MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN DATA SCIENCE

Deep Learning Applied to Code Analysis

Genin, Simon

Award date:
2019

Awarding institution:
University of Namur

Université de Namur
Faculté d'informatique
Année académique 2018–2019

Deep Learning Applied to Code Analysis

Simon Genin

Maître de stage : Prof. Benoit Frenay

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Prof. Benoit Frenay

Co-promoteur : Prof. Benoit Vanderose

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Acknowledgements

I would like to thank my promoters for their help, patience and pieces of advice as well as my family for their support.

Contents

List of Figures

Chapter 1

Introduction

Throughout the last couple of years, the frontier of what can be achieved with machine learning has expended tremendously. Many subjects have seen breakthroughs with smart usage of deep learning. One of them, still shy amongst other more pro-eminent topics, is the application of those kind of methods to improve the quality of work of the software engineering community. Natural Language processing (NLP) has seen great ways of dealing with textual data and show great promises. Very naturally, the idea of applying the same principles on code emerged. After all, source code is text and it is only fair to assume NLP techniques would dominate the field of code analysis. Alas, when it comes to reveal the semantic of a piece of code by shoving it into an NLP typical neural net architecture, it does not go that well. Many bits and pieces of NLP are still relevant though. For instance, the analysis of method and variables names to try and guess the purpose of a method. But these struggles have lead to the research of new innovative use of deep learning adapted for source code.

In this thesis, the goal is to introduce methods to analyse source code using deep learning. This work is meant for an informed reader in software engineering who wish for some deep learning insights and opportunities in this field. A deep understanding of machine learning is therefore not required. The reader should get a sense of what is currently done in code analysis with deep learning and be able to grab the fundamental principles and where to seek for further study.

Some of the questions this document tries to approach are:

- Can one interpret the features yield by the deep neural network ?
- How much of the semantic can be understood by a neural network ?
- Are the known neural network architecture relevant in this field ?
- How can code be embedded ?
- How does the tree-based convolutionnal neural network perform ?

In the first part, a brief introduction to the machine learning concepts required for a good understating of this thesis is proposed. Then, the second part contains first a chapter which is an overview of the themes of the literature and the second chapter are the tree-based convolutional network and the graph-based convolutional neural network architecture description. Then, an experience using the tree-based convolutional network is conducted.

Part I

Machine Learning

Chapter 2

Introduction

This part is meant to be a brief introduction to deep learning for a reader with very little knowledge of the field. It also only mainly mention things that matter in the code analysis. It definitively aims at being more at the reach of the average student rather than being a rigorous introduction. All the following subjects are explained because they are used in a way or another into the software analysis domain. Whether it is in an architecture which is going to be reviewed or simply relevant to understand a paper abstract in the field.

Chapter 3

Deep learning

The difference between machine learning and deep learning can be quite blurry. Some call deep learning any neural networks. Some will only use the term if their facing a truly *deep* network, because of many layers.

An other way to separate the two in an elegant way is displayed in figure 3.1, where the important part is the feature extraction. It assumes that deep learning is used when the features are found by the network, and not before like it is typically done in traditional machine learning.

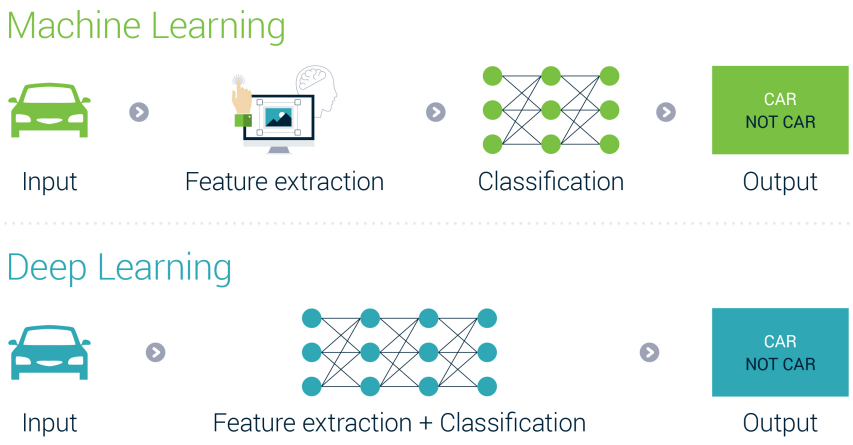

Figure 3.1: Deep learning seen as automatic feature extraction, reproduced from *https* : *//www.datavisitor.com/important − deep − learning − algorithms*

The second description is actually the one that fit the best the goal of code

analysis. Finding new features defining a piece of code.

Whatever the preferred description, the analysis of code will be based on deep learning. In truth, the goal of the researches in the field is often to find new features depicting programs behaviors and/or qualities that are different than those software engineers currently use.

Chapter 4

Neural Networks

A basic understanding of neural networks is expected for whoever reads this thesis. However, this chapter will briefly explain what it is. Feel free to skip over this chapter.

Fundamentally, a neural network defines a function. The goal being finding an approximation of another function which depicts some data. In other words, using a deep learning method just mean that somebody tries to find a close enough function to a real mathematical model. To understand how this network can be seen as a function, a closer look is needed to its structure. First, the net is a graph. Most of the time, an directed acyclic (DAG) one as seen in figure 4.1

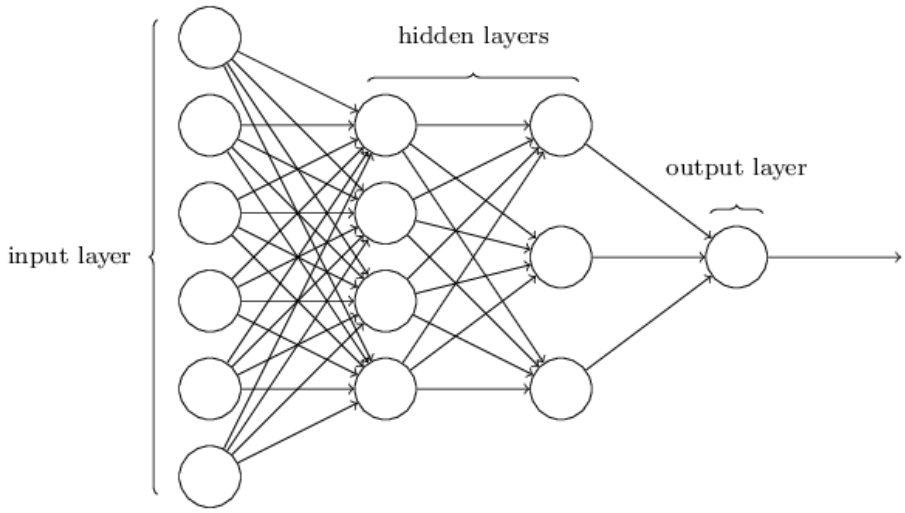

Figure 4.1: Basic neural network architecture, reproduced from `http://www.shivambansal.com/blog/neural_network_1`

The nodes are called perceptrons. The edges have a direction, obviously, and hold a weight. More on that later. Perceptrons can be seen as made out of three parts: the inputs, the algorithmic operation and the activation function. See in figure 4.2. Its inner working is rather simple, each perceptron calculates the linear combination of the inputs and then add non-linearity by applying the activation function.

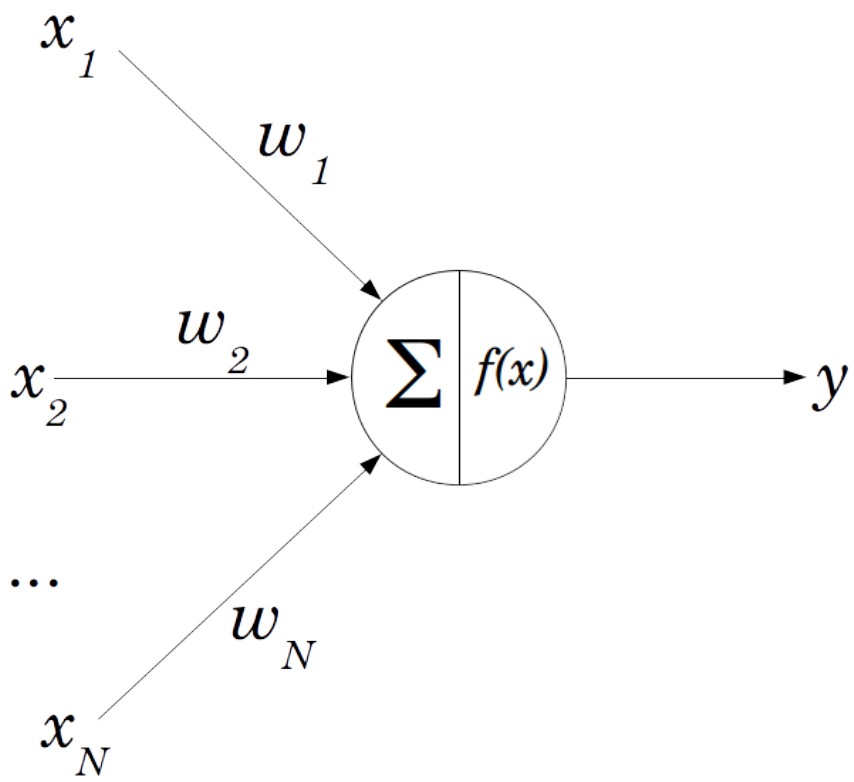

Figure 4.2: A representation of a perceptron, reproduced from `https://www.lucidarme.me/simplest-perceptron-update-rules-demonstration`

Ignore for now the w_n variables in figure 4.2. Then we clearly see the output of the perceptron being $y = f(\sum_i x_i)$ where x_n are the inputs, i the number of inputs and f the activation function.

The weights, are their name implies, are a way to regulate each input value and to a greater extend, the perceptron. If at first sight, it doesn't bring a lot to the model, it is actually the core of the deep learning algorithms.

The number of nodes, the depth of the graph, these are all variables for the model to be found to make the model as efficient as possible.

4.1 The Learning Process

At the heart of the learning process stands an algorithm: the back-propagation. It is an application of the chain rule, a formula for computing the derivative of the composition of two or more functions.

The way it works will not be discussed in details as it is not the goal of this work. All there is to understand is that there is an error function defined to calculate how wrong the neural network output is. Through the process of backpropagation, the weights are getting updated by a value given by the error function. Applied many times, neural network shall converge towards a good approximation. In theory as it is highly simplified in our explanation and there are plenty of loopholes to avoid.

4.2 The Architecture

For its structure, the neural net is simply a stack of layers, each one containing a certain number of perceptrons. The depth of the net (stack size) and the width of each layer to give the best results is to be find by tweaking and experimenting and also depends on technical and time based limitation. This is of course for its simplest forms as many different and much more complex ways of building up neural networks are possible, all aimed and optimized for specific tasks.

Chapter 5

Convolutional Neural Networks

5.1 Introduction

Convolutional Neural Networks [Goodfellow et al., 2016], depicted in figure 5.1, are very similar to ordinary neural networks except they are used for images [1]: they are made up of perceptrons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally add a non-linear operation, often called the activation function. The whole network still is one function taking matrices to a result: from the raw image pixels on one end to class scores to the other end, for classification by example. So what are the differences with a typical neural network ?

Convolution net architectures make the explicit assumption that the inputs are images, which allows to encode certain properties into the architecture of the network. These then make the forward function more efficient to implement and can vastly improve the results given in a reasonable time.

[1] In truth, it is used fro plenty of other things, but it is simple to stick to this extend while learning.

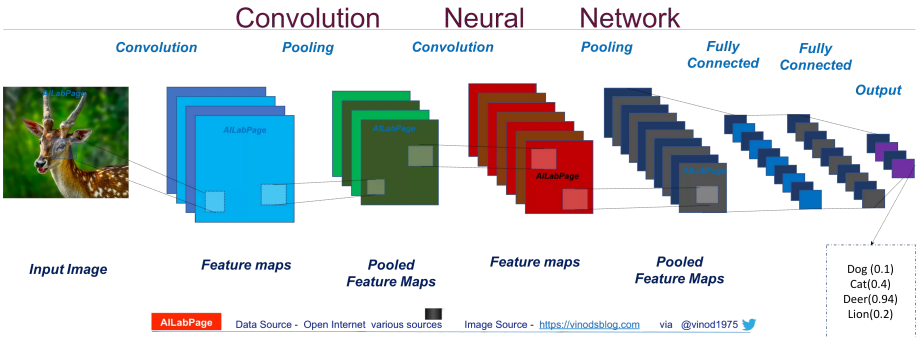

Figure 5.1: A convolutional net architecture, reproduced from `https://vinodsblog.com/2018/10/15/everything-you-need-to-know-about-convolutional-neural-networks/`

5.2 Images as Input

First, the image need to be transformed into a matrix, each value depicting a pixel. Each of these value will be an input to the network. If it can be that simple to insert an image inside a neural network, why even bother trying special kinds of architectures like convolution ? Because a fully connected layer is awfully time consuming. Working with a picture with shape 200x200px means there is already 40000 input values. This is huge, since everything will then be combined many times inside the network. Furthermore, the network would need to be deep and wide enough to capture best the features of the images. That's why more clever methods are required.

