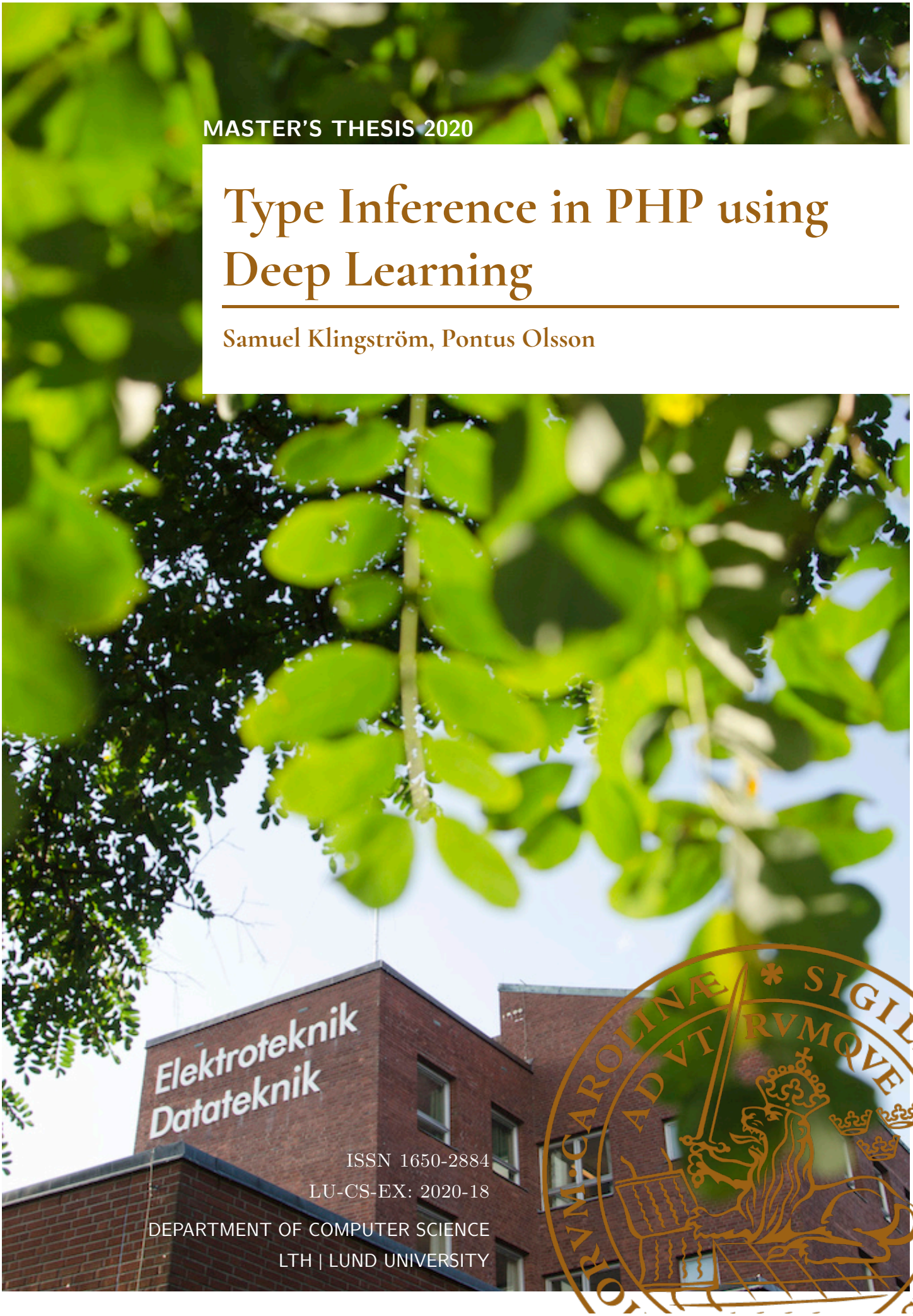


MASTER'S THESIS 2020

Type Inference in PHP using Deep Learning

Samuel Klingström, Pontus Olsson

ISSN 1650-2884
LU-CS-EX: 2020-18

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-18

Type Inference in PHP using Deep Learning

Samuel Klingström, Pontus Olsson

Type Inference in PHP using Deep Learning

Samuel Klingström
dat15skl@student.lu.se

Pontus Olsson
dat15pol@student.lu.se

June 17, 2020

Master's thesis work carried out at Axis Communications AB.

Supervisors: Niklas Fors, niklas.fors@cs.lth.se
Johannes Claesson, Johannes.Claesson@axis.com
Simon D Månsson, Simon.D.Mansson@axis.com

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Dynamically typed programming languages such as PHP, JavaScript and Python have recently started supporting gradual typing, where type annotations can be added to part of the code. Tools that can perform type inference are therefore becoming increasingly helpful as they could ease the labor intensive task of updating legacy code for developers. However, for PHP, most static code analysis tools have lacking or unsatisfactory type inference functionality.

In this thesis, we use deep learning to predict type annotations for parameters in PHP. The neural network can, given a function or method, predict the type annotations for the parameters based on their usage. The predictions are then presented in the code comment. This approach builds upon the previous work, code2vec, and is based on the idea of representing code as paths in its abstract syntax tree.

After training the model with the 10,000 most popular PHP repositories from Github, it was able to correctly predict type annotations with a top-1 accuracy of 76.2 % and a top-3 accuracy of 84.2 %. These results are better than the current code analysis tool we tested for PHP. We conclude that deep learning can successfully be used for type inference in PHP with great results.

Keywords: PHP, Type Inference, Deep Learning, Neural Networks, AST

Acknowledgements

We would like to thank our supervisor Niklas Fors for the guidance and valuable feedback throughout the thesis.

We would also like to thank our supervisors at Axis, Simon D. Månsson and Johannes Claesson, for sharing their ideas and providing us with helpful feedback.

Lastly, we would like to thank Marcus Klang for his valuable input and advice regarding any machine learning related issues we stumbled upon.

Contribution Statement

Implementation

Both authors have participated in the design and development of all parts of this thesis. The deep learning model as well as the first part of the preprocessing was for the most part implemented by Samuel. Meanwhile, Pontus added support for PHP in Astminer, made the necessary modifications to Astminer and implemented the final preprocessing. The evaluation of existing code analysis tools as well as the qualitative evaluation was done together where both authors contributed equally. Throughout the thesis, both authors have continuously updated and patched all the parts of the implementation.

Thesis Writing

Both authors wrote on all parts of the thesis and discussed all topics together. Samuel focused more on the chapters about the model, related work and evaluation while Pontus focused more on the introduction, background, implementation and qualitative evaluation. Both authors wrote the discussion, conclusion and abstract together.

Contents

Chapter 1
Introduction

Dynamically typed programming languages have become more and more popular in software development. Python is heavily used in machine-learning projects, JavaScript is heavily used in web development, and PHP is also used for web development, although PHP is overshadowed by JavaScript. Dynamically typed languages are less restrictive and less verbose compared to statically typed languages, which is part of the reason why the dynamic languages are popular. One drawback with dynamically typed languages is that it is hard to efficiently debug a program. Gao et al. [7] found that around 15 % of the bugs generated by the public JavaScript repositories they tested could have been detected if a more statically typed version of JavaScript, such as TypeScript, was used. Many of the dynamically typed programming languages have started to support gradual typing for variables and functions. Gradual typing is a type system where some variables and expressions may be given types and analysed at compile time (like static typing), and some expressions may be left untyped and checked at run-time (like dynamic typing). Python has a library called typing, PHP has support for gradual typing within the language itself and supports several types using the union type, PHP has variants such as Hack, JavaScript has variants such as TypeScript etc. With the increased support for typing in the dynamic languages themselves, developers are using more type annotations in their code.

With more developers wanting to use type annotations in their code, the refactorization of legacy code becomes progressively cumbersome the longer it takes for the developer to update the code. Since dynamically typed languages assign types to method parameters/variables during run time, it can be labor intensive to add type annotations to parameters/variables in legacy code. A tool that can be helpful in this situation is a static code analysis tool, such as a linter or a code analyzer. However, different languages have different tools, and the capabilities of the tools are varying. Currently, many of the code analysis tools for PHP lack type inference functionality, and if any type inference can be done, it is not conveyed to the user in a useful format or with satisfying results. For example, a tool that can do type inference, called Psalm, could only infer types on 9.7 % of the parameters, and with an accuracy of 44.1 %. Studies have been done to see how machine learning can be used instead of static code

analysis, with many varying methodologies, and have seen varying amount of success. One methodology that was not used for type inference, but for prediction of method names in Java, was a deep learning neural network called code2vec [3] which represented code snippets as code vectors, and can be used to find semantic properties in the code. In the paper by Alon et al. [3], they list type inference as a possible usability case of their model, and we decided that this was a good opportunity to test the model on type inference.

1.1 Goal and Purpose

In this thesis, we use a deep learning neural network to predict type annotations on parameters in PHP. The model used is based on the model used in code2vec, rewritten in Pytorch with some modification to fit our purpose. Before the code is sent into the model, the code needs to be preprocessed, so that relevant information about the code can be extracted.

The goal of this thesis is to explore how type inference can be done in PHP using machine learning, and compare how efficient machine learning type inference is compared to type inference using static analysis tools.

The main purpose for this thesis is to make debugging of legacy PHP code easier. For dynamically typed programming languages, it is especially difficult to debug code since variables and function parameters are not defined with a specific type. If type inference using machine learning can replace type inference using static code analysis, or be used in addition to code analysis tools, debugging could be done much more efficiently, and the workflow of the developer could be improved.

1.2 Research Questions

The research questions for this thesis are the following:

1. How can type inference be done in PHP using machine learning?
2. How well can type inference be done in PHP using machine learning?
3. How does type inference done using machine learning compare to type inference done with static code analysis?

We hope that this thesis will contribute to the scientific world by extending the knowledge on how machine learning can be used for type inference, and show another use of the code2vec model that might be useful for future use.

1.3 Results

The output of the model is the top k predictions from the model, based on how large k is. The output of the model is inserted into the code as a PHPDoc, which can be seen in listing 1.1. With a dataset of 10,000 PHP repositories, the model manages to predict types with a top-1 accuracy of 76.2 %, a top-3 accuracy of 84.2 %.

