TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR COMPILERBAU

Diplomarbeit

Investigating Input Representations and Representation Models of Source Code for Machine Learning

Alexander Brauckmann

17. Februar 2020

Erstgutachter: Prof. Dr.-Ing. Jeronimo Castrillon
Zweitgutachter: Prof. Dr. Markus Krötzsch
Betreuer: M.Sc. Andrés Goens

Abstract

Machine Learning methods are actively used to solve various tasks on source code, such as in Compilers to improve performance of executable code, or IDEs to boost developer productivity. While the use cases are manifold, most of these methods rely on manually-defined features that require substantial engineering efforts, while not necessarily being optimal.

In this thesis, we introduce a novel approach to encode programs as graphs that include compiler-internal semantics and use the recently discovered class of Graph Neural Networks to learn task-specific features automatically. Specifically, we design a framework for learning program representations based on Abstract Syntax Trees and Control- and Dataflow Graphs, extracted with the Clang/LLVM compiler infrastructure.

We empirically evaluate the approach in compiler heuristic use cases and show to outperform existing methods based on Recurrent Neural Networks (RNNs) in generalization performance and inference time. In the task of code generation however, we show limitations of the graph-generative architecture we used, which cause a bias towards generating samples of less size and complexity.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die am heutigen Tag eingereichte Diplomarbeit zum Thema:

Investigating Input Representations and Representation Models of Source Code for Machine Learning

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Zitate sind als solche kenntlich gemacht.

Dresden, den 17. Februar 2020

Alexander Brauckmann

Contents

1

Introduction

1.1 Motivation

Since the 1950s, software is developed in high-level programming languages and translated into low-level executable machine code by *compilers*. Having a key position in the process of software engineering, one of the main objectives in compiler research is to produce highly optimized machine code for a given target architecture. While many problems in compiler design can efficiently be solved with deterministic algorithms, some problems still remain unsolved due to their complexity or their statistic nature. In order to approximate the solutions of those problems, data-driven *Machine Learning methods* have proven to be a useful tool.

A key component of the performance of machine learning methods is the quality of its *features*, i.e. the goodness of the representation of the data sample at hand. For this reason, e.g. the computer vision and natural language processing communities have invested large efforts to devise so-called *Deep Learning models* that operate on the raw data and are optimized to extract high-quality features.

In the compiler community, several deep learning models have been proposed. However, these models tend to not leverage from the known semantics of source code. In this thesis, we will introduce a hybrid approach to feature extraction and evaluate it in various settings. In contrast to relying exclusively on the raw input data and the capabilities of deep learning models, our approach combines semantic structures known from decade-long research on

compilers with a recently discovered deep learning models for graphs.

1.2 Outline

This thesis is structured as follows: In chapter 2, we will show various use cases in compilers and software engineering, where machine learning methods have successfully been used. On the one hand, the tasks will motivate this study; on the other hand, it will present tasks that are the basis of the later evaluation.

In chapter 3, we will describe different representations of programs that can be embedded with the deep learning models shown in chapter 4. While the models are used to extract good features of the code, they need to be integrated into a higher-level architecture to solve a real task. Therefore, chapter 5 shows an architecture for predictive and one for generative tasks. As the basis for evaluation of the different representations and models, we will proceed with describing a framework that can be used to solve predictive and generative tasks on graphs, as shown in chapter 6. We evaluate the representations in two predictive and one generative task in chapter 7, along with the result interpretation and conclusion in chapter 8.

1.3 Contriubtions

The contributions of this thesis are as follows:

- To the best of our knowledge, we are the first who apply *Graph Neural Networks (GNNs)* to compiler-internal program representations in the use case of compiler heuristic models. The GNN model has proven to be well-suited for extractions of features from graphs in various other domains. In this thesis, we evaluate its performance and compare it to existing machine learning approaches for compiler heuristic tasks.

- We apply a graph-generative model based on GNNs to source code and evaluate its performance to the state-of-the-art generative model based on sequences.

- We introduce a flexible framework that supports predictive and generative tasks based on compiler-internal graph representations of programs, which can be re-used and applied to further tasks.

2

Related work

In this chapter, we will give the context for this work by describing several examples of applications, where probabilistic machine learning methods have been used. The presented applications pose potential use cases for deep learning representation models. Therefore we consider them as relevant.

In the second part of this chapter, we will focus on related work on embedding models for source code used in the application field of compilers.

2.1 Machine Learning Applications on Source Code

The fields of applications for probabilistic machine learning methods on source code are manifold. From a high-level perspective, they can be grouped into the domains of *Software Engineering* and *Optimizing Compilers*. Each of them has different reasons for the use of probabilistic models.

Software Engineering Allamanis et al. (2018) give a comprehensive review in this domain. They motivate the use of probabilistic machine learning methods for code by the fact that these methods capture the statistical properties of the source code. In contrast, strongly deterministic methods fail to do so. The reviewed work is classified into the application fields. We adopt this taxonomy and present the most relevant, as well as additional work:

- *Recommender Systems* are used to make recommendations to the user in the software engineering process. An example is *auto completion*, where the next tokens of a program are predicted for a given context (Bruch et al. (2009); Nguyen et al. (2013); Raychev et al. (2014)). Another example is *variable name recommendation* or *variable misuse prediction* as done by Allamanis et al. (2017), who used a graph neural network model to do predictions on Abstract Syntax Tree (AST) nodes.

- Inspired by the success of *Statistical Machine Translation (SMT)* in natural language, *Code Translation* methods are applied to programming languages in order to translate one into another (Karaivanov et al. (2014); Nguyen et al. (2015)). However, the translation task currently only supports languages of the same paradigms, e.g., Java to C#, but not Java to C.

- Examples for *Code-to-Text* generation are prediction of variable names (Bavishi et al. (2018)), function names (Allamanis et al. (2016)), and comments (Movshovitz-Attias and Cohen (2013)) for a given context. Integrated into IDEs, they are useful tools to assist pasting of code and generating documentation.

- In *Program Synthesis*, probabilistic methods are successfully used to generate code. Based on a specification expressed in natural language or as examples, programs that fulfill this specification are generated. Probabilistic models have not only proven to be useful as a heuristic for a guided search (Liang et al. (2010)), but also for the use case of dataset augmentation for predictive models (Cummins et al. (2017b)). However, they didn't achieve convincing results yet. (Goens et al. (2019)).

- *Vulnerability Prediction* is another relevant use case of probabilistic machine learning methods in the software engineering tools domain. After being trained on a large body known vulnerabilities, they have proven to generalize to new, previously unknown vulnerabilities (Li et al. (2018c,b))

Compilers Wang and OBoyle (2018) survey the application domain of machine learning-based compilation. They argue that machine learning is a natural fit for predicting best-performing *compiler optimizations* in the compiler task of translating a source language into an instruction set of a target architecture. They classify the use cases into the following categories:

- In *Compiler Optimization Selection*, probabilistic models have proven to achieve significant speedups in various settings by selecting optimizations, along with their tuning parameters. Early work is done by Monsifrot et al. (2002), who predict whether it is desirable to unroll a loop or not. One of the most prominent examples is the Milepost GCC compiler, which predicts the best-performing optimizations of GCC on a function-level granularity (Fursin et al. (2011)). Magni et al. (2014) introduced a probabilistic model to predict the best thread coarsening factor in OpenCL. In the proposed method that combines multiple OpenCL work items into a single thread. An entirely new compiler optimization on the other hand, enabled through probabilistic modeling was proposed by Grewe et al. (2013). They use a probabilistic model to decide whether it is better to transform OpenMP annotated code to OpenCL and execute it on the system's GPU.

Reference	Application	Code embedding	Prediction model
Monsifrot et al. (2002)	Loop unrolling	Manually-defined features	Decision Tree
Fursin et al. (2011)	Optimization selection	Manually-defined features	Decision Tree
Grewe et al. (2013)	Device Mapping	Manually-defined features	Decision Tree
Magni et al. (2014)	Thread Coarsening	Manually-defined features	Neural Network
Cummins et al. (2017a)	Device Mapping, Thread Coarsening	Learned features (RNN)	Neural Network
Ben-Nun et al. (2018)	Device Mapping, Thread Coarsening	Learned features (RNN)	Neural Network

Table 2.1: Overview of used code embeddings in different compiler applications.

- *Compiler Phase Ordering Selection* considers the order in which compiler optimizations are applied. Kulkarni and Cavazos (2012) show significant speedups in the Java Jikes RVM by using a probabilistic machine learning model. Ashouri et al. (2017) present a method that reduces the search space by clustering individual passes, then searching for good cluster orders. Implemented in the LLVM compiler framework, they achieved major speedups.

2.2 Code Embeddings in Compiler Applications

Table 2.1 gives an overview of different code embeddings and prediction models used in various compiler applications.

Several of these methods use *manually-defined features* in combination with *decision trees* and *neural networks*. Developing such a method is a two-step process: First, a domain expert defines several features and extracts them from the raw data using deterministic methods. For example, in the context of the loop unrolling task, the features are the number of statements and arithmetic operations in the loop. Second, a decision tree or a neural network model is optimized with the objective to fit best a set of provided training example pairs of input and output.

Inspired by the success of deep learning in research areas like image recognition applications (LeCun et al. (2015)), these models recently have been used in compiler applications as well. Cummins et al. (2017a) proposed the *DeepTune* model, in which a Recurrent Neural Network (RNN) is optimized to extract relevant features from the raw source code. For this, the source code is normalized for any identifiers, then represented as a sequence of tokens. The authors have shown to outperform decision tree models by far in a compiler heuristic task. In contrast to that, Ben-Nun et al. (2018) use a sequence representation based on the LLVM IR, while embedding the LLVM IR tokens with a method following the notion of word2vec (Mikolov et al. (2013)) in order to produce similar embeddings for LLVM IR tokens that share similar contexts.

3

Representations of Programs

In this chapter, we will define different representations of programs that are used as input representations to the deep learning models shown in chapter 4.

We begin by describing the typical architecture of a compiler to put different code representations and analyses into a broader context. We then narrow down to the Clang and LLVM framework as mature and modern implementation. In the second part of this chapter, we will give the concrete definitions of the representations.

Throughout the descriptions, we demonstrate the concepts on a running example of a simple program, finally building up to our representations.[1]

3.1 Compiler Intermediate Representations

In order to translate a program expressed in a source code programming language to executable machine code, the compiler sequentially processes it and passes it through a pipeline, consisting of multiple stages. Figure 3.1 shows such a pipeline with stages grouped into the *Frontend*, the *Middleend*, and the *Backend* meta components. At the beginning of the pipeline, the Frontend tokenizes and parses the source program, performs analyses and transforms it into a more universal, low-level representation. Next, the Middleend trans-

[1] More complex examples of programs and our representations are included in Appendix A, as they're too complex for the scope of this chapter.

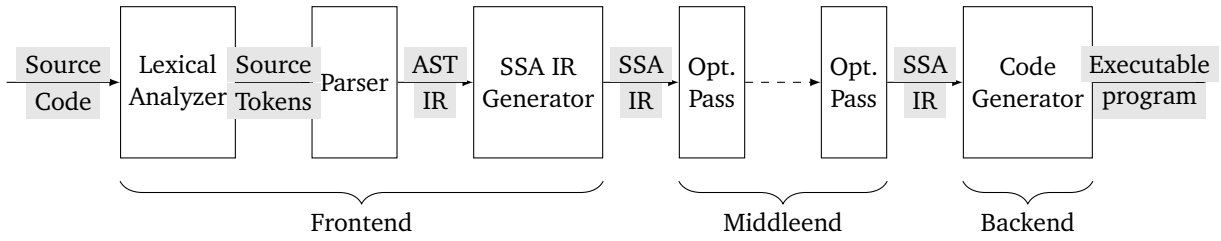


Figure 3.1: Compiler-internal pipeline architecture.

forms the representation to be more efficient by applying multiple optimization passes. Finally, the pipeline ends with the Backend that generates executable machine code supported by an *Instruction Set Architecture (ISA)*.