The convolution process is a way to find features from the images, that will help the network taking decisions. Figure 5.2 shows how features can be progressively found going through the network. The way the figure represents the network itself should be ignored as it is an overly simplified representation.

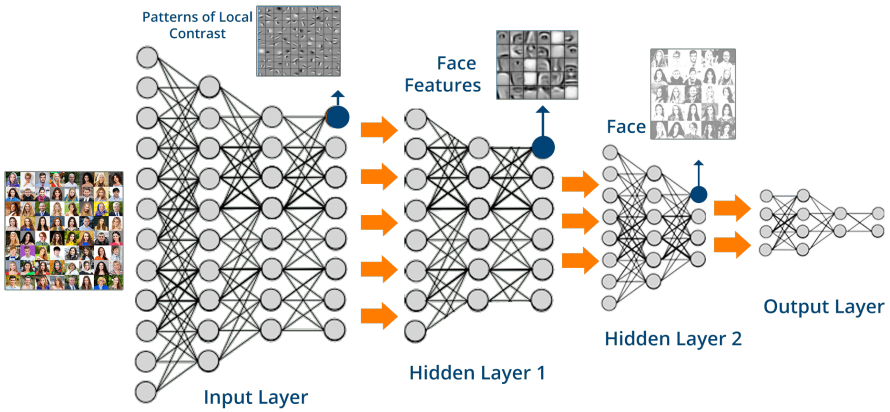

Figure 5.2: The features more and more identifiable being extracted by the network.

5.3 Kernel

To find features, kernels (also called masks or filters) are defined. They are used to slide across the input to generate several new images based on local part of the original. The kernels are made up of values so that the convolution applies transformations to help find features. Which values to put in the kernels is found by past researches.

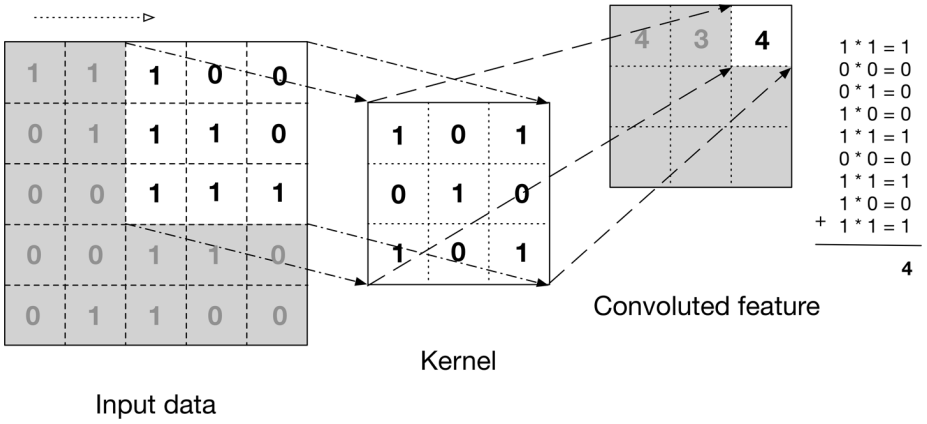

Figure 5.3: The kernel sliding over the top-right part of the image, adapted from "Deep Learning" by Adam Gibson, Josh Patterson

5.4 Pooling

Even if the convolution is less expensive than a fully connected layer, we still encounter a problem. At each convolution, we generate more sub-images. This once again can quickly drain all the calculation resources as soon as the input image is not tiny. The solution to this problem is called *pooling*.

It simply consists in a size reduction, from several numbers to one. Averaging the values, taking the maximum or other, there are several ways to apply a pooling. Which one to use is up to the developer. He may try some of them or just pick one from experience.

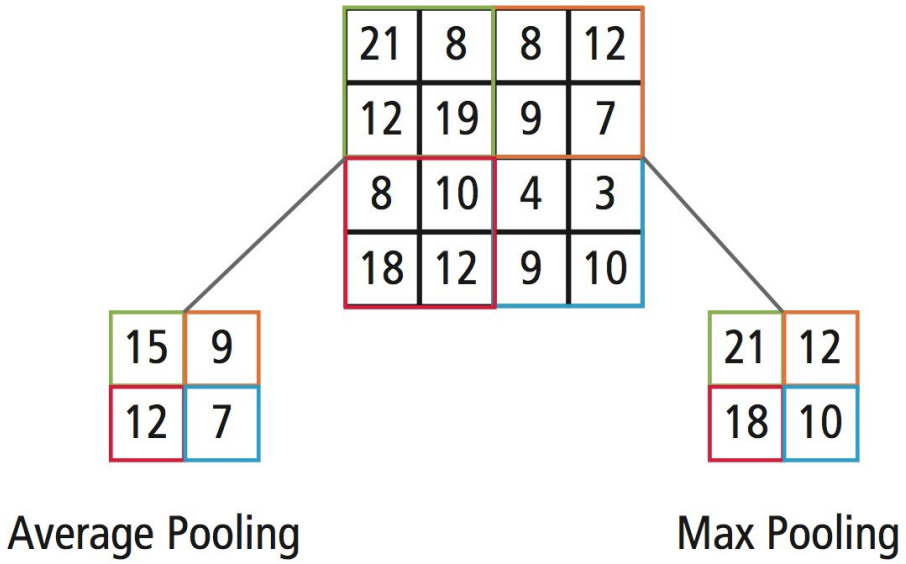

Figure 5.4: Two different pooling methods, adapted from `https://medium.com/@Aj.Cheng/convolutional-neural-network-d9f69e473feb`

The pooling being done, another convolution layer could be coming next.

5.5 Conclusions

Convolutions is really what made it possible to work with images using neural networks. Extracting efficiently features from somewhat structured data. But convolution shouldn't be too tightly linked to images anymore. Indeed, the idea behind it can work with many things. Even source code as this will be zoomed in later in this thesis.

Chapter 6

Embeddings

An embedding is a mapping from a categorical variable to a vector of continuous numbers. In the context of neural networks, learned continuous vector representations of discrete variables. Neural network embeddings are useful because they can reduce the dimensionality of categorical variables and meaningfully represent categories in the transformed space [Koehrsen, 2018]

This concept is very important for the code analysis. Indeed, a neural network cannot accept raw code as input. With images, it was easy. The pixels had RGB color values that could be used. But for any input that doesn't fit the requirements of a neural net, embeddings is a solution.

6.1 One-hot Encoding

The simplest one is the one-hot encoding. It is very useful in traditional machine learning to transform several categorical labels into a single discrete one. But in the case of embeddings for deep learning, it is a rather dummy one.

The one-hot encoding is used to transform categorical values to a sequence of discrete values without any consideration for any possible semantic. It is therefore a straightforward way to make the inputs valid, but there is a loss of information about the input data.

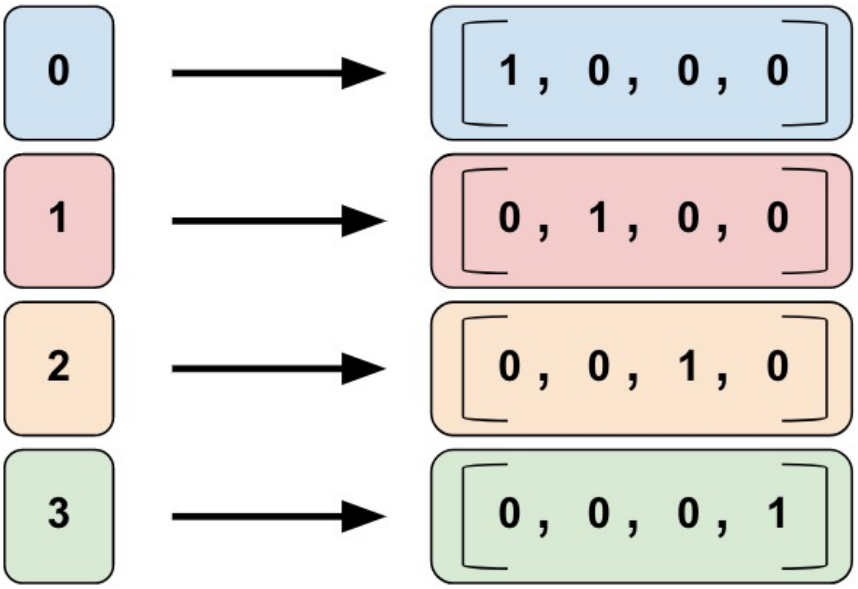

Figure 6.1: One-hot encoding, reproduced from `https://www.tensorflow.org/guide/feature_columns`

The figure 6.1 depicts 4 categorical values that are transformed into an identical format. The 4 categorical values are a bit deceiving, but it could very much be: "*plus*","*minus*","*mul*","*div*". Those would be likely inputs working with source code, that would need such embedding to be used in a net.

But as said before, the one-hot encoding just lose any meaning from the data, making it just impractical in deep learning application.

6.2 Learned Embeddings

The typical way to create an embedding is a very clever trick in deep learning. First, each input get assigned some kind a valid input of a neural network. By example, the one-hot encoding. Then, a neural network is used to encode the input to a defined sized vector, and another to decode it back to the original input. Figure 6.2 sheds light on how this process work. Like any other neural net, this gets trained until the error gets minimal, meaning that the weights in the encoder and decoder are such that $X = X'$.

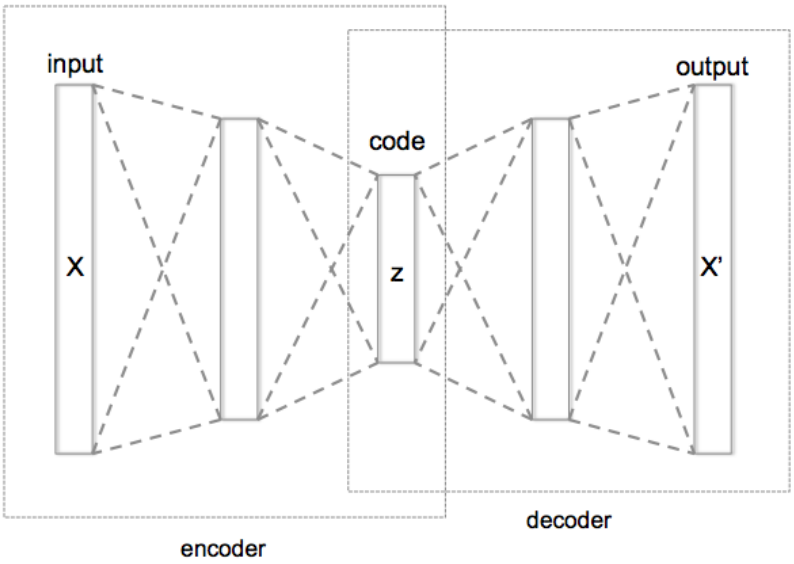

Figure 6.2: z is a embedding of $X \cong X'$, reproduced from `https://ayearofai.com/lenny-2-autoencoders-and-word-embeddings-oh-my-576403b0113a`

The inner structure of the network in the encoder and decoder is a task in itself, and there are many different architectures done for many different use cases. What is great about this embedding is it holds enough information so that the decoder can reconstruct the correct input. Which infers the semantic has been kept, despite the embedding being often way more condensed than the origin input. This is actually a great new way to compress things. What is great is that some properties can be retrieved from these embeddings. One on the most famous embedding, word2vec (transforming words into vectors), is a great demonstration. In figure 6.3 can be observed the ability to arithmetically combines vectors to find others that are semantically making sense. It would be very interesting to observe such behaviors with source code related embeddings.

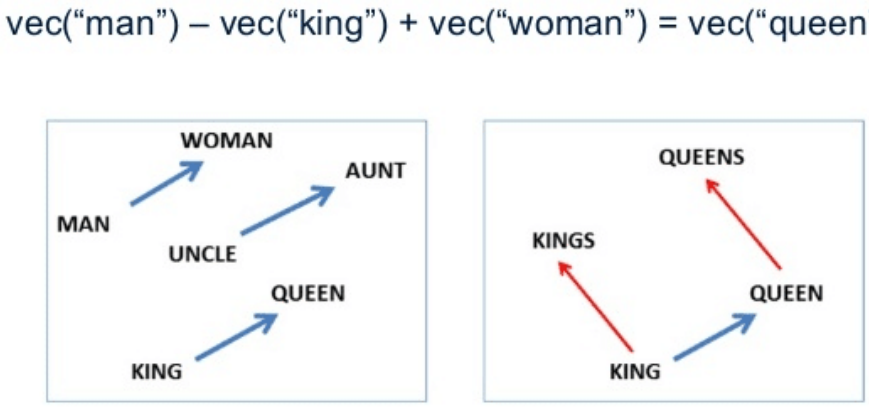

Figure 6.3: Property of word2vec, from [Mikolov et al., 2013]

It is important to note that while the behavior depicted in figure 6.3 tends to be observed, it has never actually been proven to be true and systematic. Also, it has never such a convenient accuracy. It is therefore a phenomenon nice to take note of but nothing to rely on.

