Graph- and behaviour-based machine learning models to understand program semantics

Benno Fünfstück
benno.fuenfstueck@tu-dresden.de

ABSTRACT

Applying machine learning to detect patterns in programs requires finding a suitable form to input these programs into models. In this work, we look into two recently explored directions which do not require manually feature engineering: using the graph structure of programs and using execution traces. Both representations are proposed for different reasons: graph-based models can capture the syntatical properties of programs well and perform simple reasoning on them, while execution trace models are closer to program semantics. We explain applications of both models and then compare them on a set of tasks. We find that although both models are used for similar applications, different datasets make it hard to compare them. More research is needed to better explain the strengths of each model.

1 INTRODUCTION

Many tools that operate on source code require a form of program analysis. This includes compilers needing to understand program semantics for optimization, static analyzers designed to prevent bugs in programs or development environments suggesting code completions and improvements. Traditionally program analysis relies on abstraction, since finding an exact solution to the underlying problems is often computationally infeasible or even impossible.

But with the availability of large open source code repositories such as GitHub[1], a new form of program analysis based on methods from machine learning is possible. The fundamental principle of this form of program analysis is captured by the following hypothesis, the natural code hypothesis, as stated by Allamanis et al. [2]:

> Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools.

For example, identifier names are not choosen randomly, but instead based on the meaning of the variable and therefore convey useful information for analysis. This suggests using machine learning models for code.

A straightforward method then is to take models that have proven successful in natural language processing and apply them to source code. Various applications of this pattern have been described in the literature [1, 7, 18, 24]. The advantage of this method is that models working on sequences are well understood. Additionally, since the order of tokens is preserved, the models can also learn from that information. But the disadvantage is that these models generally are not well suited to deal with long-range dependencies such as those found when referring to variables and other structures that might be far away in the token representation of source code.

For this reason, new variants of representing programs as input to machine learning models have been explored. Here, we will focus on two variants: graph-based models that rely on the graph structure of source code and behaviour-based models that make use of the executability of source code to find patterns in data gathered from execution traces. First, we give a description of both models and its application to computer programs. Then, we compare them on two tasks (method naming/classification and invariant inference). We finish we a conclusion summarizing the results and listing some interesting future research directions.

2 GRAPH-BASED MODEL

Graphs are a natural representation for source code, due to its highly interlinked structure. In this section, we will thus look at models that can learn from the graph structure. A flexible model for this task is the message passing neural network, introduced by Gilmer et al. [17]. Message passing neural networks are a variant of graph neural networks. We will first explain the general concept of message passing neural networks in section 2.1. We then compare different concrete instances of this model applied to source code in section 2.2.

Message passing neural networks are not the only kind of model that uses the graph structure for learning. One alternative is described by Raychev et al. [29] where they use conditional random fields to predict types and names of JavaScript variables. Program elements are represented as nodes in a graph with edges between them capturing dependencies in their properties. The probabilistic weight of the dependency can then be learned from data, and the model be used to predict the unknown properties from known properties, taking into accout the dependencies between the predictions. We will however focus on message passing neural networks here since they are the most popular model found in literature and have successfully been applied to several different tasks (see also section 4).

2.1 Message passing neural network

We explain the concept of message passing neural networks following Gilmer et al. [17]. Let G by a directed graph, where edges from w to v have a label represented as e_{vw}. Two nodes can be connected by multiple edges of different kinds. In the application to source code, each edge kind often has an associated backward edge, allowing information to flow in both directions. These can just be treated as an additional edge with a label different from the corresponding forward edge.

Message passing neural networks start by assigning an initial hidden state h_v^0 to each node v in the graph. The initial state is

Unpublished working draft.
Not for distribution.

