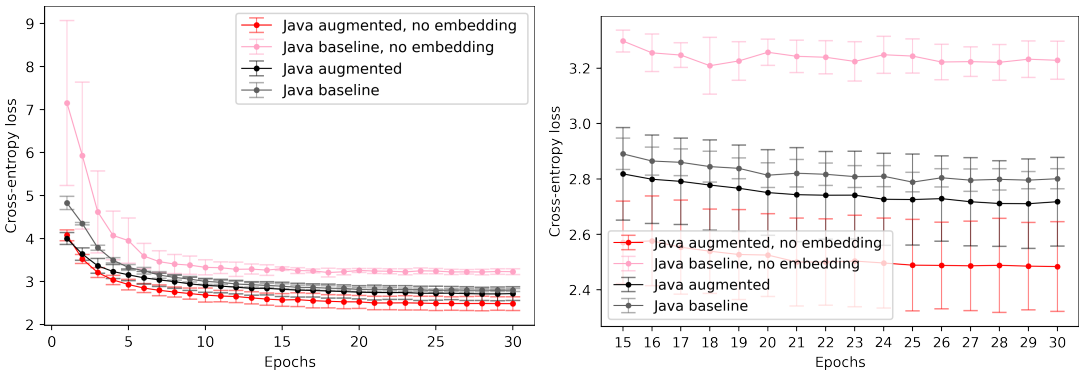
No pre-trained embeddings The first experiment we run is to evaluate whether the pre-trained embeddings learned by the LSTM were necessary. We compare the runs from section 5.2 with two new runs, running the baseline model without pre-trained embeddings and also running the augmented model without pre-trained embedding. We pre-train in both the monolingual and multilingual setting and later fine-tune in the monolingual and mixed setting. We chose the mixed setting over the multilingual setting as it delivered better results during the main investigation. The results for the monolingual pre-training can be seen in figure 5.9, and for the multilingual pre-training, we give the results in figure 5.10.

From figure 5.9 we see an interesting trend developing between the performance difference in the augmented and baseline models. From the Java plots we see that the Java baseline has the largest benefit to using pre-trained embeddings, however, the augmented model performs worse with the pre-trained embeddings. We see the same pattern in the Julia plot where the augmented model without pre-trained embeddings had the lowest loss, this time the baseline models however had no improvement with the embeddings. Finally looking at the CPP model we once again see that the augmented model performs worse with the pre-trained embeddings, and the baseline model performs better with the pre-trained embeddings.

Keeping the trend of decreased performance for the augmented model in mind we move over to the multilingual setting for the pre-training task. The results of the training can be seen in figure 5.10. The first thing we notice is that very clearly in the multilingual setting, across all languages, when we remove the pre-trained embeddings, the baseline model loses a lot of performance. However, unlike in the monolingual setting, the detrimental effect on the performance of the augmented model is not as evident. In the plots for Java, we see that there is a slight benefit to using the pre-trained embeddings for the augmented model. For Julia, we see that once again the pre-trained embeddings hurt performance in the augmented setting. Finally, in the CPP model, there is no difference between using the pre-trained embeddings and not using them for the augmented setting.

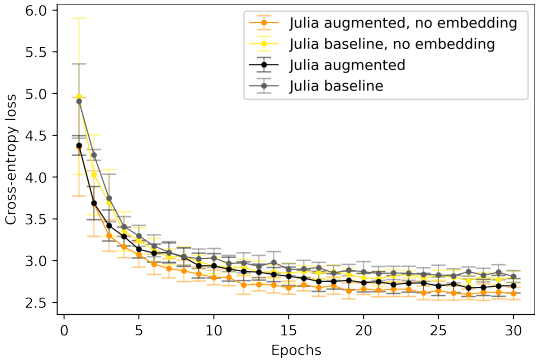
Moving on from the pre-training task, we look at the final performance of the model using both a pre-trained embedding and a randomly initialized embedding. We show the results of these runs evaluated on the validation set, test set, and the unseen dataset in table 5.3, and we show the training progress of these runs in figure 5.10

We will begin by looking at the mixed setting for all languages. These results are shown in the right column of figure 5.10. We see that in the mixed setting for Java, as seen with

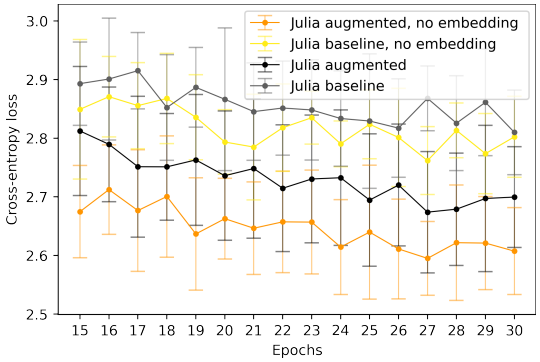


(a) Java full pre-training

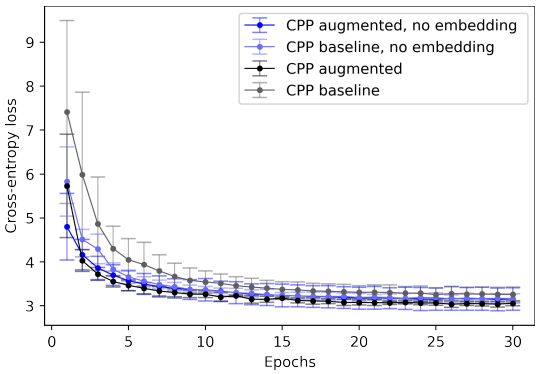
(b) Java last 15 epochs



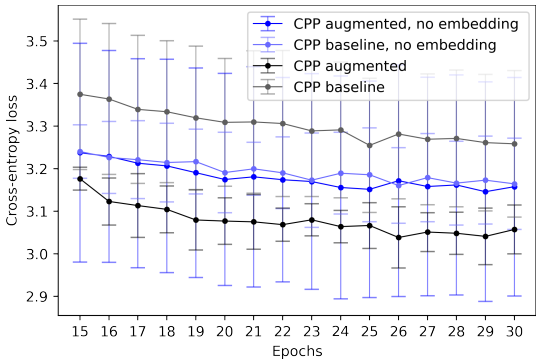
(c) Julia full pre-training



(d) Julia last 15 epochs

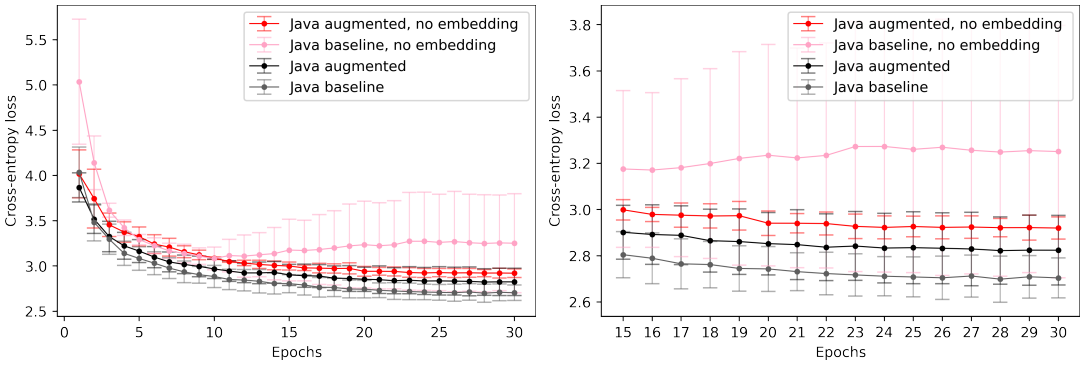


(e) CPP full pre-training



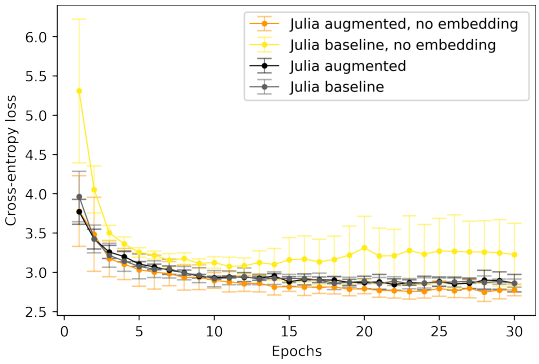
(f) CPP last 15 epochs

Figure 5.9: Model performance on the cloze task with vs without pre-trained embeddings, in the monolingual setting

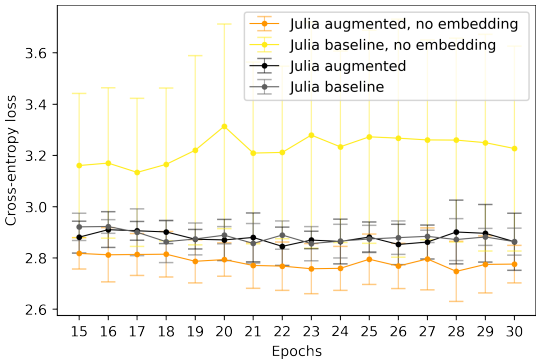


(a) Java full pre-training

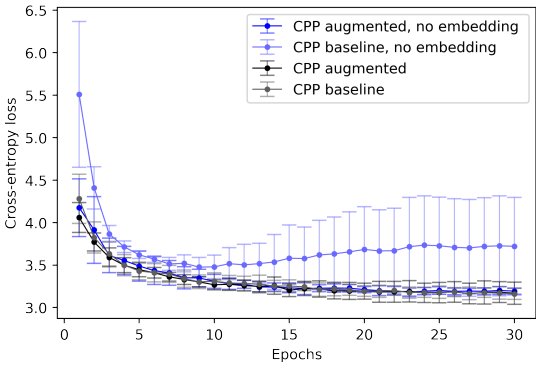
(b) Java last 15 epochs



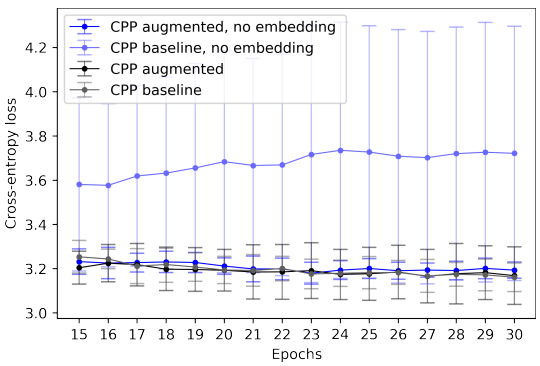
(c) Julia full pre-training



(d) Julia last 15 epochs



(e) CPP full pre-training



(f) CPP last 15 epochs

Figure 5.10: Model performance on the cloze task with vs without pre-trained embeddings, in the multilingual setting