Listing 1.1: Example of predictions made by the neural network. This function implements binary search, and our model has predicted the types array for $arr and int for $x. The certainty of the predictions can be seen next to the predictions

```php
<?php

/**
 * Predictions for $arr
 * 1. array 99.79 %
 * 2. bool 0.12 %
 * 3. int 0.07 %
 *
 * Predictions for $x
 * 1. int 98.96 %
 * 2. number 0.23 %
 * 3. numeric 0.22 %
 */
function binarySearch($arr, $x) {
    if (count($arr) === 0) {
        return false;
    }
    $low = 0;
    $high = count($arr) - 1;
    while ($low <= $high) {
        $mid = floor(($low + $high) / 2);
        if ($arr[$mid] == $x) {
            return true;
        }
        if ($x < $arr[$mid]) {
            $high = $mid - 1;
        } else {
            $low = $mid + 1;
        }
    }
    return false;
}
```

1.4 Overview

This thesis is split into 8 chapters. Chapter 1: Introduction gives an overview of the thesis, and defines the research questions and goals of the thesis. Chapter 2: Background goes over relevant background that is needed in order to understand our thesis better. Chapter 3: Model describes the theory behind the code2vec model, explains in more detail about how the model works, and how we used the model. Chapter 4: Implementation of data processing and model describes how all the data processing was implemented. Chapter 5: Evaluation explains how the model was set up, what metrics were used, the results of the model, and how it compares to other code analysis tools that exist for PHP. Chapter 6: Discussion and Future work contains discussion of the results, what improvements could be made to the model, as well as presenting opportunities of future work. Chapter 7: Related work will go over similar types of studies, that have been using other approaches of deep learning in order to do type inference. Chapter 8: Conclusions presents our conclusions of the thesis.

Chapter 2
Background

In this chapter, we will cover the theory of different concepts that are used in this thesis. Section 2.1 describes what PHP is and some parts that are important to know in order to understand this thesis. Section 2.2 gives a quick explanation of what machine learning is and introduces neural networks. Section 2.3 explains what embeddings are, and why they are useful. Section 2.4 describes context-free grammar, section 2.5 describes parse trees and abstract syntax trees and Section 2.6 presents the code2vec paper, which is the paper this thesis is based on.

2.1 PHP

PHP [8], which recursively stands for PHP: Hypertext Preprocessor, is a dynamically typed scripting language that was originally created in 1994. PHP is mostly used for web development, can use many different databases and can be embedded into HTML, among other things. Compared to JavaScript, which normally runs client-side (Node.js runs on server-side), PHP code is executed on a server, and outputs the results into HTML which is then sent to the client.

PHP has four primary types: `boolean`, `integer`, `float` (aka `double`), and `string`. There are also compound types, such as `array` or `object`; special types, such as the `null` type; and pseudo-types, such as `mixed`, which allows the variable to have any type. Like other object-oriented languages, PHP supports classes, which allows the developer to create more specific types.

As PHP is a dynamically typed language, type on variables and function parameters are set at runtime, which means that there is no need to put type annotations in the code. As of PHP 7.2+, which was released the 30 November 2017 [8], the language supports parameter type annotations and return type annotations, which allows the developer to define parameters and function return types in a more static way. However, variable type annotations are not supported until PHP 7.4+, which was released the 28 November 2019 [8]. In this thesis, we

will only look at parameter type annotations.

An example of php with and without type annotations can be seen in listing 2.1 and 2.2.

Listing 2.1: PHP code example

```php
function factorial($n) {
    $factorial = 1;
    while ($n > 0) {
        $factorial =
            $factorial * $n;
        $n--;
    }
    return $factorial;
}
```

Listing 2.2: Annotated PHP example

```php
function factorial(int $n) {
    $factorial = 1;
    while ($n > 0) {
        $factorial =
            $factorial * $n;
        $n--;
    }
    return $factorial;
}
```

2.2 Machine Learning and Neural Networks

Defining what machine learning is is hard, since the field of machine learning is very broad. Generally, machine learning is a sub field of AI that uses algorithms with the capability of learning from previous experiences. A machine learning model needs to be trained in order to function, which can be done using sets of training data and test data. There are three different types of learning when it comes to machine learning models, which is supervised learning, unsupervised learning, and semi-supervised learning [18, p. 694-695]. In supervised learning, the model gets pairs of input and output, so that the model learns to map inputs to a given output. In unsupervised learning, the model is only given input, and has to detect potentially useful patterns in the input by itself. Semi-supervised learning is a mix of supervised learning and unsupervised learning, where the model uses the a supervised approach if there is a corresponding output to an input, and an unsupervised approach if there is missing feedback. In this thesis, we use supervised learning.

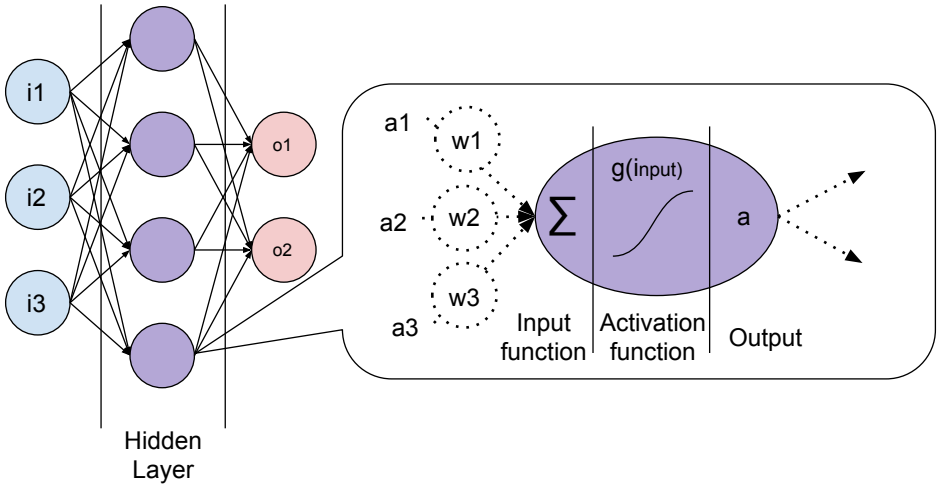
The sub field of machine learning that is used in this thesis, called *deep learning*, tries to simulate learning by mimicking how a brain learns. Deep learning is a group of methodologies which uses multiple levels of non-linear representations in order to represent more abstract representations [12]. Deep learning is in practice implemented using large *neural networks*. As a brain has neurons that are connected via synapses, a neural network has nodes, representing the neurons, that are connected to each other using edges, representing the synapses. To resemble how important a synapse is for a neuron, a neural network uses weights to give values for the edges between each node, which can be tweaked, simulating learning in the network.

A neural network is usually implemented as a series of nodes connected to each other via edges. A small example of a neural network can be seen in figure 2.1.
Each edge connects a node i to a node j, and each node creates a weighted sum of the input edges:

$$\text{input}_j = \sum_{i=0}^{n} w_{i,j} a_i \tag{2.1}$$

where w_i is the weight of the edge and a_i is the output from the activation function of node

Figure 2.1: Small example of a neural network. The content of a node is also shown



i. The node then sends an output to its output edges if the activation function is triggered:

$$a_j = g(\text{input}_j) = g(\sum_{i=0}^{n} w_{i,j} a_i) \tag{2.2}$$

The activation function g in each node is most often a hard threshold function, but can also be a logistic function. An example of an activation function would be:

$$g_w(x) = \text{Threshold}(w \cdot x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

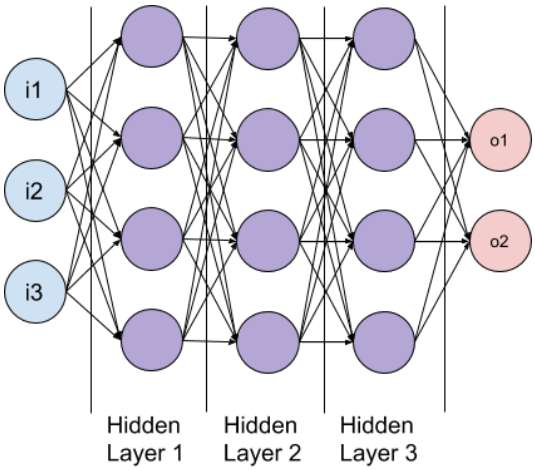
The input of a neural network is most often represented by either a matrix or a vector containing numbers, which represents the *features*, or values, of the input. Each element in the matrix/vector is passed to the corresponding input node. For the output, the shape can be changed depending on how the output is constructed. It can be a single natural number, which could represent a class in classification, or a single real number in the case of regression, or a vector of real numbers, or a vector or natural numbers etc. The output is dependent on how the output layer is constructed, and can therefore vary in size.

According to Stuart Russel and Peter Norvig [18, p. 729] there are two fundamentally distinct ways of structuring a network. The first one, a *feed-forward* network, only sends its output in one direction, which means that each node only sends its output to nodes further down in the network. Since there are no loops in a feed-forward network, the nodes do not have to store an internal state, and can fully rely on the weights from its input. The other network structure, a *recurrent* neural network, sends its own output into its input, therefore creating loops. Since the output from a node can be influenced by its previous output, the nodes have to store an internal state, which allows a recurrent neural network to support short-term memory.

For feed-forward networks, the nodes are often sectioned into layers, so that nodes in a layer can only receive input from nodes in previous layers. Layers between the input layer

and the output layer are often referred to as hidden layers, since the weights of the nodes in the layer can not be directly observed. A feed-forward network can therefore be visualized like in figure 2.2.

Figure 2.2: Example of a feed-forward network



2.2.1 Learning In Networks

In order for a neural network to learn, the network needs to be able to know the error of its output so it can change its weights. This is done by using a *loss function*. There are many different loss functions, such as the absolute error loss, mean square loss, or the cross entropy loss:

$$\text{CrossEntropy}(x) = -\sum_{i=0}^{n} y_i \log(a_i)$$

In neural networks, these loss functions are used in algorithms called *optimizers*, which are algorithms for using the loss and changing the weights of the nodes in the network by a process called back propagation. There are many optimizers, and one of the most common ones is the Adam optimizer. We will not go into more detail on the back propagation, but if you are interested in the topic, we would recommend *Artificial Intelligence*: *A Modern Approach* by Stuart Russel and Peter Norvig [18] and *Adam: A Method for Stochastic Optimization* by Diederik P. Kingma and Jimmy Ba [10].

2.3 Embeddings

Since machine learning models use numbers to represent features in its input, the text has to be converted into numbers. But simply hashing the text into a number is not enough, as the risk of collisions for the hash function becomes very large. Using a single number for a word would also hide all the similarities the word has to other words, such as small and smallest. Instead, a vector of numbers is generated for each word in the text. These vectors, called

embeddings, are then continuously changed by the back propagation of the machine learning mode, in order for the model to learn how each word is related to each other. A common way of creating embeddings for words in machine learning is to use word2vec [14]. With embeddings for each word, similar words get similar embeddings, which makes it easier to connect similar words to each other. With embeddings, similar words can be found using simple arithmetic, e.g., $vector("King") - vector("Man") + vector("Woman") \approx vector("Queen")$, which in turn makes it easier for the machine learning model to learn relations between words.