Highlighted in grey are the *Intermediate Representations (IRs)*, which are data structures representing the program throughout the compiler pipeline. Compilers typically use different forms of IRs, transforming one to another when reaching a certain stage. This is largely due to the diversity of requirements that the components have at different stages: While the components at the beginning of the pipeline require a representation for tasks close to the grammar of the source language, components in the later stages perform lower-level tasks that require an IR closer to the target ISA. Relevant structures would remain hidden when working on a higher-level IR. Also, it is an important design decision that allows for the reusability of compiler components across programming languages and ISAs.

Having representations of the program that are designed for entirely different purposes results in a variety of rich semantics rich available in the compiler. In the context of designing program representations for deep learning, these semantics pose a great opportunity.

3.2 Compiler Analyses

Along the compilation pipeline, analyses are performed in order to perform error analyses, performance optimizations, and transformations into new IRs. Common analyses are the control-flow analysis and dataflow analysis.

3.2.1 Control-Flow Analysis

Control-flow analysis makes the control structures within a program explicit, which act as the basis for further analyses, as well as the construction of IRs. Especially lower-level IRs and machine code implicitly contain control-flow information.

Allen (1970) describes control-flow analysis as a two-step process: First, instructions are partitioned into basic blocks. Then, a *control-flow graph* is constructed based on this contracted representation as defined below:

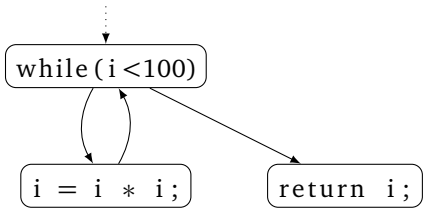
Definition 1 (Basic block (BB)). *A BB is the maximal sequence of IR instructions where each instruction has exactly one control-flow successor, except for the last instruction. The control-flow can only enter the first and leave in the last instruction.*

Definition 2 (control-flow graph (CFG)). *A CFG is a directed graph where the nodes represent BBs and the edges the control-flow.*

An example of a function defined in the C programming language, along with its corresponding CFG representation is given below. The function consists of a while statement with a conditional statement and one instruction in its body. Lastly, a return statement terminates the function. The corresponding CFG consists of three BBs that are connected with CFG edges: The while condition is followed by the while body and the return statement. The while body has a back edge to the while-condition.

```
1 int foo(int i) {
2   while(i<100) {
3     i = i * i;
4   }
5
6   return i;
7 }
```

(a) Source code.



(b) Control-flow graph.

Figure 3.2: Example function and its corresponding control-flow graph.

3.3 The Clang/LLVM Compiler Framework

The Clang/LLVM compiler framework is an example of an implementation of the pipeline architecture. It supports various programming languages and architectures. This is achieved by using rich IRs, which make it a good candidate for further studies.

From a high-level perspective, the framework is a combination of two strictly decoupled libraries: While Clang is responsible for the Frontend functionality close to the source code like lexing, parsing, and semantic analysis, LLVM performs the Middle- and Backend tasks like optimization and target code generation.

The high decoupling enables using the libraries in two use cases: First, it is a flexible framework for the implementation of a compiler, in which new programming languages and target architectures can be implemented with minimal effort by reusing and sharing various components: When implementing a new programming language, the Clang Frontend can be replaced by a custom Frontend, while the LLVM Middleend and all target architecture code generators can be kept. Similarly, the new target architectures can be supported by implementing a Backend, while preserving the support for existing Frontends. Second, the Clang Frontend is a popular framework for the development of productivity and code analysis tools in order to e.g. perform refactorings, find syntax errors, checking coding conventions, and finding vulnerabilities.

The analyses performed by Clang and LLVM operate on different forms of IRs: Clang constructs an *Abstract Syntax Tree (AST)*, LLVM an own, specially designed IR called the *LLVM IR*. In the scope of a compiler, the Clang AST is used for source-level representation and error reporting. In the use case of tooling on the other hand, it is the basis for various analyses, which is the reason for its high degree of development. In combination, the two libraries interface at the LLVM IR, which is constructed at a final stage in the Clang library by traversing the AST. The LLVM library performs further analyses with the objective of producing well-performing code while preserving the program semantics.

Because Clang and LLVM each bring a highly developed representation, along with various well-known code analyses, they are the ideal candidate for the implementation of our method.

3.3.1 Clang Abstract Syntax Tree

While a program represented as tokens allows the programmer to express a program in a very condensed form, this representation is not a good fit for further automated tasks. Therefore, Clang constructs an IR that captures the program's structure. This structure is commonly used for high-level tasks and analyses on the source code that require a mapping to the original source code, such as source-level transformations or error reporting, which requires the knowledge of the exact error location in the program source code.

The Clang AST is constructed by tokenizing the program's stream of characters, constructing a parse tree, and then reducing it to an AST as described below:

Definition 3 (Parse tree). *A parse tree (or concrete syntax tree) is a tree structure representing the derivation of a source program, expressed in a given grammar. It has a one-to-one correspondence to the sequential representation of the program, whereas nodes correspond to non-terminal symbols and leaves to terminal symbols.*

Definition 4 (Abstract syntax tree (AST)). *An AST is a parse tree that is reduced by the concrete syntax elements of the language, e.g. terminal symbols, punctuation, and parentheses. In contrast to a parse tree, it only contains the minimum information necessary to represent a program.*

Definition 5 (Clang abstract syntax tree (Clang AST)). *The Clang AST is an AST, whereas each node belongs to one of the following type hierarchies:*

- *Declaration: Representing a declaration or definition such as a variable, struct, typedef, or a function.*
- *Statement: Such as a compound statement that aggregates multiple statements, a binary operator, or a control statement.*
- *Type: Such as scalars, pointers, or complex types.*

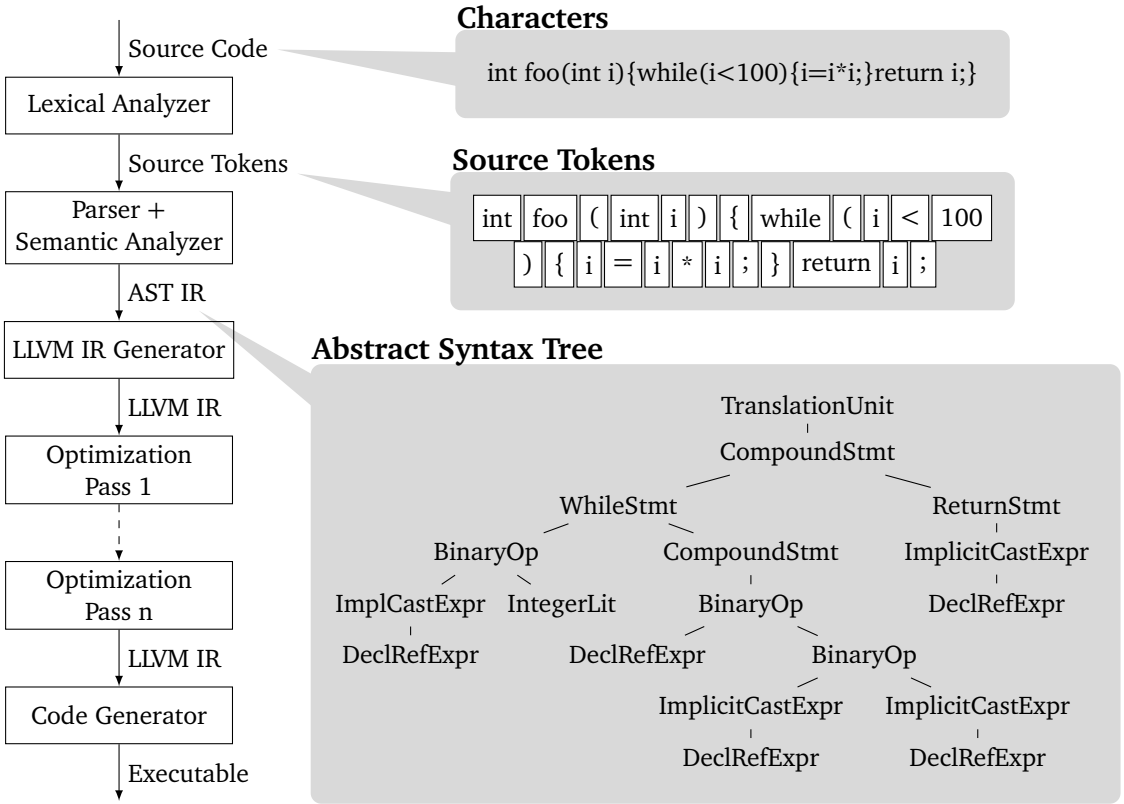


Figure 3.3: Example function in character, token, parse tree, and AST representations taken from the Clang/LLVM compiler.

Figure 3.3 gives an overview of the pipeline implemented in the Clang/LLVM library, along with the frontend-based code representation at different stages. In the phase of *lexical*

analysis, the characters of the program are split into tokens. Then, the tokens are parsed into the AST. After that, additional analyses from the semantic analyzer module are performed to enrich the AST with e.g. variable references and dataflow information.

3.3.2 LLVM IR

The LLVM IR is a low-level representation of a program, which has properties that help to meet the goal of designing a flexible, efficient, and universal IR.

It is universal to both, the high-level programming languages that are mapped to it, and the target architectures it is translated to, allowing to support of many different programming languages and target architectures. Representing the program at a lower level enables the compiler to perform analyses and optimizations that would otherwise be impossible to conduct on a high-level representation such as the AST. The lightweight and simple instruction set harmonize with the goal of flexibility and universality. The instruction set of the LLVM IR, which is similar to the Reduced Instruction Set Computer (RISC) class of architectures, allows for efficient processing. This is because the simplicity of instructions and handling of variables empower the implementation of more general analysis and optimization algorithms. On the other hand, the simple instruction set allows support for both - simple and complex target architectures. Complex ISAs can be constructed by transforming the simple instructions to more complex and high-level instructions in the final code generation stage.

Based on the descriptions of Lattner and Adve (2004), we will now define the LLVM IR. Fundamentally, it is a Three Address Code (TAC), which is transformed into a Single Static Assignment form (SSA) at a certain point:

Definition 6 (Three address code (TAC)). *A TAC is a linear code, where the individual operations have at most two operands and one result. Each element of an instruction is either an address or an operation. Addresses are variable names or constants. For representing control-flows, instructions can be referred to in jump operations by the usage of labels.*

Operations commonly include expressions of unary or binary type in the form `a = op b` *or* `a = b op c`*, copy instructions such as* `a = b`*, jumps of conditional or unconditional type like* `goto L` *or* `if a relop b goto L`*, function calls, function returns, and pointer assignments like* `a = *b` *or* `a = &b`*.*

Definition 7 (Single static assignment form (SSA)). *An SSA is a TAC with the additional requirement that each defined variable has a unique name and can be assigned only once. This is desirable because it creates explicit use-def relationships, which eases further analyses and transformations.*

Enforcing this property is problematic in cases where multiple control-flows merge as it results in ambiguities in the operands. This can be mitigated by inserting phi-functions that map a set of variables to a single variable, conditioned on the control-flow.

Definition 8 (LLVM IR). *The LLVM IR is a TAC with a RISC-like instruction set with instructions that are general and don't include processor-specific information such as specific operands or physical register addresses. Instead, it is a simple and very small instruction set with a memory model that includes an unlimited amount of virtual registers, and a virtual load/store memory.*

It is in SSA form if it includes no memory accesses. Lattner and Adve (2004) argue that a memory store through a pointer location may modify multiple variables, making it very complex to maintain the SSA requirement.

The LLVM IR also explicitly captures the control-flow information of a program. Functions frame the control-flow of basic blocks that form a CFG. Basic blocks frame a control-flow sequence of instructions.