	Fine-tuning		
	Validation Accuracy@5	Test Accuracy@5	Unseen Accuracy@5
Monolingual augmented pre-trained embedding	60.1 ± 1.0	58.6 ± 1.4	52.4 ± 1.5
Monolingual baseline pre-trained embedding	56.7 ± 3.6	55.6 ± 3.3	49.8 ± 2.8
Monolingual augmented no pre-trained embedding	51.2 ± 9.4	50.0 ± 9.0	46.0 ± 7.7
Monolingual baseline no pre-trained embedding	53.4 ± 2.0	51.7 ± 1.9	47.0 ± 1.3
Mixed augmented pre-trained embedding	55.8 ± 6.5	54.0 ± 6.4	49.0 ± 4.7
Mixed baseline pre-trained embedding	58.6 ± 0.4	57.3 ± 0.6	51.3 ± 0.5
Mixed augmented no pre-trained embedding	56.1 ± 5.0	54.8 ± 4.6	49.2 ± 3.6
Mixed baseline no pre-trained embedding	53.2 ± 2.8	52.1 ± 2.6	47.2 ± 2.0

(a) Java

	Fine-tuning		
	Validation Accuracy@5	Test Accuracy@5	Unseen Accuracy@5
Monolingual augmented pre-trained embedding	55.3 ± 2.2	52.6 ± 2.8	47.5 ± 1.4
Monolingual baseline pre-trained embedding	53.0 ± 0.9	51.5 ± 1.3	43.8 ± 0.6
Monolingual augmented no pre-trained embedding	55.9 ± 2.3	51.5 ± 1.6	47.6 ± 0.3
Monolingual baseline no pre-trained embedding	51.7 ± 4.6	49.6 ± 3.7	42.8 ± 4.3
Mixed augmented pre-trained embedding	51.2 ± 3.9	49.4 ± 2.8	44.4 ± 3.4
Mixed baseline pre-trained embedding	52.3 ± 2.0	49.7 ± 2.5	44.5 ± 0.5
Mixed augmented no pre-trained embedding	54.9 ± 6.2	50.7 ± 3.3	46.7 ± 2.5
Mixed baseline no pre-trained embedding	43.6 ± 6.4	46.1 ± 2.4	40.0 ± 4.6

(b) Julia

Table 5.2: Ablation results (without pre-trained embeddings) of the next token prediction task for Java and Julia with and without pretraining, evaluated on a validation dataset, test dataset, and 10 unseen projects from Github

	Fine-tuning		
	Validation Accuracy@5	Test Accuracy@5	Unseen Accuracy@5
Monolingual augmented pre-trained embedding	35.8 ± 8.5	35.9 ± 7.9	39.9 ± 9.3
Monolingual baseline pre-trained embedding	29.1 ± 12.4	30.2 ± 13.1	33.1 ± 14.8
Monolingual augmented no pre-trained embedding	23.3 ± 8.4	23.8 ± 8.8	25.7 ± 9.2
Monolingual baseline no pre-trained embedding	23.3 ± 5.1	24.0 ± 5.1	26.1 ± 5.9
Mixed augmented pre-trained embedding	38.5 ± 5.4	39.2 ± 5.2	43.6 ± 4.9
Mixed baseline pre-trained embedding	37.2 ± 1.0	38.9 ± 0.5	42.7 ± 1.1
Mixed augmented no pre-trained embedding	40.4 ± 2.2	42.0 ± 2.4	46.1 ± 2.4
Mixed baseline no pre-trained embedding	39.0 ± 0.5	40.0 ± 0.7	43.9 ± 0.8

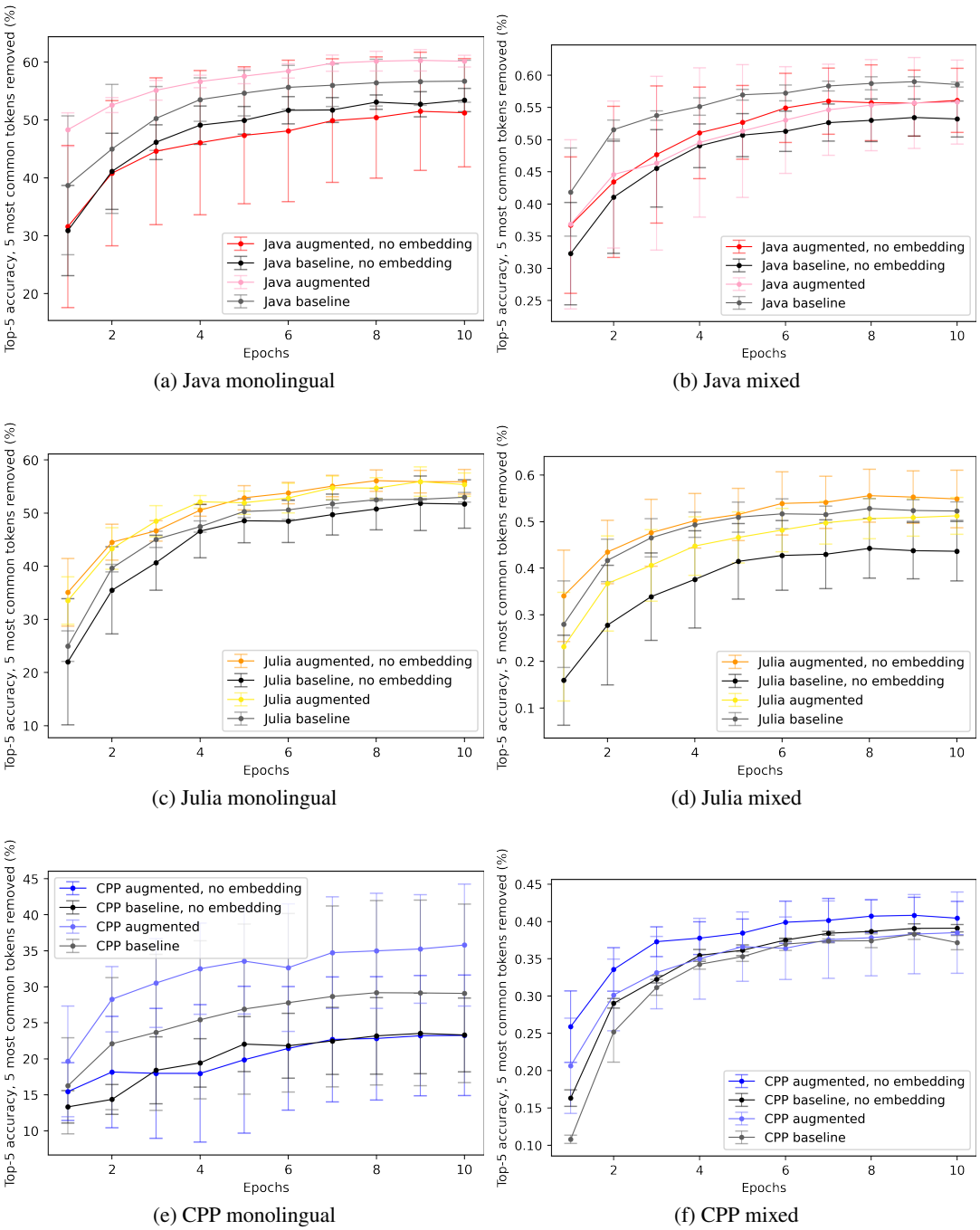
(c) CPP

Table 5.3: Ablation results (without pre-trained embeddings) of the next token prediction task for CPP with and without pretraining, evaluated on a validation dataset, test dataset, and 10 unseen projects from Github.

the pre-training, the baseline model increases in performance, while there is little effect on the augmented models when adding pre-trained embeddings. Julia shows the same trend, however, there is a larger difference between the performances of the augmented model, where the augmented model without pre-trained embeddings performs better. Finally, for CPP we see that in the mixed settings there is very little difference in performance for the baseline models, however, the augmented model, without pre-trained embeddings, seems to again perform the best.

Next, we look at the monolingual setting, this is shown in the left column of figure 5.10. Contrary to the results found in the mixed setting, the Java models with pre-trained embeddings perform better than the models without, for the augmented and baseline runs. In the case of Julia however there is very little difference in using pre-trained embeddings, with a very slight advantage going to using pre-trained embeddings for the baseline, however, the augmented models have the same performance. In CPP we see the largest benefit to using the pre-trained embeddings out of all languages in the monolingual setting. We see that the augmented model with pre-trained embeddings outperforms the augmented model without pre-trained embeddings, and the same goes for the baseline model, to a slightly lesser extent.

Finally, we look at table 5.3, here we will look at the models' ability to generalize to other datasets, as well as the consistency by looking at the variance between runs. Starting



(a) Java monolingual

(b) Java mixed

(c) Julia monolingual

(d) Julia mixed

(e) CPP monolingual

(f) CPP mixed

Figure 5.10: Performance of the models on the final task, comparing pre-trained embeddings to randomly initialized embeddings, in the monolingual and mixed setting

with Java we see that the results are similar in the models' ability to generalize, however, we do see that when comparing the performance of the monolingual augmented model with and without pre-trained embeddings, we see that the pre-trained embeddings helped the model train more consistently, reducing the standard deviation between runs from 9.4% to 1.0% on the validation set, and similar results on the other datasets. This same trend can be seen in the mixed baseline setting for Julia, the standard deviation of the runs drops from 6.4% on the validation set to 2.0% when including pre-trained embeddings, also on the unseen set the standard deviation drops from 4.6% to 0.5%. Contrary to the previous results, we see that although including pre-trained embeddings in the training process for CPP raised the performance, we see that in many runs the standard deviation of the accuracy drops when not using pre-trained embeddings.