2.4 Context-Free Grammar

In order for a parser to be able to read code, it needs to follow a set of rules. These rules describe the structure of the language that the code is written in, like how there are grammatical rules for a real world language, e.g., English. For compilers, these rules are called *context-free* grammars. Andrew W. Appel and Jens Palsberg [4, p. 40-42] has a good definition of what context-free grammar is and it can be summarized as this:
For any language, a context-free grammar describes the language as a set of productions of the form

$$\text{symbol} \longrightarrow \text{symbol symbol} \ldots \text{symbol}$$

where the amount of symbols on the right hand side is zero or more. The symbols can either be terminals or nonterminals, where a terminal symbol is a token from the alphabet defined by the language, and a nonterminal symbol is a symbol that appears on the left-hand side of any production. A terminal symbol can never appear on the left-hand side of a production, and at least one of the nonterminal symbols must be distinguished as the start symbol of the grammar.

Using a context-free grammar, the parser can check if a sentence is part of a language, by starting from the designated start symbol, and repeatedly applying the appropriate production rules until the given sentence is derived.

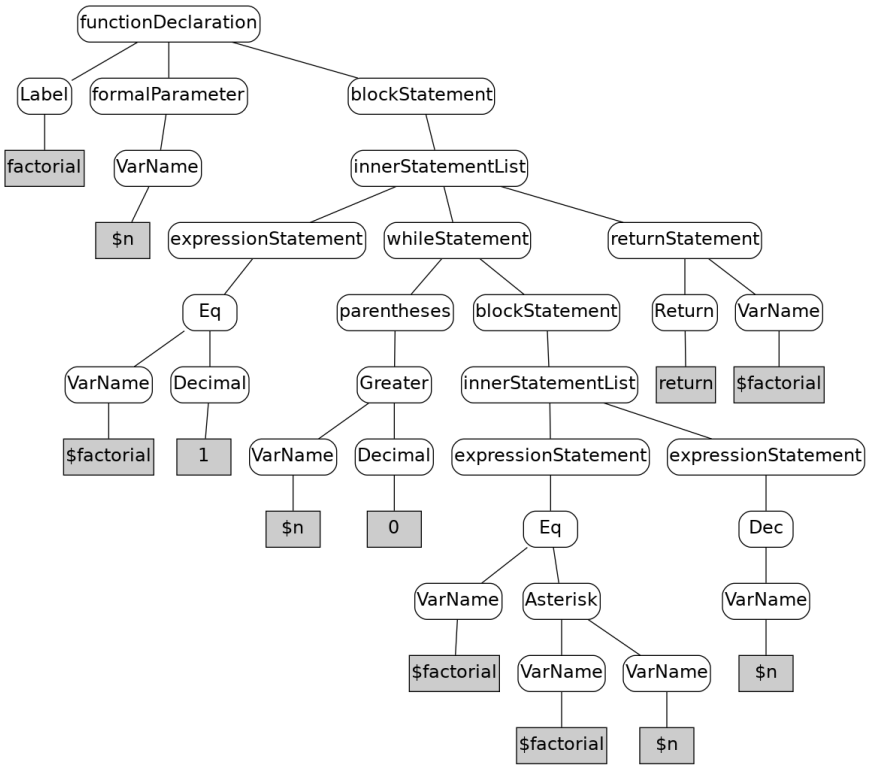
2.5 Parse Trees and Abstract Syntax Trees

A parse tree is a tree representation of source code written in a programming language according to some context-free grammar [4, p. 42]. Parse trees are made by connecting symbols of a derivation to the symbol that it was derived from. If the grammar is constructed well enough, the parser can derive the structure without ambiguity, which means that the parser can not create two different parse trees from the same code. An example of a parse tree can be seen in figure 2.3, which is based on the code in listing 2.1.

Each node in a parse tree represents nonterminals in the context-free grammar (e.g., for-loop, if-statement, variable declarations etc.). Each leaf in the tree represents a terminal (e.g., integer, boolean, string etc.). For example, an if-statement can be represented by a node with two children, where one child contains the boolean expression, and the other child contains the block of the if-statement.

A sibling to the parse tree is the abstract syntax tree (AST) [4, p. 89-91], which looks similar to the parse tree, but there are a few key differences. An AST removes all inessential punctuation and delimiters such as parentheses, braces, semicolon etc. This means that e.g.,

Figure 2.3: Simplified example of a parse tree. In this tree the values of the leaves are shown in the grey boxes, and the leaves themselves are parents to the values.



an if-statement only consists of the content of the statement, and not of the parentheses or braces. An AST also only consists of essential nodes, so the tree itself is much smaller than the parse tree, which makes it easier to work with. Normally, a compiler uses ASTs instead of parse trees in order to operate more efficiently.

2.6 Path-Based Code Representation

In April 2018 Alon et al. presented a new approach, which uses a path-based code representation, for predicting properties and attributes within source code [2]. This approach is based on the idea of representing code snippets as a set of paths in its abstract syntax tree as opposed to representing it as a sequence of characters or tokens. These paths between nodes in the abstract syntax tree are then used as input features when training a model to predict program properties. The theory behind this approach is that an abstract syntax tree will capture the syntactic structures of source code in a better way for a learning model to take advantage of.

To evaluate their approach of using paths in the abstract syntax tree as features they tested it for three different tasks and in a few different languages (JavaScript, Java, Python and C#). Those tasks were prediction of variable names, prediction of method names and prediction of data types for expressions. Each task was then tested and evaluated with two different

models, one that was based on conditional random fields (CRF) and one that was based on Word2Vec.

With their best model they achieved an accuracy of around 60 % for all tested languages for the variable name prediction task. For the method name prediction task they achieved an accuracy of around 50 % for all tested languages. The task of predicting data types for expressions was only tested for Java for which they achieved an accuracy of 69.1 %.

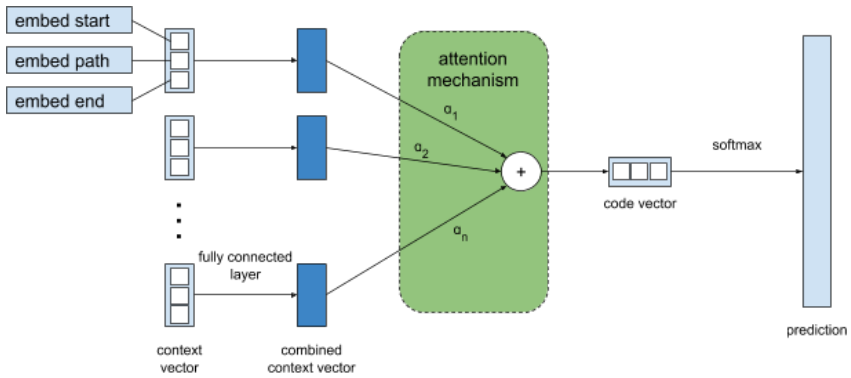
Later the same year Alon et al. published code2vec [3], where they presented an advancement of their previous approach. In this new and improved approach they present a model that can aggregate a collection of paths from an abstract syntax tree into a fixed-length code vector. The idea is that a code snippet can be represented using code embeddings. How their model accomplished this will be described in more detail in chapter 3 as this is the approach we took in our thesis for the task of predicting parameter types in PHP.

When Alon et al. evaluated this improved approach on the task of predicting method names in Java they achieved a precision of 63.1 %, a recall of 54.4 % and a F1-score of 58.4 %.

Chapter 3
Model

In this chapter we will describe how the model used in code2vec [3] works, and how we applied it to the problem of type inference in PHP. An overview of the model can be seen in figure 3.1 and each part of it will be described in detail in its own section of this chapter. Section 4.1 describes how to represent code using AST paths, and how those paths were filtered to be used in our approach. Section 4.2 describes how embeddings are used to convert the input into vectors. Section 4.3 describes how the fully connected layer of the model works. Section 4.4 describes the attention mechanism and section 4.5 describes how the predictions are made.

Figure 3.1: Overview of the model

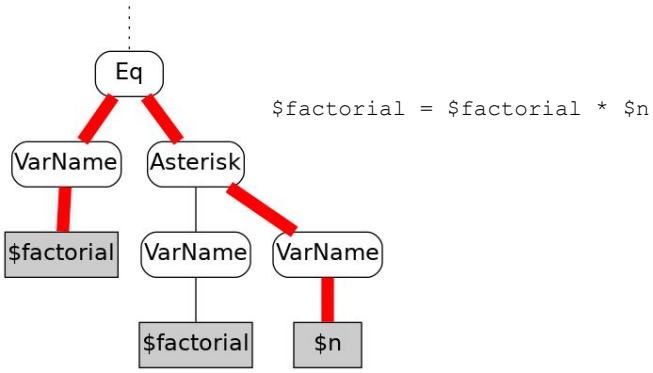


3.1 Representing Code as AST Paths

The main contribution from the work of Alon et al. [2], which our thesis is built upon, is the idea of representing code using paths from its AST. The first step is therefore to convert the

code into syntax trees and to extract these paths. Alon et al. has a clear definition of what an AST-path is in their paper, but in short, it is a sequence of connected nodes in the AST, where each node is combined with a direction in the tree (UP or DOWN). Only paths between leaf nodes in the tree are extracted and considered in this implementation and every such path is then combined with the actual tokens corresponding to the leaf nodes it connects. This combination of a start token, a path and an end token form a *path-context*. An example of such a path-context can be seen in figure 3.2. The same path-context in its textual form would be:

Figure 3.2: Example of a path-context in a subtree of an AST



$factorial, (VarName UP; Eq UP; Eq DOWN; Asterisk DOWN; VarName DOWN)**, $n**

Where $factorial is the token corresponding to one of the leaf nodes, and $n is the token corresponding to the other leaf node.

While Alon et al. uses code2vec to predict the name of methods in Java, we want to adapt the same approach and model and use it to predict type annotations for method and function parameters in PHP. This means that we need to adjust the selection of the path-contexts to better suit our purposes. Since Alon et al. aimed to predict the name of methods, they wanted to use path-contexts that captured as much of the syntactical structure and information as possible. This was done by extracting all the path-contexts that could possibly be found in the AST for a method. These path-contexts would then hopefully contain enough information to be able to summarize the method into a single method name. We on the other hand only want to capture the usage of the parameter in order to summarize it with a single type annotation. This was done by only extracting the path-contexts that satisfied the following two rules for the given parameter:

- The path-context's start or end token has to be the name of the parameter
- The path-context's start and end token has to begin with $, a letter, or a digit

These rules will filter out the path-contexts that are relevant for the purpose of predicting the type annotation for the parameter. By only including path-contexts that start or end with the parameters name we can focus on the path-contexts that capture how the parameter

is used. Since each parameter is checked independently, this will also make sure that two parameters for the same method are distinguishable as they will result in different sets of path-contexts. The second rule is then enforced to avoid paths that lead to uninformative tokens such as parentheses and semicolons.