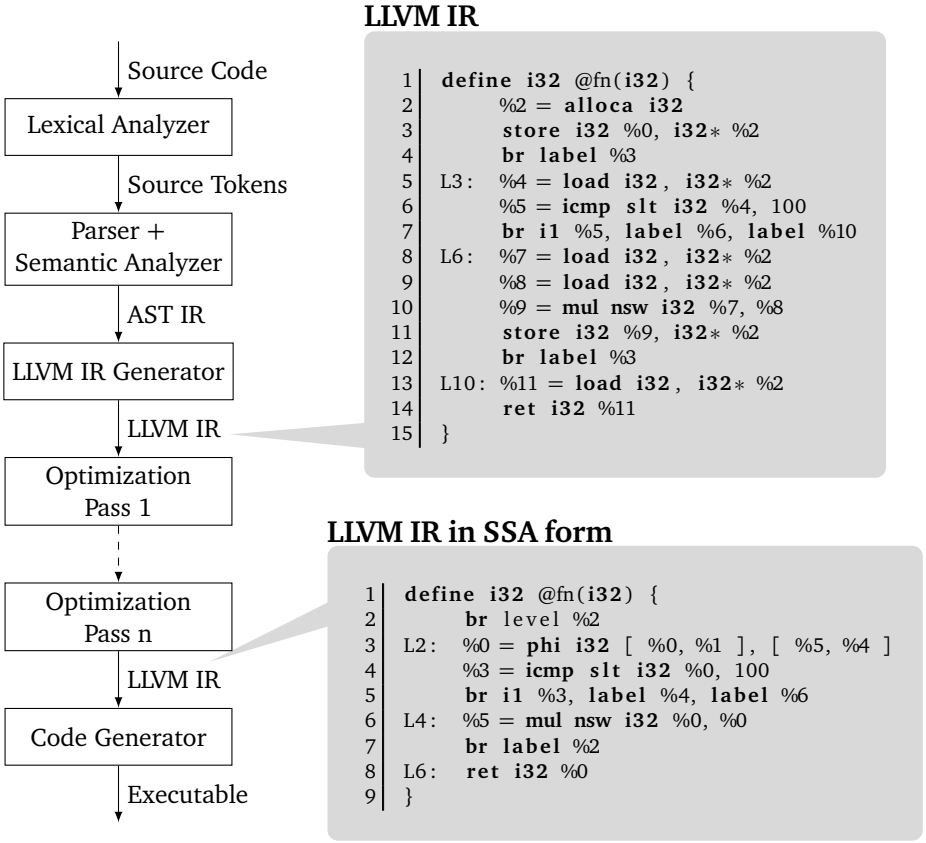


Figure 3.4: Example function in sequentialized LLVM IR representation before and after optimization.

Figure 3.4 continues the overview of the pipeline implemented in the Clang/LLVM library, along with the backend-based code representation at different stages. The Clang AST is transformed into LLVM IR by the LLVM IR Generator component. The LLVM IR is unoptimized, yet universal. Then, multiple optimizations transform the LLVM IR code into a more efficient variant. Here, we show the `mem2reg` pass as a relevant example.

Clang's LLVM IR Generator component solves the control-flow ambiguity problem by allocating memory for the ambiguous variable. In the LLVM IR representation in the top of Figure 3.4, the `alloca` instruction (line 2) allocates a memory location that is accessed by a patterns of `store` and `load` in the different BBs: BB1 (lines 2-4) initializes the memory location with a value. BB3 (lines 8-12) performs a computation and stores the result in the memory location. The conditional statement in BB2 (lines 5-7) loads the value stored at the memory location. BB4 also loads the value at the memory location in order to return it to the caller.

As shown in the bottom of the figure, the LLVM `mem2reg` optimization pass transforms the structures where `alloca` instructions are only followed by `store - load` by replacing them with phi-functions that are conditioned on the control-flow, which results in the LLVM IR in SSA form.

3.4 Program Representations for Deep Learning

As a result of planning to evaluate the representations in two different types of tasks, we need to consider different requirements, as summarized below:

Requirement	Generative tasks	Predictive tasks
Compactness	✓	
Completeness	✓	
Expressiveness	✓	✓

Table 3.1: Requirements on the representations.

Generative tasks In the use case of code generation, the representation needs to be complete and compact. Completeness is mandatory as it allows for transforming the graph representation back into a compiler-internal representation to produce executable code. Compactness is crucial as it impacts model performance: As shown in chapter 4, the generative models perform better on smaller-sized representations because their performance drops with longer decision sequences. Analyses on the generated dataset (chapter 7) reveal that the source-based representation is generally more compact than the LLVM IR based representation. The comprehensive type system that LLVM uses additionally contributes to the non-compactness. Therefore, we choose a source-based representation for this task.

Predictive tasks Deep learning-based embedding models are capable of extracting rich structural features from the data sample at hand. However, no study about the performance of different compiler-internal representations exist. Therefore, we define several sequence- and graph-based representations. Both, the Clang AST and the LLVM IR have complementary advantages that could be the basis of a well-performing representation: On the one hand, Clang's representations are generally more compact than the LLVM IR, resulting in a better model performance also for the predictive use case because information in GNN-based models has a limited reach. On the other hand, the LLVM IR contains semantics, uncovered through compiler-internal analyses that remain hidden in the Clang representation. Another important advantage of an LLVM IR based representation is that a Clang-based representation contains semantics specific to the programmer, such as coding style and coding conventions. A representation based on the LLVM IR is invariant to this because different constructs with the same computational semantics are translated into the same LLVM IR code. Because both representations have unique properties, we decide to design and evaluate both representations.

We will proceed with the description of four representations that are based on the Clang/LLVM compiler framework. Figure 3.5 gives an overview of these representations. Two of them, *Source Token Sequence* and the *Source Dataflow Graph* are high-level representations extracted from the Clang AST. The other representations, specifically the *LLVM IR Token Sequence* and the *LLVM IR Control- and Dataflow Graph*, are lower-level representations, constructed on the LLVM IR.

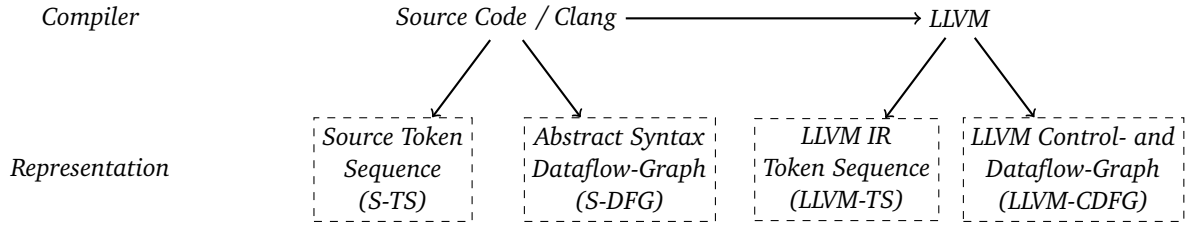


Figure 3.5: Representations of code for deep learning in the context of a compiler.

3.4.1 Source Token Sequence

As a sequential representation close to the source code, we define:

Definition 9 (Source Token Sequence (S-TS)). *An S-TS is a sequence of C language tokens. The token types are C-language builtins, data types, keywords, and normalized identifiers.*

Following the representation described in Cummins et al. (2017a), the source token sequence is constructed as follows: First, the source code is *normalized* so that the model is invariant to trivial differences such as comments and method names. This is achieved by eliminating any comments and by replacing the method and variable names with a fixed scheme. In a second step, *tokens* are produced by splitting the resulting sequence of characters.

Figure 3.6 shows the transformation of the running example into the S-TS-representation. The original source code (Figure 3.6a) is normalized for variable and method names (Figure 3.6b), and finally tokenized (Figure 3.6c).

```
1 int foo(int i) {
2   while(i<100) {
3     i = i * i;
4   }
5
6   return i;
7 }
```

(a) Source code.

```
1 int fn1(int var1) {
2   while(var1<100) {
3     var1 = var1 * var1;
4   }
5
6   return var1;
7 }
```

(b) Normalized source code.

```
int fn1 ( int var1
) { while ( var1
< 100 ) { var1
= var1 * var1 ; }
return var1 ;
```

(c) Normalized source tokens.

Figure 3.6: Transformation of the example function into the normalized source sequence representation.

3.4.2 LLVM IR Token Sequence

As a sequential representation close to the LLVM IR code, we define:

Definition 10 (LLVM IR Token Sequence (LLVM-TS)). *An LLVM-TS is a sequence of LLVM language tokens. The token types are LLVM IR instructions, data types, and normalized identifiers.*

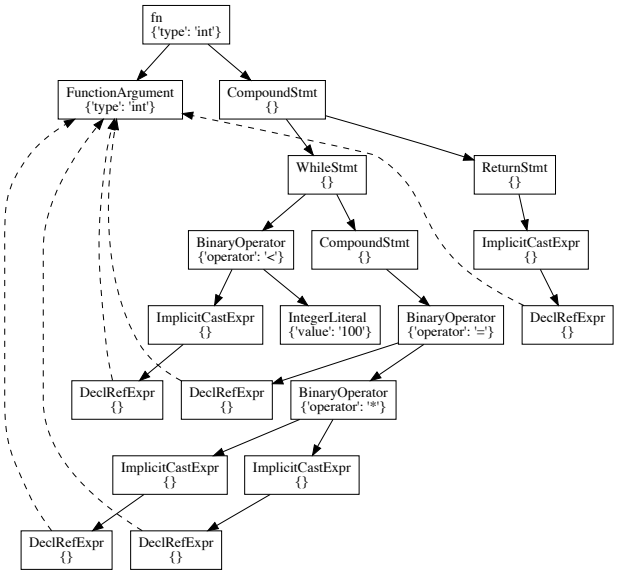
3.4.3 Source Dataflow Graph

As a graph representation close to the source code, we define:

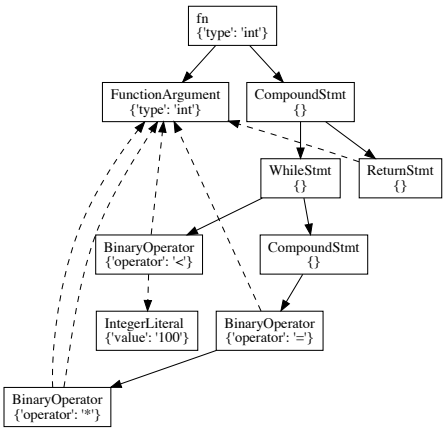
Definition 11 (Source Dataflow Graph (S-DFG)). *An S-DFG is a directed graph, where the nodes are Declarations, Statements, and Types from the Clang AST. The edges are of type AST, representing the child-of relationships within the AST, and dataflow, representing the dataflow.*

S-DFGs are constructed using the Clang AST and applying two modifications to it. First, the tree representation is enriched with dataflow edges that are extracted from Clang's semantic analysis module, resulting in a graph. Secondly, trivial chains of AST nodes of type `ImplicitCastExpr` and `ImplicitCastExpr` are eliminated because they are a regularity and bloat up the graph representation. Having small graphs is especially an advantage when considering graph-generative models. This also holds true in the embedding models we use because fewer hops help to propagate information farther.

The running example is continued in Figure 3.7, showing the original Clang AST enriched with dataflow edges (Figure 3.7a) and the described S-DFG representation (Figure 3.7b).



(a) Original Clang AST enriched with dataflow edges.



(b) S-DFG representation.

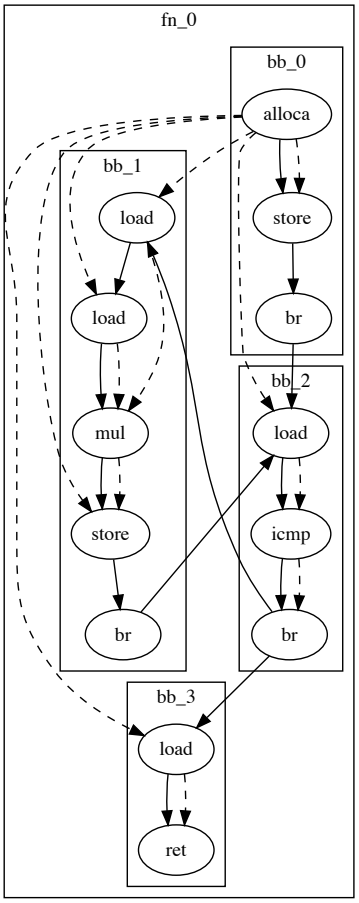
Figure 3.7: Transformation of the example function into the S-DFG representation by eliminating certain node types. AST child edges are in solid, dataflow edges in dashed style.