[1] https://github.com

usually computed as an embedding of the features of the node. At every timestep, the network then computes a message for each edge using the *message function* M. The messages of all edges with the same target are aggregated by an *aggregation function* h. This is a slight generalization of the original formulation, where h is always the sum of all incoming messages. The aggregated messages are then used to update the node hidden state, using the *update function* U for the node. The following two equations describe this process:

$$m_v^{t+1} = h(\{M(h_v^t, h_w^t, e_{vw}) | w \in N(v)\}) \tag{1}$$

$$h_v^{t+1} = U(h_v^t, m^{t+1}) \tag{2}$$

where $N(v)$ is the set of all nodes w for which there is an edge from w to v. The steps are repeated for T timesteps. Finally, the overall prediction is computed by applying the readout function to the final hidden states $\{h_v^T\}$ of all nodes:

$$\hat{y} = R(\{h_v^T | v \in G\}) \tag{3}$$

In this computation M, U and R are all learnable differentiable functions. Because the whole system is differentiable, parameters can be learned with gradient-based optimization.

2.2 Applications to source code

Table 1 shows some applications of (variations of) message passing neural networks to source code. We will first look at different representations of source code as a graph. Then, we describe the concrete variant of message passing neural network used in these applications.

2.2.1 AST representations. A syntatic way to represent source code as a graph is the form of an abstract syntax tree (AST). This representation was first introduced by Allamanis et al. [4] and many further works are based on variations [3, 11, 16, 19, 20, 22, 30]. Semantic information is added to the AST in the form of additional edge types. An overview of the edge types found in these works is given in table 2. These edges add data flow and control flow information for variables to the graph. Since in practice the message passing network only uses a small number of timesteps, these additional edges are useful to capture dependencies that can potentially be far away in the graph. Allamanis et al. found that these semantic edges greatly increase the performance. Schrouff et al. [30] also use an AST representation, but group their 96 relationship types into only two categories: AST relation like child/parent and reference/traverse which are a kind of semantic edge.

Because each edge type requires its own parameters for the message generation step of the model, restricting the edge types can improve training time. Which edges are useful however depends on the task. In an ablation study, Allamanis et al. [4] found that restricting the edges to semantic edges (removing NextToken and Child edges) affected the performance on the VarMisuse task much more than for the VarNaming task.

2.2.2 CFG representations. AST representations include information about syntax that may not be relevant for semantics, such as dead code. By basing their representation on a ontrol flow graph (CFG), CFG representations eliminate some of this "noise". The tradeoff is that some of the superfluous information might still communicate useful information. For example, the order of statements in a control flow graph will be different to the source order, so this information can no longer be used by the model. Brauckmann et al. [11] compare both representations on the same tasks and find that there is no single best representation. They base their graph on a CFG extracted from the LLVM intermediate representation, so the nodes in the graph are LLVM IR instructions. A similar approach is followed by Cummins et al. [14]. They also construct a control flow graph and enrich it with data flow and call flow. In contrast to the work of Brauckmann et al. they include edge positions which differentiate the first operand/branch target from the second and so on. But nodes do not always need to be IR statements in a CFG representation. Si et al. [31] describe a variant where they first construct a control flow graph over statements over the SSA (static single assignment) form but then use the AST representation of each statement as nodes. Since the SSA form splits a single variable in potentially many differently named ones, they introduce variable linking which connects all instances of a single variable to a node named after the original name.

2.2.3 Application-specific graphs. A third possibility to represent programs as graphs is shown by Wei et al. [37]. They construct a type inference graph, which only contains predicates believed to be relevant for the type inference problem, such as subtype and usage edges. Nodes in their graph represent types that are known or need to be assigned. In contrast to the previous representation methods, this requires more thought about which edges to include for a given application and is thus a less general method.

2.2.4 Propagation models. Most of the applications listed in table 1 are based on a special kind of message passing neural network named Gated Graph Neural Network (GGNN) [23]. These networks use a simple linear layer on the source node as the message function (with different trainable parameters for each edge kind) and the gated recurrent unit (GRU) [13] as the node update function. For message aggregation, GGNNs sum all messages arriving at a single node.

There are only two models that are not based on the GGNN variant. Wei et al. [37] use a different model where they define specialized learnable message functions for each of their edge types. Since their graph is a hypergraph, they specify how to generate messages for each argument of a hyperedge. The specialized message functions are mostly based on multilayer feedforward networks. For aggregation and updating, they use a variant of the attentation based aggregation operator proposed in graph attentation networks [33].