In figure 6.4 is a concrete example of what a 2D dimensionality reduction of an high dimensional embedding on the C programming language looks like, done by Harer et al. [A. Harer et al., 2018]

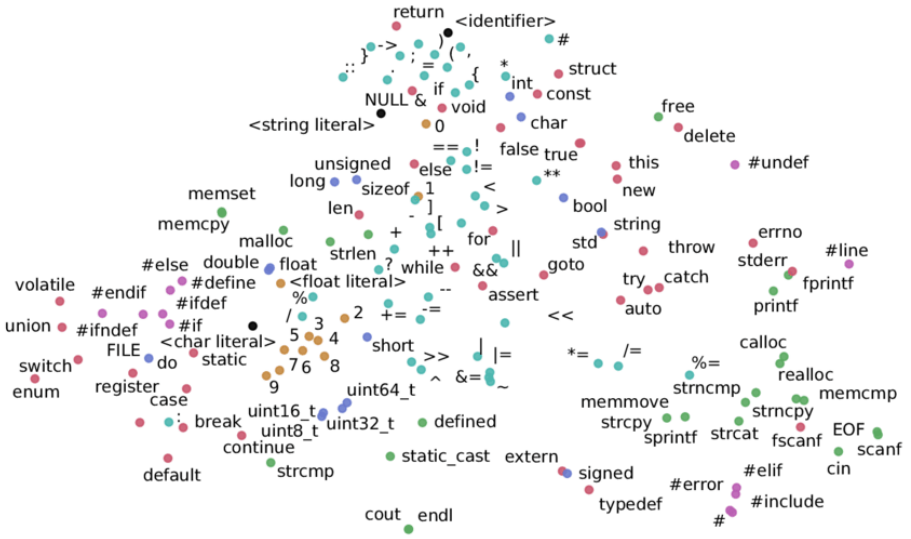

Figure 6.4: Embedding of C# tokens, reproduced from [A. Harer et al., 2018]

Part II

Contribution

Chapter 7

Research Focuses

Is proposed now an overview of different topics brought by research recently. It contains quite a wide variety of application in software analysis and quality. Here are a few of them to get a glance of where the state-of-art methods currently stand.

7.1 Clone Detection

Clone detection can be of great use in education. How much a program differs from another semantically, perhaps even with compiling errors. Automatic grading could rise from this field of study and counter plagiarism at the same time.

Ding et al work on a system that detects clones using only assembly code. With their tool Asm2Vec, they aim at getting a good semantic understanding and bypassing the many compilers optimisation and code obfuscation that can occur. [Ding et al., 2019]

Bch and Andrzejak studied more the impact to the results of the machine learning model used and propose some good practices. [Büch and Andrzejak, 2019]

7.2 Code Generation and Summarization

Code generation is a very broad field. Generating code from a domain specification in natural language looks promising. Simply giving a text depicting the application domain could generate a working SQL code and related ORM. And much more can be thought of. It is also one of the hardest subject as it means that the semantic of the text, extracted via NLP, must then be ported to a valid

model performing code generation. Matching these two is a non-trivial task.

Sun et al propose to use a convolutional neural net (CNN) instead of a recurrent neural net (RNN) to generate code as it contains much more tokens than a natural language sentence. Their network predicts the grammar rules of the programming language and then generates a program. [Sun et al., 2019]

Gao et al use an encoder-decoder with attention to help naming method names and other software artifacts. It is hard for developers to name properly things, especially beginners. They use natural language functional description to infer results. [Gao et al., 2019]

Kacmajor and Kelleher explain it is hard to create new solution for code generation as there is not really a lot of good code that is fully annotated [1]. But it is widely adopted to name test functions in a very self-documentation way (Example: $it_validates_the_user_password$). They propose to make dataset from them and observed good results generating code from quasi-natural language description. [Kacmajor and Kelleher, 2019]

Similar to code generation, there is also the opposite. Generating comments or annotation from code. Or making a summary of the code.

Yao et al propose the authors use reinforcement learning to annotate pieces of code. It is interesting as reinforcement learning is rarely used for those kind of feats. This proposed tool is specifically generating annotation for later code retrieval. It is therefore making sentences optimized for natural language search. Services like Github or any code bank would benefit from these. [Yao et al., 2019]

Xu et al discuss about the important of commit message for a code base comprehension. And yet they're often neglicted. They point that existing summarization methods tend to ignore the code structure information and suffer from the out-of-vocabulary issue. They propose another method that will both extract code structure and semantic and use all of this along some tweaks to generate commit messages that perform better than state-of-the-art. [Xu et al., 2019]

7.3 Code Suggestion and Completion

Code suggestion and completion are very nice features that can empower IDE. As a matter of fact, more and more IDE try to include statistical analysis and some machine learning to try and improve the experience of the developer. It is interesting in education as it often proposes code the student mightn't have thought of. The subject probably targets intermediate students as beginners must first learn to think properly before completion does the work for them. Here are a few works done in that domain.

Natural language used to infer types in non-strongly typed languages such as

[1] Machine learning loves labeled data

JavaScript. The idea is to understand enough of the comments and the function signature to bring more help to the IDE. [Rabee Sohail Malik et al.,]

Rabee et al propose Smart code snippet pasting. It is also part of the program repair field. When lines of code are being pasted, the surrounding or/and inner code gets modified to make it immediately syntactically correct. It doesn't sound like much at first, but could actually be a big productivity boost. How many times one does not copy and paste code from some website and then spent the next 10 minutes making it work with the current code. Changing variables names, language level (API version by example) and others. It is a very interesting topic, that can be declined in many little improvements. [Rabee Sohail Malik et al.,]

Bhoopchand et al introduce a neural language model with a sparse pointer network aimed at capturing very long range dependencies. Is it a optimised use of a pointer network which is itself an architecture that relies on the attention mechanism used in NLP methods. It can yield better results for application that needs to look for tokens much further back in the source code. [Bhoopchand et al., 2016]

7.4 Code Optimisation

Cummins et al propose a way to learn to build heuristic to optimize complex programs without any hand crafted features like it is usually done. The learned heuristics can be then passed to others programs to apply optimisation and learn more. On heterogeneous parallelism and GPU thread coarsening factors, they come up with solutions that match and beat the state-of-the-art methods that use engineered features. [Cummins et al., 2017]

7.5 Embeddings

There are plenty of different methods deep learning can be applied to source code. But one of the most important step is the embedding. A good and relevant embedding tend to yield better results than whichever neural net used with a wrongly chosen embedding. Which one to use is the heart of the problem of course. Chen et al discussed the matter in a nice structured way [Chen and Monperrus, 2019]. This paper points back to many others and is a great study of the current state of embedding in the field. Four main categories are put forward.

7.5.1 Embeddings of Tokens

Code can be seen as a bunch of tokens. The idea is to map every one of those to a vector. It is the most intuitive idea. It can still be declined into more

advanced methods though. Like taking into account the context of a token. Its vector embedding could be generated not only from the token data, but also by its neighborhood.

7.5.2 Embeddings of Functions or Methods and Embeddings of Sequences or Sets of Methods Calls

Gathering two categories in one. Those two methods share a same idea: a many-to-one mapping. It can be a complete method path generated walking an AST. Or a node coupled with metadata about it such as its depth, its type and its relationship between the node and its parent. Or simply a whole piece of code [2]. In other words, any mapping taking several elements [3] to be embed as a single vector. In comparison to the one-to-one token technique, the way to create the embedding will heavily depend on what results are sought.

7.5.3 Embedding of Binary Code

Using binary code to create an embedding is pretty self-explanatory. It is mainly used to observe similarity between codes with perhaps a different architecture or another programming language.

[2] Still composed of tokens of course, not just plain text.
[3] tokens, nodes, metadata, etc.

Chapter 8

Tree and Graph Architectures

Code can be quite well represented as a tree or a graph. In this document, we are therefore interested in how it might work to use neural networks with these kind of data structure for code. It is good to note that these two are not the only methods used.

8.1 Tree-based Convolutional Neural Network

A tree-based convolutional neural network is a neural network that accepts a tree like structure as input, in opposition with the more traditional way of accepting only a matrix of values. Tree structures are a great way to have relations amongst the data modeled as edges relying nodes. It is handy for source code as we cannot imagine a way to put code directly into a traditional net. Indeed, each source code will have a completely different length and structure. Resizing, padding and other manipulation as we see with images is therefore not an option. And converting code to a tree is not difficult.

8.1.1 Convolution

Convolution has proven its usefulness, mainly with images but not only. It is therefore not crazy to think of convolutions on source code. If a convolution operation takes a piece of a whole image and try to extract a feature from i, it could maybe be used to take a piece of code from a program and do the same.

Comparing image and source code with convolution is fine, but whereas images are easily inserted into a neural net as a matrix of values, code don't have this chance. So there comes a crucial part in the tree-based convolutional

neural network: match the source code to a valid input. A valid input is tree based as the name suggests. Source code needs to be transformed into a tree.

8.1.2 The Network Input

Abstract Syntax Tree

One of the source code strength can be used: its structural formality. The abstract syntax tree (AST), defining the syntax of the code becomes a precious ally in automatic software analysis. It gives exactly what the TBCNNs want as input. An AST can be seen in figure 8.1.

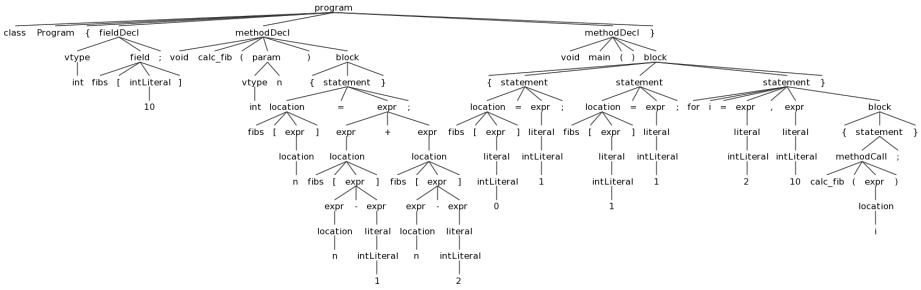

Figure 8.1: An example of abstract syntax tree.

Nodes and Embeddings

The interesting part is now what to put in each node of the AST. It is not as simple as generating the AST and feed it in. Each node should contain a vector. And this vector should be the embedded data of what it desired in this particular node. By example, every node could simply be done using a dummy one-hot encoding, so that there is a clear but simple distinction between each node. Of course this one would be useless, but with a relevant embedding on relevant data, this AST filled with embeddings is now made of vectors and contains the wanted data.

Custom Abstract Tree

AST seems like a general solution. But the only important thing is that the final structure is a tree. So, there is no necessary need of using a default AST. Actually, ASTs tend to be differently generated from one library to the other and with a great amount of clutter information. A cleaning process before the embedding to remove all unwanted miscellaneous data will always be required. Also, an AST often means we care about the tokens of the code more than

anything else. But a tree made with different data, like method names and such is perfectly viable. It really depends on the end goad of the task to perform using the network. In a way, it is the dataset cleaning and preparing phase like any other machine learning workflow.

8.1.3 The Network Process

The TBCNN is conceptually rather simple. It holds in three main steps.

The Network Convolution Step

With the input being a tree, a convolution operation on it needs to be defined. It still has a kernel (or mask), it still slides over the different values, and it still will need pooling later. The goal is to keep it as familiar as possible.

For a node, the convolution is going to be an aggregate of that node and its neighbors, that will generate a new node in another tree of the same size and shape. To make that more clear, here is a figure of the phenomenon 8.2.