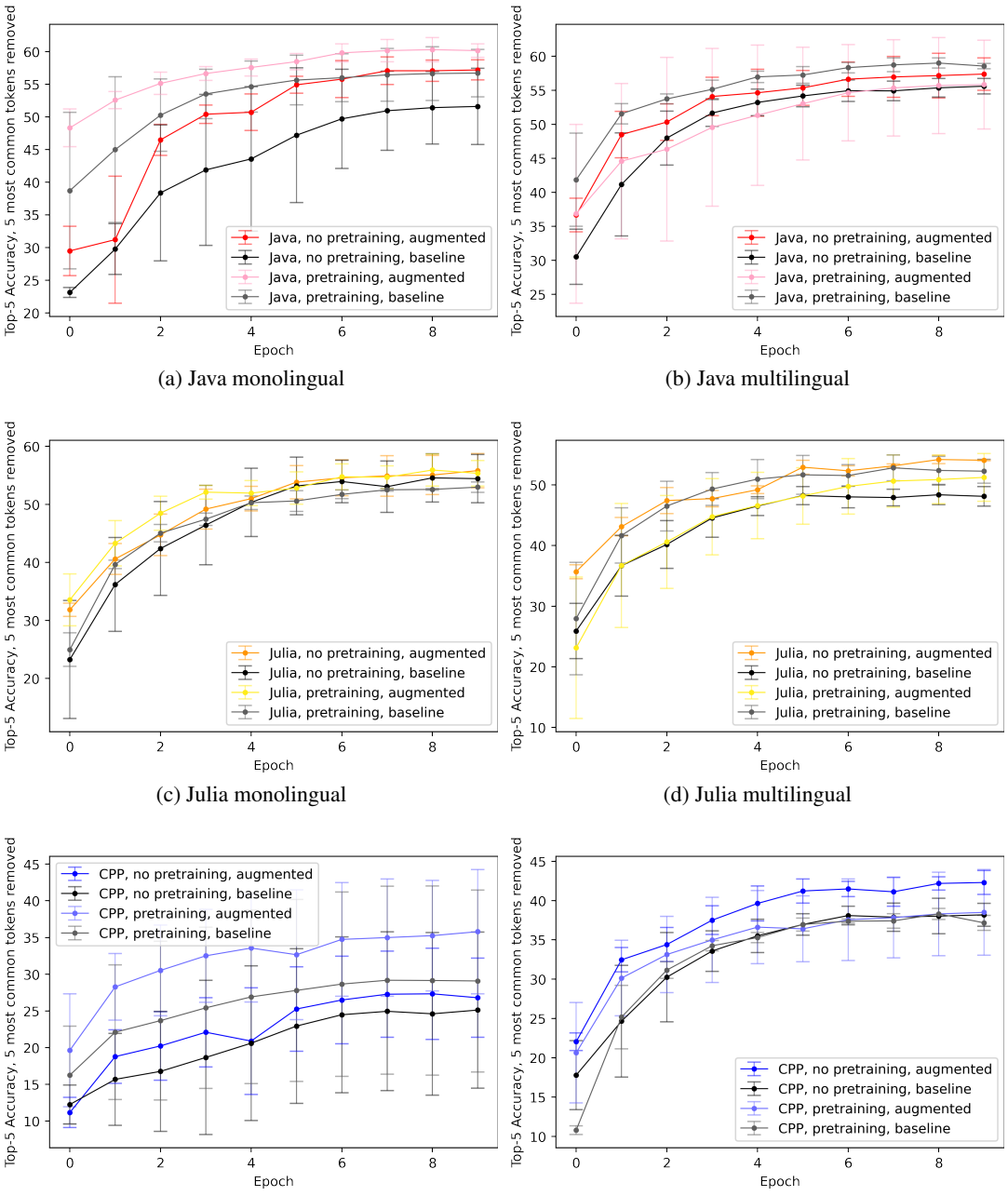
No pre-training task Following up on the need to investigate the utility of the pre-trained embeddings, we now focus on the need to pre-train the model on the cloze task as was done in the original BERT paper [18]. For this experiment, we will use the results from chapter 5.2 once more and compare them to the same runs, only without pre-training beforehand. The results of these runs are given in figure 5.11. The evaluation of the runs on the same datasets as in 5.2 are given in table 5.5. We use the accuracy of the correct token being in the top 5 predictions with the 5 most common tokens removed as a metric. The error used is one standard deviation of the trials gathered by running each model 3 times.

To evaluate the performance of the pre-training task, we will be looking at the results gathered on a per language basis. An overview of all results is shown in figure 5.11 and in table 5.5. We compare the performances of both the augmented and baseline models in the monolingual as well as the multilingual setting.

We begin by looking at Java, from the graphs in figure 5.11 we see that in the monolingual setting there is a benefit to pre-training in both the augmented as well as the baseline models, the benefit being slightly larger for the baseline model. Moving to the multilingual setting, we see that while the baseline model still benefited from pre-training, the augmented model was worse.

Following up with Julia, we see that now there is no benefit to using pretraining in the monolingual setting for either the augmented model or the baseline model. In the multilingual setting, however, there is a difference in performance. Similar to Java the augmented model without pre-training performed better in the multilingual setting yet the baseline model still benefited from pre-training.

Finally, for CPP, we once again see in the monolingual setting that there is a benefit to pre-training in the monolingual setting for both the augmented and the baseline model. Looking at the multilingual setting, we see that while the baseline models no longer saw an increase in performance from pre-training, the augmented model still had a substantial decrease in performance due to the pretraining.



(a) Java monolingual

(b) Java multilingual

(c) Julia monolingual

(d) Julia multilingual

(e) CPP monolingual

(f) CPP multilingual

Figure 5.11: Performance on the next token prediction task, comparing the performance with and without pre-training

	Fine-tuning		
	Validation Accuracy@5	Test Accuracy@5	Unseen Accuracy@5
Monolingual augmented pre-trained	60.1 ± 1.0	58.6 ± 1.4	52.4 ± 1.5
Monolingual baseline pre-trained	56.7 ± 3.6	55.6 ± 3.3	49.8 ± 2.8
Monolingual augmented no pre-training	57.2 ± 1.5	55.9 ± 1.6	50.6 ± 1.2
Monolingual baseline no pre-training	51.6 ± 5.8	50.7 ± 4.9	45.8 ± 3.6
Multilingual augmented pre-trained	57.5 ± 2.8	55.9 ± 2.0	49.9 ± 1.6
Multilingual baseline pre-trained	56.3 ± 2.3	57.2 ± 2.0	50.9 ± 1.2
Multilingual augmented no pre-training	57.4 ± 2.4	55.7 ± 2.8	49.4 ± 2.5
Multilingual baseline no pre-training	55.6 ± 1.1	54.4 ± 1.6	48.3 ± 1.2

(a) Java

	Fine-tuning		
	Validation Accuracy@5	Test Accuracy@5	Unseen Accuracy@5
Monolingual augmented pre-trained	55.3 ± 2.2	52.6 ± 2.8	47.5 ± 1.4
Monolingual baseline pre-trained	53.0 ± 0.9	51.5 ± 1.3	43.8 ± 0.6
Monolingual augmented no pre-training	55.8 ± 3.0	53.9 ± 1.6	46.6 ± 1.5
Monolingual baseline no pre-training	54.4 ± 4.2	53.0 ± 3.5	44.8 ± 2.5
Multilingual augmented pre-trained	52.0 ± 1.9	50.1 ± 2.1	44.8 ± 2.2
Multilingual baseline pre-trained	49.7 ± 2.7	49.9 ± 2.8	44.9 ± 1.5
Multilingual augmented no pre-training	54.1 ± 0.1	53.5 ± 0.8	46.8 ± 0.3
Multilingual baseline no pre-training	48.1 ± 1.6	47.6 ± 0.5	40.1 ± 0.7

(b) Julia

Table 5.4: Results of the next token prediction task for Java and Julia with and without pretraining, evaluated on a validation dataset, test dataset, and 10 unseen projects from Github.

	Fine-tuning		
	Validation Accuracy@5	Test Accuracy@5	Unseen Accuracy@5
Monolingual augmented pre-trained	35.8 ± 8.5	35.9 ± 7.9	39.9 ± 9.3
Monolingual baseline pre-trained	29.1 ± 12.4	30.2 ± 13.1	33.1 ± 14.8
Monolingual augmented no pre-training	26.8 ± 5.4	27.2 ± 5.9	32.0 ± 6.0
Monolingual baseline no pre-training	25.1 ± 10.6	25.7 ± 10.9	28.9 ± 12.6
Multilingual augmented pre-trained	41.0 ± 3.3	42.1 ± 2.7	45.9 ± 2.9
Multilingual baseline pre-trained	40.3 ± 1.5	41.8 ± 2.1	45.6 ± 1.9
Multilingual augmented no pre-training	42.3 ± 1.5	43.5 ± 1.1	47.3 ± 1.2
Multilingual baseline no pre-training	38.2 ± 1.5	39.7 ± 1.9	43.9 ± 1.9

(c) CPP

Table 5.5: Results of the next token prediction task for CPP with and without pretraining, evaluated on a validation dataset, test dataset, and 10 unseen projects from Github.