Si et al. [31] also use a linear function to compute messages for each edges. Messages are aggregated separately for each edge type by summing and applying a nonlinear activation function. The update is performed by transforming each aggregated result by a linear function followed by a nonlinear activation function. Interesting to note here is that the update function is not based on the hidden state of the current node. This means that the model cannot directly propagate information from a node to itself in one step.

The remaining models either use the GGNN model directly [4, 11, 16, 19, 20, 22, 30] or apply slight changes to the message or

Unpublished working draft. Not for distribution.

Reference	Graph	Task
Allamanis et al. [4]	enriched AST	VarNaming, VarMisuse
Allamanis et al. [3]	enriched AST	predict python type annotations
Brauckmann et al. [11]	enriched AST, CDFG+calls+mem	OpenCL device mapping, OpenCL thread corsening
Cummins et al. [14]	IR graph + control-, data- and call-flow	graph algorithms, device mapping, algorithm classification
Fernandes et al. [16]	enriched AST	MethodNaming, MethodDoc
Hellendoorn et al. [19]	enriched AST	checking if extracted invariants are valid
Hellendoorn et al. [20]	enriched AST	VarMisuse
Li et al. [22]	enriched AST	predict log level for log statements
Schrouff et al. [30]	AST + reference/traverse	predict javascript type annotations
Si et al. [31]	SSA-CFG with AST + variable-linking	embed semantics for loop invariant prediction
Wei et al. [37]	type variables with relationships	predict python type annotations

Table 1: Different applications of message passing neural networks to source code tasks

Unpublished working draft.
Not for distribution.

Edge	Edge between ...
NextToken	two consecutive token nodes or sub-token nodes
InToken	sub-tokens and the AST token node
NextLexicalUse	lexical uses of the same variable (independent of data flow)
Child	AST syntax nodes and their children
NextMayUse	variable tokens and next potential uses of that variable
AssignedFrom	left hand side of an assignment and right hand side
ComputedFrom	variable appearing in the LHS of an assignment and all variables appearing in the RHS
OccurrenceOf	tokens referencing a symbol and nodes that bind this symbol
FormalArgName	arguments in method calls and formal parameters in the signature
LastWrite	syntax node at which a variable was written to and variable token
LastRead	syntax node at which a variable was read from and variable token
ReturnsTo	return token and the method declaration
GuardedBy	variable to enclosing guard expression that must be true for this statement to be executed and uses this variable
GuardedByNegation	variable to enclosing guard expression that must be false for this statement to be executed and uses this variable

Table 2: Edge types used to enrich AST graphs

aggregation function. All models use the GRU unit as update function. For type prediction, Allamanis et al. [3] use the elementwise maximum instead of sum for aggregation, as it lead to better results. Intuitively, the elementwise maximum corresponds to a meet-like operation on a lattice defined over $\mathbb{R}^N$. Cummins et al. [14] change the message function to add a sinusodial positional encoding of the edge position to the hidden state of the source node before applying a linear function. All other models in table 1 use GGNN unmodified.

2.2.5 Initial node embedding and readout. Before message propagation, an initial node embedding must be computed for all nodes in the graph. These initial node embeddings are commonly computed from a set of features of the nodes. For AST representations, these features include the ast node type (corresponding to the grammar symbol represented by this node), node properties (a BinaryExpression ast node might have a node property specifying the operator) and node value (for literal and terminal nodes) [4, 11, 30]. In statically typed languages, it is also possible to include types of variables as features [4]. All features are then converted to vectors using an embedding layer. For the terminal AST nodes which correspond the the source code tokens, an alternative is to use embeddings generated by a sequence neural network over the original token sequence. This approach is followed by Fernandes et al. [16] and Hellendoorn et al. [20]. For graphs containing IR instructions, the space of possible node values is large due to the possible operand values. Cummins et al. [14] therefore construct the embedding by applying an embedding layer to the normalized instruction using using inst2vec [8].