3.4.4 LLVM IR Control- and Dataflow Graph

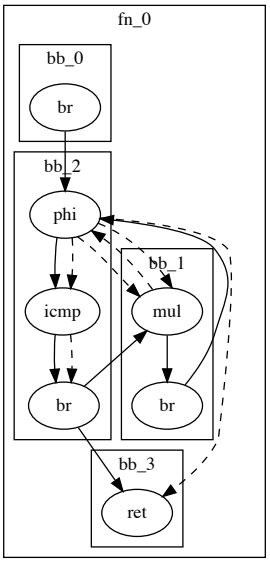
As a graph representation close to the LLVM IR code, we define:

Definition 12 (LLVM IR Control- and Dataflow Graph (LLVM-CDFG)). *An LLVM-CDFG is a directed graph, where the nodes are LLVM IR instructions, and the edges are of type control- and dataflow.*

Definition 13 (LLVM IR SSA Control- and Dataflow Graph (LLVM-SSA-CDFG)). *An LLVM-SSA-CDFG is a variant of an LLVM-CDFG. Memory accesses are replaced with register dependencies that are expressed with phi instructions. The edges are of type control- and dataflow.*



(a) LLVM-CDFG graph.



(b) LLVM-SSA-CDFG graph.

Figure 3.8: Example function in LLVM IR-based representations. Control-flow edges are in solid, dataflow edges in dashed style.

4

Embedding programs with Artificial Neural Networks

In this chapter, we will describe the deep learning models for feature extraction of the program code representations introduced in chapter 3. Outlined in Figure 4.1, these models are part of the architectures for the predictive and generative tasks shown in chapter 5.

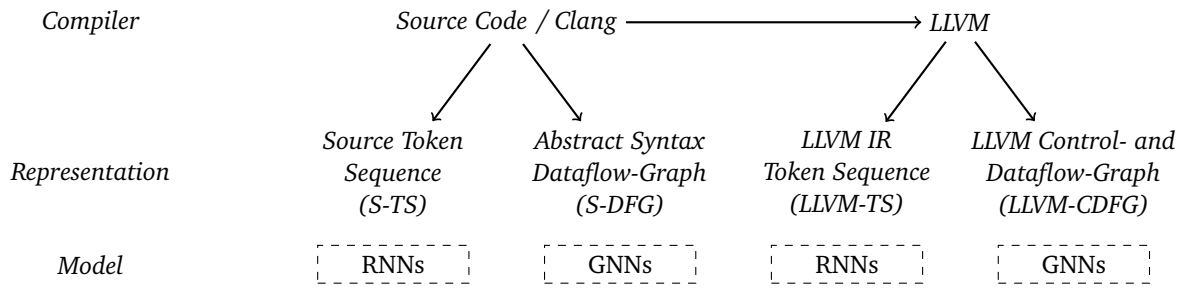


Figure 4.1: Representations of code in the compiler with their respective machine learning models. Please note that this Figure is an extension of Figure 3.5

4.1 Artificial Neural Networks

4.1.1 Multilayer Perceptrons

A *Multilayer Perceptron (MLP)* is an Artificial Neural Network that consists of multiple perceptrons that are hierarchically grouped into layers. The right side of Figure 4.2 shows such a perceptron which consists of an input layer, at least one hidden layer, and an output layer. The perceptrons in each of the layers depend only on the outputs of the previous layer's perceptrons, enabling efficient computation.

The left side of Figure 4.2 shows the computation of the output of a single perceptron according to the model defined by Rosenblatt (1958). By summing the element-wise product of the outputs x of the perceptrons in the previous layers with individual, learnable weights w, the network's activation is computed. Another part of the perceptron's activation is a bias that is multiplied with a learnable weight. The bias shifts the perceptron's activation independently of the previous perceptrons. Finally, a non-linear activation function f is applied to compute the perceptron's output.

In algebraic notation, the output of a MLP with one hidden layer is defined for each hidden perceptron of index i and each output perceptron of index i, whereas w and b are learnable weights:

$$h_i = f\left(\sum_{j=1}^{n} w_{ji}^1 x_j + b_i^1\right) \tag{4.1.1}$$

$$y_i = f\left(\sum_{j=1}^{n} w_{ji}^2 h_j + b_i^2\right) \tag{4.1.2}$$

In vectorized notation:

$$h = f(W^1 x + b^1) \tag{4.1.3}$$

$$y = f(W^2 h + b^2) \tag{4.1.4}$$

4.1.2 Activation Functions

In a biological perceptron, the decision to fire is made by an activation threshold. The activation function models this behavior. Two of the most desirable properties for activation functions are differentiability and non-linearity. As gradient-based optimization methods use the derivatives to compute the model weight updates, it is mandatory for the activation

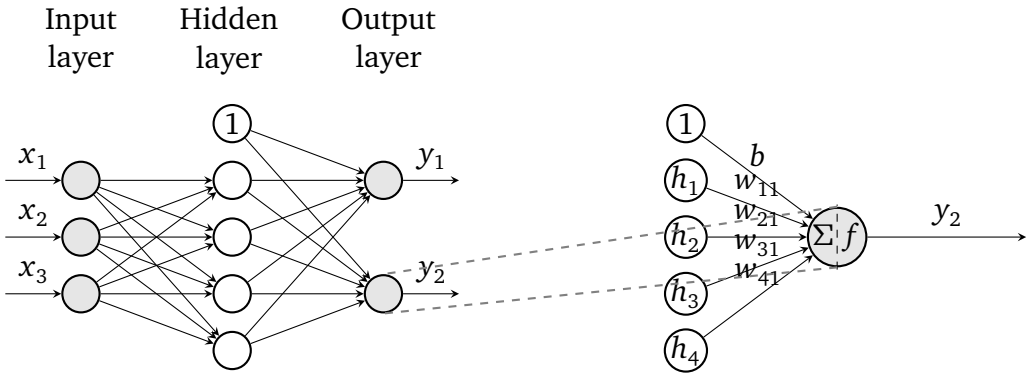


Figure 4.2: An Illustration of a MLP (left) and an individual perceptron (right).

function to be differentiable. Non-linearity is important because a network with multiple layers with linear activation functions can be reduced to a single layer, which is undesirable.

The *Unit Step* function serves as the activation function in the initial perceptron model by Rosenblatt (1958). While this function is capable of modeling a decision boundary for a single neuron, it is insufficient for more complex functions such as regressions. Also, it cannot be optimized with gradient-based methods because the *Unit Step* function is not differentiable.

Multiple activation functions that are both differentiable and non-linear have been proposed for different use cases. Relevant for this work are the *Logistic Sigmoid*, the *Hyperbolic Tangent*, and the *Softmax* activation functions, shown below.

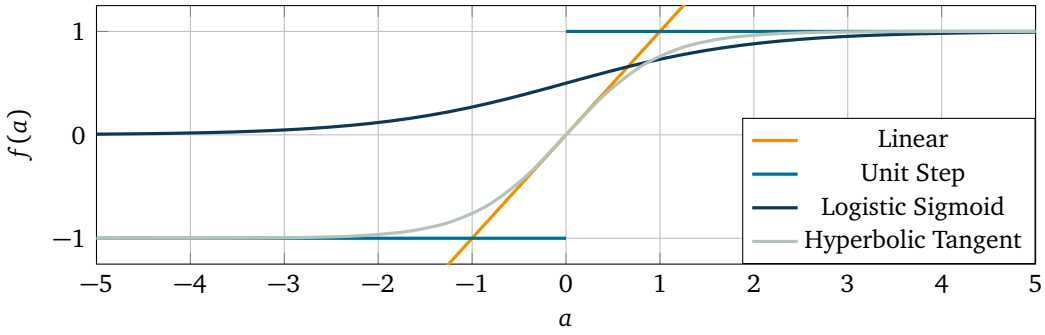


Figure 4.3: Visualization of common activation functions

Logistic Sigmoid The logistic Sigmoid function, denoted as σ, is commonly used in binary classification. Its output range of $(0, 1)$ is a natural fit for this type of problem, representing a probability distribution of p and $1-p$ of two classes.

It is defined as:

$$f(a) = \frac{1}{1+e^{-a}} \tag{4.1.5}$$

Hyperbolic Tangent The hyperbolic tangent function, commonly denoted as $tanh$, has similar properties as the logistic sigmoid function. However, with its range of $(-1, 1)$, its gra-

dients are steeper and it suffers less from the optimization problems.

The hyperbolic tangent function is defined as:

$$f(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{4.1.6}$$

Softmax For multiclass classification problems, the softmax function is a natural choice. Taking a vector of real-valued numbers x as input, they are normalized into a probability distribution proportional to the exponentials of its input numbers.

For a perceptron's activation a_i out of all n perceptrons activations in this layer, it is defined as:

$$f(a_i) = \frac{e^{a_i}}{\sum_{j=1}^{n} e^{a_j}} \tag{4.1.7}$$

4.1.3 Optimization with Supervised Learning

A perceptron is a function with an input x and an output y. Given tuples of (x_i, y_i), the learnable weights w of the MLP function is optimized with the objective that for each input x_i the function produces the desired output y_i, which is commonly referred to as *supervised learning*, the tuples to as training samples.

The objective is defined as a *loss function*, which measures the fit of the output of the training data and the models' predicted output. A loss function for *regression problems* is the *mean-square error* is defined below, whereas f is the model, and θ the learnable weights for a single training sample:

$$l_i(x_i, y_i, \theta) = (f(x_i, \theta) - y_i)^2 \tag{4.1.8}$$

A loss function for *classification problems*, such as the ones we will show in the next chapter, is the *cross entropy*, whereas x_i and y_i are one-hot encoded vectors of size m:

$$l_i(x_i, y_i, \theta) = \sum_{j=1}^{m} y_{ij}\ log(f(x_{ij}, \theta)) \tag{4.1.9}$$

For all training samples, we obtain:

$$L(X, Y, \theta) = \frac{1}{n} \sum_{i=1}^{n} l_i(x_i, y_i, \theta) \tag{4.1.10}$$

Algorithm 1: Gradient Descent

initialize w, b;
foreach *sample* $\in$ *samples* **do**
  $w = w + \Delta w$;
  $b = b + \Delta b$;
end

Algorithm 2: Batch Gradient Descent

initialize w, b;
$\Delta W = 0$;
$\Delta B = 0$;
foreach *sample* $\in$ *samples* **do**
  $\Delta W = \Delta W + \Delta w$;
  $\Delta B = \Delta B + \Delta b$;
end
$w = w + \Delta W$;
$b = b + \Delta B$;

Gradient Descent Algorithm Optimization of the loss function, commonly referred to as *model training*, can efficiently be done with gradient-based algorithms. The most basic algorithm is *Gradient Descent*, which iteratively updates the model weights according to the following update rules that compute a delta that is applied to calculate the new model weights, whereas η is the *learning rate*:

$$\Delta w = -\eta \sum_i \frac{\partial l_i(\theta)}{\partial w} \tag{4.1.11}$$

$$\Delta b = -\eta \sum_i \frac{\partial l_i(\theta)}{\partial b} \tag{4.1.12}$$

The learning rate controls the step size of the weight updates. A small learning rate will result in a smaller step size, a higher learning rate in larger step size.

Multiple variants of the *Gradient Descent* algorithm exist. While in the basic version, the weights are updated per sample, and the model is estimated with those new weights for the next sample, the *Batch Gradient Descent* variant collects the weight updates for the whole training set, before updating them. This reduces the variance and therefore results in a more stable training. However, the optimization can converge in local optima, as shown in Figure 4.4.

The *Stochastic Gradient Descent* variant mitigates this problem by estimating the new parameters based on a random subset called *minibatches* of the training set. The size of this subset is commonly referred to as *batch size* influences the models' convergence and training stability.