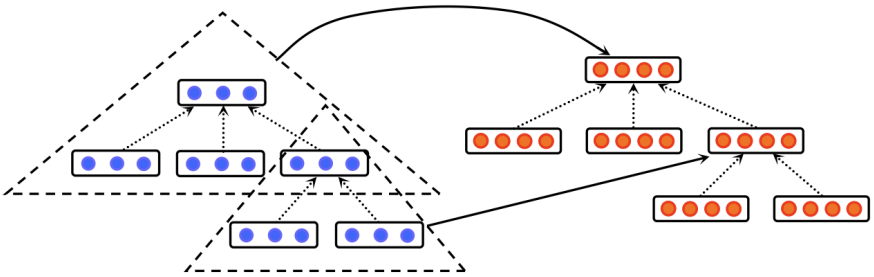

Figure 8.2: Convolution on a tree, reproduced from [Mou et al., 2016]

The triangle window the figure 8.2 shows is the kernel (or mask), that will slide on every node. Its depth and other possible coefficients are parameters of the convolution step.

The Network Pooling Step

The convolution step took a tree of a certain size and generate a new one of the same size. Pooling could seem useless. And it would between several layers of convolution. But to get back to a fully connected layer, a pooling layer is

required. Indeed, any program varies in size, therefore its AST will too. But the fully connected layer has a fixed size.

Pooling with TBCNN is quite rough. The max pooling is working like show in figure 8.3.

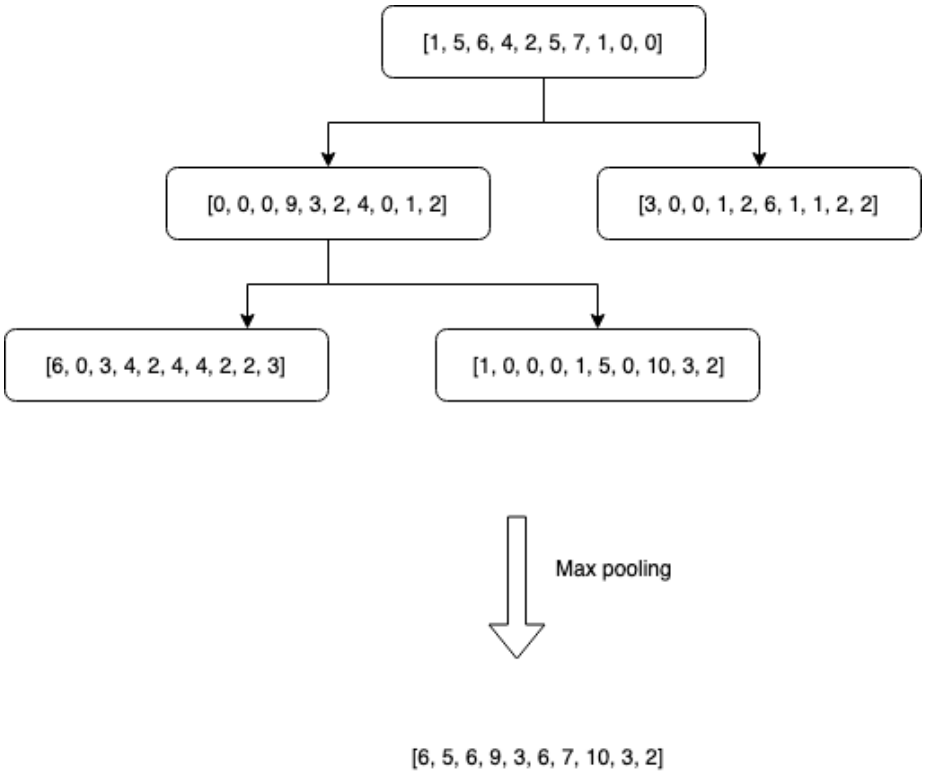

Figure 8.3: Max pooling on a tree

The result in the figure 8.3 is the value that is going through to the fully connected layer, taking the final decision. It means that a lot of the data information has been lost during the pooling process. Hopefully, more performing methods will emerge. Note that there are already alternatives to the max-pooling, but they work in a similar fashion by not taking into account most of the tree.

The Network Decision

After the pooling, there is a regular fully connected layer and softmax. At this point, the network works on vector and not trees, so there is nothing special to add.

8.1.4 Advanced Architectures

More advanced derived architectures exists, such as the bi-TBCNN [Bui et al., 2018]. It gets the perks of the TBCNN and improve upon it. What it does is accepting two inputs alongside and working just as two normal separate TBCNN instances all the way up to the fully connected layer, where they merge. It's like double checking a result. A nice use case would be to better capture the semantic of a piece of code by sending it to the network coded in two distinct languages. It would force the network to care less about the syntax and more about the semantic. Figure 8.4 depicts the phenomenon.

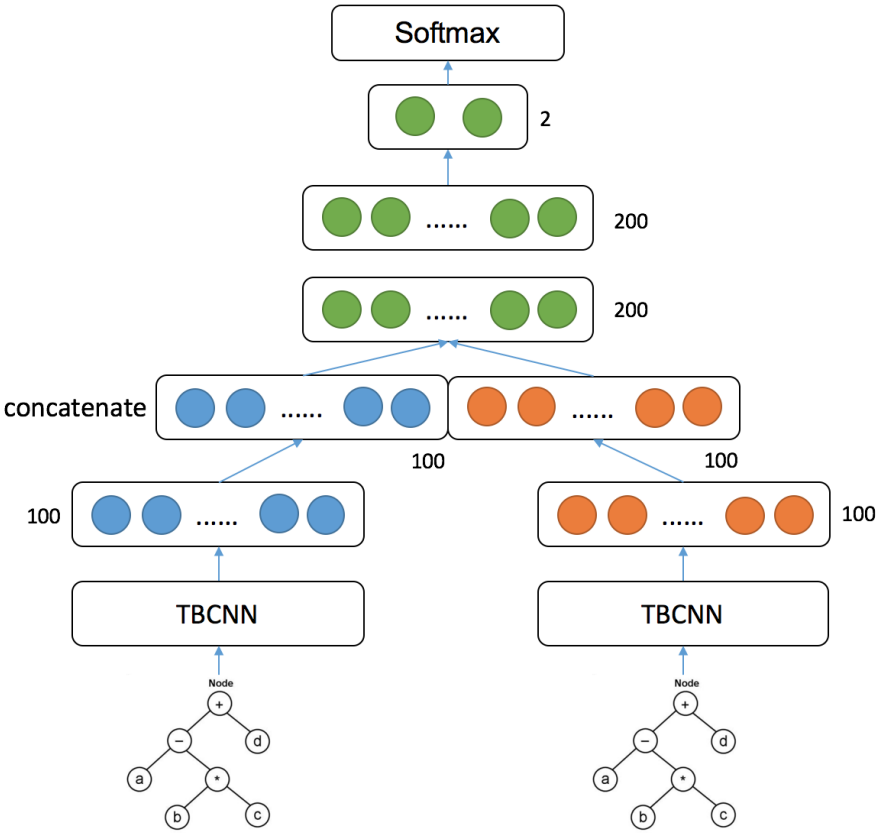

Figure 8.4: A bi-TBCNN simplified architecture.

8.2 Graph-based Convolution Neural Network

Code can be represented as an AST. But if trees are great to use because they are so easy to generate from code, they sure have also limitations. They contain

nodes and *only one* relationship: the child nodes. It is enough to depict the simple raw syntax syntax of source code, but what if one wants more ? The difference between a tree and a graph is that the graph can have as *many* relations from and towards *any* nodes. Those relations are data:

- A relation from a token towards the next token in the source code
- A relation from a variable towards the place it got declared
- A relation from a variable towards the place it got assigned
- And so on

There are as many possibilities as one can think of. It's only a matter of what information can be relevant for a certain task. Put in the graph, any nodes and relations would be part of the embedding.

It also is more general. After all, a tree is a sub-part of a graph. Something standardized is better as it often serves as a base and the future technologies are built on top of it. A developer will spend less time searching how to implement a neural network based on graphs if those are well defined in the frameworks. As a matter of fact, the pytorch (one of the top deep learning library) ecosystem recently welcomed a library just to work with graph structures in neural networks. This will prevent anyone from trying to create again and again a decent implantation and focus more on the actual problems.

Working of such structure instead of a TBCNN implies some differences in the actual implementation concepts. The convolution would now need to not to only be an aggregate of the children nodes, but all the neighbours node. If there are several links between the nodes that represent different relations, those should be taken into account and weighted (for the most important ones have more impacts). The pooling though would still face the same problem of being harsh since the fully connected layer would still be of fixed size.

Use Cases

In figure 8.5, we have an example of a simple line of source code $Assert.NotNull(clazz)$ being transformed in as AST. The benefit in comparison with an typical tree comes up. There is now a "next token" property for the leaf tokens, represented as the orange arrow. It means that this added link can be used by the network to find features. With just this little work, the model now is aware of the order of the element, which wasn't the case before. It might seem insignificant, but in some case, be a crucial information for the network to work properly.

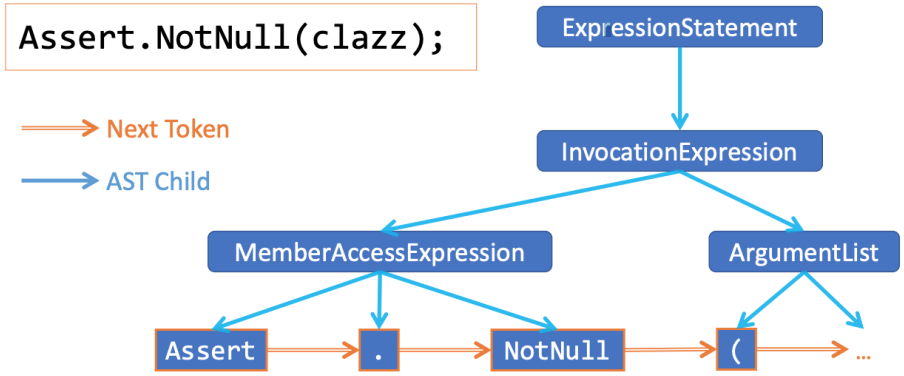

Figure 8.5: AST comes with blue arrows and adding the orange arrows makes it graph, reproduced from `https://miltos.allamanis.com/files/slides/2019fosdem.pdf`

In figure 8.6 can be seen a piece of code to which three kind of relations were added. The later generated graph structure of this code will contain much more information because of the "last write", "last use" and "computed from" properties. Before, with just a tree structure, there was no real way to obtain such information.

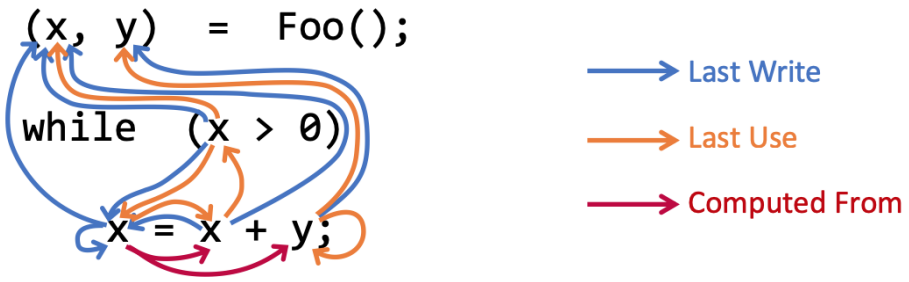

Figure 8.6: Graph including the flow of the source code, reproduced from `https://miltos.allamanis.com/files/slides/2019fosdem.pdf`

The graph structure is therefore very useful to add quickly and simply a lot of features. And it is not so much harder to implement. Indeed, the graph structure is getting used in several fields of deep learning. Because of that, libraries slowly start to support those by default. And since a tree can be considered as a graph, it is fair to say that graphs are a preferred approach over

the tree-based convolutionnal network.

Chapter 9

Deeper Dive into the TBCNN

The tree-based convolutional neural network was greatly defined by Mou et al [Mou et al., 2016]. Some parts are quite vague and it is therefore important to try and reproduce the work to this if it actually performs well.

9.1 Hypothesis

Tree-based convolutional neural network provides good results for code classification. More accurately, it is able to differentiate programs that are different based on a label.

9.2 Experience

An experience to see the results of an application of the TBCNN is conducted.

9.2.1 Dataset

The data to work with are 6 sorting algorithms. For each one, there is roughly more than a hundred piece of code that each implements it. All of this data is scraped from Github repositories. It is good measure to ensure there are the least amount of duplicates as it tends to happen a lot with Github data scrapping.

9.2.2 Data Preparation

The data preparation step is where the programs are transformed into abstract syntax trees. There are a few tools to do that, including one from the official python API. The tree retrieved from the AST generation needs cleaning. ASTs can hold a lot of data and only some of it matter for our experiment. Only the tokens matter, the raw syntax of the program. After this step, there are hundreds of clean ASTs. It is of course required to still keep what is the algorithm in question, thus a label is applied for each AST.

9.2.3 Embedding

The original Mou et al's paper reference back to another one of their paper [Peng et al., 2014] where they generate embedding actually using a complete TBCNN architecture. They get decent results, and similarities can be observed between tokens just like in figure 9.1.