After message propagation, the resulting graph needs to be interpreted in the context of the concrete application. Predictions about properties of single nodes can directly be computed from the hidden state of that node [3, 22]. To compute a graph-level vector, a weighted average of a projection of each hidden state can be computed [11, 16]. Both the weight (attention) and the projection are learnable functions that take the hidden state and in some variants [14] also the initial embedding of the node as an input. Fernandes et al. [16] combine this graph vector with the sequence embedding by another linear layer. Instead of computing the average over all nodes of the graph, Hellendoorn et al. [19]

only average the nodes that appear in the invariant. A similar approach is followed by Allamanis et al. [4], where they average over all nodes corresponding the slots of the variable whose name they want to predict. Si et al. [31] keep all embeddings as external memory which can be queried by another model through an attention-based mechanism.

3 BEHAVIOUR-BASED MODEL

Behaviour-based models make inferences about properties of source code by learning from data gathered from its runtime behaviour. They provide an interesting alternative to the graph-based models in applications that focus on how the code behaves (semantics) instead of how it was written (syntax). These models should therefore be more resistant to syntactic modifications. Some applications of execution-based program features are shown in table 3.

A behaviour-based model of a program works by learning features from the executions of programs. The first question hence is how to represent these executions. The most common form here is to consider executions as a sequence of program states, where each program state is represented by a set of variable assignments [25–27, 34, 38]. Since such trace data contains a lot of unique states, Henkel et al. [21] suggest abstracting the trace to only include a subset of relevant information represented as symbolic events. Events include for example function calls, function returns, equivalence of return values and function parameters. Symbolic traces are also used in the model of Wang and Su [36] in combination to the concrete traces. This is an improvement to prior work by the same authors [34] which only used concrete traces. Instead of simple vectors of program variables, we can also represent the program state as a graph. It is useful if the underlying programming language supports references or pointers, where variables or memory locations can naturally “point” to other locations in the program state. This representation is used by Brockschmidt et al. [12] and Li et al. [23] in a system that learns a heuristic to generate predicates in separation logic that describe invariants of data structures on the program heap (a problem named shape analysis). Whereas the former work uses simple manually extracted features from the graph, Li et al. [23] use the gated graph neural networks described in the previous section to automatically learn features for this task.

There are two fundamentally different approaches to apply machine learning to the data extracted from program executions. The first variant is to train a single embedding or model on executions of different programs. This is then later applied to new unseen programs. Almost all the listed works in table 3 follow this approach, except two. Yao et al. [38] train a model to capture the behaviour of a single program, and then extract a loop invariant from the parameters of this model. A combination between both approaches (learning a shared model for all possible executions and representing a single program by an optimized model) is described by Piech et al. [27]. They learn an encoder from the program state into a new space where the training programs can approximately be described as linear functions. Using these linear functions as features of programs, they then train a classifier that can predict whether feedback given to one program can be propagated to a program by a different student. Both these applications make use of the fact that machine learning models are often good generic function approximators. The approach has similarities to the field of program synthesis, especially the task of inferring a program or formula that satisfies certain input output examples.

Compared to graph-based models, there appear to be less works that use executions as a feature for learning over programs, especially ones using deep learning methods. Reasons for this may be that it is harder to gather execution data since it requires running the program which is impossible in some situations (such as incomplete programs during development, although this can be mitigated slightly by symbolic techniques which can execute single functions) and must be implemented separately for each supported programming language. That makes this approach more complex to implement compared to the graph-based one. Main applications include the use in tutoring systems to give feedback to students [25, 27] and inferring invariants [12, 26, 38]. Intelligent tutoring systems often contain a very controlled programming environment, sidestepping some of the issues associated with gathering execution traces.

4 EVALUATION

We have seen two different approaches to capture the meaning of source code with machine learning. In this section, we attempt to look at how well these two approaches perform. Unfortunately, comparision between the different architectures is difficult, since they are often applied to different tasks. Even when the tasks are similar (for example, both models applied to infer types), sometimes the results are not comparable because they are evaluated on different datasets. These factors prevent a clear comparision. This issue has been noted before by Bourgeois [10, p. 71]. Therefore, we instead will look at the results for some specific tasks were we find some overlap between different papers and note interesting observations and research questions for each task where applicable.