3.2 Embeddings

The next step is to transform the extracted path-contexts into a format that can be understood and handled by the neural network. This is done with the help of two separate embeddings, one that is used for the start and end tokens and one that is used for the paths. The dimensionality of the embedding spaces for both of these embeddings are set to 128, which was the dimensionality that worked the best for code2vec. This means that that every input value to the network, i.e., start token, path and end token, will be embedded into a vector in $\mathbb{R}^{128}$. Thus, every path-context will be embedded as three vectors.

The two embeddings are initially assigned with random normally distributed numbers as weights which are then learned while training the network. The goal here is for the embedding to learn the "meaning" of the tokens and paths and distribute that meaning across the 128 vector components in a way such that tokens or paths with similar meaning will be embedded into similar vectors.

3.3 Fully Connected Layer

Once each part of the path-contexts in the input has been embedded into vectors they will be joined into context vectors representing each of the path-contexts. This is done by simply concatenating the corresponding embeddings for each of the three parts of the path-context which will result in a new vector with a dimension of 384. This new vector, called a context vector, is then sent through a fully connected layer.

The purpose of this fully connected layer is for the network to learn how to combine the different components of the context vector in the best way possible. This is necessary since some tokens might have a bigger effect on the output when seen in combination with certain tokens and paths than with others. By allowing the fully connected layer to learn this it will be able to attend different combinations of tokens and path differently later on.

The activation function of the fully connected layer is the hyperbolic tangent function, which means that the fully connected layer can be described as:

$$\hat{c} = \tanh(W \cdot c) = \frac{e^{W \cdot c} - e^{-(W \cdot c)}}{e^{W \cdot c} + e^{-(W \cdot c)}} \tag{3.1}$$

Where c is the context vector, W is the learned weights in the layer and $\hat{c}$ is the output, the combined context vector.

3.4 Attention Mechanism

One difficulty when dealing with problems that generate large sets or bags of features is the selection of which of the features to use as input for the model. Including all of them would be

one solution, but that could easily lead to overfitting [18, p. 736]. Selecting only some of them would be problematic as well, as it can be difficult to know which of the features to use. There is even a possibility that some features might be important to make the correct prediction for some input, while some completely different features might be more important to make the correct prediction for a different input.

When given a large set of path-contexts as input, the network will therefore have to select which of the path-contexts to use. This is done by the attention mechanism which is the main component of this path-attention network which was originally proposed by Alon et al. This way, it is possible to feed all extracted path-contexts to the network and then let the attention mechanism decide which of the path-contexts to pay attention to. The attention mechanism takes the combined context vectors as input and uses soft attention to decide how much focus each of them should be given. Soft attention here means that the network will consider all the combined context vectors and combine them using a weighted sum where more important vectors will be given more weight. This can be compared to hard attention which instead only would consider the single most important vector and use that one.

Given a set of combined context vectors, the attentions mechanism will therefore combine them into a single, fixed length vector as:

$$v = \sum_{i=1}^{n} \alpha_i \cdot c_i \tag{3.2}$$

Where α_i is the attention weight corresponding to the combined context vector c_i. These attention weights are calculated as the softmax of the dot product of the combined context vectors, c_i, and the global attention vector a:

$$\alpha_i = \frac{\exp(c_i \cdot a)}{\sum_{j=1}^{n} \exp(c_j \cdot a)} \tag{3.3}$$

Similar to the embeddings, the global attention vector a is initialized as a vector of random normally distributed values and learned when training the neural network.

The output of the attention mechanism is the code vector, v, which represents all the path-contexts in the input, and therefore also the usage of the function or method parameter, as a single vector.

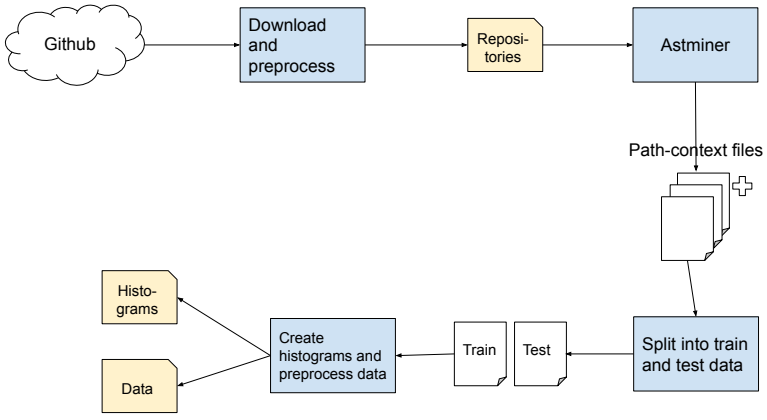
3.5 Predictions

The last step is to predict the type annotation for the given parameter which is done using the resulting code vector from the attention mechanism. The *logits*, or raw predictions, are first computed as the dot product between the code vector and each of the embeddings for the type annotations, which are learned while training the network. These logits are then normalized with the softmax function to get the probability distribution over all the type annotations. The final prediction will then be the type annotation with the highest probability.

Chapter 4
Implementation of Data Processing

Before any tests or evaluations could be performed we needed data in order to train the model. In this chapter we will describe how this data was gathered and processed, all the way from raw source code until data that could be used as input to the model. An overview of the entire data processing can be seen in figure 4.1 and each part of it will be described in further detail in this chapter. Section 4.1 explains how the data was gathered. Section 4.2 describes how the raw source code was preprocessed. Section 4.3 describes how the path-contexts were extracted and section 4.4 describes the final preprocessing of these extracted path-contexts.

Figure 4.1: Overview of the data processing



4.1 Data Gathering

In order to train a model to predict type annotations in PHP we first needed data for it to learn from. We therefore had to gather a large quantity of PHP code with existing type

annotations. This was done by collecting projects from Github that predominantly consisted of PHP code. Since we assumed that more popular projects were more likely to be better documented and use type annotations to a greater extent, as well as containing more code, we decided to sort the repositories by star count. The number of projects in our dataset grew over time, but ultimately, a dataset consisting of the top 10,000 repositories, sorted by most stars, was used.

4.2 Data Processing

Before any path-contexts could be extracted from the collected data, it had to be preprocessed. This was done for several reasons. The main three reasons were to perform some basic lemmatization of the type annotations, to save storage space, and to move type annotations from PHPDoc comments to the function or method headers. Most of this preprocessing was done using PHP-Parser [16], a tool which allowed us to generate an AST for each PHP file and then traverse the tree while performing some modifications to it.

One of the objectives for the preprocessing was to lemmatize and combine some of the type annotations. During the collection of the data we noticed that some type annotations were written differently in different repositories since PHP allows for several spelling variations of them. For example, some projects used `int` and `bool` as type annotations while other projects used `integer` and `boolean`. To prevent the model from trying to learn how to distinguish between these type annotation we therefore replaced all `integer` and `boolean` type annotations with `int` and `bool`. Similarly, all the capitalized scalar type annotations were replaced by their lowercase versions. E.g., replacing `String` and `Array` with `string` and `array`.

Another objective of the preprocessing was to generalize the code slightly and prevent the model from learning from specific values in the code, such as string and int literals. We don't want the model to treat an addition with 5, for instance, different from an addition with 7, since the specific value is unlikely to matter for the prediction of the type annotation. Neither do we want it to treat a string concatenation with "Hello" different from a string concatenation with "World" etc. While traversing the AST using PHP-Parser, we therefore replaced all integer literals with the value 0, float literals with the value 0.0, string literals with the string "s" and inline HTML blocks with an empty HTML block.

The preprocessing was also used to standardize the placement of type annotations across the dataset. Since different projects used different conventions, some projects placed the type annotations in the documentation comment while other projects placed them directly in the header of the function or method. This standardization was done by moving all type annotations found in documentation comments to the header. A placement that would make them easier to access when extracting the path-contexts and creating the data samples for the model to learn from.

To prevent code duplication in the dataset, the preprocessing was also responsible for removing all folders named vendor as these folders contain the code dependencies. By removing these folders we could prevent cases where several repositories had a mutual dependency. Thus, eliminating duplicated code. Removing these folders, as well as removing any files that were not PHP files, also saved storage space as it drastically reduced the size of the repositories in the dataset.

4.3 Extraction of Path-Contexts

Once all the repositories were preprocessed, the next step of the data processing was to extract the path-contexts. This was done using astminer, a library for mining path-based representations of code [11]. However, several modifications of astminer were necessary in order to adapt the tool for our purposes and to extract the path-contexts in the desired format.

The main modification we had to make was to add support for parsing PHP files. This could be done by either implementing a PHP parser from scratch or by generating a parser using an ANTLR grammar file and only implementing a wrapper around the generated parser. As writing your own parser can be a complex and time consuming project we decided to generate a parser using a predefined grammar file.

Unfortunately, the generated parser was only able to convert code into parse trees, and not into abstract syntax trees. While a parse tree is a more detailed representation of the syntactic structure of the code, an AST is more compact while still containing all the essential information. This means that by extracting the path-contexts from an AST, the use of a parameter could be represented in a more compact and concentrated way since fewer path-contexts would be needed to represent the same amount of information. For this reason we had to modify astminer further in order to treat the generated parse tree more like an AST.

One such modification was to filter out and ignore path-contexts with a start or end token that would have been omitted in an AST. This way, path-contexts containing uninformative tokens such as parentheses, semicolons and braces could be disregarded.

Another modification that was done was to change how unary and binary expressions were represented in the parse tree and to make these parts of the tree more similar to an AST. In a parse tree, an addition, for instance, is represented as a binary expression node with three children, one for the operator and one for each operand. This means that we would need two path-contexts to cover both of the operands and the operator. By changing it and representing expressions in a way that is more similar to an AST, an addition would instead be represented as an addition node with two children, one for each operand. This makes it possible to cover both of the operands and the operator with a single path-context.

Once all of the necessary modifications had been done to astminer, it could be used to extract the path-contexts. This was done by giving the repositories in the dataset, one at the time, as input to astminer. Each PHP file in a given repository was then parsed and all path-contexts representing the use of a parameter with a known type annotation were extracted. These were then written to a separate line of a file together with the type annotation of the corresponding parameter. Thus, producing a file with one data sample per line. To prevent this file from being excessively large, the path of each path-context was first hashed and only the hash value was written to the file, making each path-context more compact.

The result of this step of the data processing was therefore 10,000 files, one per repository, containing data samples for the model to learn from. These files could then be split into a training dataset and a testing dataset.

4.4 Final Preprocessing

In order for the model to be able to handle any input, it needed to know what input values it could expect and which type annotations it could chose between when making the predic-

tions. In other words, it needed vocabularies containing all the possible values for the tokens, paths and type annotations.