Query	Results		
	Most Similar		**Most Dissimilar**
ID	BinaryOp, Constant, ArrayRef, Assignment, StructRef	···	PtrDecl, Compound, Root, Decl, TypeDecl
Constant	ID, UnaryOp, StructRef, ArrayRef, Cast	···	EnumeratorList, ExprList, If, FuncDef, Compound
BinaryOp	ArrayRef, Assignment, StructRef, UnaryOp, ID	···	PtrDecl, Compound, FuncDecl, Decl, TypeDecl
ArrayRef	BinaryOp, StructRef, UnaryOp, Assignment, Return	···	Compound, PtrDecl, FuncDecl, Decl, TypeDecl
If	For, Compound, Break, While, Case	···	BinaryOp, TypeDecl, Constant, Decl, ID
For	If, While, Case, Break, Struct	···	BinaryOp, Constant, ID, TypeDecl, Decl
Break	While, Case, Continue, Switch, InitList	···	BinaryOp, Constant, TypeDecl, Decl, ID
While	Switch , Continue , Label , Goto	···	BinaryOp, Constant, Decl, TypeDecl, ID
FuncDecl	ArrayDecl, PtrDecl, FuncDef, Typename, Root	···	ArrayRef, FuncCall, IdentifierType, BinaryOp, ID
ArrayDecl	FuncDecl, PtrDecl, Typename, FuncDef, While	···	BinaryOp, Constant, FuncCall, IdentifierType, ID
PtrDecl	FuncDecl, Typename, FuncDef, ArrayDecl	···	FuncCall, ArrayRef, Constant, BinaryOp, ID

Figure 9.1: Example of the nearest neighbour query results on the C programming language, reproduced from [Peng et al., 2014]

But from their tests, this embedding do not work better or barely than other ones that are much simpler to implement. Therefore, the implementation of the experiment uses an embedding called ast2vec.This trains a vector embedding of an AST using a strategy similar to word2vec, but applied to the context of ASTs. The ast2vec networks need to be trained with all python tokens and after using it, the result is an AST with the nodes embedded. It is quite complicated to make it work properly as code written in python 2 and python 3 have different tokens names. Here is the final list of existing tokens used for training:

```
TOKEN_LIST = [
    'Module','Interactive','Expression','FunctionDef','ClassDef','Return',
    'Delete','Assign','AugAssign','Print','For','While','If','With','Raise',
    'TryExcept','TryFinally','Assert','Import','ImportFrom','Exec','Global',
    'Expr','Pass','Break','Continue','attributes','BoolOp','BinOp','UnaryOp',
```

```
    'Lambda','IfExp','Dict','Set','ListComp','SetComp','DictComp',
    'GeneratorExp','Yield','Compare','Call','Repr','Num','Str','Attribute',
    'Subscript','Name','List','Tuple','Load','Store','Del',
    'AugLoad','AugStore','Param','Ellipsis','Slice','ExtSlice','Index','And',
    'Or','Add','Sub','Mult','Div','Mod','Pow','LShift','RShift','BitOr','BitXor',
    'BitAnd','FloorDiv','Invert','Not','UAdd','USub','Eq','NotEq','Lt',
    'LtE','Gt','GtE','Is','IsNot','In','NotIn','comprehension','ExceptHandler',
    'arguments','keyword','alias', 'arg', 'NameConstant', 'JoinedStr',
    'FormattedValue',  'withitem', 'Try', 'Starred'
]
```

This will accept python 2 and python 3 syntax from the dataset.

9.2.4 Network Implementation

Information were scarce for this part. No example and some steps hard to understand. Here are a few of the things encountered during the implementation.

Network input

There is two paths possible here. The first one, to actually work with a tree like structure, manipulating it through all the network. This is basically how it should be and how it was presented in the TBCNN part. This method is way better for understanding and maintaining the code. But libraries support for graph like structures only start to come up, and doing it yourself with a typical pytorch or tensorflow workflow is awfully complicated. They are not made for tree structures at all, but just vectors. Our personal implementation using this method suffered heavy performances issues, making the model very neatly written but totally unpractical. It can be seen in appendix B. The second one is to transform this structure into a high dimension vector. Conceptually, there is a tree, but the network input is a vector. It is hard to understand unless perhaps with the habit of working in the field. It is basically a huge tensor of 4 or more dimensions, that contains all of the same information than the graph but in a completely different way. Working with such data is overwhelming. But the bright side is that the performance are great, since current deep learning libraries are optimized for this kind of data structures.

Whatever the chosen methods, the network will keep the same mathematical description. But every bits of the implantation will change. Using the second method is somewhat annoying though, as it loses many of the advantages cited for the TBCNN. Recent coming libraries are the answer to this problem.

9.3 Validity

The results are good. It can say with a 95% accuracy what the sort algorithm was. Here is an example with a bubble sort. The code is:

```
def bubble_sort(data):
        swap_flag = True
        index = 0
        swap_count = 0

        while swap_flag == True:
                while index < len(data)-1:
                        if data[index] > data[index+1]:
                                data[index], data[index+1] = data[index+1],data[index]
                                swap_count += 1
                        index += 1

                if swap_count == 0:
                        swap_flag = False
                else:
                        swap_count = 0
                        index = 0

        return data

assert bubble_sort([6,4,7,8]) == [4,6,7,8]
assert bubble_sort([4,3,1,2]) == [1,2,3,4]
```

The network successfully class this as a bubble sort implementation. Note that it doesn't read the names of the functions and variables in our case, so it is not cheating by looking up the $bubble_sort$ identifier. Each token have an impact on the final decision. Here is a graph in figure 9.2 showing this importance, from most impact top-left to the least bottom-right.

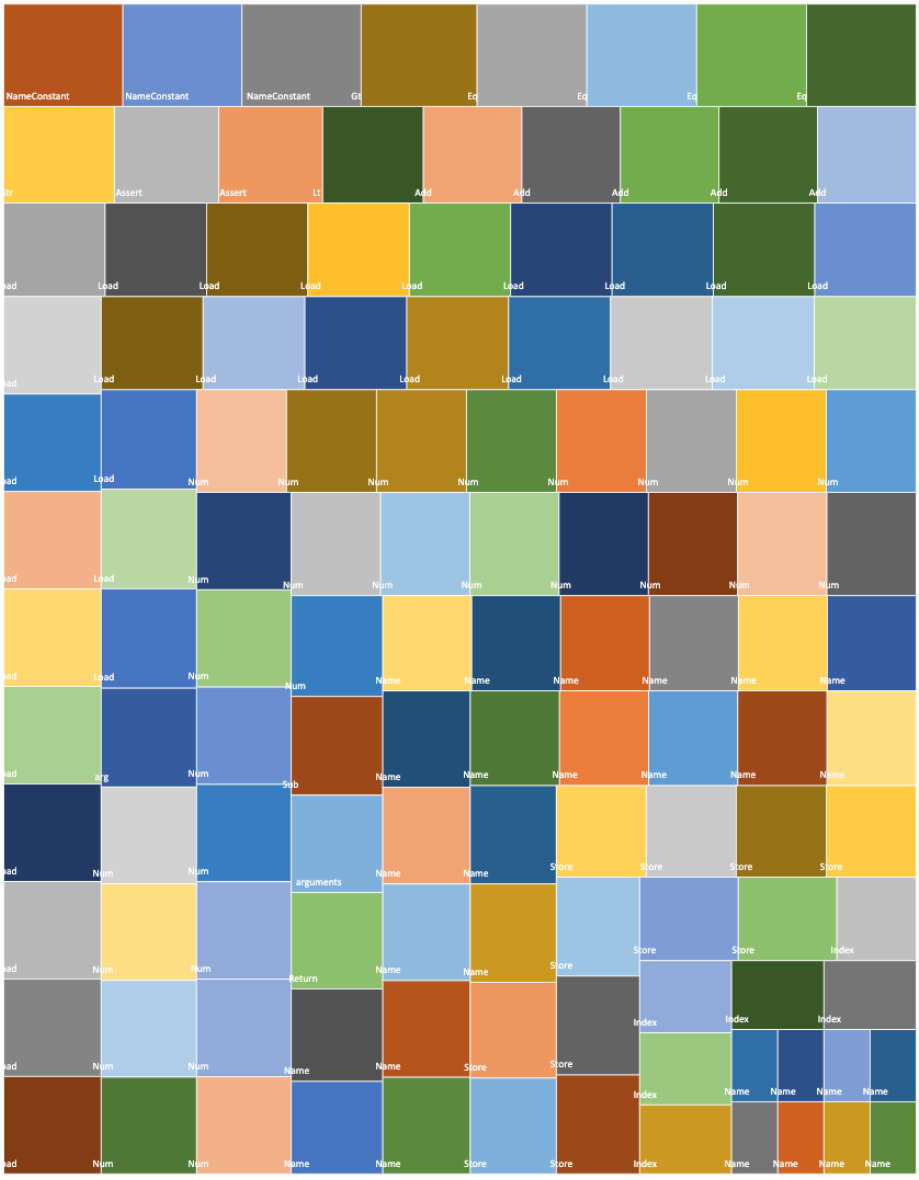

Figure 9.2: Graph depicted the importance of each token of the tree towards the decision of the classification.

It is normal that many tokens are duplicates. The same token type can have different importance depending its position. In appendix 2, more data about the experiment is available.

We can though say that it did extract some new features. But it is hard to tell if they are relevant. After all, this version of the TBCNN just look for

the existances of some tokens to classify code. It doesn't feel like this network learned much of the semantic of our program. It is hard to interpret them as well, because there is nothing to really interpret here.

But adding more relations to the data structure like it has been discussed in the GBCNN chapter might be a good solution towards the right direction.

Chapter 10

Conclusion

Deep learning applied to code analysis is definitely a field in expansion. Studies on all fronts of the software quality and analysis domain are being done.

For the TBCNN, good results with small pieces of code are common but there is still work on full length programs. Indeed, it was shown that the features extracted do not seem to truly have a semantic meaning. It is now interesting to see what the future can bring, with new rising more complex network architectures.

For future work, a very interesting direction to take for all this would be education. Grading a lot of students is a lot of work. And it can be hard to get it properly done as beginner programmers can use wrong syntax but have a valid semantic and vice-versa. Using deep learning, one could possibly get out how far from a default source code a student is. It would greatly help giving students feedback on how their thinking process might be wrong. It also could bring some nice visualisations on how the class perform a certain task or see the programming patterns students follow. Another interesting topic is the actual analysis of code depots and not just source code. Code depots are services like PyPi, NPM, Composer and alikes. It is the places where developers upload the librairies they build so others can have them as dependencies. An example comes from Bommarito and Bommarito proposing an empirical study on "PyPi", python library services. It brings insight not just from the source code, but from other data types such as how often they update their packages, how they version their packages, how many actually never make it to a production stage, and so on. [Bommarito and Bommarito, 2019].

Appendix A

TBCNN Implementations

Here are the codes for the networks used. Of course, the projects contain more than just the network implementation but this is the part of interest.