4.1 Method naming/classification

In this category, we look at models solving the task of assigning a set of labels or names to methods. We consider two slightly different tasks: MethodNaming and program classification. MethodNaming requires the model to predict the name of a method from its body. This task is often used to evaluated the accurracy of embeddings, since gathering data for it is easy (take a dataset with methods and remove the names) and the name of a method provides a succint description of its meaning. Program classification is a related task that instead requires classifying short programs according to the implemented algorithm or solved task. The implementations for both tasks considered in this section are shown in table 4. Only results relevant to the comparision between graph and behavioural approaches are shown.

A baseline for MethodNaming is given by a non-graph based model by Alon et al. [6]. They train a model that learns to embed and aggregate paths through the AST of the method automatically. In contrast to their previous code2vec [7] model, code2seq uses a sequence based decoder based on this embedding. The model is evaluated on a dataset of 11 large java projects (java-small [5])

Unpublished working draft.
Not for distribution.

Reference	Description
Yao et al. [38]	extract non-linear loop invariants from learned model of loop behaviour
Henkel et al. [21]	generate function vectors from usage contexts
Wang [34]	learn representation of program semantics from execution trace
Padhi et al. [26]	learn preconditions from program traces
Li et al. [23]	learn separation logic formulas for heap objects
Brockschmidt et al. [12]	learning shape analysis
Piech et al. [27]	learn program embedding to propagate student feedback
Paaßen et al. [25]	identify algorithm, errors and error location for student feedback

Table 3: Application using runtime features of program executions

and two new datasets consisting of top-starred java projects from GitHub (java-medium, java-large). Fernandes et al. [16] improve on this result by using graph neural networks for the encoding step. Their best result is a combination of a gated graph neural network combined with sequence neural networks for encoding, combined with a new decoder (an LSTM network with a pointer network-style copying mechanism). An interesting result for this comparision is that they find networks augmented with graph neural networks to outperform baseline methods which do not use any graph networks. There is also one implementation of a technique based on concrete and symbolic execution traces for the method naming tasks by Wang and Su [36]. However, the authors evaluate their model on a different dataset, so comparision to state of the art is difficult. The motiviation they give for using their own dataset is that it is more complex by using algorithmically focused methods from company interviews, in contrast to the java dataset which contains a lot of simple functions such as getters and setters. They retrain code2seq for this dataset, reporting lower results, which suggests the hypothesis that the dataset is more complex. It would be interesting to see how the model by Fernandes et al. [16] performs in comparision on this dataset, and how well their model performs on the java-small and java-large datasets.

The LiGeR model is an extension that adds support for symbolic traces to an earlier model called DyPro by the same authors which was also behaviour-based. Both of these models have also been tested for the challenge of program classification, which is a more natural task for behaviour based models due to being inherently tied to the semantics of the program. The evaluations for this model are based on a custom dataset known as CoSet [35] which consists of hand-labelled programs from an online coding competition. The results show that the behaviour-based methods (DyPro, LiGeR) significantly outperform the gated graph neural network variant. However, the exact AST graph representation used for the GGNN variant was left unspecified in this evaluation. That the graph representation matters is shown by Cummins et al. [14]. Their graph representation is flow, position and value sensitive, and in the evaluation they show that this graph representation leads to improved classification rates using a GGNN on the POJ-104 dataset. The POJ-104 dataset contains implementations of 104 different algorithms submitted to a judge system. They compare their result to prior work by Ben-Nun et al. [8] which didn't use graph neural networks and also used a graph representation that did not preserve the node positional order. This shows that the learning from structure that graph neural networks provide is useful. The question that remains is how well the LiGeR approach would perform on this dataset.

In summary, we can observe that graph-based representations provide useful signal for both tasks, improving over prior sequence based implementations. There are some indications that behavioural approaches might have an advantage when the dataset contains many algorithmic-heavy functions but a direct comparision is not possible due to different implementations and datasets.