From these results, we see that the augmented model does not benefit from pretraining in the multilingual setting, which leads us to believe that the shape of the ASTs may change too much across languages and across tasks for there to be any beneficial carryover. However, as we saw from the CPP model in section 5.2 we believe that the models being used are at the smallest size possible to show any good results, if using larger models such as in the original BERT paper (110 million parameters in BERT, vs 40 million for this work) [18], we may see more beneficial behavior.

```
function add_zero_counts!(o::Mosaic)
    v = value(o)
    akeys = [k[1] for k in keys(v)]
    bkeys = [k[!<loc>
    for ai in akeys, bi in bkeys
        get!(v, (ai, bi), 0)
    end
end
```

Target : NUM

Token	certainty
NUM	35.7
UNK	17.2
g	6.0
i	2.1
m	1.8
a	1.7
p	1.7
t	1.6
:	1.3
[	1.3

Example 5.1: Token prediction for incomplete list indexing in Julia

```
function add_zero_counts!(o::Mosaic)
    v = value(o)
    akeys = [k[1] for k in keys(v)]
    bkeys = [k[2] !<loc>
    for ai in akeys, bi in bkeys
        get!(v, (ai, bi), 0)
    end
end
```

Target : for

Token	certainty
,	75.2
==	3.8
]	3.1
)	2.9
UNK	2.0
;	1.6
EOP	1.5
.	1.3
+	1.0
=>	0.8

Example 5.2: Token prediction for incomplete list comprehension in Julia

5.4 Qualitative analysis

Following the quantitative analysis we will evaluate the performance of the model on a number of randomly sampled files from the validation set. We do this to evaluate where the strengths and weaknesses of the model lie and to gain some insight into the predictions.

For the qualitative evaluation we will first look at the performance of the model on Julia, followed by Java and finally CPP.

Julia In Julia there are a set of common construct such as lists, list comprehensions, and function calls. To get a good idea of expected performance we will look at an example of a list comprehension, a list access, a function name, and a function call.

Starting with the list access in example 5.1 we can see that in this case, the model is very certain that it should be a number, however, it is followed by the unknown token. Both these tokens are logical completions at this location, followed by common index variables such as 'i', 'a', 'm', and ':', all of these suggested tokens could be a correct prediction, however the final suggestion [is the only invalid suggestion as it would result in a syntax

```
function add_zero_counts!(o::Mosaic)
    v = value(o)
    akeys = [k[1] for k in keys(v)]
    bkeys = [k[2] for k in !<loc>
    for ai in akeys, bi in bkeys
        get!(v, (ai, bi), 0)
    end
end
```

Target : keys

Token	certainty
EOP	44.1
==	3.8
[	2.3
NUM	2.3
.	1.1
(	0.9
f	0.8
UNK	0.7
EON	0.6
end	0.6

Example 5.3: Token prediction for incomplete list comprehension in Julia

```
function add_zero_counts!(o::Mosaic)
    v = value(o)
    akeys = [k[1] for k in keys(v)]
    bkeys = [k[2] for k in keys(v)]
    for ai in akeys, bi in bkeys
        get!(v, (ai, bi), 0 !<loc>
    end
end
```

Target :)

Token	certainty
,	47.0
EOP	36.4
)	12.5
;	1.1
=>	0.7
]	0.4
==	0.3
+	0.2
-	0.2
*	0.1

Example 5.4: Token prediction for completing a function call in Julia

error. Interestingly the model does not suggest any other variable names used earlier or later in the code snippet.

Looking at the list comprehension in example 5.2 we see that in this case, the model does not seem to know about the possibility of a list comprehension. Although the model is very certain that we are instantiating a list, as can be seen by the high certainty of predicting the ’,’ token, the only other solutions that could make sense are ’]’ or ’+’. The absence of the ’for’ token from the predictions shows that language-specific structure of the list comprehension seems to not be a possibility even though it is a monolingual model.

The worst prediction we found in the qualitative analysis was predicting a function call in the list comprehension, as seen in example 5.3. Although the model could have copied the line previous to it, the only predictions that would have been logical are the start of a list as given by ’[’ with only 2.3% certainty, or ’f’, or ’UNK’ both with less than 1% certainty.

Finally, when looking at the end of a function call with parameters, in example 5.4 we can see that the top 3 tokens are far more common than all the other tokens, the correct token ’)’ has a certainty of 12.5%, and ’,’, which is very common in a function call, has a certainty of 47%. The presence of the EOP token can be explained by the setup of the task,

```
public class PGSetTest2 extends PGTest{
    protected void setUp() throws Exception{
        super.setUp();
        System.out.!<loc>
    }
}
```

Target : println

Token	certainty
get	16.2
on	7.0
class	5.1
create	3.5
add	3.1
is	3.0
write	2.9
set	2.6
visit	2.1
to	1.4

Example 5.5: Token prediction for a common library call in Java

```
public class Array_newInstance01 extends
↪  JTTTest{
    public static boolean test(int i){
        return
        ↪  Array.newInstance(Array_newInstance01.class,
        ↪  i) != null;
    }
    @Test
    public void run1() throws Throwable{
        run !<loc>
    }
}
```

Target : test

Token	certainty
equals	11.7
l	6.2
'test	5.7
class	3.1
EON	2.1
i	1.6
that	1.6
e	1.5
null	1.2
assert	0.9

Example 5.6: Token prediction for a test method call in Java

where an expression such as `1+1 <EOP>` may be in a function call as it is an expression and a valid prediction location, although the model should have been able to eliminate this option due to the absence of a closing parenthesis.

Java Following on with the qualitative analysis of the Java examples, we start by looking at the very common library call `System.out.println(<STRING>)` in example 5.5. What we notice is that although the model is not able to predict the println token, we do see that the model has an understanding that we expect a function call. Compared to some of the results from Julia we see that there are no operators or syntactic symbols in the prediction, but mainly tokens which are common in function calls.

Similar to the common print function, Java also has a large number of tests, to see the models' performance on these, usually very short, test functions we ran the model on example 5.6. Here we see that the model recognizes that either it is a run function being

```
if( obj == null){
  return false;
}
if( this.getClass() != obj.getClass()){
  return !<loc>
}
AttachRequestArguments other
↪  =(AttachRequestArguments) obj;
```

Target : false

Token	certainty
this	21.2
null	14.5
false	9.4
new	8.7
true	5.0
UNK	2.0
get	1.9
return	1.5
NUM	1.0
is	1.0

Example 5.7: Token prediction to complete return value of an equals method in Java (truncated to fit)

```
@Override public boolean equals( Object
↪  obj){
    if(!<loc>){
    return true;
    }
    if( obj == null){
    return false;
    }
```

Target : this

Token	certainty
null	13.4
this	4.4
UNK	3.8
i	2.3
==	1.9
!	1.7
!=	1.5
value	1.4
type	1.3
false	1.2

Example 5.8: Token prediction to complete null check statement in an equals method (truncated to fit)

called, or 'runEquals', 'runL', or 'runTest'. The correct answer is 'runTest'. We can further see from the less likely results the model predicted that it has made a connection between the token 'assert' and being in a test function, as well as the token 'null' and 'that', these are all common tokens that could be used in a test class yet, in this case, they are not correct.

Moving on from test classes we look at the model's performance when working with an equals methods, a very common construct within Java. For both example 5.7 and example 5.8 we run the prediction for a part of an equals method. In example 5.7 we want to return the value that should be returned when the objects are not equal and for example 5.8 we want to predict the check if the memory addresses are equal with `this == obj`.

Looking at example 5.7 we see that the top two predictions would not make any logical return values, however, they are tokens that as can be seen from the examples appear frequently in an equals methods. The correct value 'false' is however predicted with almost twice the certainty of 'true'.

From example 5.8 we see that the correct value of the prediction is the second most

```
if(max_icp_distance_ > 0)
    truncateDepth(depths_curr_[0],
    ↪ max_icp_distance_);
for(int i = 1; i < LEVELS; ++ !<loc>)
    pyrDown(depths_curr_[i - 1],
    ↪ depths_curr_[i]);
for(int i = 0; i < LEVELS; ++ i){
    createVMap(cam_intrinsics(i),
    ↪ depths_curr_[i], vmaps_curr_[i]);
    computeNormalsEigen(vmaps_curr_[i],
    ↪ nmaps_curr_[i]);
}
```

Token	certainty
EOP	42.8
i	28.3
++	11.8
j	1.7
UNK	1.1
NUM	0.4
int	0.4
!=	0.3
const	0.3
c	0.3

Target : i

Example 5.9: Token prediction to complete a counter increment in a for loop in CPP (truncated to fit)

```
NIVissimConflictArea :: DictType
↪ NIVissimConflictArea :: myDict;
NIVissimConflictArea *
↪ NIVissimConflictArea::dictionary(int id){
  DictType::iterator i = myDict.!<loc>
  if(i == myDict.end()) {
    return nullptr;
  }
  return (*i).second;
}
```

Token	certainty
get	12.0
m	8.8
set	2.3
size	1.9
UNK	1.7
push	1.5
to	1.4
(	1.3
is	1.2
NUM	1.1

Target : find

Example 5.10: Token prediction to complete a counter increment in a for loop in CPP (truncated to fit)

likely predicted by the model. Furthermore, we see that the other predictions suggest that the model understands that there must be a boolean operation in the if statement as it predicts the operators, '==', '!', and '!=', which are not seen in example 5.7 although it is a prediction within the same equals method.

CPP Finally, the CPP results show similar trends as the Julia and Java results. We start with example 5.9, here we are predicting the counter increment in a for loop. From the predictions of the model, we can see that there is an understanding that an operation with a number and a variable need to be done. The correct target 'i' is given as the second most likely token after EOP. We believe this is due to the use of a prefix increment operator instead of the postfix increment operator. Both the operations `++ i` and `i ++` would increment

```
if(max_icp_distance_ > 0)
    truncateDepth(depths_curr_[0],
    ↪ max_icp_distance_);
for(int i = 1; i < LEVELS; ++ i)
    pyrDown(depths_curr_[i !<loc>
for(int i = 0; i < LEVELS; ++ i){
    createVMap(cam_intrinsics(i),
    ↪ depths_curr_[i], vmaps_curr_[i]);
```

Target : -

Token	certainty
]	92.9
[	2.9
+	0.9
.	0.8
*	0.5
,	0.5
i	0.3
::	0.2
->	0.2
-	0.1

Example 5.11: Token prediction to complete an array access with an equation in CPP (truncated to fit)

```
void wait(){
  std::cout << "Press enter to continue";
  getchar();
  std::cout << !<loc>
}
```

Target : STRING

Token	certainty
UNK	19.6
std	13.7
NUM	3.4
endl	3.3
boost	2.6
STRING	2.4
return	2.2
test	1.2
cout	0.9
true	0.8

Example 5.12: Token prediction to complete print statement in CPP (truncated to fit)

the counter `i` by 1, making the order unimportant, however, this did manage to confuse the network slightly.