These vocabularies could be created in different ways. One option would have been to include every value that could be found in the training dataset. However, that would also include values that only appeared a handful of times. Since it would be difficult to detect any patterns and learn anything useful from these values we decided to omit them. Instead we created histograms representing the value frequencies and introduced a threshold such that only values with a frequency above the selected threshold were added to the vocabulary. This way we could prevent the model from trying to learn anything from the most uncommon values. Several different threshold values were tested, but ultimately it was set to 10 as this seemed to give the best results.

The final preprocessing step was also responsible for limiting the number of path-contexts per sample to a maximum of 200. For any sample exceeding this limit, a selection of 200 path-contexts would be kept while the rest would be discarded. To prevent any path-contexts with valuable information to be discarded, priority would be given to those where the path and tokens were part of the vocabulary. The value 200 for the limit was selected since this was the value that had worked the best for Alon et al. in their work of code2vec [3].

Once the final preprocessing was done, the data could be used to start training the model.

Chapter 5
Evaluation

In this chapter, we will present the results of the model, as well as evaluating the model and comparing it with current code analysis tools in PHP. Section 5.1 will go over the setup for the model. Section 5.2 covers the metrics used for the evaluation. Section 5.3 presents the results for the model. Section 5.4 contains a qualitative evaluation of the model, that looks at how the model works on repositories that have not been seen in the training data. Section 5.5 compares the model with current PHP code analysis tools and section 5.6 covers code analysis tools that were tested but were not used for comparisons with the model.

5.1 Experimental Setup

The preprocessing was done on Intel® Core™ i7-9750H CPU @ 2.60GHz × 12 processor. For a dataset of 10,000 repositories, the preprocessing took ~12 hours, depending on the path depth astminer was configured with. The neural network was trained on a Nvidia Tesla T4 with 8 epochs per model. Each epoch for the dataset took ~50 min, which resulted in a training time of ~7 hours.

5.2 Metrics

To evaluate the model, we used k-fold cross validation [18, p. 708] in order to get a result that can be generalized better, since splitting the data into only one training set and one test set might give a poor estimate of the general performance of the model. The evaluation was done with k = 10, which means that the dataset of all repositories, R, was first split into ten disjoint subsets T_1, T_2, ... ,T_{10}. Each subset T_i was then used as the test set in an independent round of training using the training set $R \setminus T_i$. Once all ten rounds of training were done, the result was calculated as the average for each metric between the ten rounds. The metrics used were the top-k accuracy, precision, recall and F1-score.

The top-k accuracy of the model is defined as:

$$\text{Top-k accuracy} = \frac{C}{D}$$

Where C is the number of predictions where the correct type is within the top k predictions, and D is the total number of data samples in the test set.

The precision for a specific type annotation is defined as:

$$\text{Precision}_{\text{type}} = \frac{C_{\text{type}}}{P_{\text{type}}}$$

Where C_{type} is the number of correct predictions for the specific type annotation, and P_{type} is the total number of times the model predicted the type annotation.

The recall for a specific type annotation is defined as:

$$\text{Recall}_{\text{type}} = \frac{C_{\text{type}}}{D_{\text{type}}}$$

Where C_{type} is the number of correct predictions for the specific type annotation, and D_{type} is the total number of data samples of the type annotation.

The F1-score for a specific type annotation is defined as:

$$\text{F1-score}_{\text{type}} = 2 \cdot \frac{\text{Precision}_{\text{type}} \cdot \text{Recall}_{\text{type}}}{\text{Precision}_{\text{type}} + \text{Recall}_{\text{type}}}$$

In other words, the F1-score for a type annotation measures the harmonic mean between the precision and the recall for that specific type annotation.

The overall precision, recall and F1-score for the model after a round of training was then calculated as the weighted average for these metrics for each type annotation. Each of these values, as well as the top-k accuracy, was then averaged over the ten rounds of training to obtain the final evaluation measurements for the model.

5.3 Results

After training the model with the ten different splits of the data, we got an average top-1 accuracy of 76.2 %, and an average top-3 accuracy of 84.2 %. An overview of the results can be seen in table 5.1.

Table 5.1: Evaluation metrics for different models

Path-contexts used	Top-1 accuracy	Top-3 accuracy	Precision	Recall	F1
Only local contexts	76.2	84.2	72.7	76.2	73.2
Local + whole file contexts	75.3	84.0	71.3	75.3	72.1

To get a better understanding of how the model performs for larger functions and methods in comparison to smaller ones, we created a plot visualizing how the accuracy depends

on the number of path-context in the samples. The assumption was that fewer path-contexts could be extracted from smaller functions and methods, giving the model less information to base its prediction on. Thus, resulting in a worse accuracy for these samples. As can be seen in figure 5.1a, the assumption seems to be correct as the accuracy appears to be slightly worse for samples with a number of path-context below 50.

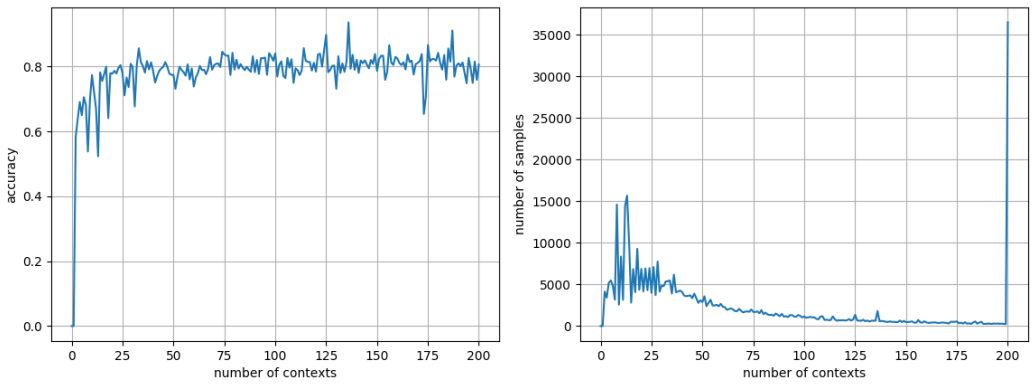
Looking at figure 5.1b, which shows how the samples are distributed across the number of path-contexts they consist of, we can see that a lot of the samples have rather few path-contexts. This means that the majority of the samples falls within the range with a slightly weaker accuracy. The spike at 200 in the graph occurs due to the threshold set for the maximum number of path-contexts per sample.

To remedy this problem we made an attempt at increasing the number of path-contexts in each sample. This was done by considering entire files when extracting path-contexts as opposed to looking at each function or method independently. Since we assume that several methods within the same class might share a parameter, we could extract more path-context by looking at how parameters with the same name are used in other methods. With this approach we obtained a top-1 accuracy of 75.3 %, and a top-3 accuracy of 84.0 %. However, as can be seen in table 5.1, this approach did not make much of an improvement to the results, which indicates that our assumption might have been wrong.

Figure 5.1: Graphs surrounding samples

(a) Accuracies depending on amount of path-contexts in a sample

(b) Distribution of samples in test data

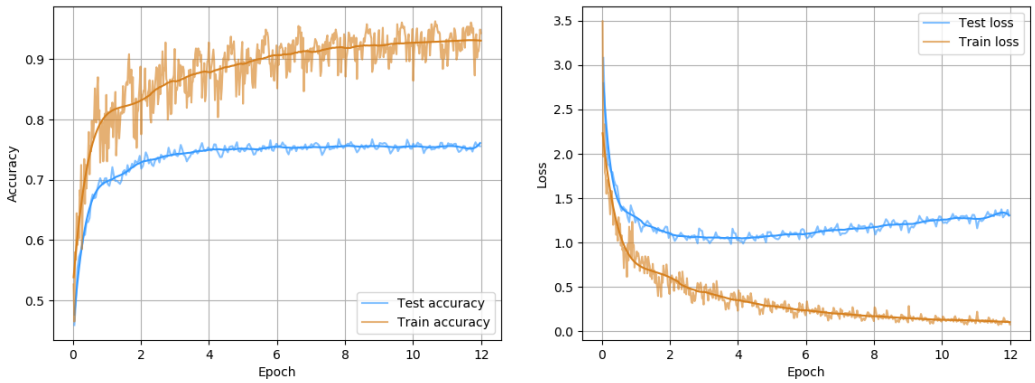


In figure 5.2a, we can see that the accuracy for the training data keep improving the longer we train the model. The accuracy for the test set on the other hand seems to stabilize at around 76 % after approximately six epochs. Similarly, by looking at figure 5.2b, we can see that training loss keeps decreasing the longer the model is trained while the test loss reaches a minimum after about five epochs before it starts to increase. This reveals that the model will start to overfit to the training data if it is trained for too many epochs. It also suggests that five or six epochs might have been better than the eight epochs we decided to train for. However, these graphs only show the accuracy and loss for one individual round of training. After performing all ten rounds of training it turned out that eight epochs gave the best results.

Figure 5.2: Graphs surrounding training and test data

(a) Accuracies of training and test data over epochs

(b) Cross entropy loss of training and test data over epochs



We also looked at how the model performs when predicting the type annotation for parameters whose names had not been observed during training. This way we would get an understanding of how dependent the model is on good parameter names in order to make accurate predictions. Since every path-context extracted contained the parameter name as either the start token or end token we could evaluate this by modifying the data in the test set. By changing all tokens containing a parameters name to a new, previously unseen token, we could create a new test set which simulates samples for parameters with unknown names.

On this test set the model was able to predict type annotations with a top-1 accuracy of 55.9 % and a top-3 accuracy of 67.3 %, as can be seen in table 5.2. These results are significantly worse than the results we obtained with the normal test set, which indicates that the model is dependent on good and descriptive parameter names in order to make accurate predictions.

Table 5.2: Metrics for different models based on path-context structure

Path-context structure	Top-1 accuracy	Top-3 accuracy	Precision	Recall	F1
Full path-context	76.2	84.2	72.7	76.2	73.2
Unknown parameter name	55.9	67.3	49.9	55.9	50.0

Another aspect we looked at was how the model performs at different certainty thresholds, where the model has to reach a certain certainty on the prediction in order for the prediction to be valid. As can be seen in table 5.3, the higher the threshold is the fewer predictions will be made. With a certainty threshold of 99 %, for instance, it will only be able to make a prediction for approximately half of the samples. However, it will have an accuracy of 97 % on those predictions. With a threshold of 50 % on the other hand, it will be able to make a prediction for as many as 87.3 % of the samples, but with a lower accuracy.