A.1 Using Tensorflow and Vectors with High Dimensions, adapted from crestonbunch on Github

```
"""Build a CNN network that learns a convolution over a tree structure as
described in Lili Mou et al. (2015) https://arxiv.org/pdf/1409.5718.pdf"""

import math
import tensorflow as tf


def init_net(feature_size, label_size):
    """Initialize an empty network."""

    with tf.name_scope('inputs'):
        """
        Notes:
            Number of nodes is decided by the biggest tree in the batch
            children number is decided by the max number of children there are
        """

        # shape is (batch_size, number_of_nodes, feature_size)
```

```
19          nodes = tf.placeholder(tf.float32, shape=(None, None,
            ↪  feature_size), name='tree')
20
21          # shape is (batch_size, number_of_nodes, child_number)
22          children = tf.placeholder(tf.int32, shape=(None, None,
            ↪  None), name='children')
23
24      with tf.name_scope('network'):
25          conv1 = conv_layer(10, 50, nodes, children, feature_size)
26          # conv2 = conv_layer(1, 10, conv1, children, 100)
27          pooling = pooling_layer(conv1)
28          hidden = hidden_layer(pooling, 500, label_size)
29
30      with tf.name_scope('summaries'):
31          tf.summary.scalar('tree_size', tf.shape(nodes)[1])
32          tf.summary.scalar('child_size', tf.shape(children)[2])
33          tf.summary.histogram('logits', hidden)
34          tf.summary.image('inputs', tf.expand_dims(nodes, axis=3))
35          tf.summary.image('conv1', tf.expand_dims(conv1, axis=3))
36          # tf.summary.image('conv2', tf.expand_dims(conv2,
            ↪  axis=3))
37
38      return nodes, children, hidden
39
40
41  def conv_layer(num_conv, output_size, nodes, children,
    ↪  feature_size):
42      """Creates a convolution layer with num_conv convolutions
        ↪  merged together at
43      the output. Final output will be a tensor with shape
44      [batch_size, num_nodes, output_size * num_conv]"""
45
46      with tf.name_scope('conv_layer'):
47          nodes = [
48              conv_node(nodes, children, feature_size, output_size)
49              for _ in range(num_conv)
50          ]
51          return tf.concat(nodes, axis=2)
52
53
54  def conv_node(nodes, children, feature_size, output_size):
55      """Perform convolutions over every batch sample."""
56      with tf.name_scope('conv_node'):
57          std = 1.0 / math.sqrt(feature_size)
58          w_t, w_l, w_r = (
59              tf.Variable(tf.truncated_normal([feature_size,
                ↪  output_size], stddev=std), name='Wt'),
60              tf.Variable(tf.truncated_normal([feature_size,
                ↪  output_size], stddev=std), name='Wl'),
```

```
61              tf.Variable(tf.truncated_normal([feature_size,
                ↪  output_size], stddev=std), name='Wr'),
62          )
63          init = tf.truncated_normal([output_size, ],
            ↪  stddev=math.sqrt(2.0 / feature_size))
64          # init = tf.zeros([output_size,])
65          b_conv = tf.Variable(init, name='b_conv')
66
67          with tf.name_scope('summaries'):
68              tf.summary.histogram('w_t', [w_t])
69              tf.summary.histogram('w_l', [w_l])
70              tf.summary.histogram('w_r', [w_r])
71              tf.summary.histogram('b_conv', [b_conv])
72
73          return conv_step(nodes, children, feature_size, w_t, w_r,
            ↪  w_l, b_conv)
74
75
76  def children_tensor(nodes, children, feature_size):
77      """Build the children tensor from the input nodes and child
        ↪  lookup."""
78      with tf.name_scope('children_tensor'):
79          max_children = tf.shape(children)[2]
80          batch_size = tf.shape(nodes)[0]
81          num_nodes = tf.shape(nodes)[1]
82
83          # replace the root node with the zero vector so lookups
            ↪  for the 0th
84          # vector return 0 instead of the root vector
85          # zero_vecs is (batch_size, num_nodes, 1)
86          zero_vecs = tf.zeros((batch_size, 1, feature_size))
87          # vector_lookup is (batch_size x num_nodes x
            ↪  feature_size)
88          vector_lookup = tf.concat([zero_vecs, nodes[:, 1:, :]],
            ↪  axis=1)
89          # children is (batch_size x num_nodes x num_children x 1)
90          children = tf.expand_dims(children, axis=3)
91          # prepend the batch indices to the 4th dimension of
            ↪  children
92          # batch_indices is (batch_size x 1 x 1 x 1)
93          batch_indices = tf.reshape(tf.range(0, batch_size),
            ↪  (batch_size, 1, 1, 1))
94          # batch_indices is (batch_size x num_nodes x num_children
            ↪  x 1)
95          batch_indices = tf.tile(batch_indices, [1, num_nodes,
            ↪  max_children, 1])
96          # children is (batch_size x num_nodes x num_children x 2)
97          children = tf.concat([batch_indices, children], axis=3)
98          # output will have shape (batch_size x num_nodes x
            ↪  num_children x feature_size)
```

```
99          # NOTE: tf < 1.1 contains a bug that makes backprop not
            ↪ work for this!
100         return tf.gather_nd(vector_lookup, children,
            ↪ name='children')
101
102
103 def eta_t(children):
104     """Compute weight matrix for how much each vector belongs to
        ↪ the 'top'"""
105     with tf.name_scope('coef_t'):
106         # children is shape (batch_size x max_tree_size x
            ↪ max_children)
107         batch_size = tf.shape(children)[0]
108         max_tree_size = tf.shape(children)[1]
109         max_children = tf.shape(children)[2]
110         # eta_t is shape (batch_size x max_tree_size x
            ↪ max_children + 1)
111         return tf.tile(tf.expand_dims(tf.concat(
112             [tf.ones((max_tree_size, 1)),
                ↪ tf.zeros((max_tree_size, max_children))],
113             axis=1), axis=0,
114         ), [batch_size, 1, 1], name='coef_t')
115
116
117 def eta_r(children, t_coef):
118     """Compute weight matrix for how much each vector belongs to
        ↪ the 'right'"""
119     with tf.name_scope('coef_r'):
120         # children is shape (batch_size x max_tree_size x
            ↪ max_children)
121         children = tf.cast(children, tf.float32)
122         batch_size = tf.shape(children)[0]
123         max_tree_size = tf.shape(children)[1]
124         max_children = tf.shape(children)[2]
125
126         # num_siblings is shape (batch_size x max_tree_size x 1)
127         num_siblings = tf.cast(
128             tf.count_nonzero(children, axis=2, keep_dims=True),
129             dtype=tf.float32
130         )
131         # num_siblings is shape (batch_size x max_tree_size x
            ↪ max_children + 1)
132         num_siblings = tf.tile(
133             num_siblings, [1, 1, max_children + 1],
                ↪ name='num_siblings'
134         )
135         # creates a mask of 1's and 0's where 1 means there is a
            ↪ child there
136         # has shape (batch_size x max_tree_size x max_children +
            ↪ 1)
```

```
137         mask = tf.concat(
138             [tf.zeros((batch_size, max_tree_size, 1)),
139              tf.minimum(children, tf.ones(tf.shape(children)))],
140             axis=2, name='mask'
141         )
142 
143         # child indices for every tree (batch_size x
            ↪ max_tree_size x max_children + 1)
144         child_indices = tf.multiply(tf.tile(
145             tf.expand_dims(
146                 tf.expand_dims(
147                     tf.range(-1.0, tf.cast(max_children,
                        ↪ tf.float32), 1.0, dtype=tf.float32),
148                     axis=0
149                 ),
150                 axis=0
151             ),
152             [batch_size, max_tree_size, 1]
153         ), mask, name='child_indices')
154 
155         # weights for every tree node in the case that
            ↪ num_siblings = 0
156         # shape is (batch_size x max_tree_size x max_children +
            ↪ 1)
157         singles = tf.concat(
158             [tf.zeros((batch_size, max_tree_size, 1)),
159              tf.fill((batch_size, max_tree_size, 1), 0.5),
160              tf.zeros((batch_size, max_tree_size, max_children -
                 ↪ 1))],
161             axis=2, name='singles')
162 
163         # eta_r is shape (batch_size x max_tree_size x
            ↪ max_children + 1)
164         return tf.where(
165             tf.equal(num_siblings, 1.0),
166             # avoid division by 0 when num_siblings == 1
167             singles,
168             # the normal case where num_siblings != 1
169             tf.multiply((1.0 - t_coef), tf.divide(child_indices,
                ↪ num_siblings - 1.0)),
170             name='coef_r'
171         )
172 
173 
174 def eta_l(children, coef_t, coef_r):
175     """Compute weight matrix for how much each vector belongs to
        ↪ the 'left'"""
176     with tf.name_scope('coef_l'):
177         children = tf.cast(children, tf.float32)
178         batch_size = tf.shape(children)[0]
```

```
179         max_tree_size = tf.shape(children)[1]
180         # creates a mask of 1's and 0's where 1 means there is a
            ↪ child there
181         # has shape (batch_size x max_tree_size x max_children +
            ↪ 1)
182         mask = tf.concat(
183             [tf.zeros((batch_size, max_tree_size, 1)),
184              tf.minimum(children, tf.ones(tf.shape(children)))],
185             axis=2,
186             name='mask'
187         )
188
189         # eta_l is shape (batch_size x max_tree_size x
            ↪ max_children + 1)
190         return tf.multiply(
191             tf.multiply((1.0 - coef_t), (1.0 - coef_r)), mask,
                ↪ name='coef_l'
192         )
193
194
195 def conv_step(nodes, children, feature_size, w_t, w_r, w_l,
    ↪ b_conv):
196     """Convolve a batch of nodes and children.
197
198     Lots of high dimensional tensors in this function.
    ↪ Intuitively it makes
199     more sense if we did this work with while loops, but
    ↪ computationally this
200     is more efficient. Don't try to wrap your head around all the
    ↪ tensor dot
201     products, just follow the trail of dimensions.
202     """
203     with tf.name_scope('conv_step'):
204         # nodes is shape (batch_size x max_tree_size x
            ↪ feature_size)
205         # children is shape (batch_size x max_tree_size x
            ↪ max_children)
206
207         with tf.name_scope('trees'):
208             # children_vectors will have shape
209             # (batch_size x max_tree_size x max_children x
                ↪ feature_size)
210             children_vectors = children_tensor(nodes, children,
                ↪ feature_size)
211
212             # add a 4th dimension to the nodes tensor
213             nodes = tf.expand_dims(nodes, axis=2)
214             # tree_tensor is shape
215             # (batch_size x max_tree_size x max_children + 1 x
                ↪ feature_size)
```

```
216             tree_tensor = tf.concat([nodes, children_vectors],
                ↪  axis=2, name='trees')
217
218         with tf.name_scope('coefficients'):
219             # coefficient tensors are shape (batch_size x
                ↪  max_tree_size x max_children + 1)
220             c_t = eta_t(children)
221             c_r = eta_r(children, c_t)
222             c_l = eta_l(children, c_t, c_r)
223
224             # concatenate the position coefficients into a tensor
225             # (batch_size x max_tree_size x max_children + 1 x 3)
226             coef = tf.stack([c_t, c_r, c_l], axis=3, name='coef')
227
228         with tf.name_scope('weights'):
229             # stack weight matrices on top to make a weight
                ↪  tensor
230             # (3, feature_size, output_size)
231             weights = tf.stack([w_t, w_r, w_l], axis=0)
232
233         with tf.name_scope('combine'):
234             batch_size = tf.shape(children)[0]
235             max_tree_size = tf.shape(children)[1]
236             max_children = tf.shape(children)[2]
237
238             # reshape for matrix multiplication
239             x = batch_size * max_tree_size
240             y = max_children + 1
241             result = tf.reshape(tree_tensor, (x, y,
                ↪  feature_size))
242             coef = tf.reshape(coef, (x, y, 3))
243             result = tf.matmul(result, coef, transpose_a=True)
244             result = tf.reshape(result, (batch_size,
                ↪  max_tree_size, 3, feature_size))
245
246             # output is (batch_size, max_tree_size, output_size)
247             result = tf.tensordot(result, weights, [[2, 3], [0,
                ↪  1]])
248
249             # output is (batch_size, max_tree_size, output_size)
250             return tf.nn.tanh(result + b_conv, name='conv')
251
252 \begin{lstlisting}[language=Python]
253 def pooling_layer(nodes):
254     """Creates a max dynamic pooling layer from the nodes."""
255     with tf.name_scope("pooling"):
256         pooled = tf.reduce_max(nodes, axis=1)
257         return pooled
258
259
```

```
260 def hidden_layer(pooled, input_size, output_size):
261     """Create a hidden feedforward layer."""
262     with tf.name_scope("hidden"):
263         weights = tf.Variable(
264             tf.truncated_normal(
265                 [input_size, output_size], stddev=1.0 /
                    ↪  math.sqrt(input_size)
266             ),
267             name='weights'
268         )
269 
270         init = tf.truncated_normal([output_size, ],
             ↪  stddev=math.sqrt(2.0 / input_size))
271         # init = tf.zeros([output_size,])
272         biases = tf.Variable(init, name='biases')
273 
274         with tf.name_scope('summaries'):
275             tf.summary.histogram('weights', [weights])
276             tf.summary.histogram('biases', [biases])
277 
278         return tf.nn.tanh(tf.matmul(pooled, weights) + biases)
279 
280 
281 def loss_layer(logits_node, label_size):
282     """Create a loss layer for training."""
283 
284     labels = tf.placeholder(tf.int32, (None, label_size,))
285 
286     with tf.name_scope('loss_layer'):
287         cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
288             labels=labels, logits=logits_node,
                ↪  name='cross_entropy'
289         )
290 
291         loss = tf.reduce_mean(cross_entropy,
             ↪  name='cross_entropy_mean')
292 
293         return labels, loss
294 
295 
296 def out_layer(logits_node):
297     """Apply softmax to the output layer."""
298     with tf.name_scope('output'):
299         return tf.nn.softmax(logits_node)
```