Looking at example 5.10 we attempt to predict the name of the function used to retrieve the value of an index in a dictionary. As we can see the correct token 'find' is not predicted, however, we do see that the function name 'get' is the most likely prediction given by the model. Looking at this from a programmer's point of view the 'get' and 'find' methods are very closely related. Furthermore, we see that some of the other tokens predicted are also common method names that may be seen together with a dictionary, such as, 'set', 'size', and 'push'.

Analyzing the performance in list indexes, we can see in example 5.11 that there is a very high chance that the model will predict the ']' token after an index variable. This is often the correct choice looking at the frequency this happens in the rest of the method. However, in this case, we expected the '-' token which was the 10^{th} prediction. we acknowledge that the other predictions given by the model all make sense and aside from repeating the 'i' token would all return a valid program at this location.

Augmented > Baseline		Baseline > Augmented	
Token	Δ Accuracy	Token	Δ Accuracy
that	97.6	extends	100.0
1	87.9	]	97.3
helper	76.9	string	95.2
hash	66.7	api	83.5
require	50.0	manager	18.5

(a) Java

Augmented > Baseline		Baseline > Augmented	
Token	Δ Accuracy	Token	Δ Accuracy
double	100.0	code	100.0
int8	100.0	ptr	50.8
invalid	66.7	convert	21.4
=>	63.6	get	20.0
p2	63.6	k	10.0

(b) Julia

Augmented > Baseline		Baseline > Augmented	
Token	Δ Accuracy	Token	Δ Accuracy
rsp	100.0	stub	42.9
endl	89.8	dispatch	29.6
push	78.3	data	9.0
back	75.0	type	7.2
cout	70.1	d	4.5

(c) CPP

Table 5.6: Comparing tokens contributing most to the difference in performance between the augmented model and the baseline in the monolingual setting.

Finally, in example 5.12 we attempt to predict the string being printed in a common print statement. The print statement in CPP as can be seen in the example is different from most other languages so is an interesting example to look at. From the results we see that the model has made some associations with common tokens that are often used during the print statement such as 'std', 'endl', and 'cout', however, the correct value is the 6^{th} prediction.

5.5 Token performance

Finally, on top of analysing the performance on random samples from the dataset, we will also evaluate the areas where each model performed better on a per token basis. For this we use the results collected in section 5.2 and look at which tokens had the largest difference in performance for each model. For the comparisons we will use the monolingual models and models trained in the mixed setting as these were the best performing models in general. In table 5.6 we give the difference in performance of the augmented to baseline model in the monolingual setting. In 5.7 we give the difference in performance of the baseline and augmented model but in the mixed setting. Finally, in 5.8 we compare the performance between augmented models trained in the monolingual and mixed setting.

Java Starting with Java there are some interesting aspects to note. From tables 5.6, 5.7 and 5.8 we see that when comparing the types of tokens (from Figure 3.7) we see that Java has the highest number of shared tokens among the tokens that changed most in accuracy

Augmented > Baseline		Baseline > Augmented	
Token	Δ Accuracy	Token	Δ Accuracy
<	95.7	}	76.5
export	86.7	{	59.9
helper	82.7	require	40.5
[	76.2	num	37.8
run	50.6	scheduler	33.3

(a) Java

Augmented > Baseline		Baseline > Augmented	
Token	Δ Accuracy	Token	Δ Accuracy
object	100.0	code	100.0
rand	50.0	int8	100.0
quote	30.0	double	90.0
default	27.2	plot	47.6
float64	22.2	=>	36.4

(b) Julia

Augmented > baseline		Baseline > Augmented	
Token	Δ Accuracy	Token	Δ Accuracy
push	58.5	size	14.5
set	37.7	g	14.3
static	29.6	id	13.7
!	27.9	define	13.2
&	24.8	std	9.7

(c) CPP

Table 5.7: Comparing tokens contributing most to the difference in performance between the augmented model and the baseline in the mixed setting.

Monolingual > Mixed		Mixed > Monolingual	
Token	Δ Accuracy	Token	Δ Accuracy
}	76.5	extends	100.0
{	57.8	]	99.0
require	55.9	string	93.9
p	50.4	api	83.5
visit	48.4	operation	50.0

(a) Java

Monolingual > Mixed		Mixed > Monolingual	
Token	Δ Accuracy	Token	Δ Accuracy
double	100.0	object	100.0
int8	100.0	ptr	79.4
<:	75.0	convert	50.0
invalid	66.7	rand	50.0
=>	54.5	type	40.0

(b) Julia

Monolingual > Mixed		Mixed > Monolingual	
Token	Δ Accuracy	Token	Δ Accuracy
implement	100.0	stub	47.6
rsp	100.0	name	10.1
endl	99.4	size	8.8
back	75.9	type	6.5
cout	70.1	dispatch	5.0

(c) CPP

Table 5.8: Comparing tokens contributing most to the difference in performance between the monolingual and mixed setting for the augmented model.

across the runs. This is interesting to see as we also saw that the Java model was most sensitive to the inclusion of pre-trained embeddings (see Figure 5.10) as well as being the only model to benefit in the multilingual pretraining setting over the monolingual (see Table 5.1). Furthermore, when looking at the baseline vs augmented models in tables 5.6 and 5.7 we see that the tokens 'that', 'helper', 'hash', and 'run' contribute to the increase in performance of the augmented model. This backs up the findings in Examples 5.6, 5.7, 5.8 that the model when augmented with ASTs can identify when it is working with a test or equals method.

Julia Moving on to Julia we see a different trend in the data. Instead of shared tokens having a large difference in the performance of the models we see that when comparing the mixed to the monolingual setting, a large increase in performance comes from the use of tokens that are (almost) exclusive to Julia. From table 5.8 we see that the token 'int8' (a type used only in Julia), '<:' (an operator used only in Julia) and '=>' (the pair operator used more frequently in Julia than other languages) make up the majority of the list where the monolingual outperforms the mixed model. In the mixed setting, however, the performance was mostly boosted by tokens such as 'ptr', 'object', 'rand', and 'convert' which are fairly common tokens to use across all languages. A similar trend is also seen when the language unique tokens outperform the baseline in the monolingual setting. We believe this shows that for a resource-scarce language such as Julia, there is an advantage to adding structural information when predicting language-specific tokens, however, in the mixed setting, this benefit may be improved by the larger amount of data for shared tokens stemming from other languages.

CPP Finally, for CPP we see some of the trends previously described for Julia. In the mixed approach, the augmented model outperformed the baseline model on the common tokens 'push', 'set', 'static' and '!'. Surprisingly the '&' token was also present, which withing CPP can be used for pointer operations, however, in Java and Julia is predominantly used in boolean operations. This shows us that the inclusion of the ASTs in the mixed setting was able to help the model distinguish between a boolean operation and a pointer operation when working with the '&' symbol. Looking at the monolingual performances, however, we see that the tokens 'endl' and 'cout' are both tokens that contribute largely to the performance of the augmented vs baseline setting as well as the monolingual vs mixed setting. The increase in performance for these tokens in the monolingual augmented vs baseline models, as well as the increase in performance in the monolingual setting vs the mixed setting, reinforces our idea that the ASTs are too different across languages, as this is a structure only seen in CPP which was able to increase performance in the monolingual augmented setting, however, did not boost performance compared to the mixed setting. We also saw in example 5.12 that the model is able to recognize when it is predicting in the common print expression in CPP `cout << <STRING> << endl;`.

Chapter 6

Conclusion

To conclude this thesis, we will give a quick summary of the findings we found in chapter 5, followed by the areas where we believe the model may have issues performing in the Threats to validity section, and we end with an overview of interesting areas for future work.

6.1 Summary

The use of ASTs has become more common when looking at machine learning for software engineering, we combine the use of ASTs with the creation of high-quality embeddings and use this to teach a network to complete code in a multilingual environment. We will begin by discussing the results we found to answer the research questions from chapter 1.1, followed by the findings from the ablation study in chapter 5.3.

Following the order of the experimental setup, we begin by summarizing the results of the embedding LSTM. To answer RQ1 we saw that there was no benefit overall to the performance of the model when learning from multiple languages compared to one language at a time. Interestingly, however, there was also not a drop in performance as was found in other works [57], showing that it is possible to represent multiple languages effectively in a single model. The findings for RQ2 follow a similar path. We found no clear benefit to finetuning on a final language, however, there was also no detriment to the final performance. The model did however converge faster at the start of training, showing promise if convergence speed is important.

Moving on to the transformer model we summarize the findings for RQ3. We found that for every language the overall best performing model was trained with the input that was augmented by ASTs. When looking at a pairwise comparison there was also a benefit in most cases to using ASTs in the input although there were a few cases where it was not beneficial. Furthermore, we found that the inclusion of ASTs in the input increased the complexity of the task, which we saw by the augmented models having a higher variance on the final outputs across multiple runs than their baseline counterparts. Looking at the pre-training part we saw that oftentimes, the monolingual models were preferred for the lowest loss, letting us believe that there is an extra layer of difficulty due to different shapes

of ASTs that may require a larger model to effectively utilize.

Comparing the performance of the monolingual, mixed and multilingual models for RQ4, we saw that for the comparison of the pre-training task, the monolingual model performed best, which we suspect was due to different AST structures as mentioned previously. For the final task, we compared the model in a monolingual setting, a mixed setting, and a multilingual setting. This showed us that even though the pre-training task was worse in the multilingual setting, the mixed approach to the final task performed best on most languages.