Table 5.3: Accuracy of the model depending on certainty threshold

Threshold	Accuracy	Samples predicted	Percentage of samples predicted
50 %	84.5	439,433	87.3
90 %	93.2	348,937	69.3
99 %	97.0	261,869	52.0

5.4 Qualitative Evaluation

To analyse the usefulness of the model in a developer scenario, we used the model to predict types on a repository that has not been seen in the training data, and manually checked if the predictions were useful. An useful prediction would be a prediction which is either correct, has the correct type annotation within the top-3 predictions, or has a type annotation within the top-3 predictions which is close enough to the correct type annotation. We checked 200 functions, and classified the correctness into three categories: correct prediction, useful prediction, and wrong prediction. A correct prediction was a prediction where the correct type annotation of the parameter was the top-1 prediction. A useful prediction was a prediction where the type annotation was not the top-1 prediction, but was in the top-3 prediction or was close enough to the correct type annotation that it could give some clues. A wrong prediction was a prediction where the type annotation was not within the top-3 prediction. From the manual testing, we found that 116 predictions were correct, 51 predictions were useful, and 33 predictions were wrong, which would give an usefulness percentage of 83.5 %.

To show what the model actually outputs, we want to show some examples of where the model has correctly predicted the type annotations and some cases where it predicts the wrong type annotation. All code examples were taken from an older version of Axis PHP code, and the type annotations in the functions are kept in order to check if the model is correct or not. The types that are left in the parameter declaration and in the comments are not used in the predictions themselves.

Our first example, which can be seen in listing 5.1, shows a small function, where the model predicted the correct type on the parameter `$audioInputId`. All the predictions the model makes is inserted into the PHPDoc comment of the function, instead of directly into the function. This lets the developer decide what type annotation that should be there, instead of letting the model insert a type annotation that might be wrong. We also insert the certainty of the prediction, so that the developer can get more information of what the model thinks, in order to make better decisions. In this example, the model correctly predicts that the parameter `$audioInputId` should be a string with a certainty of 99.59 %.

Listing 5.1: Small code example where the model predicted the right type

```
1  /**
2   * Determines if the user is authorized to update an audio input
3   *
4   * @param string $audioInputId The id of the audio input.
5   * @return bool - True if user is authorized, False otherwise.
6   *
7   * Predictions for $audioInputId
8   * 1. string 99.59 %
9   * 2. mixed 0.07 %
10  * 3. array 0.05 %
11  */
12 public function updateRule(string $audioInputId) : bool
13 {
14     $explodedAudioInputId = explode('-', $audioInputId);
15     $deviceId = $explodedAudioInputId[0];
16     return $this->permissionChecker->verifyDeviceACE($deviceId, '
           DEV_MANAGE_AUDIO');
17 }
```

The second example, shown in listing 5.2, shows a larger example with several parameters, many with differing type annotations. The model correctly predicts the type annotations on all the parameters, and each prediction is done in the order that the parameter is implemented. The prediction is also inserted in the order the parameters are presented, to easily see which prediction corresponds to which parameter.

Listing 5.2: Large code example where the model predicted the right type on all parameters

```php
/**
 * Builds a LIMIT/OFFSET clause.
 *
 * @param array $params Reference to the parameters for the prepared statement which is populated with the values.
 * @param array $types Reference to the types for the prepared statement which is populated with the types of each value.
 * @param int $limit Max number of results
 * @param int $offset Offset from first match
 * @return string SQL clause or empty string if $limit and $offset is null.
 * @throws InvalidArgumentException if $limit is null while $offset is not null.
 *
 * Predictions for $params
 * 1. array 99.99 %
 * 2. string_arr 0.01 %
 * 3. array_null 0.0 %
 *
 * Predictions for $types
 * 1. array 99.8 %
 * 2. string_arr 0.15 %
 * 3. array_null 0.01 %
 *
 * Predictions for $limit
 * 1. int 77.61 %
 * 2. int_null 11.69 %
 * 3. string 6.54 %
 *
 * Predictions for $offset
 * 1. int 99.6 %
 * 2. int_null 0.16 %
 * 3. string 0.1 %
 */
public function buildLimitOffset(array &$params, array &$types, $limit = null, $offset = null)
{
    if (null === $limit && null !== $offset) {
        throw new \InvalidArgumentException('Offset␣requires␣a␣limit');
    }
    $sql = '';
    if (null !== $limit) {
        $sql .= 'LIMIT␣?';
        $params['limit'] = $limit;
        $types[] = 'i';
        if (null !== $offset) {
            $sql .= '␣OFFSET␣?';
            $params['offset'] = $offset;
            $types[] = 'i';
        }
    }
    return $sql;
}
```

But our model is not always correct. Our third example, which can be seen in listing 5.3, shows a small function where the model predicted the wrong type annotation with a high certainty. An explanation to why the model predicts the wrong type annotation might be that the model has seen $siteId or something similar before, but the variable has mostly been used with integers instead of strings.

Listing 5.3: Example where the model predicted the wrong type

```
1   /**
2    * Verifies if a user is authorized to a site.
3    *
4    * @param string $siteId The id of the site.
5    * @return bool True if user is authorized and site found, False
         otherwise.
6    *
7    * Predictions for $siteId
8    * 1. int 99.56 %
9    * 2. string 0.35 %
10   * 3. null 0.02 %
11   */
12  public function verifyUserAuthorizedToSite(string $siteId) :
        bool
13  {
14      $siteIds = $this->getSiteIdsThatUserIsAuthorizedFor();
15      if (in_array($siteId, $siteIds)) {
16          return true;
17      }
18      return false;
19  }
```

Our last example, which can be seen in listing 5.4, shows a function where the model predicted the wrong type, but the prediction is close enough to the correct type annotation. In this example, the type `AbstractProperty` does not exist in the vocabulary of the model, but from the variable name and how it is used, it predicts type annotations that are similar to the correct type annotation. This can be useful for the developer in cases such as this, since it can help the developer decide the type annotation if there is a custom type annotation that is similar to the predicted type annotations.

Listing 5.4: Example where the model predicted a type similar to the correct type

```
/**
 * Add a property.
 *
 * @param AbstractProperty $property
 * @throws InvalidArgumentException if property name already
     exists in the JSON Schema.
 *
 * Predictions for $property
 * 1. Property 77.94 %
 * 2. PhpProperty 13.86 %
 * 3. \ReflectionProperty 0.84 %
 */
public function addProperty(AbstractProperty $property)
{
    if ($this->inJsonSchema($property->getName())) {
        throw new InvalidArgumentException("Invalid␣argument,␣
            property␣with␣name␣'{$property->getName()}'␣already␣
            exists.");
    }
    $this->properties[] = $property;
}
```

5.5 Comparison With Static Code Analysis Tools

Currently, there is a decent amount of code analysis tools for PHP [6]. Many of them are more niche than general purpose, like tools that find uses of non-standard libraries, tools that find security flaws in code, tools that find copy/pasted code etc. There is a large amount of linters that can scan the code and report flaws, but many of the tools are very small, and can not do type inference. Of the code-analysis tools that exists for PHP, some the most popular static code-analysis tools: PHPStan, Psalm, and Phan; and a dynamic code-analysis tool called phpweaver, were selected for testing.

The tool that was tested the most, Psalm [20], is different to the other tools due to the fact that it can modify code if it finds some flaws, e.g., missing parameter types. Psalm can infer type annotations from functions by looking how a function is called and the function's content, but it does not always work. In listing 5.6, Psalm can infer that function g should have a parameter that is a string, but it cannot infer type annotations on some simpler examples, such as the code seen in listing 5.5.

Listing 5.5: An even simpler php example

```php
<?php

class Test {
    public static function f($s) {
        print($s);
    }
}

f("Hello");
```

However, Psalm can infer return types for functions and methods, which the other tools could not do. Since Psalm can change code, we decided to test the limits of Psalms type inference.

When testing Psalm with the top 10,000 rated PHP projects on github where we removed all type annotations from both the PHPdoc comments and function signatures, Psalm could infer type annotations on 9.7 % of all parameters. If we are only looking at parameters that were annotated to begin with, Psalm could infer type annotations correctly on around 7.0 % of the parameters. If we are looking at each type annotation by itself, the numbers are a bit different. Psalm could infer 17.8 % for booleans, 10.3 % for integers, 10.4 % for arrays, 10.7 % for strings, and 4.16 % for floats. An overview of the evaluation can be seen in table 5.4 and 5.5.

Table 5.4: Results of evaluating Psalm

	Amount of typed parameters	Total amount of parameters	Percentage typed parameters
Developer typed types	3922989	6381656	61.5 %
Psalm inferred types	620979	6381656	9.7 %
Correctly inferred precision	273949	620979	44.1 %
Correctly inferred recall	273949	3922989	7.0 %

Table 5.5: Results of individual types

Type	Amount of typed parameters	Total amount of parameters	Percentage typed parameters
String	136099	1272266	10.7 %
Array	52440	506027	10.4 %
Integer	25985	252934	10.3 %
Boolean	19926	111970	17.8 %
Float	1015	24393	4.2 %

We believe that the numbers are higher in reality, since we have some limitations. We are only accepting the cases where Psalm inferred the exact same type as the type the parameter had before we removed all type annotations in the code. In some cases Psalm can infer correctly, e.g., a parameter that has the type `string|null` while Psalm inferred string, or that the parameter was typed with string and Psalm inferred `string|null`. However one would have to manually examine and verify that the inferred type annotation actually works. Another possibility is that Psalm believes that a parameter can get null while in reality the parameter should not be able to be `null`. We are also not accepting types from external packages where Psalm could infer correctly, but not in the exact same way as it was annotated in the beginning. There are also cases where the developer gave the parameter the `mixed` type annotation while Psalm infers a more specific type annotation, e.g., `int|string`. There can also be cases where the developer gave a parameter the wrong type annotation while Psalm inferred the realistically correct type.

We are not taking into account how well Psalm can infer type annotations on parameters that did not have a type annotation to begin with, since it is difficult to actually verify that the inferred type annotation is correct without executing the program. Even if the inferred type is correct, the code would have to be manually tested and reviewed to see if the inferred type actually is a good fit in the code. In a lot of cases it might be very simple functions that Psalm infers the type on, but since there is no simple way to automatize the verification process for the code, we have decided to not take it into account.

If we are only looking at raw numbers from Psalm and our model, the machine learning model outperforms Psalm. With a top-1 accuracy of 76.2 %, there are improvements compared to Psalm's accuracy of 44.1 % on type annotations it actually inferred, and 7.0 % on all the parameters in its dataset. However, there are some important distinctions between how Psalm and the machine learning model infer type annotations. Psalm will not infer a type on a parameter unless the program is absolutely sure that the inferred type is correct, while the machine learning model will guess the type annotation on the parameter no matter what. When evaluating Psalm, we could only count exact matches, so even though Psalm might have inferred a type annotation that is correct but in a different order compared to what the developer typed, e.g., `int|string` instead of `string|int`, it will count as a failure. If a more fair evaluation could be done for Psalm, the numbers for the accuracy of Psalm would most likely be higher.