A.2 Using Pytorch and Graph Architecture

```
from torch import nn
import torch
from torch.nn import init
import torch.nn.functional as F

from network.convolutional_layer import ConvolutionalLayer


class TBCNNMaxPooling(nn.Module):

    def __init__(self):
        super(TBCNNMaxPooling, self).__init__()

    def forward(self, nodes):
        return nodes.max(1)[0]

class TBCNN(nn.Module):
    """
    Tree-Based Convolutional Neural Network (TBCNN)
    """

    def __init__(self, convolutional_features, output_size,
    ↪  target_classes_number):
        super(TBCNN, self).__init__()

        # Grab the info we need
        self.convolutional_features = convolutional_features
        self.hidden_size = output_size
        self.target_classes_number = target_classes_number

        self.conv_layer = ConvolutionalLayer(30,
        ↪  self.hidden_size)

        self.pooling_layer = TBCNNMaxPooling()

        self.layer_linear_1 = nn.Linear(self.hidden_size,
        ↪  self.hidden_size)
        self.layer_relu = nn.LeakyReLU()
        self.layer_linear_2 = nn.Linear(self.hidden_size,
        ↪  self.target_classes_number)

        self.features = [self.conv_layer, self.pooling_layer,
        ↪  self.layer_linear_1, self.layer_relu,
        ↪  self.layer_linear_2]

        # Define the convolutional layer

```

```
        # Initialize the network
        self._initialization()


    def _initialization(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                init.xavier_normal_(m.weight.data)
                if m.bias is not None:
                    init.normal_(m.bias.data)

    def forward(self, nodes, children):

        conv = self.conv_layer(1, nodes, children)

        pooling = self.pooling_layer(conv)

        features = self.layer_linear_1(pooling)

        features = self.layer_relu(features)

        features = self.layer_linear_2(features)

        return features
```

```
from typing import List, Any

import torch
from torch import nn

from network.convolutional_node import ConvolutionalNode


class ConvolutionalLayer(nn.Module):

    def __init__(self, num_features, output_size):
        super(ConvolutionalLayer, self).__init__()

        self.num_features = num_features
        self.output_size = output_size

        self.conv_node = ConvolutionalNode(num_features,
        ↪  output_size)

    def forward(self, num_conv, nodes, children):

        nodes = [
            self.conv_node(nodes, children, self.num_features)
            for _ in range(num_conv)
```

```
        ]

        result = torch.cat(nodes, 2)

        return result


import torch
from torch import nn

from network.convolutional_step import ConvolutionalStep


class ConvolutionalNode(nn.Module):

    def __init__(self, num_features, output_size):
        super(ConvolutionalNode, self).__init__()

        self.num_features = num_features
        self.output_size = output_size
        self.conv_step = ConvolutionalStep(num_features)

        self.w_t =
        ↪  torch.nn.Parameter(data=torch.Tensor(self.num_features,
        ↪  self.output_size), requires_grad=True)
        self.w_t.data.uniform_(-1, 1)

        self.w_r =
        ↪  torch.nn.Parameter(data=torch.Tensor(self.num_features,
        ↪  self.output_size), requires_grad=True)
        self.w_r.data.uniform_(-1, 1)

        self.w_l =
        ↪  torch.nn.Parameter(data=torch.Tensor(self.num_features,
        ↪  self.output_size), requires_grad=True)
        self.w_l.data.uniform_(-1, 1)

        self.b_conv =
        ↪  torch.nn.Parameter(data=torch.Tensor(self.output_size),
        ↪  requires_grad=True)
        self.b_conv.data.uniform_(-1, 1)

        # self.w_t = torch.randn(self.num_features,
        ↪  self.output_size, requires_grad=True)
        # self.w_l = torch.randn(self.num_features,
        ↪  self.output_size, requires_grad=True)
        # self.w_r = torch.randn(self.num_features,
        ↪  self.output_size, requires_grad=True)
        # self.b_conv = Variable(torch.randn(self.output_size))

```

```
33      def forward(self, nodes, children, feature_size):
34          """Perform convolutions over every batch sample."""
35
36          batch_size = children.shape[0]
37
38          max_tree_size = children.shape[1]
39
40          max_children = children.shape[2]
41
42          conv_result = self.conv_step(nodes, children,
            ↪  feature_size, self.w_t, self.w_r, self.w_l,
            ↪  self.b_conv)
43
44          return conv_result


1   import torch
2   from torch import nn
3
4   from network.childrentensor import ChildrenTensor
5   from function_util import tensordot, tile
6
7
8   class ConvolutionalStep(nn.Module):
9       def __init__(self, num_features):
10          super(ConvolutionalStep, self).__init__()
11
12          self.num_features = num_features
13          self.children_tensor = ChildrenTensor(num_features)
14
15      def eta_l(self, children, coef_t, coef_r):
16          """Compute weight matrix for how much each vector belongs
            ↪  to the 'left'"""
17          # creates a mask of 1's and 0's where 1 means there is a
            ↪  child there
18          # has shape (batch_size x max_tree_size x max_children +
            ↪  1)
19          batch_size = children.shape[0]
20          max_tree_size = children.shape[1]
21          max_children = children.shape[2]
22
23          mask = torch.cat((
24              torch.zeros((batch_size, max_tree_size, 1)),
25              torch.min(children, torch.ones((batch_size,
                ↪  max_tree_size, max_children))))
26              , 2)
27
28          # eta_l is shape (batch_size x max_tree_size x
            ↪  max_children + 1)
```

```
29          result = torch.mul(torch.mul((1.0 - coef_t), (1.0 -
            ↪ coef_r)), mask)
30          return result
31
32      def eta_t(self, children):
33          """
34          Compute weight matrix for how much each vector belongs to
    ↪ the 'top'
35
36          This part is tricky, this implementation only slide over
    ↪ a window of depth `, which means a child node in a window
37          always has depth = 1, according to the formula in the
    ↪ original paper, top-coefficient in this case is alwasy 0/1 =
    ↪ 1
38          """
39
40          batch_size = children.shape[0]
41          max_tree_size = children.shape[1]
42          max_children = children.shape[2]
43
44          # eta_t is shape (batch_size x max_tree_size x
            ↪ max_children + 1)
45          eta = torch.cat((torch.ones((max_tree_size, 1)),
            ↪ torch.zeros((max_tree_size, max_children))), 1)
46          eta = eta.unsqueeze(0)
47          eta = tile(eta, 0, batch_size)
48          return eta
49
50      def eta_r(self, children, coef_t):
51          """Compute weight matrix for how much each vector belogs
            ↪ to the 'right'"""
52          # children is batch_size x max_tree_size x max_children
53          batch_size = children.shape[0]
54          max_tree_size = children.shape[1]
55          max_children = children.shape[2]
56
57          # num_siblings is shape (batch_size x max_tree_size x 1)
58          num_siblings = max_children - (children == 0).sum(dim=2,
            ↪ keepdim=True)
59
60          # num_siblings is shape (batch_size x max_tree_size x
            ↪ max_children + 1)
61          num_siblings = tile(num_siblings, 2, max_children +
            ↪ 1).float()
62
63          # creates a mask of 1's and 0's where 1 means there is a
            ↪ child there
64          # has shape (batch_size x max_tree_size x max_children +
            ↪ 1)
65          mask = torch.cat((
```

```
            torch.zeros((batch_size, max_tree_size, 1)),
            torch.min(children.float(), torch.ones((batch_size,
             ↪  max_tree_size, max_children)))
        ), 2)

        # child indices for every tree (batch_size x
         ↪  max_tree_size x max_children + 1)
        child_indices = torch.arange(-1.0, float(max_children),
         ↪  1.0)
        child_indices = child_indices.unsqueeze(0)
        child_indices = child_indices.unsqueeze(0)
        child_indices = tile(child_indices, 0, batch_size)
        child_indices = tile(child_indices, 1, max_tree_size)
        child_indices = torch.mul(child_indices, mask)

        # weights for every tree node in the case that
         ↪  num_siblings = 0
        # shape is (batch_size x max_tree_size x max_children +
         ↪  1)
        """
        singles = torch.cat((torch.zeros((batch_size,
    ↪  max_tree_size, 1)),

    ↪  torch.tensor(()).new_full((batch_size, max_tree_size, 1),
    ↪  0.5),
                             torch.zeros((batch_size,
    ↪  max_tree_size, max_children - 1))), 2)
        """



        # eta_r is shape (batch_size x max_tree_size x
         ↪  max_children + 1)
        num_siblings = torch.where(
            torch.eq(num_siblings, 1),
            torch.ones(num_siblings.shape) + 1,
            num_siblings
        )


        result = torch.mul((1.0 - coef_t),
         ↪  torch.div(child_indices, num_siblings - 1.0))

        return result

    def forward(self, nodes, children, feature_size, w_t, w_r,
     ↪  w_l, b_conv):
        """Convolve a batch of nodes and children.

```

```
103         Lots of high dimensional tensors in this function.
    ↪ Intuitively it makes
104         more sense if we did this work with while loops, but
    ↪ computationally this
105         is more efficient. Don't try to wrap your head around all
    ↪ the tensor dot
106         products, just follow the trail of dimensions.
107         """
108
109         # nodes is shape (batch_size x max_tree_size x
            ↪ feature_size)
110         batch_size = children.shape[0]
111         max_tree_size = children.shape[1]
112         max_children = children.shape[2]
113
114         # children is shape (batch_size x max_tree_size x
            ↪ max_children)
115         # children_tensor = CHILDREN_TENSOR(self.max_children,
            ↪ self.batch_size, self.max_tree_size, feature_size,
            ↪ self.opt)
116         children_vectors = self.children_tensor(nodes, children,
            ↪ feature_size)
117         # add a 4th dimension to the nodes tensor
118         nodes = nodes.unsqueeze(2)
119         # tree_tensor is shape (batch_size x max_tree_size x
            ↪ max_children + 1 x feature_size)
120
121         tree_tensor = torch.cat((nodes, children_vectors), 2)
122
123         c_t = self.eta_t(children)
124         c_r = self.eta_r(children, c_t)
125         c_l = self.eta_l(children, c_t, c_r)
126         # coef = self.coefficients(children)
127         coef = torch.stack((c_t, c_r, c_l), 3)
128
129         weights = torch.stack([w_t, w_r, w_l], 0)
130
131         # reshape for matrix multiplication
132         x = batch_size * max_tree_size
133         y = max_children + 1
134
135         result = tree_tensor.view(x, y, feature_size)
136
137         coef = coef.view(x, y, 3)
138
139         result = torch.matmul(torch.transpose(result, 1, 2),
            ↪ coef)
140         result = result.view(batch_size, max_tree_size, 3,
            ↪ feature_size)
141
```

```
        # # output is (batch_size, max_tree_size, output_size)
        result = tensordot(result, weights, [[2, 3], [0, 1]])
        # # output is (batch_size, max_tree_size, output_size)

        return torch.tanh(result + b_conv)
```

```
import torch
from torch import nn

from function_util import tile


class ChildrenTensor(nn.Module):

    def __init__(self, num_features):
        super(ChildrenTensor, self).__init__()

        self.num_features = num_features

    def forward(self, nodes, children, feature_size):

        # children is batch x num_nodes
        batch_size = children.shape[0]
        num_nodes = children.shape[1]
        max_children = children.shape[2]

        # replace the root node with the zero vector so lookups
        ↪  for the 0th
        # vector return 0 instead of the root vector
        # zero_vecs is (batch_size, num_nodes, 1)
        zero_vecs = torch.zeros((batch_size, 1,
        ↪  self.num_features))

        # vector_lookup is (batch_size x num_nodes x
        ↪  feature_size)
        # print("Shape zero vec : " + str(zero_vecs.shape))
        vector_lookup = torch.cat((zero_vecs, nodes[:, 1:, :]),
        ↪  1)

        # print("Vector look up : " + str(vector_lookup.shape))
        # children is (batch_size x num_nodes x num_children x 1)
        children = children.unsqueeze(3)

        # prepend the batch indices to the 4th dimension of
        ↪  children
        # batch_indices is (batch_size x 1 x 1 x 1)
        batch_indices = torch.arange(0, batch_size)
        batch_indices = batch_indices.view(batch_size, 1, 1, 1)

```

```
        # batch_indices is (batch_size x num_nodes x num_children x 1)
        batch_indices = tile(batch_indices, 1, num_nodes)
        batch_indices = tile(batch_indices, 2, max_children)

        # children is (batch_size x num_nodes x num_children x 2)
        children = torch.cat((batch_indices.float(), children), 3)

        # output will have shape (batch_size x num_nodes x num_children x feature_size)
        # NOTE: tf < 1.1 contains a bug that makes backprop not work for this!

        result = vector_lookup[children[:, :, :, 0].long(), children[:, :, :, 1].long(), :]

        return result
```