Taking a closer look at the token performances from all models showed us that the difference in performance between baseline and augmented models can be attributed to the augmented models' performance to predict a wider variety of tokens correctly than the baseline model in the monolingual setting. When comparing the distribution of correct tokens for the monolingual versus the mixed setting, we found that for Java and Julia the distribution is more similar, however, for CPP there was a definite benefit to a more spread-out distribution in the monolingual setting. Overall the performance of the models when looking at the tokens being predicted correctly was low with all correct predictions being only in the top 300 tokens by occurrence frequency.

complementing the findings from the investigation we will look into which parts of the model were necessary in the ablation study. Starting with the need for the LSTM model and the pre-trained embeddings we saw that the baseline model, which is more similar to other models that have been shown to benefit from pre-trained embeddings, showed a definite increase in performance. The augmented model, however, saw a drop in performance. This was seen in both the pre-training as well as the finetuning part of training, however, the pre-training of the baseline models showed the need for pre-trained embeddings the most.

Finally, we looked at the need for the pre-training task. Similar to the need for pre-trained embeddings we saw that the augmented model was not as dependent on pre-training as expected. the monolingual models showed a slight increase in performance in Java, no increase for Julia, and only CPP which was the worst performing model and was underfit to the data had a definite increase in performance when using pre-training. For the multilingual models, we saw that for all languages the models trained without pre-training performed the best, showing us that this expensive step is not necessary when working with ASTs. We do believe however that given a larger model the results may vary.

Overall, we have shown that there is a benefit to augmenting the input with ASTs when working on the next token prediction task in source code. We saw that for the LSTM models there was no downside to training in a multilingual setting as was seen in other papers. For Transformer models, it was often best to work in a mixed setting, although if working in a multilingual setting it was not necessary to pre-train the model saving computation time.

```
(if (> a b) (write-line a) (write-line b))
```

Snippet 6.1: Example lisp program.

6.2 Threats to validity

As for all research, certain areas could be problematic for this model. Here we will lay out the major problems that we foresee when it comes to the use of the augmented transformer in any form of application. We divide these threats to validity into two main groups. First, we will discuss the limitations from a theoretical point of view, analysing the aspects of what could be detrimental to the transformer, based purely on the theory. Secondly, we will go over the issues that may arrive if the proposed transformer model is ever to be used in a real-world setting.

6.2.1 Theoretical threats to validity

From the design of the model and the experiments, we have identified two areas that may be theoretical threats to this model's validity. We will explain the theoretical threat and how it affects the model with regards to scaling and the choice of languages.

Scaling One common issue that many neural networks have to deal with is scaling. This stems from the nature of neural connection in linear layers, or the $O(n^2)$ scaling of attention calculations when looking at the input sequence length. This scaling issue is present in the graph Transformer. As this transformer works on the AST generated from a snippet of source code, in the best case, the calculation would scale quadratically. Here we assume there is no grammar which only returns the code as the AST. However, as this would be a normal transformer we must look at what is expected to happen with scaling in a normal use case. The actual cost of increasing the sequence length used in predictions is $O(m^2)$ where $m = O(n^x)$ where n is the length of the sequence and x is the scaling factor of the AST generated for its language. $x > 1$ for all languages unless there is some grammar where tokens are deleted from the AST. Combining this with the need for both a normal transformer block, together with a structural transformer block shows that the proposed model scales at least twice as quickly as the comparable BERT [18] model.

Languages used A further threat to the validity of this model is that in the experiments, we have shown it to work for Java, CPP, and Julia. These languages look similar in grammatical structure and vary only slightly in the ordering of tokens. There are however, a group of programming languages that, although they may not be as popular at the moment as the mentioned three programming languages, are also programming languages. These languages can be categorized as list-based programming languages.

A well-known list-based programming language is Lisp [42], in snippet 6.1 we can see an example of a list program. We can see that the structure of such a program is different from the languages studied in this paper, so the performance of the model can not be guaranteed in this setting.

6.2.2 Real world threats to validity

Looking at the design of the model we can see a few threats to validity that could be an issue when deploying this model in the real world.

Computational complexity Computational complexity can be a catch-all for any problems that could appear in the real world, in this case, we specify that a few key aspects of the transformer model may require a lot of computation when running on a computer. The main area outside of the transformer model that would require a large amount of computation is the calculation of the eigenvalues of the laplacian, and the generation of the AST. These are common problems that can be completed decently fast. The generation of ASTs is often done by IDEs for use in text highlighting, and work has gone into calculating eigenvalues rapidly in many linear algebra packages such as BLAS and LAPACK, however, even though it can be done fast, it may require a more modern computer to integrate it into an IDE efficiently. Secondly, the issue with ASTs is also addressed in some regard by IDEs that use AST generation for features such as collapsible code blocks and text highlighting, however, there is no standard for the kinds of nodes used so these trees may vary by IDE as well as by plugin which would require the IDE to generate an AST only for the use in the transformer separately which only adds onto the computational load.

Some, papers that have looked at the use of transformers for source code completions have seen the same issues, albeit only for the use of a transformer model trained on source code, such as IntelliCode complete [57]. The solution that the developers of this model came up with was to keep a local cache of frequent completions and run all other calculations on a dedicated server. This showed to be possible, however, would require larger development efforts to implement.

Incomplete ASTs One further threat to this model in the real world is that it requires the ability to generate ASTs that are incomplete. As has been shown by the existence of this project this is possible to do, and also can be done rather simply by editing an existing grammar to accept special tokens for where code may be missing (given that we know where the cursor is). However as one of the goals of this project is to have a transformer network that can perform well on a language that has very little source code available, it would require the creation of such a grammar. Given that new languages such as Julia may not even have a public grammar and change their languages frequently at the start of its life this would require constant updates to the grammar and keeping up with all the changes in the language.

Hardware availability As is often the case with machine learning models, they require many powerful GPUs to train large-scale models. Due to only having reliable access to a single Nvidia GeForce RTX 3080 with 10 GB of RAM at a time the presented model with all its results fits only on 1 GPU. This is an order of magnitude smaller than other models such as the BERT model, which was trained on 16 Cloud TPUs for 4 days [18] and GPT-c models which were trained on 80 v100 GPUs for just under 500 hours [57]. This gives an idea of the difference in scale for the other transformer model, due to this our model is

trained on less data, and for less time, the results may vary given a larger dataset/training setup.

6.3 Future Work

During the explorations conducted in this thesis, there have been points where we had to decide to limit the scope to ensure that it would be completed. This has left us with several areas that would be interesting for the future of the use of transformer models in machine learning for software engineering. We will divide the future work section into two main groups. The initial ideas for future research will stem from areas that seemed interesting yet were omitted to remain in the scope of this research. Then we will progress to discuss the areas of future work that have come out of the results gathered in this thesis.

6.3.1 More pre-training tasks

Given the similarity we have seen between the other graph transformers and language models such as BERT [67], we propose an extension to this work that mirrors the approach used in the training of the BERT-transformer [18] and the GPT 1,2,3 transformers [48, 49, 12]. We see that models based upon the GPT transformer architecture seem to have a larger number of pre-training tasks, we give an example of several tasks that could be adapted to the current model setup.

Task 1 The first task that comes to mind is the Next Sentence Prediction task (NSP) which is used in training the BERT transformer as helping understand the relationships in language inference and question-answer tasks. We redefine the NSP task as the Is SubTree task (IST).

Given the same rationale as BERT, we set up the task as shown in figure 6.1. For this task we make use of the structure of an AST, knowing that if we replace any nodes in the graph, we can connect edges like the order edges and possible other edges such as variable use and returns to edges (from other papers [4]) to the AST.

This lets us ask a network whether a given augmented AST is made up of a real program or whether at any point a subtree of the augmented AST got replaced with that from another program. Intuitively this would be beneficial if a downstream task of the transformer were to be error detection/bug localization.

Task 2 For our second task, we look toward the GPT models. One task trained upon by the GPT models is the summarization task of news articles, in this task, the transformer is asked to summarize what the news article is reporting on. Another task is the translation task where two languages are translated from one to another. In the following task, we propose a hybrid task of both which is comment generation which can be seen as both a translation (from code to English) or a summarization problem.

In the GPT papers, the model was asked to enter the summarization after a TL;DR tag [49], when given source code however we can insert a comment at any location. For this purpose, there is already a specific token applicable in most languages. To incorporate this into the AST structures, there would be two options of adding support for including comments in an AST (which is not common) or to only generate summaries of a method, not inline comments [7, 8].

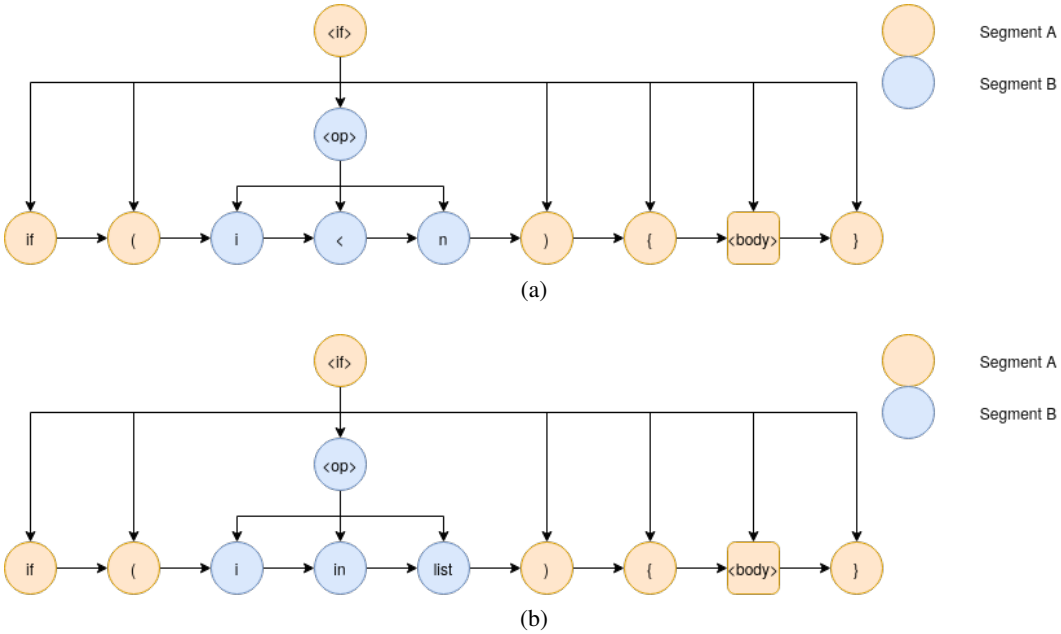


Figure 6.1: Example of the IST task, the colors represent the segment embedding. In 6.1a an if statement is given, where the body has been truncated. In 6.1b the original if statement has had the condition replace by a different condition from somewhere in the code base, the body is once again truncated.