If the developer is looking for suggestions to type annotations instead of what is actually correct, the machine learning model would be better than Psalm. But if the developer wants to automatically change code, Psalm would be better, since the type annotations inferred by Psalm will work, while the type annotations inferred by the machine learning model might not be correct.

5.6 Other Code Analysis Tools

Although we tested other code analysis tools than Psalm, they did not fill certain functionalities that we were looking for, or had other problems. Here, we would like to present the other tools.

5.6.1 PHPStan

PHPStan [15] is a code analyser that can check for many code flaws such as incorrect function or method calls, too few parameters in a function call, wrong type annotations, return types etc. A functionality that we found was missing was the ability to infer type annotations. PHPStan can find flaws such as the lack of type annotations for function parameters or that parameters sent to a function call were of the wrong type, but it can not infer type annotations based on the content of the function or how a parameter is used. Let us look at the example in listing 5.6.

Listing 5.6: A simple php example

```
1  <?php
2
3  class Example {
4      public function f(): void {
5          $this->g("Hello World");
6      }
7
8      public function g($s): void {
9          print($s);
10     }
11 }
```

```
Method Example::g() has parameter $s with no typehint specified.
```

PHPStan can see that the variable $s is missing a type annotation, but it can not say what type annotation. A developer would have instantly seen that the variable $s should be a string, but PHPStan can not infer it at all, since it does not analyse the content of the function. We also found that PHPStan has no ability to alter files, and can only analyse the files. This means that the tool can only give hints on what might be problems within the code, but can in no way aid the developer in how to solve the problems. Since we could not actually change the code with PHPStan or extract the tools internal type inference (if it has one), we decided to not test the tool further.

5.6.2 Phan

Phan [17] was the least useful tool that was tested, since it does not give any errors around type annotations or return types (even if return types is not as useful as type annotating in this case). In order to do some kind of inference, the tool needs to be able to find types by itself or output some kind of marker (i.e., an error) which can then be used to find the places in the code where type inference is needed. It might be Phan's "minimize false-positives" approach that might be the problem, since it might not want to do type inference since it does not know what type the variable/function might be. Since Phan gives no feedback on type annotations and can not infer types, we decided to not test the program further.

5.6.3 Phpweaver

Phpweaver [19] is a tool that combines static and dynamic code-analysis. This tool uses xdebug, an extension to PHP, in order to dynamically analyse function calls and see what types

all the arguments for the called functions actually had during execution. Phpweaver then uses the output from xdebug to generate PHPdoc comments for all the functions which contain type annotations with the types that was seen during the dynamic code-analysis.

This tool works well for code bases where one can be sure that exactly all code is executed with all possible inputs, which means that it works well for projects with 100 % test coverage. However, it was infeasible to execute all code and evaluate phpweaver as many repositories did not have any tests. We also thought such an evaluation would only measure the test coverage of the repositories rather than the usefulness of the tool. We therefore decided to not test phpweaver any further. However, this might have been a mistake, since an evaluation of phpweaver would have strengthened the comparison of existing tools with the model.

Chapter 6
Discussion and Future work

In this chapter we will discuss the model, its practical applications, some design choices and what future work there is to be done. Section 6.1 contains the discussion of the model in its entirety and section 6.2 lists some suggestions on future work that can be done.

6.1 Discussion

There are a couple of different subjects we would like to discuss here. Not only the results of the model and how well it works, but also some practical applications of it and how the model can be used in a real-world setting. We will also discuss a few improvements that we believe would work, but which we did not have time to actually implement. Lastly, we will also discuss a few limitations of the model.

6.1.1 Does the Approach Work?

Our results show that deep learning successfully can be used for type inference, and compared to the static analysis tools that are currently available for PHP, it might even be better in some cases. However, it all depends on how the model is used. Even with a 76.2 % top-1 accuracy and a 84.2 % top-3 accuracy, there is still a large margin of error. This means that predictions made by the model never can be blindly trusted to be correct as that most certainly would introduce errors into the code. If the predictions made by the model only were to be viewed as suggestions instead, where the developer ultimately makes the final decision, we believe that the model could serve as a helpful and convenient tool.

Looking at how the model performs when predicting type annotations for parameters with previously unseen names, we could see a significant drop in the accuracy. This does not only suggest that the model is highly dependent on good and descriptive parameter names to make accurate predictions. It also suggests that the approach actually works and that the model is learning something. It learns that certain parameter names usually correspond to

certain type annotations. At the same time, the accuracy is still high enough to suggest that the model is not randomly guessing. In other words, it also appears to learn how to correlate the usage of parameters with their type annotations.

Overall we would say that our results indicate that the approach works very well, but in the end it depends on how the model is used. In the following section we will discuss a few practical applications of the model.

6.1.2 Practical Applications of the Model

In this thesis we added all the predictions of the model to the documentation comments of the corresponding methods or functions. This was solely done for demonstration purposes and to make it clear which prediction belongs to which parameter while evaluating the model. From a practical point of view this would most likely just annoy the developers as it clutters the code comments with many unnecessary predictions as well. However, there are still several ways in which this model can be used in a real-world setting to enhance the developer workflow.

One possibility would be to implement the model as a web service. In other words, giving it a web interface where developers can submit code for the model to predict type annotations in. This would also make it possible for the model to improve over time by learning from that submitted code and extend its vocabulary. Allowing the model to continuously learn in such a web service imposes a risk of overfitting though. If a developer repeatedly submits the same code or code from a single project or company, the model might start to overfit to that particular data.

Another possibility would be to integrate the model into a continuous integration system, such as jenkins, so that the model can be used in automated testing. This could also make it easier for companies to start using it as they might already have a tool stack in place which allows for easy integration of new testing tools. Including the model into the feedback loop of the project might be helpful to the developers if its used in the right way. As mentioned previously, the predictions made by the model are merely guesses and may be wrong. The predictions from the model should therefore not be used in critical tests that may cause a system build to break. However, if the model is used in the right setting and its predictions are treated the right way, we believe that the model can be of great use to a developer.

The model could also be implemented as a plugin for an IDE that gives the predictions as suggestions directly in the editor. By making a plugin, the model would be much easier to use, and if implemented in a similar style to a linter it would not clutter the code comments unless the developer decides that it should.

The fact that the model performs significantly worse when predicting type annotations for parameters with previously unseen names also opens up the door for another possible application of the model. That is to evaluate how good and descriptive the parameter names in a project are. Since the model is trained on the most popular PHP repositories on github, which we assume use good names and follows naming conventions, we can use the models dependency on good parameter names to our advantage. In other words, if the model struggles to predict type annotations in a certain project and obtains a low accuracy one might suspect that the parameter names are not good and descriptive enough.

6.1.3 Using ASTs Instead of Parse Trees

In the works of Alon et al. they extracted the path-contexts from ASTs [2, 3], but in this thesis we extracted them from parse trees instead. This might have been a mistake.

Astminer [11], which we used to extract the path-contexts, is capable of creating and mining ASTs, which can be seen when looking at the examples in the paper by Kovalenko et al. However, the trees we generated for PHP were more akin to parse trees than ASTs. The reason for this was that we decided to generate and use a parser based on a predefined ANTLR grammar file, and this particular parser could only transform code into parse trees. Since an AST is more compact with its information concentrated to fewer nodes it might have been beneficial to utilize ASTs instead of parse trees. The fact that the generated parser we used created parse trees instead of ASTs was unfortunately not discovered until it was too late to replace it. If we wanted to use a parser that created and mined ASTs instead we would have had to implement the parser ourselves. Either from scratch or by modifying an existing parser. Even though both of these options would have been too time consuming we did make an effort to alter the parse trees slightly once they were created to make them more similar to ASTs. This modification led to a small improvement of the results which leads us to believe that our results can be improved further by fully utilizing ASTs.

6.1.4 Limitations

One limitation of the model is that it is confined to its vocabulary when making predictions. This prevents the model from ever correctly predicting the type annotation of a parameter whose correct type annotation is not part of the vocabulary. Even though this is expected as it is difficult for a model to guess something it has never seen before, it can limit the usability of the model in certain cases. Particularly when it is used in a project that uses a lot of custom in-house classes and objects with names unique to that project. If these in-house objects are commonly used as parameters, the overall accuracy of the model will suffer as the model will fail to predict the type annotations for these parameters.

Another limitation of the model is that it is only capable of predicting type annotations for parameters. One could possibly argue and say that it also works for local variables since path-contexts for these can be extracted similarly to how we extracted path-contexts for the parameters. However, this would presumably result in a lower accuracy since we believe that local variables generally have less descriptive names than parameters. Return expressions on the other hand is something that the model is completely incapable of predicting type annotations for. Since a return expression might consist of several variables, literals, function calls, operators etc, we can not simply treat it the same way as a parameter. The fact that one function or method can have multiple return expressions might also be problematic as these expressions might look vastly different. We can therefore not see any obvious ways to use this model to predict type annotations for return expressions.

6.2 Future work

Throughout this thesis we thought of several ways to improve either the performance or the usability of the model. Unfortunately we did not have time to implement all of them. Instead

we will leave some of them as future work. Here follows a few of those ideas we had:

Gather more paths by checking how the method or function is invoked. Considering entire files when extracting path-contexts did not result in an increase of the accuracy. However, we still believe that considering other functions or methods could improve the accuracy of the model. A solution that we did not have time to implement and test is to check how the parameter that is analysed is used in the context of the function or method call that the parameter is part of. By looking at how a parameter is used in a function or method call, we gather more context from how it is used.

Use code2seq instead of code2vec. Alon et al. developed another model after their code2vec model, called code2seq [1]. The code2seq model represents the paths in a path-contexts as a series of nodes instead of a single entity. This allows paths that share a majority of nodes with other paths to be more similar to those path embeddings. The use of this model could increase the accuracy of type inference, and we believe that it would be a good idea to compare how the code2seq model performs compared to the code2vec model.

Combine the model with a static code analysis tool. Instead of using the model by itself, you could combine the model with a static code analysis tool. This would strengthen the code analysis tool when it is unsure on what it should infer, while reducing the probability of the model inferring the wrong type annotation when the code analysis tool is confident in it's inference.

Implement the model as a service. Implementing the model as a service would allow developers to quickly test small snippets of code without having to train a model beforehand. It would be interesting to see what challenges there are in implementing such a solution.

Handle out of vocabulary type annotations in a better way. For obvious reasons the model is unable to predict the type annotation for parameters whose correct type annotation is not part of the vocabulary. However, for these parameters we noticed that the model frequently predicted type annotations that were similar to the correct annotation. This leads us to believe that further research could be done to find a way to predict these out of vocabulary type annotations more accurately.

Use a proper AST instead of a parse tree. Since we unfortunately did not use a proper AST, we could not utilize all the benefits of an AST. Alon et al. [3] used ASTs in their solution, and we believe that higher accuracy can be achieved when utilizing an AST.