Appendix B

TBCNN raw saliency data

```
Expr          +0.050902309826   : '\nThe bubble sort algorithm compares every two items which are next to each other, \nand swaps them if they are in the wrong order. \nAn array of n elements can be sorted within n-1 passes. \n\nFor example, in this array of 4 items:\n\nFirst pass\n(4, 3, 1, 2) > (3, 4, 1, 2)\n(3, 4, 1, 2) > (3, 1, 4, 2)\n(3, 1, 4, 2) > (3, 1, 2, 4)\n\nSecond pass:\n(3, 1, 2, 4) > (1, 3, 2, 4)\n(1, 3, 2, 4) > (1, 2, 3, 4)\n(1, 2, 3, 4) > (1, 2, 3, 4)\n\nThird pass:\n(1, 2, 3, 4) > (1, 2, 3, 4)\n(1, 2, 3, 4) > (1, 2, 3, 4)\n(1, 2, 3, 4) > (1, 2, 3, 4)\n\n\n'
FunctionDef   -0.222295941589   : def bubble_sort(data):
    swap_flag = True
    index = 0
    swap_count = 0
    while (swap_flag == True):
        while (index < (len(data) - 1)):
            if (data[index] > data[(index + 1)]):
                (data[index], data[(index + 1)]) = (data[(index + 1)], data[index])
                swap_count += 1
            index += 1
        if (swap_count == 0):
            swap_flag = False
        else:
            swap_count = 0
            index = 0
    return data
Assert        +0.048353655056   : assert (bubble_sort([6, 4, 7, 8]) == [4, 6, 7, 8])
Assert        +0.048353655056   : assert (bubble_sort([4, 3, 1, 2]) == [1, 2, 3, 4])
```

```
Str           +0.051473548412   : '\nThe bubble sort algorithm compares every two items which are next to each other, \nand swaps them if they are in the wrong order. \nAn array of n elements can be sorted within n-1 passes. \n\nFor example, in this array of 4 items:\n\nFirst pass\n(4, 3, 1, 2) > (3, 4, 1, 2)\n(3, 4, 1, 2) > (3, 1, 4, 2)\n(3, 1, 4, 2) > (3, 1, 2, 4)\n\nSecond pass:\n(3, 1, 2, 4) > (1, 3, 2, 4)\n(1, 3, 2, 4) > (1, 2, 3, 4)\n(1, 2, 3, 4) > (1, 2, 3, 4)\n\nThird pass:\n(1, 2, 3, 4) > (1, 2, 3, 4)\n(1, 2, 3, 4) > (1, 2, 3, 4)\n(1, 2, 3, 4) > (1, 2, 3, 4)\n\n\n'
arguments     +0.042746567113   : data
Assign        -0.035972142407   : swap_flag = True
Assign        -0.052940859930   : index = 0
Assign        -0.052940859930   : swap_count = 0
While         -0.115974933376   : while (swap_flag == True):
    while (index < (len(data) - 1)):
        if (data[index] > data[(index + 1)]):
            (data[index], data[(index + 1)]) = (data[(index + 1)], data[index])
            swap_count += 1
        index += 1
    if (swap_count == 0):
        swap_flag = False
    else:
        swap_count = 0
        index = 0
Return        +0.042530625347   : return data
Compare       -0.145189321499   : (bubble_sort([6, 4, 7, 8]) == [4, 6, 7, 8])
Compare       -0.145189321499   : (bubble_sort([4, 3, 1, 2]) == [1, 2, 3, 4])
arg           +0.045342837831   : data
Name          +0.016020464230   : swap_flag
NameConstant  +0.058632828971   : True
Name          +0.016020464230   : index
Num           +0.044367977678   : 0
Name          +0.016020464230   : swap_count
Num           +0.044367977678   : 0
Compare       -0.139545036217   : (swap_flag == True)
While         -0.115558239599   : while (index < (len(data) - 1)):
    if (data[index] > data[(index + 1)]):
        (data[index], data[(index + 1)]) = (data[(index + 1)], data[index])
        swap_count += 1
    index += 1
If            -0.195986491735   : if (swap_count == 0):
    swap_flag = False
else:
    swap_count = 0
```

```
    index = 0
Name          +0.040657457101   : data
Call          -0.058420679766   : bubble_sort([6, 4, 7, 8])
Eq            +0.054043814823   : ==
List          -0.164477496848   : [4, 6, 7, 8]
Call          -0.058420679766   : bubble_sort([4, 3, 1, 2])
Eq            +0.054043814823   : ==
List          -0.164477496848   : [1, 2, 3, 4]
Store         +0.039619160410   : Memory operation
Store         +0.039619160410   : Memory operation
Store         +0.039619160410   : Memory operation
Name          +0.040657457101   : swap_flag
Eq            +0.054043814823   : ==
NameConstant  +0.058632828971   : True
Compare       -0.137680653876   : (index < (len(data) - 1))
If            -0.175283048812   : if (data[index] > data[(index +
↪  1)]):
    (data[index], data[(index + 1)]) = (data[(index + 1)],
    ↪  data[index])
    swap_count += 1
AugAssign     -0.173897925075   : index += 1
Compare       -0.146076451541   : (swap_count == 0)
Assign        -0.035972142407   : swap_flag = False
Assign        -0.052940859930   : swap_count = 0
Assign        -0.052940859930   : index = 0
Load          +0.045631365260   : Memory operation
Name          +0.040657457101   : bubble_sort
List          -0.164477496848   : [6, 4, 7, 8]
Num           +0.044367977678   : 4
Num           +0.044367977678   : 6
Num           +0.044367977678   : 7
Num           +0.044367977678   : 8
Load          +0.045631365260   : Memory operation
Name          +0.040657457101   : bubble_sort
List          -0.164477496848   : [4, 3, 1, 2]
Num           +0.044367977678   : 1
Num           +0.044367977678   : 2
Num           +0.044367977678   : 3
Num           +0.044367977678   : 4
Load          +0.045631365260   : Memory operation
Load          +0.045631365260   : Memory operation
Name          +0.040657457101   : index
Lt            +0.046764030796   : <
BinOp         -0.150902470394   : (len(data) - 1)
Compare       -0.120960774529   : (data[index] > data[(index +
↪  1)])
Assign        -0.068009975267   : (data[index], data[(index +
↪  1)]) = (data[(index + 1)], data[index])
AugAssign     -0.173897925075   : swap_count += 1
Name          +0.016020464230   : index
```

```
Add             +0.045801584061   : +
Num             +0.044367977678   : 1
Name            +0.040657457101   : swap_count
Eq              +0.054043814823   : ==
Num             +0.044367977678   : 0
Name            +0.016020464230   : swap_flag
NameConstant    +0.058632828971   : False
Name            +0.016020464230   : swap_count
Num             +0.044367977678   : 0
Name            +0.016020464230   : index
Num             +0.044367977678   : 0
Load            +0.045631365260   : Memory operation
Num             +0.044367977678   : 6
Num             +0.044367981276   : 4
Num             +0.044367981276   : 7
Num             +0.044367977678   : 8
Load            +0.045631365260   : Memory operation
Load            +0.045631365260   : Memory operation
Num             +0.044367977678   : 4
Num             +0.044367977678   : 3
Num             +0.044367977678   : 1
Num             +0.044367977678   : 2
Load            +0.045631365260   : Memory operation
Load            +0.045631365260   : Memory operation
Call            -0.043237801409   : len(data)
Sub             +0.043407705019   : -
Num             +0.044367977678   : 1
Subscript       -0.167453723011   : data[index]
Gt              +0.057127803132   : >
Subscript       -0.167453723011   : data[(index + 1)]
Tuple           -0.119074295875   : (data[index], data[(index +
    ↪ 1)])
Tuple           -0.107141554343   : (data[(index + 1)],
    ↪ data[index])
Name            +0.016020464230   : swap_count
Add             +0.045801584061   : +
Num             +0.044367977678   : 1
Store           +0.039619160410   : Memory operation
Load            +0.045631365260   : Memory operation
Store           +0.039619160410   : Memory operation
Store           +0.039619160410   : Memory operation
Store           +0.039619160410   : Memory operation
Name            +0.040657457101   : len
Name            +0.040657457101   : data
Name            +0.040657457101   : data
Index           +0.030521157554   : index
Load            +0.045631365260   : Memory operation
Name            +0.040657457101   : data
Index           +0.031862087594   : (index + 1)
Load            +0.045631365260   : Memory operation
```

```
Subscript     -0.178836106945   : data[index]
Subscript     -0.178836106945   : data[(index + 1)]
Store         +0.039619160410   : Memory operation
Subscript     -0.167453723011   : data[(index + 1)]
Subscript     -0.167453723011   : data[index]
Load          +0.045631365260   : Memory operation
Store         +0.039619160410   : Memory operation
Load          +0.045631365260   : Memory operation
Load          +0.045631365260   : Memory operation
Load          +0.045631365260   : Memory operation
Name          +0.040657457101   : index
Load          +0.045631365260   : Memory operation
BinOp         -0.149667067717   : (index + 1)
Name          +0.040657457101   : data
Index         +0.030521157554   : index
Store         +0.039619160410   : Memory operation
Name          +0.040657457101   : data
Index         +0.031862087594   : (index + 1)
Store         +0.039619160410   : Memory operation
Name          +0.040657457101   : data
Index         +0.031862087594   : (index + 1)
Load          +0.045631365260   : Memory operation
Name          +0.040657457101   : data
Index         +0.030521157554   : index
Load          +0.045631365260   : Memory operation
Load          +0.045631365260   : Memory operation
Name          +0.040657457101   : index
Add           +0.045801584061   : +
Num           +0.044367977678   : 1
Load          +0.045631365260   : Memory operation
Name          +0.040657457101   : index
Load          +0.045631365260   : Memory operation
BinOp         -0.149667067717   : (index + 1)
Load          +0.045631365260   : Memory operation
BinOp         -0.149667067717   : (index + 1)
Load          +0.045631365260   : Memory operation
Name          +0.040657457101   : index
Load          +0.045631365260   : Memory operation
Load          +0.045631365260   : Memory operation
Name          +0.040657457101   : index
Add           +0.045801584061   : +
Num           +0.044367977678   : 1
Name          +0.040657457101   : index
Add           +0.045801584061   : +
Num           +0.044367977678   : 1
Load          +0.045631365260   : Memory operation
Load          +0.045631365260   : Memory operation
Load          +0.045631365260   : Memory operation
```

Bibliography

[A. Harer et al., 2018] A. Harer, J., Kim, L., L. Russell, R., Ozdemir, O., R. Kosta, L., Rangamani, A., H. Hamilton, L., I. Centeno, G., R. Key, J., M. Ellingwood, P., W. McConley, M., M. Opper, J., Chin, S., and Lazovich, T. (2018). Automated software vulnerability detection with machine learning.

[Bhoopchand et al., 2016] Bhoopchand, A., Rocktäschel, T., Barr, E., and Riedel, S. (2016). Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*.

[Bommarito and Bommarito, 2019] Bommarito, E. and Bommarito, M. (2019). An empirical analysis of the python package index (pypi). *CoRR*, abs/1907.11073.

[Büch and Andrzejak, 2019] Büch, L. and Andrzejak, A. (2019). Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104. IEEE.

[Bui et al., 2018] Bui, N. D., Jiang, L., and Yu, Y. (2018). Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.

[Chen and Monperrus, 2019] Chen, Z. and Monperrus, M. (2019). A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*.

[Cummins et al., 2017] Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. (2017). End-to-end deep learning of optimization heuristics. *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232.

[Ding et al., 2019] Ding, S. H., Fung, B. C., and Charland, P. (2019). Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization*, page 0. IEEE.

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

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Kacmajor and Kelleher, 2019] Kacmajor, M. and Kelleher, J. D. (2019). Automatic acquisition of annotated training corpora for test-code generation. *Information*, 10(2):66.

[Koehrsen, 2018] Koehrsen, W. (2018). Neural network embeddings explained.

[Mikolov et al., 2013] Mikolov, T., Yih, S. W.-t., and Zweig, G. (2013). Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT-2013)*. Association for Computational Linguistics.

[Mou et al., 2016] Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. (2016). Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[Peng et al., 2014] Peng, H., Mou, L., Li, G., Liu, Y., Zhang, L., and Jin, Z. (2014). Building program vector representations for deep learning. *ArXiv*, abs/1409.3358.

[Rabee Sohail Malik et al.,] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. NL2type: Inferring JavaScript Function Types from Natural Language Information.

[Sun et al., 2019] Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G., and Zhang, L. (2019). A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7055–7062.

[Xu et al., 2019] Xu, S., Yao, Y., Hegde, S., Gu, T., Tong, H., and Lü, J. (2019). Commit message generation for source code changes. In *IJCAI*.

[Yao et al., 2019] Yao, Z., Peddamail, J. R., and Sun, H. (2019). Coacor: Code annotation for code retrieval with reinforcement learning. pages 2203–2214.