Task 3 The final proposed task we believe could have benefits in training a general-purpose transformer for source code language tasks is a variation of the Children's book task (CBT) [49] in this task the model needs to identify whether a word is a named entity, noun, or verb, this is done as a cloze (masking) task where a choice of 10 words is given.

This is also related to the Token type task from the CodeFill [28] paper where the model needs to determine for a given token whether it is a function name, library name, local variable, or module.

Implementation of this task would include a slight modification of the Universal ASTs as they would need to distinguish between local variables, function names, module names, and others, yet from the papers analyzed this is a promising alleyway to explore in boosting performance.

6.3.2 Weisfeiler-Lehman embeddings/multiple embeddings

A further area of interest is to look at the embedding of nodes used in the transformer. Looking at the used edge type in the explored architecture, we can see that there is a large difference between the AST and the order edges. For the AST we can see that the Laplacian embedding is intuitive, however, we saw that using the Laplacian as a positional embedding for the order edges was not beneficial. This leads us to believe there may be better suited positional encodings to add to the model. Furthermore, from many transformers [18, 67,

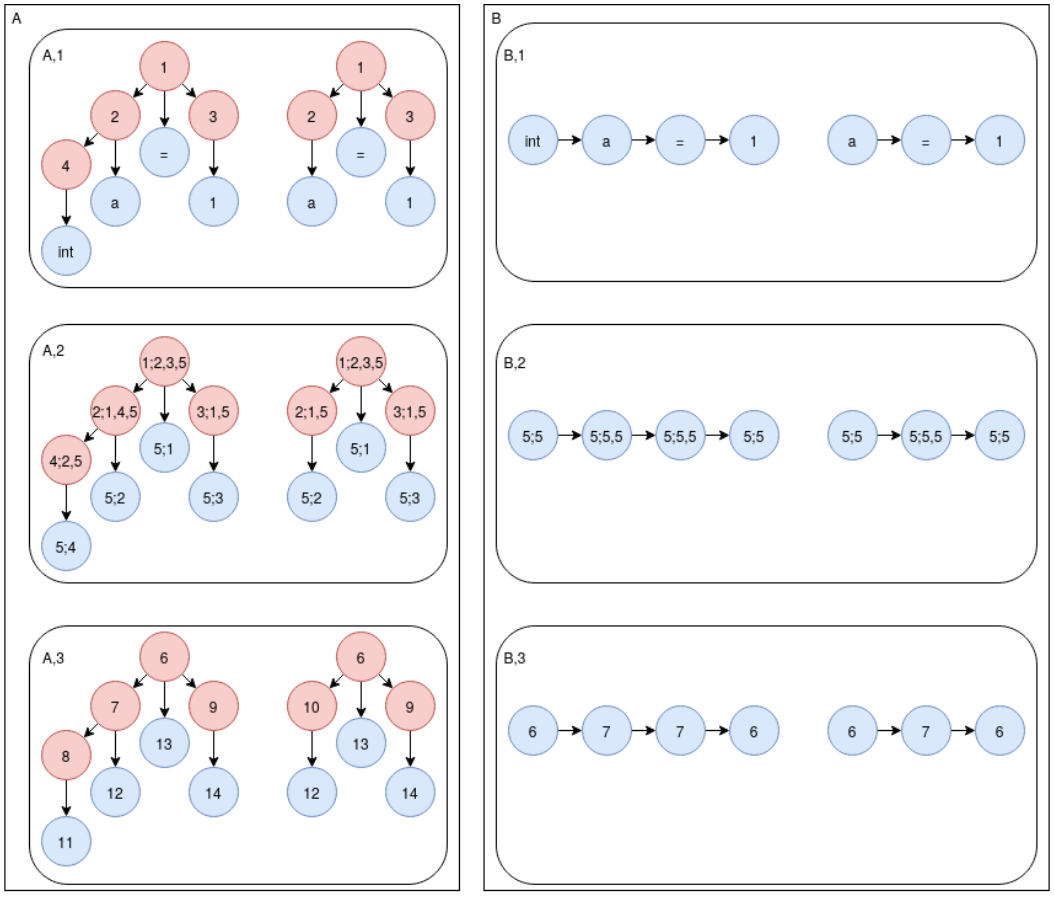


Figure 6.2: Example of WL-encoding proposed. It is run for 1 iteration of the algorithm.

48, 49, 12, 57] we can see that it is quite common to use multiple embeddings. This also allows us to explore other potential embeddings for the transformer.

We have seen that using the code generated in the Weisfeiler-Lehman isomorphism algorithm can be used as a potential embedding [67] where the code given to the node is referred to as its ”role” in the graph. Looking at the node embeddings of this paper we shall explore the role of the node types further.

In figure 6.2 we give an example of a Weisfeiler-Lehman-encoding (WL-encoding) for 2 Augmented ASTs. Figures 6.2,A,1:3 contains the AST edges of both Augmented ASTs and figures 6.2,B,1:3 contains all the order edges of the augmented AST. In Both A and B, the first image is the nodes with an example node type, the second is the intermediate representation of the node types after 1 iteration of the WL-isomorphic algorithm, and the third figure is the new node types after renumbering, based on the intermediate output of the WL-isomorphism algorithm.

The example we are using in figure 6.2 is that of 2 snippets of any language where a type can be specified before a variable (as is the case for Java and C), the snippets are shown in snippet 6.2

```
int a = 1
```
(a) Typed

```
a = 1
```
(b) No type

Snippet 6.2: Example of a variable assignment with and without type declaration.

From both ASTs we can see that it is an assignment. However, in the case of languages such as Java and C, if there is a type definition in front of the variable name, we know that it is the first occurrence of this variable. Looking at figure 6.2, we can see that the node corresponding to the variable name `a` gets two different values after one iteration of the WL algorithm. This encodes the knowledge that it is a typed variable definition instead of a normal variable definition. This example hopefully shows that there is information that can be generated through using an encoding like the WL-encoding, which may be beneficial to the network in highlighting common code structures in source code.

On the other hand looking at figure 6.2, B we can see that there is little use in using this encoding when looking at the order edge as it only highlights which are the first tokens and last tokens and which tokens are in the middle. This is information that would be given by the beginning of file and end of file tokens and could be misleading when looking at a smaller sub-sampled graph. This is another reason we see to closely evaluate the kind of edges being used in an augmented AST and especially which positional encoding to use.

Finally, it must be noted that for the WL-encoding, there is an extra layer of complexity as different code structures will be highlighted at different depths of the WL algorithm, making this an expensive choice (in regards to feature engineering) but which may also improve performance if the main common structures are appropriately highlighted.

This high cost of feature engineering may be offset by using an approach that automatically selects the best performing features, for this, we suggest an automatic selection mechanism as was sen in the Graph Transformer Network [61].

6.3.3 Auto-regressive completions

One major drawback to the proposed model is that it can only generate a single token prediction for the given input. We suggest that in future work one may attempt to generate an auto-regressive model similar to the decoder only architecture of the GPT transformers [48, 49, 12]. To facilitate this behavior it would require an additional grammar to generate ASTs for incomplete statements that can then be inserted into the graph. We believe this will be a beneficial avenue to explore as it lets us mimic behavior similar to the CM3 [2] model, which has been shown to be beneficial in other NLP areas.

6.3.4 Cross lingual clone detection

An interesting area of software engineering is the managing of large code bases, more specifically, the presence and detection of code clones (snippets/files of code that give the same output). Static methods have been relatively effective at detecting semantically similar code clones, and machine learning methods have shown to perform well with semantic clones that are hard to detect for textual or lexical approaches [51].

6.3.5 summarize rest of file as input

Finally, aside from the possible application/training tasks, there is also a more practical problem that needs solving. This problem relates to the inability of a transformer to work for varying lengths of input, requiring most inputs to be truncated in practice. While it can be argued that giving a large enough neighborhood of a certain node to the model should give enough information, we see issues arising in situations where a variable is defined at the top of the file, and only used towards the end. This type of long-range dependency will likely not be addressed in most subsampling algorithms. A solution to this has been attempted in the TransformerXL architecture [17]. The TransformerXL architecture is designed for sequences, and works by splitting the input into fixed-length segments. However, when starting a new segment the hidden state of the previous segment is used to summarize the knowledge gained from the previous segment, therefore, increasing the length of dependencies that can be modeled.

This same idea leads us to the final area of future research we propose. In order to preserve the knowledge of the entire graph when looking at one segment, we propose to work on summarizing the knowledge in a node. The intuition behind this is that given a subsampled augmented-AST, we can run the model on all or a set of nodes where the AST has been truncated and learn a representation of these nodes that will summarize the node.

As a final potential benefit to this approach, we look at the nature of data that is contained in a file of source code, In many languages, it is possible to import a set of code from another file. When looking at only a large repository of code, we can learn a representation of all tokens, and the name of libraries may be encoded in such a way that they contain vital information about their use. However, if we can summarize the information for a library in one node, we may choose to use representations learned on the contents of a file rather than the appearances of its tokens.