4.2 Invariant inference

Invariant inference is a task from program verification. An invariant is a property that is preserved by a certain code sequence whenever it is executed. One use case for invariants is proving statements about loops. Because the invariant is preserved no matter how many times to loop body is executed, it allows reasoning about the state after the loop without explictly considering the number of iterations. Automatic loop invariant inference is thus an important problem for program verification.

Because the task requires inferring correct invariants, a natural choice is to apply a formal checker to the candidate invariants predicted by the model. The checker can also return additional counter examples to improve the prediction. This iterative process is known as counterexample guided inductive synthesis (CeGIS [32]). This approach is followed by Si et al. [31], where they use graph based model to convert the program to a structural external memory. It then uses this external memory in a reinforcment-based learning process interacting with the formal checker. The model can solve 106 of 133 benchmark programs compared to 100 solved by the best state of the art algorithm, but only if given a larger timeout than in the original competition. We can see from this result that graph-based embeddings are useful to predict loop invariants. They also perform an ablation study with an LSTM instead of the graph neural network, showing that the LSTM-variant can solve 13 fewer instances. This is supported by work of Hellendoorn et al. [19], where they use a gated graph neural network to reduce the false positive rate of invariants extracted by the dynamic analysis tool Daikon [15].

Behavioural based models should be well suited to this problem, since loop invariants essentially require fitting formulas that hold over all possible runtime traces. And indeed, as demonstrated by Yao et al. [38], fitting a model (gated continous logic networks) to

Unpublished working draft.
Not for distribution.

Reference	Task (Metric)	Dataset	Results (*model* score)
code2seq [6]	MethodNaming (F1)	java-small	*code2vec* 18.62 *code2seq* 43.02 *TreeLSTM* 35.46
code2seq [6]	MethodNaming (F1)	java-large	*code2vec* 42.73 *code2seq* 59.19 *TreeLSTM* 53.63
structured neural summarization [16]	MethodNaming (F1)	java-small	*BiLSTM+GNN → LSTM+POINTER* 51.4 *code2seq* 43.0
LiGeR [36]	MethodNaming (F1)	named methods from algorithmic focused company interviews	*code2seq* 0.39 *LiGer* 0.57
LiGeR [36]	program classification (F1)	CoSet [35]	*code2vec* 0.68 *GGNN* 0.70 *DyPro* 0.81 *LiGer* 0.85
DyPro [34]	program classification (F1)	handpicked problems from popular online coding platforms	*TreeLSTM* 0.61 *GGNN* 0.66 *DyPro* 0.78
NCC [8]	program classification (accuracy)	POJ-104	*NCC* 94.83 *TBCNN* 94.0
ProGraML	program classification (accuracy)	POJ-104	*NCC* 94.83 *XFG-GGNN* 95.71 *GGNN-s* 96.13 *GGNN* 96.22 *Transformer* 96.67

Table 4: Implementations for method summarization/classification

Unpublished working draft.
Not for distribution.

the traces and then extracting the formula from that model can perform well, solving all but 9 of the 133 problems in the benchmark set of Si et al. [31]. The remaining 9 problems are claimed to be theoretically unsolvable.

The gated continous logic network by Yao et al. [38] is trained for each single program separately, learning to represent only the behaviour of that program. In contrast, Wang [34] show a different approach where they train a single generic model to embed program traces, and apply this model to the task of verifying whether an invariant for a loop is valid or not. They evaluate their approach on a dataset generated by formally checking invariants produced from Daikon to obtain positive examples and add negative examples based on mutations of those. In both accurracy and F1-score, the model beats a similar model trained using a gated graph neural network and one using a TreeLSTM. They do however simply use the hidden state of the root node of the AST to extract a single embedding from the GGNN, and leave the concrete edges they used in their graph representation of code unspecified. It would be interesting to see the results for the GGNN for on an enriched AST, where the embedding is extracted by averaging the hidden state of the nodes which are present in the invariant, as in Hellendoorn et al. [19].