Algorithm 3: Stochastic Gradient Descent

initialize w, b;
$\Delta W = 0$;
$\Delta B = 0$;
minibatches = partition(shuffle(samples));
foreach *minibatch* $\in$ *minibatches* **do**
  foreach *sample* $\in$ *minibatch* **do**
    $\Delta W = \Delta W + \Delta w$;
    $\Delta B = \Delta B + \Delta b$;
  end
  $w = w + \Delta W$;
  $b = b + \Delta B$;
end

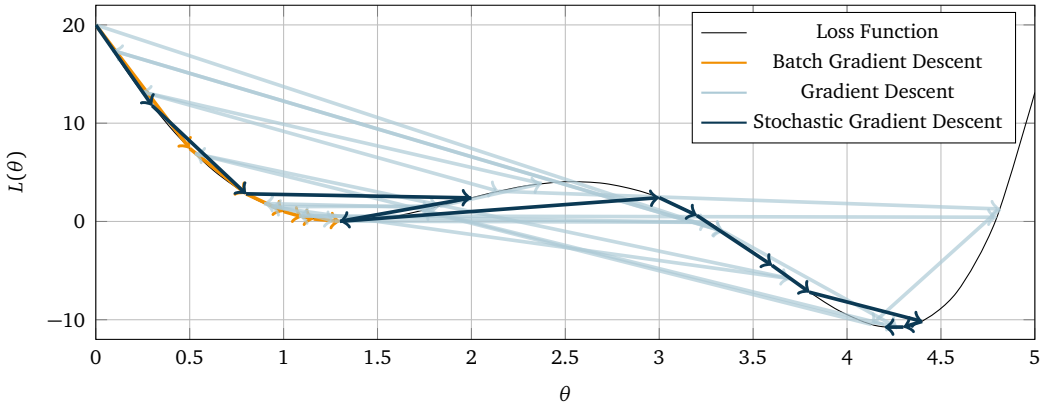


Figure 4.4: Visualization of Gradient Descent algorithm variants.

4.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a family of models that are well-suited for extracting features about dependencies in sequential data. They have successfully been used in the areas of *Natural Language Processing (NLP)*, *Speech Recognition*, and *Time-series prediction*.

Figure 4.5 shows the architecture of an RNN on the left side. In comparison to feed-forward neural networks such as MLPs, they have a feedback loop, which makes the network's output consumable as input in the next iteration. This enables the network to hold a state and learn features over time.

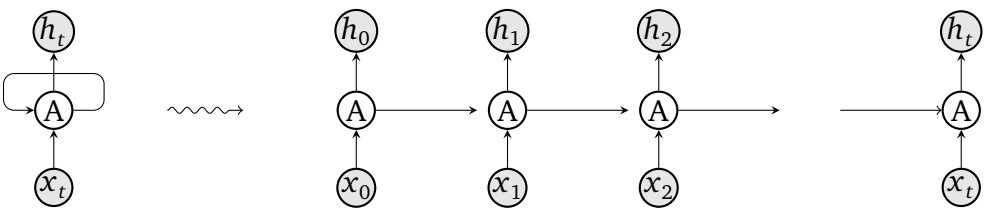


Figure 4.5: A static and a unrolled representation of a Recurrent Neural Network.

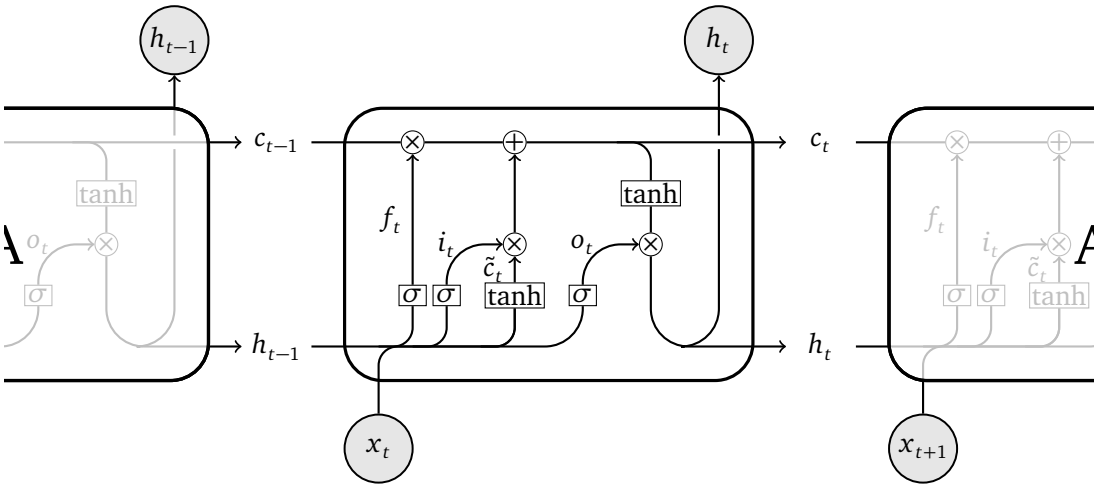


Figure 4.6: Internal architecture of a LSTM cell.

In order to train RNNs, Werbos et al. (1990) introduced a method called *Backpropagation through time (BPTT)*. As shown in Figure 4.5 on the right side, the RNN is unrolled for several iterations, while the individual instances of the RNN cell share the same learnable parameters. Then it can be efficiently optimized using gradient-based methods, such as *Backpropagation*. BPPT works well for short sequences. However, for long sequences a number of problems arise: First, the network needs to be unrolled for the length of the sequence in order to extract features ranging into the past. Because computing the gradients involves a multiplication, it is likely for gradients to result in 0 or infinity, if the individual gradients are less or greater than 1. These so-called *vanishing gradients* and *exploding gradients* problems can be solved by using *gradient clipping* (Pascanu et al. (2013)) and modern RNN cell architectures such as the *Long Short-term Memory* (Hochreiter and Schmidhuber (1997)) and the *Gated Recurrent Unit* (Cho et al. (2014)) that can pass information, therefore also the gradients unchanged over long distances.

4.2.1 Long Short-Term Memory

The Long Short-Term Memory (LSTM), proposed by Hochreiter and Schmidhuber (1997), is an RNN architecture that solves the *vanishing gradients* problem, which causes gradients to converge to zero when training an RNN with BPTT. Therefore, LSTMs are able to extract long-term features in sequential data and caused breakthroughs of deep learning methods in various domains.

Mitigation of the *vanishing gradients* problem is achieved by an internal cell architecture that allows the information and gradient to be passed through the network unaffected, intuitively in the scope of a long-term dependency. The cell architecture, as shown in Figure 4.6 consists of a *cell state*, an *input gate*, an *output gate*, and a *forget gate*.

The *cell state* is the internal memory of the LSTM. To obtain the next state, the previous *cell state* c_{t-1} is modified by multiplying it with the output of the *forget gate*. Also part of the

sum is the input of the current time step, modified by the *input gate*:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{4.2.1}$$

The first component of the new *cell state* is controlled by the *forget gate*. Its task is to decide the information to keep from the cell state c_{t-1} of the previous time step. It is a learnable function based on the hidden state h_{t-1} and the input x, using the sigmoid activation function. When used in the described context, the function's output - a vector of values in the range $[0, 1]$ - represents the importance of the individual components of the previous *cell state*.

$$f_t = \sigma\left(W_f \cdot h_{t-1} + W_f \cdot x_t + b_f\right) \tag{4.2.2}$$

The second component consists of a modified representation of the input, controlled by the *input gate*. The input is modified by a learnable function with $tanh$ activation. Its task is to normalize the input to a range of $[-1, 1]$.

$$\tilde{c}_t = tanh\left(W_c \cdot h_{t-1} + W_c \cdot x_t + b_c\right) \tag{4.2.3}$$

The *input gate* then decides what information is relevant of the current input x. Its implementation is the same as the *forget gate*, but with a different set of learnable weights.

$$i_t = \sigma\left(W_i \cdot h_{t-1} + W_i \cdot x_t + b_i\right) \tag{4.2.4}$$

After the *cell state* has been computed, it is passed on to the next timestep. In order to compute the output h_t of the cell, the *output gate* is applied to the *cell state*, which is normalized by a $tanh$ activation function. Again, the function of *output gate* is the same as the *forget gate*, just with a different set of learnable weights. The output also acts as the new hidden state h_t.

$$o_t = \sigma\left(W_o \cdot h_{t-1} + W_o \cdot x_t + b_o\right) \tag{4.2.5}$$

$$h_t = o_t \odot tanh\left(c_t\right) \tag{4.2.6}$$

4.2.2 Gated Recurrent Unit

Another recent architecture of an RNN is the Gated Recurrent Unit (GRU), proposed by Cho et al. (2014). As shown in Chung et al. (2014), it is on-par with the LSTM architecture in different types of tasks where the data is of sequential nature. As it can be seen in Figure 4.7, the GRU cell has fewer gates than the LSTM cell, which reduces the model's memory requirements. Nevertheless, LSTM and GRU models are commonly used in practice as both offer comparable levels of prediction performance.

In contrast to the LSTM, the GRU has only one recurrent state, which we will denote as *cell state*. It is updated using a sum of the previous *cell state* h_{t-1} and a *candidate state* $\tilde{h}_t$,

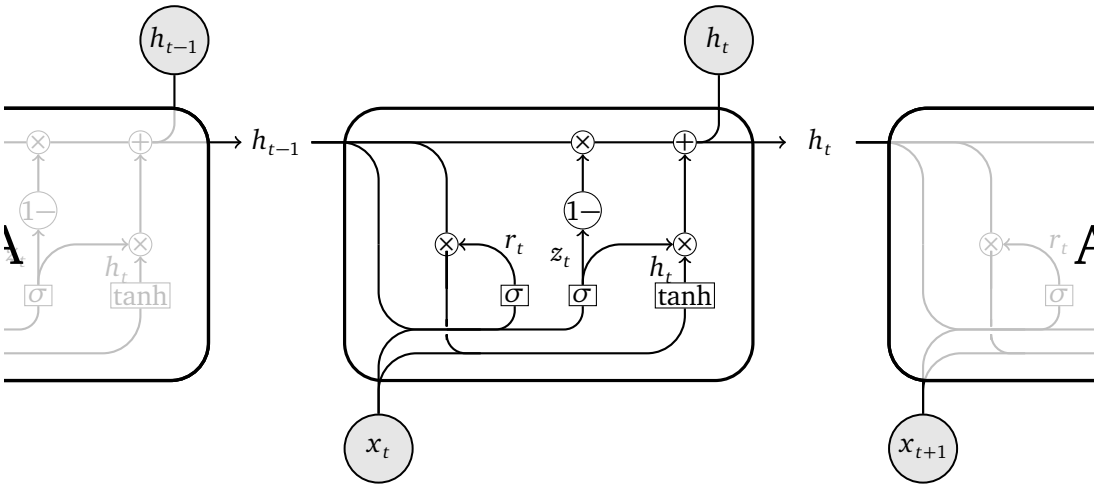


Figure 4.7: Internal architecture of a GRU cell.

whereas an *update gate* decides about the ratio of the vectors individual components.

$$h_t = (1 - z_t) \odot h_{t-1} + (z_t) \odot \tilde{h}_t \tag{4.2.7}$$

Similarly to the LSTM, the *update gate* is implemented with a learnable function based on the *cell state* h_{t-1} and the input x, using the sigmoid activation function. When used in the described context, the function's output - a vector of values in the range $[0, 1]$ - represents the importance of the individual components of the previous *cell state*.

$$z_t = \sigma(W_z \cdot h_{t-1} + W_z \cdot x_t) \tag{4.2.8}$$

The *candidate state* consists of the sum of the weighted *cell input* x and the gated previous state, using the *reset gate* that computes a vector r_t representing the importance.

$$r_t = \sigma(W_r \cdot h_{t-1} + W_r \cdot x_t) \tag{4.2.9}$$

$$\tilde{h}_t = tanh(W_o \cdot r_t \odot h_{t-1} + W_o \cdot x_t) \tag{4.2.10}$$

4.3 Graph Neural Networks

Graph Neural Networks (GNNs) are deep learning models that are well-suited for extracting features from graph data. In contrast to other node and graph embedding models, they can be optimized end-to-end as part of a task-specific architecture and therefore produce task-specific embeddings. Johnson (2017) and Li et al. (2016) have shown that GNNs can outperform RNNs in the BaBi reasoning task benchmark of Weston et al. (2015), reaching a perfect accuracy of 100%. Up to today, multiple variants of the GNN model have been

proposed. Here, we will focus on a recurrent and a convolutional variant.