Chapter 7
Related Work

Using machine learning to infer data types or to predict properties and attributes within source code is something that has been done before. In this chapter we will present a couple of previous attempts at type inference using machine learning.

7.1 Token-Based Code Representation

One approach which have been tried is to take advantage of an aligned corpus of tokens and data types and use this to teach a deep learning model which types normally correspond to certain tokens and in certain contexts. This is the approach Hellendoorn et al. took when they created DeepTyper [9], a machine learning tool that provides type suggestions in JavaScript.

The inspiration for this approach comes from the well known part-of-speech tagging problem where the goal is to assign a part-of-speech tag, such as adjective, adverb or determiner, to every word in a given text corpus. Since source code can be tokenized into a sequence of tokens, analogous to how text can be tokenized into a sequence of words, it makes it possible to treat type inference as a part-of-speech tagging problem. However, instead of tagging words with a part-of-speech tag, tokens has to be tagged with a data type.

When Hellendoorn et al. created DeepTyper they utilized the similarities between JavaScript and TypeScript to train their model. Since TypeScript is a strict syntactical superset of JavaScript, with the additional support of static typing, they were able to create their datasets based on source code written in TypeScript. By first compiling the TypeScript code they could obtain a sequence of data types for all the identifiers in the code. By then removing all static type annotations they obtained code that resembled JavaScript code. This code was then tokenized and aligned with the sequence of data types inferred by the compiler in order to produce the training data. This data, both tokens and data types, was then vectorized and fed as input to the deep learning model. In their approach they used a bidirectional GRU, which is a type of recurrent neural network which is proven to work well with sequence data such as natural language.

After training and evaluating their model with a dataset consisting of code from 776 TypeScript projects they achieved a top-1 accuracy of 56.9 % and a top-5 accuracy of 81.1 %.

Using a token-based code representation is an approach we considered going for before starting this thesis. However, there are issues which makes it infeasible to apply this approach for type inference in PHP. One such issue is that this approach requires a fully type annotated dataset to learn from. Not only would the function and method parameters need to be type annotated but also the local variables as the neural network is unable to differentiate between these. The problem with this is that type annotations for local variables were not supported in PHP until version 7.4. This makes it difficult to find enough training data which makes use of these type annotations for local variables. The only reason Hellendoorn et al. were able to use this approach for JavaScript was that they could utilize the similarities between JavaScript and TypeScript to obtain training data. In a similar way we considered creating training data from code written in Hack, which in a sense is to PHP what TypeScript is to JavaScript. However, we were unable to find enough code written in Hack to be able to create a sufficient amount of training data.

7.2 Exploiting Natural Language Features

Another approach which has been tried is to take advantage of the natural language features in source code which normally gets neglected by the traditional type inference algorithms commonly found in static analyzers. This includes features extracted from comments, function names and parameter names.

Malik et al. used this approach when they created NL2Type [13], a machine learning tool to predict type annotations for functions in JavaScript. In their approach they first extract the function name, parameter names, parameter types, return type and comments associated with the function, parameters and return value. The obtained natural language information is then filtered such that prepositions, determiners, punctuation and other uninformative words and characters are removed. The remaining words are then converted into vectors using two learnt word embeddings based on Word2vec [14]. One embedding was used for the words in the comments and another embedding was used for the words in the function name and parameter names. These vectors are then chained into a sequence, padded or truncated to a specific length and used as input to the neural network, which in their approach was a bidirectional LSTM-based recurrent neural network.

After training and evaluating their model on a dataset consisting of 618,990 data points, where each data point represents either a return type or a parameter type, they achieved a precision of 84.1 %, a recall of 78.9 % and a F1-score of 81.4 %.

This approach was also used by Boone et al. when they created DLTPy [5], a tool similar to NL2Type, but for predicting types in Python instead. In their approach they took advantage of natural language context in a similar way, but with the exception that they also included the docstring of the function and a list of the return expressions. The extracted data was then lemmatized and preprocessed in a similar way to NL2Type before it was vectorized and used as input to their neural networks. Three different models were implemented, however all three of them were variations of the LSTM-based recurrent neural network which was used in the work of Malik et al.

After training and evaluating the models with five different datasets they managed to

achieve a precision of 81.7 %, a recall of 83.2 % and a F1-score of 82.4 % on their best model and with the best dataset.

By simply looking at these numbers it might seem like an approach that exploits the natural language features in source code is slightly superior to the approach we explored in this thesis. However, we can't really compare the results of these approaches and give a definite answer to which one is better. Since NL2Type and DLTPy performs type inference in other languages we can not know for sure if their results are generalizable to PHP. They both also use different methods for evaluation. In the case of DLTPy, for instance, they achieved their best results using a dataset where all natural language features were present for all data points. In other words, all their data points were created from functions that had a comment associated to itself, to its return value and to each parameter. As a matter of fact, their model performs significantly worse for data that are missing any of these features. We therefore cannot say whether the approach we explored in this thesis is better or worse than the approach used by NL2Type or DLTPy.

Chapter 8
Conclusions

This thesis examines if a deep learning model based on code2vec can be used for type inference in PHP. Since the original model used in code2vec was created for predicting method names, our model had to be modified in order to predict type annotations for parameters. This was done by adjusting the selection of path-contexts to better suit our purpose. By restricting the path-context to require a parameter as a start or end token, the model can learn patterns specifically regarding the parameters, instead of looking at path-contexts that have nothing to do with the parameter.

After training the model with the top 10,000 PHP repositories from github, the model can predict type annotations with a top-1 accuracy of 76.2 % and a top-3 accuracy of 84.2 %. When comparing these results with the static code analysis tool Psalm, which could infer type annotations on 9.7 % of the dataset with an accuracy of 44.1 %, the improvement is substantial. However, since the predictions made by the model are merely guesses, a developer cannot blindly trust the model in the same way as they can trust Psalm. The general usefulness of the model therefore greatly depends on how it is used. Several practical applications of the model were discussed. Everything from implementing it as a web service, to integrating it into a continuous integration system or using it in a plugin for an IDE.

A dependency on good parameter names also seemed to reveal that the model works as intended. At the same time it opened up for possibilities to evaluate how good and descriptive the parameter names in a project are.

Overall, this thesis shows promising results for future usage of machine learning for type inference. We conclude that the machine learning approach explored in this thesis works as intended and that it works with great results. However, its usefulness strongly depends on how the model is used.

References

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2018, arXiv:1808.01400 [cs.LG].

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 404–419, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.

[4] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, USA, 2nd edition, 2003.

[5] Casper Boone, Niels de Bruin, Arjan Langerak, and Fabian Stelmach. Dltpy: Deep learning type inference of python function signatures using natural language context, 2019, arXiv:1912.00680 [cs.SE].

[6] exakat. Static analysis tools for php. `https://github.com/exakat/php-static-analysis-tools` Accessed: 2020-04-29.

[7] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 758–769, 2017.

[8] PHP Group. Php: Hypertext preprocessor. `https://www.php.net/` Accessed: 2020-04-14.

[9] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014, arXiv:1412.6980 [cs.LG].

[11] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 13–17. IEEE Press, 2019.

[12] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[13] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: Inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 304–315. IEEE Press, 2019.

[14] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013, arXiv:1301.3781 [cs.CL].

[15] Ondřej Mirtes. Phpstan. `https://phpstan.org/` Accessed: 2020-05-14.

[16] nikic. Php-parser. `https://github.com/nikic/PHP-Parser` Accessed: 2020-04-29.

[17] phan. Phan. `https://github.com/phan/phan` Accessed: 2020-05-14.

[18] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[19] troelskn. phpweaver. `https://github.com/troelskn/phpweaver` Accessed: 2020-05-14.

[20] Vimeo. Psalm - a static analysis tool for php. `https://www.php.net/` Accessed: 2020-05-14.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-06-11

EXAMENSARBETE Type Inference in PHP using Deep Learning
STUDENTER Samuel Klingström, Pontus Olsson
HANDLEDARE Niklas Fors (LTH)
EXAMINATOR Görel Hedin (LTH)

Kan maskininlärning användas för typinferens?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Samuel Klingström, Pontus Olsson**

Stöd för typannoteringar i dynamiska programmeringsspråk blir allt vanligare, men att uppdatera äldre kod med typannoteringar kan vara ett väldigt tidskrävande jobb. Detta arbete använder maskininlärning för att gissa typer på funktioner och metoder.

Dynamiska programmeringsspråk som t.ex PHP, Python, och JavaScript har under de senaste åren börjat stödja typannoteringar likt statiskt typade programmeringsspråk som t.ex Java och C++. För tillfället finns det många verktyg som kan analysera kod för att hitta fel, men för PHP är det få av verktygen som klarar av att inferera typer.

I vårt examensarbete har vi skapat ett verktyg som använder maskininlärning för att gissa typer på parametrar i funktioner och metoder för PHP. Maskininlärningsmodellen kan, givet en funktion eller metod, gissa vilka typer som hade passat varje parameter baserat på hur den används. För varje parameter rapporterar verktyget sina tre bästa gissningarna direkt i kodkommentaren, samt hur säker modellen är på varje gissning. Ett exempel visas figuren där nätverket gissar typen int för parametern $n.

Genom att ge förslag på typannoteringar till utvecklare, blir det mycket enklare för utvecklare att uppdatera kod som inte har typannoteringar. Att uppdatera gammal kod som saknar typannoteringar är en väldigt tidskrävande process, då utvecklare måste sätta sig in i hur koden fungerar för att kunna uppdatera koden korrekt. Ett verktyg som detta underlättar processen, och utvecklare kan istället spendera mer tid på att skapa ny funktionalitet.

```
1  /**
2   * Predictions for $n
3   * 1. int 99.92 %
4   * 2. float 0.07 %
5   * 3. double 0.0 %
6   */
7  function factorial($n) {
8      $factorial = 1;
9      while ($n > 0) {
10         $factorial = $factorial * $n;
11         $n--;
12     }
13     return $factorial;
14 }
```

För att modellen ska kunna göra gissningar på kod, måste koden representeras på ett sätt som modellen förstår. För varje fil som ska användas läses koden in och omvandlas till ett träd, där varje nod i trädet representerar en del av strukturen hos koden. Utifrån trädet kan sen information kring koden fås genom att titta på hur de olika noderna i trädet är kopplade till varandra.

Efter att modellen tränats med de 10 000 mest populära PHP-projekten på Github, kunde den gissa typannoteringar med en korrekthet på 76 % om den fick en gissning, och 84 % om den fick tre gissningar. Dessa resultat är betydligt bättre än vad nuvarande statiska kodanalysverktyg presterar. Att använda maskininlärning för typinferens ser därför ut att vara väldigt lovande.