In general, there are still not enough comparable works on predicting loop invariants with different models to reach a definite answer on which model is better. While both architectures capture enough information to be useful predictors for invariants, they are evaluated in different settings (verifying invariants vs generating) and on different datasets. A possible future research question could be to evaluate them in a common setting, to obtain insights into how well the different program representations perform.

5 CONCLUSION

We have seen that both graph- and behaviour-based models have their use for learning tasks over source code. Although they are applied for similar tasks, a direct comparision is often not possible due to different datasets and slightly different applications. Some future research directions could be:

Can we design a generic benchmark dataset to compare these models on? As seen in the evaluation, models are often evaluated on ad-hoc tasks and benchmark sets. It would be good to have a standarised, commonly accepted set of tasks along with datasets to compare different approaches on. This could also be helpful in determining if there are generic, pre-trainable models that work well for a wide variety of tasks as is the case for NLP.

What are the weaknesses and strengths of each model? Graph based models can exploit syntatical similarities taking hints from the programmer, which behaviour based models cannot. There are indications that this makes behaviour based models better for complex code that is algorithmic in nature. To explore this, a comparision of state of the art models for both tasks on the same dataset is needed. An interesting avenue in this direction is also work on adversarial examples, which provide insight into what properties of the input data these model bases their predictions on [9, 28, 39, 40].

Can both models be useful in combination? Related to the previous question, if these models have different strengths, a variant that capitalizes on both models' strength by combining could be explored.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. *arXiv:2005.00653 [cs, stat]* (May 2020). http://arxiv.org/abs/2005.00653 arXiv: 2005.00653.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *arXiv:1709.06182 [cs]* (May 2018). http://arxiv.org/abs/1709.06182 arXiv: 1709.06182.

[3] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. *CoRR* abs/2004.10657 (2020). https://arxiv.org/abs/2004.10657 _eprint: 2004.10657.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. *ICLR* (2018).

[5] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. 2091–2100.

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. *arXiv:1808.01400 [cs, stat]* (Feb. 2019). http://arxiv.org/abs/1808.01400 arXiv: 1808.01400.

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 40:1–40:29. https://doi.org/10/ggssk3

[8] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 3585–3597. http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics.pdf

[9] Pavol Bielik and Martin Vechev. 2020. Adversarial Robustness for Code. *arXiv:2002.04694 [cs, stat]* (Feb. 2020). http://arxiv.org/abs/2002.04694 arXiv: 2002.04694.

[10] Dylan Bourgeois. 2019. Learning Representations of Source Code from Structure and Context. https://infoscience.epfl.ch/record/277163 Library Catalog: infoscience.epfl.ch Number: STUDENT.

[11] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jerónimo Castrillón. 2020. Compiler-based graph representations for deep learning models of code. *CC* (2020). https://doi.org/10/gg3j57

[12] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning Shape Analysis. In *Static Analysis (Lecture Notes in Computer Science)*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 66–87. https://doi.org/10/gg3j6r

[13] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv:1409.1259 [cs, stat]* (Oct. 2014). http://arxiv.org/abs/1409.1259 arXiv: 1409.1259.

[14] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. 2020. ProGraML: Graph-based Deep Learning for Program Optimization and Analysis. *ArXiv* (2020).

[15] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45. https://doi.org/10/drc63v Publisher: Elsevier.

[16] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Structured Neural Summarization. *arXiv:1811.01824 [cs, stat]* (May 2020). http://arxiv.org/abs/1811.01824 arXiv: 1811.01824.

[17] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. *arXiv:1704.01212 [cs]* (June 2017). http://arxiv.org/abs/1704.01212 arXiv: 1704.01212.

[18] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, Paderborn, Germany, 763–773. https://doi.org/10/gg3j53

[19] Vincent J. Hellendoorn, Premkumar T. Devanbu, Oleksandr Polozov, and Mark Marron. 2019. Are My Invariants Valid? A Learning Approach. *ArXiv* (2019).