The GNN model consists of two-phases: The input structure to the *propagation phase* is an annotated graph $G = (V, E)$. Nodes $v \in V$ are annotated with their type $v_t \in \mathbb{N}$, edges $e \in E$ with their type $e_t \in \mathbb{N}$. The outputs of this phase are *node embedding vectors* $h_v \in \mathbb{R}^d$ that are assigned to all $v \in V$. In the following *output phase*, a graph embedding vector is produced by aggregating the node embedding vectors.

4.3.1 Propagation Phase

Nodes of the input graph are represented as one-hot encoded vectors, i.e. vectors $(e_j)_i = \delta_{i,j}$, where $\delta_{i,j}$ is the *Kronecker Delta* which is 1 iff $i = j$ and 0 otherwise. Figure 4.8 illustrates this process on a simple graph, which we will further use as a running example.

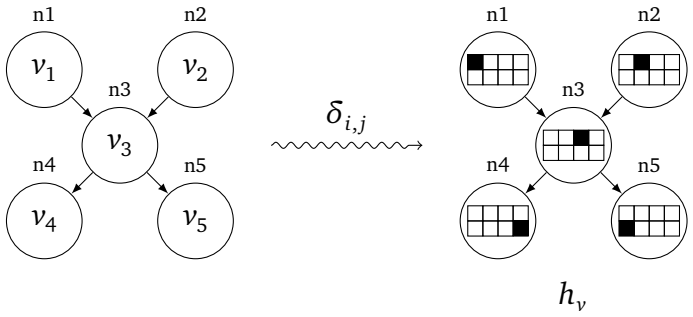


Figure 4.8: An illustration of the initial node embedding as one-hot encoded vectors.

By applying an iterative information propagation scheme, node embedding vectors are increasingly enriched with structural information of the graph. For a number of iteration steps T, the node embeddings are propagated to their directly adjacent nodes and then aggregated. This eventually results in node embeddings that contain information about a T-sized neighborhood. Figure 4.9 illustrates this scheme, where the cells represent the information encoded as embedding vectors.[1] For example, the embedding of the node n5 includes embedding information of the nodes n1, n2, and n3.

Gilmer et al. (2017) consolidate the notions of various GNN models and define a common framework, named the *Message Passing Neural Network (MPNN)*. For a consistent notation of the different GNN-based models, we will use this framework for further definitions:

$$m_v = \sum_{u:(u,v)\in E} f_{\text{msg}}(h_u, h_v, e_t) \; \forall v \in V \tag{4.3.1}$$

$$h'_v = f_{\text{prop}}(m_v, h_v) \; \forall v \in V \tag{4.3.2}$$

In each iteration, messages are formed for each node and each edge by using the output of a

[1] Note that this is an artificial example to give an intuition of the model. Depending on the task, the propagated embeddings will most likely be much more complex than shown in this example.

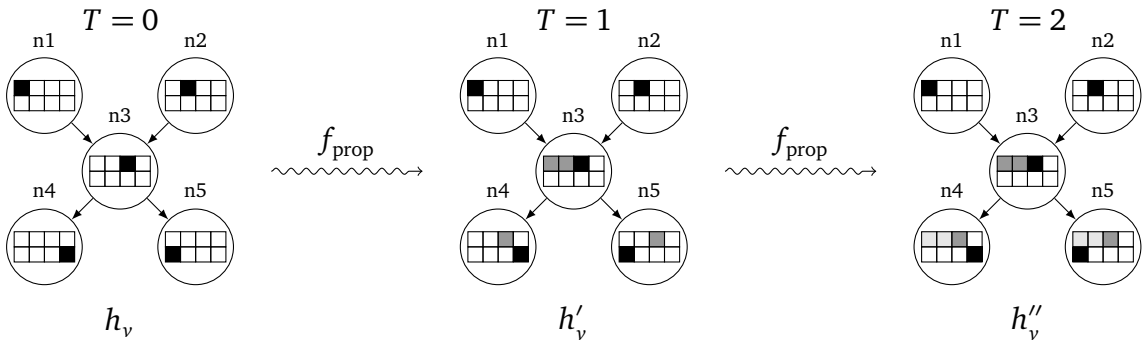


Figure 4.9: An intuitive illustration of the GNN node embedding propagation scheme across 3 iterations.

message function $f_{\text{msg}}(h_u, h_v, e_t)$, based on the embedding informations h_u, h_v, and the edge type information e_t. Summing the messages per target node results in message vectors m_v.

New node embedding vectors h'_v are formed by applying a learnable update function $f_{\text{prop}}(m_v, h_v)$, which are based on the corresponding message vectors m_v and the current node embeddings h_v.

4.3.2 Output Phase

After T iterations of the propagation scheme, the node embeddings reach a final state. Li et al. (2016) define an attention-based mechanism to aggregate a variable amount of node embeddings to a single fixed-sized graph embedding vector:

$$h_v^G = f_m(h'_v) \tag{4.3.3}$$

$$g_v^G = g_m(h'_v) \tag{4.3.4}$$

$$h_G = \sum_{v \in V} g_v^G \odot h_v^G \tag{4.3.5}$$

After mapping the node embedding vectors to a higher size with the learnable function $f_m(h'_v)$ and $g_m(h'_v)$, they are summed to the fixed-sized graph embedding vector h_G. The attention mechanism is implemented by the element-wise product of $f_m(h'_v)$ and $g_m(h'_v)$, denoted as $\odot$. This decides the relevance of individual nodes in the current task. Figure 4.10 illustrates the aggregation scheme.

The functions are implemented as MLPs with hyperbolic tangent and sigmoid activation functions. This makes the model differentiable and restricts the range of the output values to $(-1, 1)$ for the hyperbolic tangent and $(0, 1)$ for the sigmoid function. The range of the sigmoid function is especially well-suited for an attention mechanism because it represents a

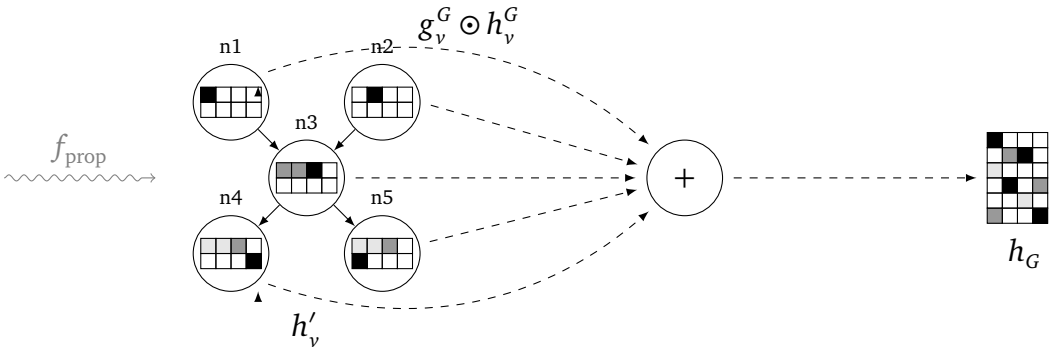


Figure 4.10: An illustration of the aggragation of node embedding vectors to a single, fixed-sized graph embedding vector.

weight function. The hyperbolic tangent function on the other hand normalizes the outputs, which improves the model's numerical stability and overall performance.

4.3.3 Recurrent Propagation Schemes

Initially, the notion of the *recurrent graph neural network* has been described by Scarselli et al. (2008).

In this initial version of the graph neural network, the function f_{prop} is implemented as a MLP and the propagation scheme is applied until convergence to compute the gradients when training the model. To ensure convergence, the node embeddings have to be initialized with predefined values that are known to converge in a fixpoint. Therefore, no typed nodes are supported, which is a significant limitation of this model.

The *Gated Graph Neural Network (GGNN)* model by Li et al. (2016) overcomes this limitation by replacing the MLP of the function f_{prop} with a GRU, a modern variant of an RNN. In the notation of the MPNN framework:

$$f_{\text{msg}}(h_u, h_v, e_t) = A_{e_t} \cdot h_v + b_{e_t} \tag{4.3.6}$$

$$f_{\text{prop}}(m_v, h_v) = \text{GRU}(m_v, h_v) \tag{4.3.7}$$

The RNN is unrolled for the T number of iterations, which enables computation of the gradients for graphs without constrained initial node embeddings. This enables initializing the model with arbitrary node embeddings. Therefore, e.g. the type of a node can serve as an additional source of information.

The message function consists of learnable parameters A and b. A is multiplied to the source node embedding and bias b is added. To support multiple edge types e_t, multiple edge type-specific message passing functions f_{msg} can exist.

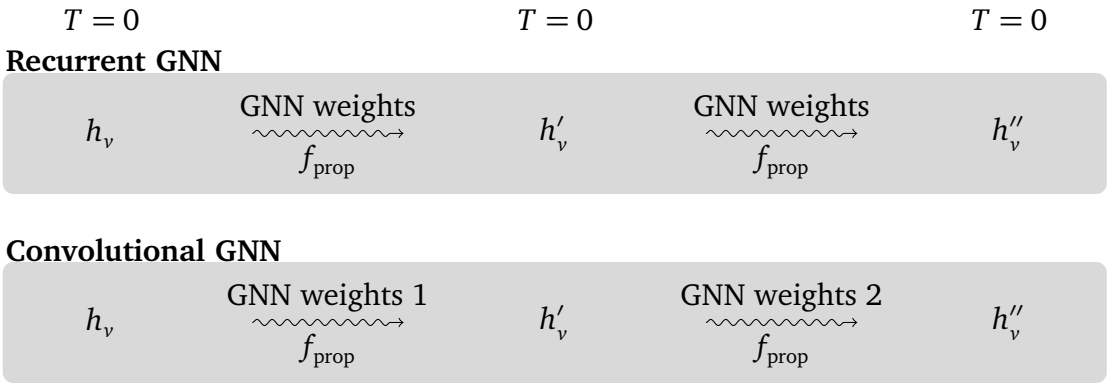


Figure 4.11: A comparison between recurrent and convolutional GNN propagation schemes. The recurrent GNN model shares the learnable weights of f_{prop} across the iterations, the convolutional GNN model uses different learnable weights for each iteration.

4.3.4 Convolutional Propagation Schemes

Convolutional Graph Neural Networks (GCNs) are a class of GNNs that generalize the convolution operation from the euclidian space to an arbitrary amount of adjacent entities.

Kipf and Welling (2017) initially define the GCN. In the notation of the MPNN framework, we obtain:

$$f_{\text{msg}}(h_u, h_v, e_t) = (\deg(u) \cdot \deg(v))^{-1/2} \cdot h_v \tag{4.3.8}$$

$$f_{\text{prop}}(m_v, h_v) = \text{ReLU}(W^t \cdot m_v) \tag{4.3.9}$$

The message function doesn't contain any learnable parameters, but a normalization mechanism only, more concretely the symmetric normalized Laplacian.

The update function multiplies a learnable weight matrix to the message vector m_v. This learnable weight matrix is unique for each time step, which is a major difference to the GGNN, illustrated in Figure 4.11.

5

Task-specific Architectures

In order to be practically usable and to fulfill tasks, the deep learning-based embedding models are integrated into higher-level architectures, which will be the subject of this chapter. Specifically, we will introduce two architectures: One for performing *predictive tasks* and one for *generative tasks*.