6.3.6 Future work with more hardware

Finally we will address future work that we see as an interesting avenue to explore that we could not implement due to hardware availability.

Tokenization As previously mentioned in this thesis, we decided to use a subword encoding for the source code over any other form of encoding. While subword encoding works as a tokenization method, there are many more methods that have been shown to be beneficial to NLP models such as Byte-Pair Encoding (BPE) [53], WordPiece [64] and SentencePiece [35]. The overarching methodology behind these methods is to find an optimal combination of subtokens to express the language as efficiently as possible. However, due to the splitting of the words into subtokens, we would be forced to reduce the amount of source code the model can take as an input. As all methods split the subwords into smaller tokens this would require more leaf nodes. This reduces the amount of source code we can take as an input because we are limited to 100 nodes in the augmented AST, and each token has its own node.

```
bigInt = var_name;
```

(a) Full line of code

```
<start>big<sep>int<EON> = <start>var<sep>name<EON>;
```

(b) Initial split according to naming convention

```
[WordPiece/SentencePiece sequence] <=> [WordPiece/SentencePiece sequence]<;>
```

(c) Final input, the tokenization of the programmer defined variables is dependent on the model so a placeholder is used.

Snippet 6.3: Example tokenization when using a WordPiece/SentencePiece model.

Looking at the designs of the BPE, WordPiece, and SentencePiece models we can see that they are similar to one another, however, they also differ in one important aspect. BPE compared to WordPiece and SentencePiece only combines the most common byte pairs. Also, information such as whitespace may be lost [53]. WordPiece and SentencePiece both include a way to determine where the whitespace in a sentence is. Although this may seem trivial. The method they use is of importance to us. For Wordpiece A special token, '_', is used to denote the start of a word [64], whereas in the SentencePiece model whitespace is inserted as an '_' [35]. We focus on the use of a separation token such as '_' when looking at splitting programmer-defined tokens.

We suggest the following approach to using either WordPiece or SentencePiece tokenization. We start with looking at the nature of tokens once again. We see that the largest part of the vocabulary belongs to the group of tokens from the programmer-defined words. Also, we notice that each language-defined token, can not be merged with any other tokens as this would require nodes to be merged. We use this information to limit the use of either the WordPiece or SentencePiece model for programmer-defined tokens within one definition node.

Within these programmer-defined tokens, we once again look at the naming conventions and abstract away from them. We split the programmer-defined tokens into subwords according to their convention. Then we add the extra tokens needed for the models. For the transformer model, we add the <EON> token at the end of all programmer-defined tokens, and for the WordPiece/SentencePiece tokenizers, we insert a <sep> token to separate the words and a <start> token to denote the beginning of a programmer-defined word.

Now we have the data split in such a way that we can use the SentencePiece/WordPiece tokenizers on the programmer-defined tokens, while we keep a set of special tokens reserved for the (shared) language-specific tokens. We give a step-by-step example in snippet 6.3.

Scaling up the model Furthermore, we have seen when looking at other models in chapter 2 we see that the smallest comparable models have in the order of 100 million parameters [18]. While most models are slightly bigger [57], and some are much bigger [48, 49, 12]. It would be interesting to see whether the performance of the model increases with a larger number of parameters. The opposite has been shown by the IntelliCode [57]

model, which lost performance when using knowledge distillation to decrease the number of parameters from 300 million to 100 million.

Extra languages Looking at other papers published, we have seen that there are several languages commonly used in papers on code completion. We have implemented the languages Java and CPP, both of which are very popular. However, there are many other languages such as Python [57] [28] [33] [63], and JavaScript [57] [63]. After scaling up the model to a competitive size, it would be interesting to evaluate its performance against other models in the language they were designed and evaluated in.

Bibliography

[1] A self-attentional neural architecture for code completion with multi-task learning. *CoRR*, abs/1909.06983, 2019. URL http://arxiv.org/abs/1909.06983. Withdrawn.

[2] Armen Aghajanyan, Bernie Huang, Candace Ross, Vladimir Karpukhin, Hu Xu, Naman Goyal, Dmytro Okhonko, Mandar Joshi, Gargi Ghosh, Mike Lewis, and Luke Zettlemoyer. Cm3: A causal masked multimodal model of the internet. *CoRR*, abs/2201.07520, 2022. URL https://arxiv.org/abs/2201.07520.

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pretraining for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.211. URL https://aclanthology.org/2021.naacl-main.211.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

[5] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=i80OPhOCVH2.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *SIGPLAN Not.*, 53(4):404–419, jun 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192412. URL https://doi.org/10.1145/3296979.3192412.

[7] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1gKYo09tX.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290353. URL https://doi.org/10.1145/3290353.

[9] Avishkar Bhoopchand, Tim Rocktäschel, Earl T. Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network. *CoRR*, abs/1611.08307, 2016. URL http://arxiv.org/abs/1611.08307.

[10] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017. doi: 10.1162/tacl_a_00051. URL https://aclanthology.org/Q17-1010.

[11] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bke4KsA5FX.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[13] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 213–222, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605580012. doi: 10.1145/1595696.1595728. URL https://doi.org/10.1145/1595696.1595728.

[14] Wenlin Chen, David Grangier, and Michael Auli. Strategies for training large vocabulary neural language models. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1975–1985, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1186. URL https://aclanthology.org/P16-1186.

[15] Roberto Cipolla, Yarin Gal, and Alex Kendall. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7482–7491, 2018. doi: 10.1109/CVPR.2018.00781.

[16] Alexis Conneau, Shijie Wu, Haoran Li, Luke Zettlemoyer, and Veselin Stoyanov. Emerging cross-lingual structure in pretrained language models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6022–6034, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.536. URL https://aclanthology.org/2020.acl-main.536.

[17] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1285. URL https://aclanthology.org/P19-1285.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

[19] Thomas Dowdell and Hongyu Zhang. Language modelling for source code with transformer-xl. *CoRR*, abs/2007.15813, 2020. URL https://arxiv.org/abs/2007.15813.

[20] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs, 2021.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

[22] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2022. URL https://arxiv.org/abs/2204.05999.

[23] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487132.

[24] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=B1lnbRNtwr.

[25] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016. URL http://arxiv.org/abs/1606.08415.

[26] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

[27] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1195. URL https://aclanthology.org/P16-1195.

[28] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: Multi-token code completion by jointly learning from structure and naming sequences. *arXiv preprint arXiv:2202.06689*, 2022.

[29] M I Jordan. Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986. 5 1986. URL https://www.osti.gov/biblio/6910294.

[30] Chung Fan R K. *Chapter 1 Eigenvalues and the Laplacian of a graph*. The American Mathematical Society, 1997.

[31] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[32] Rafael-Michael Karampatsis and Charles Sutton. Scelmo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214*, 2020.

[33] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162, 2021. doi: 10.1109/ICSE43902.2021.00026.

[34] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1007. URL https://aclanthology.org/P18-1007.

[35] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-2012. URL https://aclanthology.org/D18-2012.

[36] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 4159–4165. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/578. URL https://doi.org/10.24963/ijcai.2018/578.

[37] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05493.

[38] Chang Liu, Xin Wang, Richard Shin, Joseph E. Gonzalez, and Dawn Song. Neural code completion. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=rJbPBt9lg.

[39] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkgz2aEKDr.

[40] Peter J. Liu*, Mohammad Saleh*, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=Hyg0vbWC-.

[41] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL http://arxiv.org/abs/1907.11692.

[42] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[43] Oren Melamud, Jacob Goldberger, and Ido Dagan. context2vec: Learning generic context embedding with bidirectional lstm. In *Proceedings of the 20th SIGNLL conference on computational natural language learning*, pages 51–61, 2016.

[44] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013. URL http://arxiv.org/abs/1301.3781.

[45] Mariá C.V. Nascimento and André C.P.L.F. de Carvalho. Spectral methods for graph clustering – a survey. *European Journal of Operational Research*, 211(2):221–231, 2011. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2010.08.012. URL https://www.sciencedirect.com/science/article/pii/S0377221710005497.

[46] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[47] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1202. URL https://aclanthology.org/N18-1202.

[48] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[49] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[50] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747, oct 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984041. URL https://doi.org/10.1145/3022671.2984041.

[51] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2009.02.007. URL https://www.sciencedirect.com/science/article/pii/S0167642309000367.

[52] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter, 2019.

[53] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162.

[54] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-2074. URL https://aclanthology.org/N18-2074.

[55] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011.

[56] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '19, page 2727–2735, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362016. doi: 10.1145/3292500.3330699. URL https://doi.org/10.1145/3292500.3330699.

[57] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. *IntelliCode Compose: Code Generation Using Transformer*, page 1433–1443. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450370431. URL https://doi.org/10.1145/3368089.3417058.

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

[59] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ByloJ20qtm.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[61] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[62] Takashi Wada, Tomoharu Iwata, and Yuji Matsumoto. Unsupervised multilingual word embedding with limited resources using neural language models. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3113–3124, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1300. URL https://aclanthology.org/P19-1300.

[63] Yanlin Wang and Hui Li. Code completion by modeling flattened abstract syntax trees as graphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35 (16):14015–14023, May 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/17650.

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

translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL http://arxiv.org/abs/1609.08144.

[65] Shafiq Joty Steven C.H. Hoi Yue Wang, Weishi Wang. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, 2021.

[66] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. Graph transformer networks. *Advances in neural information processing systems*, 32, 2019.

[67] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. Graph-bert: Only attention is needed for learning graph representations. *arXiv preprint arXiv:2001.05140*, 2020.

[68] Jingyu Zhao, Feiqing Huang, Jia Lv, Yanjie Duan, Zhen Qin, Guodong Li, and Guangjian Tian. Do RNN and LSTM have long memory? In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 11365–11375. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/zhao20c.html.

[69] George Kingsley Zipf. *On the economy of words*, page 22–47. Addison Wesley Press, Inc.