[20] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global Relational Models of Source Code. https://openreview.net/forum?id=B1lnbRNtwr&noteId=B1lnbRNtwr

[21] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: understanding programs through embedded abstracted symbolic traces. 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)*. Association for Computing Machinery, Lake Buena Vista, FL, USA, 163–174. https://doi.org/10/gf6gbq

[22] Mingzhe Li, Jianrui Pei, Jin He, Kevin Song, Frank Che, Yongfeng Huang, and Chitai Wang. 2019. Using GGNN to recommend log statement level. *arXiv:1912.05097 [cs]* (Dec. 2019). http://arxiv.org/abs/1912.05097 arXiv: 1912.05097.

[23] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2017. Gated Graph Sequence Neural Networks. *arXiv:1511.05493 [cs, stat]* (Sept. 2017). http://arxiv.org/abs/1511.05493 arXiv: 1511.05493.

[24] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 304–315. https://doi.org/10/gg3j6v ISSN: 1558-1225.

[25] Benjamin Paaßen, Joris Jensen, and Barbara Hammer. 2016. *Execution Traces as a Powerful Data Representation for Intelligent Tutoring Systems for Programming*. International Educational Data Mining Society. https://eric.ed.gov/?id=ED592662 Publication Title: International Educational Data Mining Society.

[26] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, Santa Barbara, CA, USA, 42–56. https://doi.org/10/gg3j59

[27] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *International Conference on Machine Learning*. 1093–1102. http://proceedings.mlr.press/v37/piech15.html ISSN: 1938-7228 Section: Machine Learning.

[28] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2020. Semantic Robustness of Models of Source Code. *arXiv:2002.03043 [cs, stat]* (June 2020). http://arxiv.org/abs/2002.03043 arXiv: 2002.03043.

[29] Veselin Raychev, Martin Vechev, and Andreas Krause. 2019. Predicting program properties from "big code". *Commun. ACM* 62, 3 (Feb. 2019), 99–107. https://doi.org/10/gg3j66 Publisher: Association for Computing Machinery (ACM).

[30] Jessica Schrouff, Kai Wohlfahrt, Bruno Marnette, and Liam Atkinson. 2019. Inferring Javascript types using Graph Neural Networks. *arXiv:1905.06707 [cs, stat]* (May 2019). http://arxiv.org/abs/1905.06707 arXiv: 1905.06707.

[31] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 7751–7762. http://papers.nips.cc/paper/8001-learning-loop-invariants-for-program-verification.pdf

[32] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *ACM SIGARCH Computer Architecture News* 34, 5 (Oct. 2006), 404–415. https://doi.org/10/dz9ppd

[33] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *arXiv:1710.10903 [cs, stat]* (Feb. 2018). http://arxiv.org/abs/1710.10903 arXiv: 1710.10903.

[34] Ke Wang. 2019. Learning Scalable and Precise Representation of Program Semantics. *ArXiv* (2019).

[35] Ke Wang and Mihai Christodorescu. 2019. COSET: A Benchmark for Evaluating Neural Program Embeddings. *arXiv:1905.11445 [cs, stat]* (May 2019). http://arxiv.org/abs/1905.11445 arXiv: 1905.11445.

[36] Ke Wang and Zhendong Su. 2019. Learning Blended, Precise Semantic Program Embeddings. *arXiv:1907.02136 [cs]* (July 2019). http://arxiv.org/abs/1907.02136 arXiv: 1907.02136.

[37] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. *arXiv:2005.02161 [cs, stat]* (April 2020). http://arxiv.org/abs/2005.02161 arXiv: 2005.02161.

[38] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning Nonlinear Loop Invariants With Gated Continuous Logic Networks. *CoRR* (2020). http://arxiv.org/abs/2003.07959v3 _eprint: 2003.07959v3.

[39] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial Examples for Models of Code. *arXiv:1910.07517 [cs]* (May 2020). http://arxiv.org/abs/1910.07517 arXiv: 1910.07517.

[40] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 01 (April 2020), 1169–1176. https://doi.org/10/gg3j6p Number: 01.

Unpublished working draft. Not for distribution.