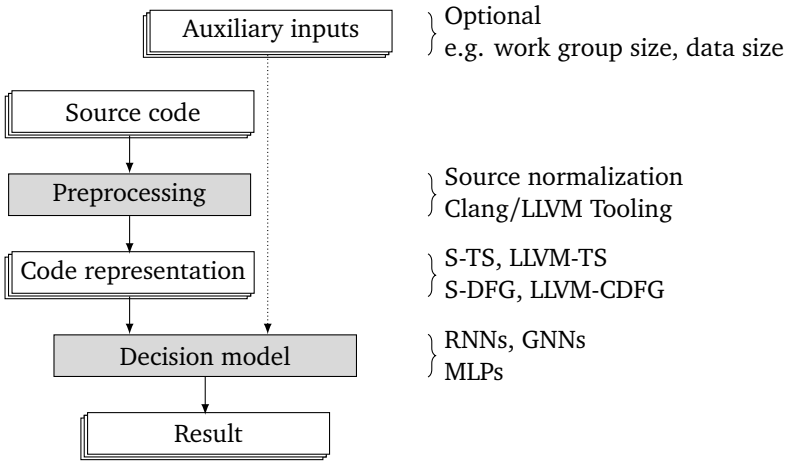


Figure 5.1: High-level architecture of the predictive models.

5.1 Predictive Tasks

Figure 5.1 illustrates the high-level architecture for predictive tasks. The input to the architecture is an original sample of *source code*. Optimally, *auxiliary inputs* information that is critical for the predictive task, but cannot be derived from the source code sample itself can be added. Examples for this are the data size of the input of a function or properties of the execution environment, such as the work-group size of an OpenCL implementation.

The source code sample is processed using a *preprocessing* component that extracts one of the *code representations* introduced in chapter 3 from a given source code sample.

The extracted code representation and the auxiliary inputs then serve as input to a *decision model* that outputs the *result* of the predictive model.

5.1.1 RNN-based

Originally introduced by Cummins et al. (2017a) and adopted by Ben-Nun et al. (2018), the RNN-based architecture supports predictions based on *sequences*, such as the S-TS and LLVM-TS representations.

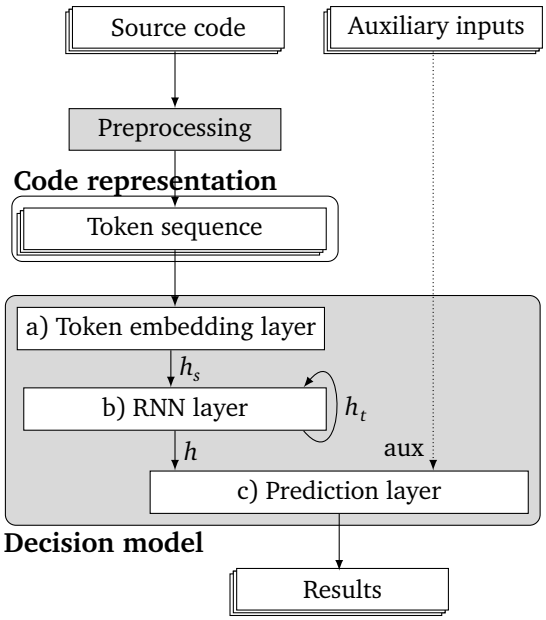


Figure 5.2: Architecture of the predictive model based on Recurrent Neural Networks.

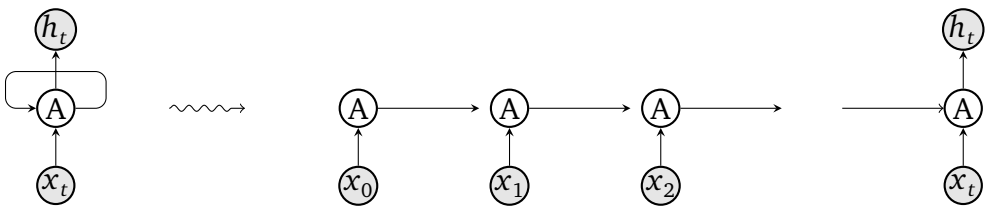


Figure 5.3: Sequence-to-vector architecture of a Recurrent Neural Network.

Figure 5.2 shows this architecture. It consists of a *initial embedding layer* for creating initial embeddings, a *RNN layer*, which extracts features from the token sequence, and a *prediction layer* for performing a prediction.

a) Token embedding layer The number of possible token types can be arbitrarily large in practice, which causes a large dimension when applying a one-hot encoding to the token. Therefore, this layer reduces the large dimension of those one-hot encoded embedding vectors to embedding vectors of a smaller size that matches the size of the model. This is done by a learnable function $f_{\text{init}}(s_t)$ that maps the one-hot encoded token type s_t to token embedding vectors h_s, implemented as a MLP.

$$h_s = f_{\text{init}}(s_t) \tag{5.1.1}$$

b) RNN layer Given a sequence of token embedding vectors h_s, a learnable function f_{seq} computes a vector of features, denoted as h_t. The function is implemented as a RNN neural network as shown in Figure 5.3, which shows a static and dynamic view of it. The vectors h_s correspond to the network's input x.

$$h_t = f_{\text{seq}}(h_s) \tag{5.1.2}$$

c) Prediction layer The final layer maps the extracted features, optionally concatenated with auxiliary inputs, to a probability distribution by applying a learnable function f_{out} to it. It is implemented as a MLP with softmax activation.

$$\text{out} = f_{\text{out}}(h_t, \text{aux}) \tag{5.1.3}$$

5.1.2 GNN-based

Figure 5.4 illustrates our predictive model for source code based on *graphs* with its corresponding components and their relationships. Our architecture consists of a *initial embedding layer* for creating initial embeddings, a *embedding propagation layer* for enriching the initial embeddings with structural information, and a *prediction layer* that aggregates the propagated embeddings and performs a prediction. The input of the decision model is a code representation graph (e.g. S-DFG, LLVM-CDFG), the output a n-sized vector representing a probability distribution, with n being the number of classes. We will proceed with a description of the layers in greater detail.

a) Initial embedding layer Since the number of possible node types tends to be very large in practice, we introduce this embedding layer to reduce the dimension of these one-hot encoded node vectors, to smaller size $d \in \mathbb{N}$. The node embedding vectors h_v are computed

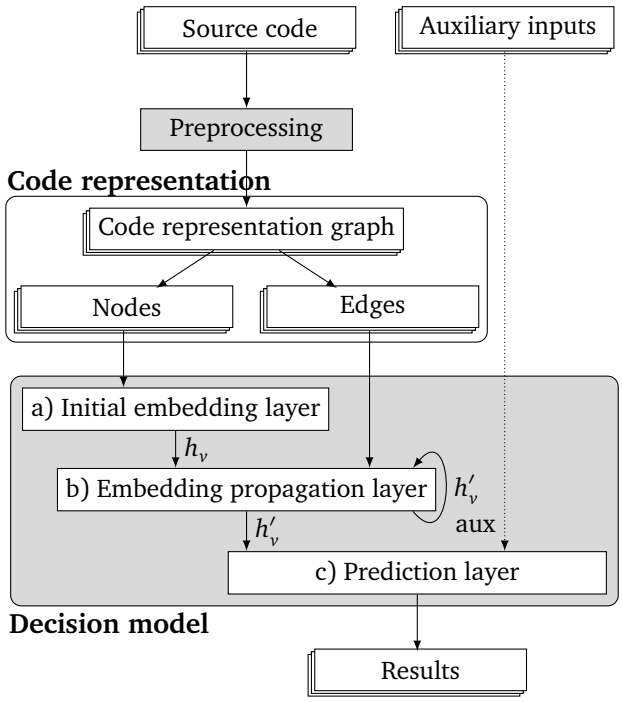


Figure 5.4: Architecture of the predictive model based on Graph Neural Networks.

by applying a learnable function $f_{\text{init}}(v_t)$ to the node annotation vector v_t. The learnable function is implemented as a MLP neural network.

$$h_v = f_{\text{init}}(v_t) \tag{5.1.4}$$

b) Graph neural network layer With the initial embeddings based only on the node type, they do not contain any structural information of the graph, i.e. as expressed by the edges. By applying a GNN propagation scheme for T iterations, the initial node embedding vectors produced by the initial embedding layer are enriched with this structural graph information.

$$h'_v = \text{prop}^T(h_v, G) \tag{5.1.5}$$

c) Prediction layer This layer maps the final node embedding vectors to a probability distribution. To achieve this, we follow the notion of the output model described earlier: A fixed-size graph embedding vector is computed by aggregating the final node embedding vectors.

A final learnable function $f_{out}(h_G)$ computes the output based on the graph embedding vector h_G that is optionally concatenated with the auxiliary inputs aux. This function is implemented as a MLP with a softmax activation function.

$$\text{out} = f_{\text{out}}(h_G, \text{aux}) \tag{5.1.6}$$

5.2 Generative Tasks

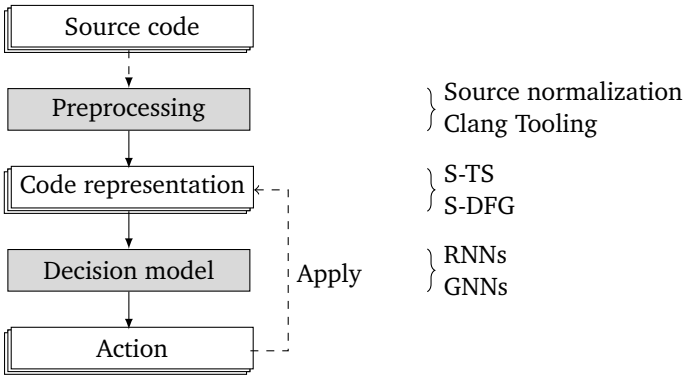


Figure 5.5: High-level architecture of the generative models.

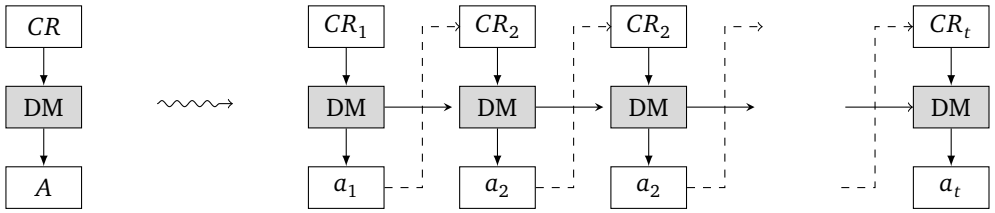


Figure 5.6: High-level dynamic view of the generative models.

5.2.1 RNN-based

Used by the method of Cummins et al. (2017b), a sequence-to-vector architecture incrementally predicts a sequence of tokens. This architecture closely follows Figure 5.6, whereas the model predicts an action A based on the current state, which is the weights of the decision model DM, and the current code representation CR. When sampling an action, the model's state is changed and the action is applied to the current state of the code representation CR.

5.2.2 GNN-based

For generating program graphs, we choose the graph-generative method by Li et al. (2018a) that sets new state-of-the-art results in the task of chemical molecule generation. The method outperforms a sequence-based generation process, in which the molecule graphs are encoded in the SMILES[1] grammar and embedded by an LSTM model, based on Hochreiter and Schmidhuber (1997) by 5 percentage points in a validity and by 10 percentage points in a novelty metric. Despite the different domains, the chemical molecule graphs also contain similarly rich structures and properties as the proposed program graphs, as they have typed edges, representing bonds and typed nodes, representing atoms.

[1]The simplified molecular-input line-entry system (SMILES) is a compact sequential notation of molecule graphs

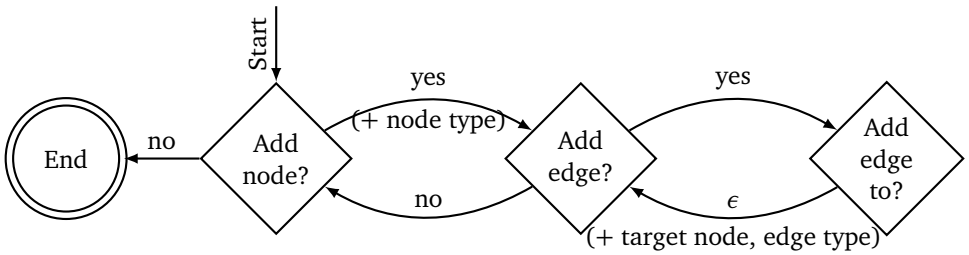


Figure 5.7: State diagram of the incremental graph generation process.

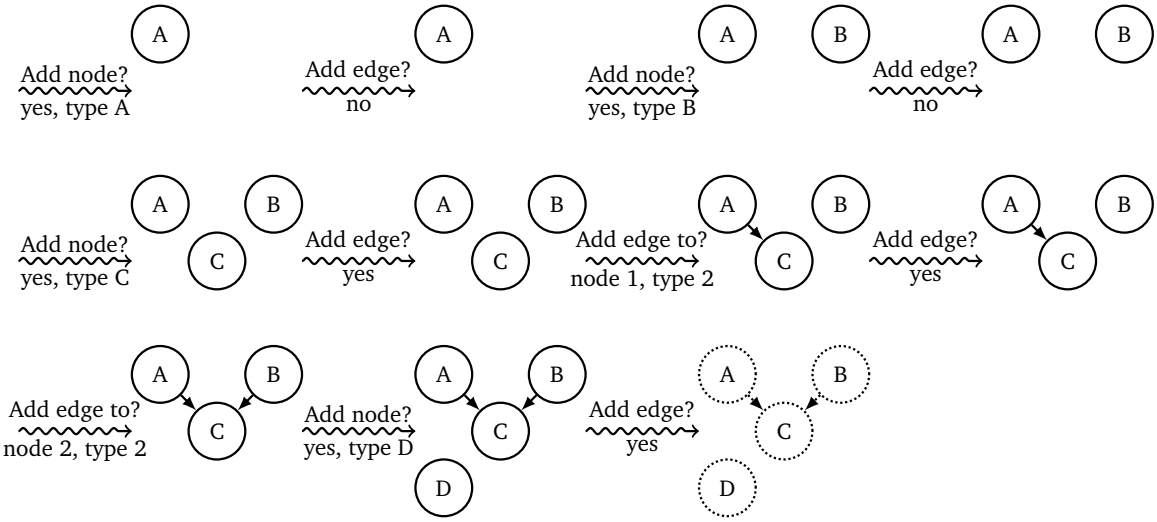


Figure 5.8: Example of the incremental graph generation process.

The graph-generative model consists of three steps: In a first preprocessing step, the graphs are transformed into a sequence of structure-building actions. By applying these actions, the graph can be re-generated in an incremental fashion. In the second step, a GNN-based model is optimized on a large body of training data with the objective of predicting those structure-building actions. Finally, the optimized generative model is used to predict structure-building actions that construct a graph incrementally.

This model also follows the dynamics shown in Figure 5.6. The code representation *CR* is a current state of the graph, *A* a structure-building action. Figure 5.7 shows the state transitions of the graph generative model. Starting with an empty graph, the model transitions into the *Add node?* state and predicts whether to add a node of a specific type. If so, a node of the specific type is added to the current state of the graph and the model transitions into the *Add edge?* state and predicts whether to add an edge. In case, it transitions into the *Add edge to?* state and predicts the target node and the edge type. Then, this edge is added to the current state of the graph. An example of this incremental process is given in Figure 5.8. The arrows represent the actions that are applied to the current state of the graph.

In Figure 5.9, we show the model architecture with its components and their relationships. After a code representation component embeds the current state of the graph, subsequent action-specific modules predict a probability distribution of the output. This output is sampled and formed into an action, which is applied to the current state, potentially leading to a new state of the graph. We will proceed with a description of the layers in greater detail.

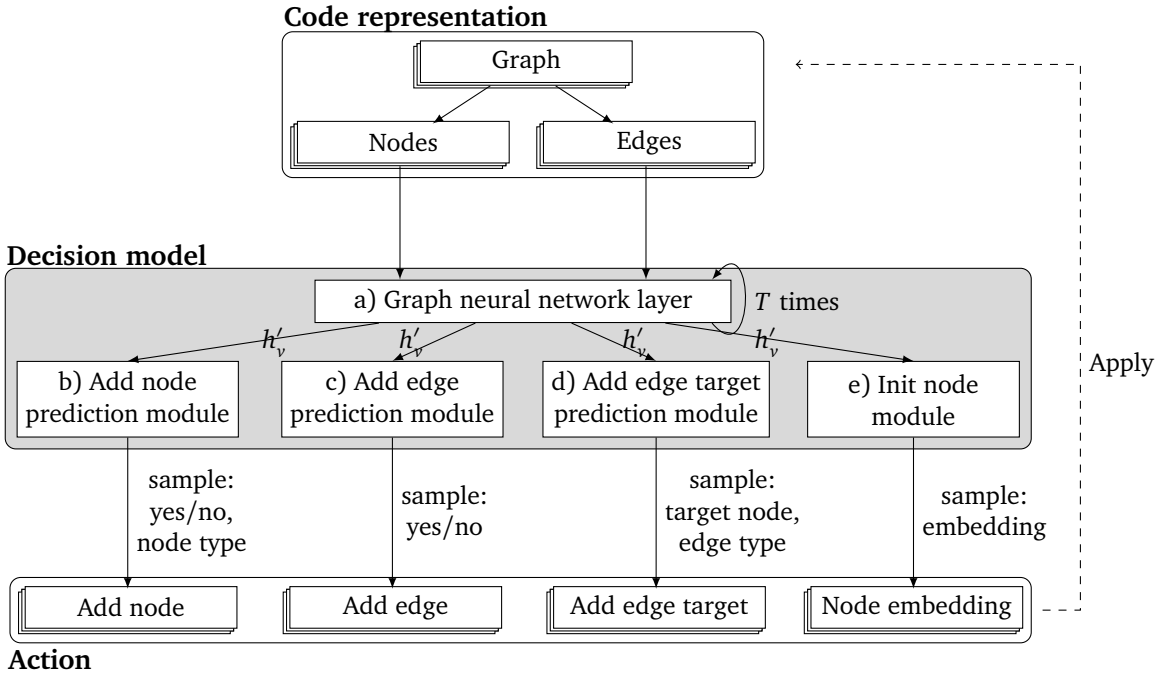


Figure 5.9: Architecture of the generative model based on Graph Neural Networks.

a) Graph neural network layer The current state of the graph is encoded with the one-hot encoding described earlier. With the initial embeddings based only on the node type, they do not contain any structural information of the graph, i.e. as expressed by the edges. By applying a GNN propagation scheme for T iterations, the initial node embedding vectors produced by the initial embedding layer are enriched with this structural graph information.

b) Add node prediction module Based on a graph embedding shown in Equation 4.3.5, this module uses a learnable function f_{an} that outputs a probability distribution representing a *no* and a node type output. It is implemented as a MLP, whereas the softmax activation function makes the module differentiable and normalizes the probabilities.

$$f_{\text{addnode}}(G) = softmax(f_{\text{an}}(h_G)) \tag{5.2.1}$$

After adding a new node to the graph, its node embedding needs to be initialized. The authors suggest to initialize it by a function f_{init} that concatenates an aggregation across all existing node embeddings with another instance of the function shown in Equation 4.3.5, and a node feature x_v. In this case, we take the node type as this node feature.

$$h_v = f_{\text{init}}(h_G, x_v) \tag{5.2.2}$$

c) Add edge prediction module The learnable function f_{addedge}, which is implemented as a MLP, outputs a single probability for adding an edge or not. Its inputs are a graph embed-

ding, shown in Equation 4.3.5 and the node embedding of the last added node.

$$f_{\text{addedge}}(G, v) = \sigma(f_{\text{ae}}(h_G, h'_v)) \tag{5.2.3}$$

d) Add edge target prediction module This module computes the probability of adding an edge between the last added node for all nodes that exist in the current graph. The learnable function f_s is implemented as a MLP, that calculates a single score for each pair of the current node embedding and the other nodes embeddings. The edge type is represented in an additional dimension. In a second step, the scores of all pairs of all edge types are concatenated and normalized by applying the softmax function.

$$s_u = f_s(h'_u, h'_v) \ \forall u \in V \tag{5.2.4}$$

$$f_{\text{nodes}}(G, v) = softmax(s) \tag{5.2.5}$$

e) Init node module After a node has been added to the current state of the graph, this particular node embedding, denoted as h_v, doesn't exist. A naïve solution to solve this could be to initialize the embedding with a one-hot encoding of the predicted node type. However, this approach has the shortcoming that the embedding doesn't contain any information about its neighborhood. Therefore, a learnable function f_{init} is used that outputs an initial embedding for the last newly added node, conditioned on the node type e and the graph embedding h_G^{init}, which uses a separate set of learnable parameters than the other modules and therefore is a different function than h_G. The function f_{init} itself is implemented as a MLP that outputs a vector of the node embedding's dimension.

$$h_v = f_{\text{init}}(e, h_G^{init}) \tag{5.2.6}$$

6

Design and Implementation

In this chapter, we will describe the design and implementation of a framework that enables an evaluation of the graph-based program representations introduced in chapter 3 and the graph embedding models described in chapter 4 and chapter 5 in specific tasks. After analyzing the requirements, we will describe our design and highlight our key decisions.

6.1 Requirement Analysis

The functional requirements of the framework to be designed can be grouped into requirements of the *models*, the *code representations*, the *tasks*, and the *experiments*. Besides that, *non-functional* requirements apply. In Table 6.1, a summary of the requirements is shown, while are more detailed description of them is given below.

Code representation The code representations that need to be extracted from raw C source code are the ones described in chapter 3: The Source Dataflow Graph (S-DFG), the LLVM IR Control- and Dataflow Graph (LLVM-CDFG), and the LLVM IR SSA Control- and Dataflow Graph Representation (LLVM-SSA-CDFG) (R1, R2, R3). The implementation of the sequential code representations on the other hand is already part of existing published source code artifacts of Cummins et al. (2017b) and Cummins et al. (2017a); Ben-Nun et al. (2018). Therefore, they don't require a implementation in the context of this work.

Name	Category	Description
R1	Representation	Extraction of the S-DFG representation from C code
R2	Representation	Extraction of the LLVM-CDFG representation from C code
R3	Representation	Extraction of LLVM-SSA-CDFG representation from C code
R4	Representation	Translation to a numerical representation
R5	Representation	Extraction of statistics
R6	Representation	Generation of an action sequences for the S-DFG
R7	Representation	Generation to a compilable C code from the S-DFG
M1	Model	Support for multiple neural network models
M2	Model	Scalability of the graph models to large-size samples
M3	Model	Support for predictive tasks
M4	Model	Support for generative tasks
E1	Experiment	Support for multiple experiments with multiple steps (Preprocessing, Execution, Evaluation)
E2	Experiment	Execution provisioning in SLURM cluster- and workstation environments
NF1	Non-functional	Extensibility

Table 6.1: Summary of the requirements.

In order to allow for greater flexibility, the data structure of the graph code representations should support interactions and transformations. Examples for interactions are the mapping of the in-memory data structure to a numerical model with node type id encodings or the generation of action sequences (R4), and extraction of statistics of a sample in a certain representation for the later evaluation (R5). Examples for transformations are the elimination of certain node types by merging for the S-DFG representation or incrementally constructing an S-DFG by predicting the next action in the generative model (R6).

Model Starting with the model requirements, the framework needs to support all of the graph neural network models that are subject to this work, specifically the GGNN, and the GCN models (M1). Based on these models, task-specific architectures, specifically for predictive (M3) and generative tasks (M4), should be supported. Furthermore, the graph models require a design that supports source code samples of larger sizes (M2).

Experiment Multiple experiments should be supported, along with its preprocessing, execution, and evaluation steps that should be executed in a convenient way (E1). Automatic provisioning in SLURM cluster- and workstation environments is a mandatory requirement for effective execution (E2). Besides being very demanding in terms of computational resources, dependencies between different experiments complicate their execution. Therefore, an automatic and coordinated way provisioning is essential.

Non-functional For future work, the framework should be extensible in multiple ways (NF1): It should be possible to add new graph models and the framework should be applicable to new tasks without changing the architecture.

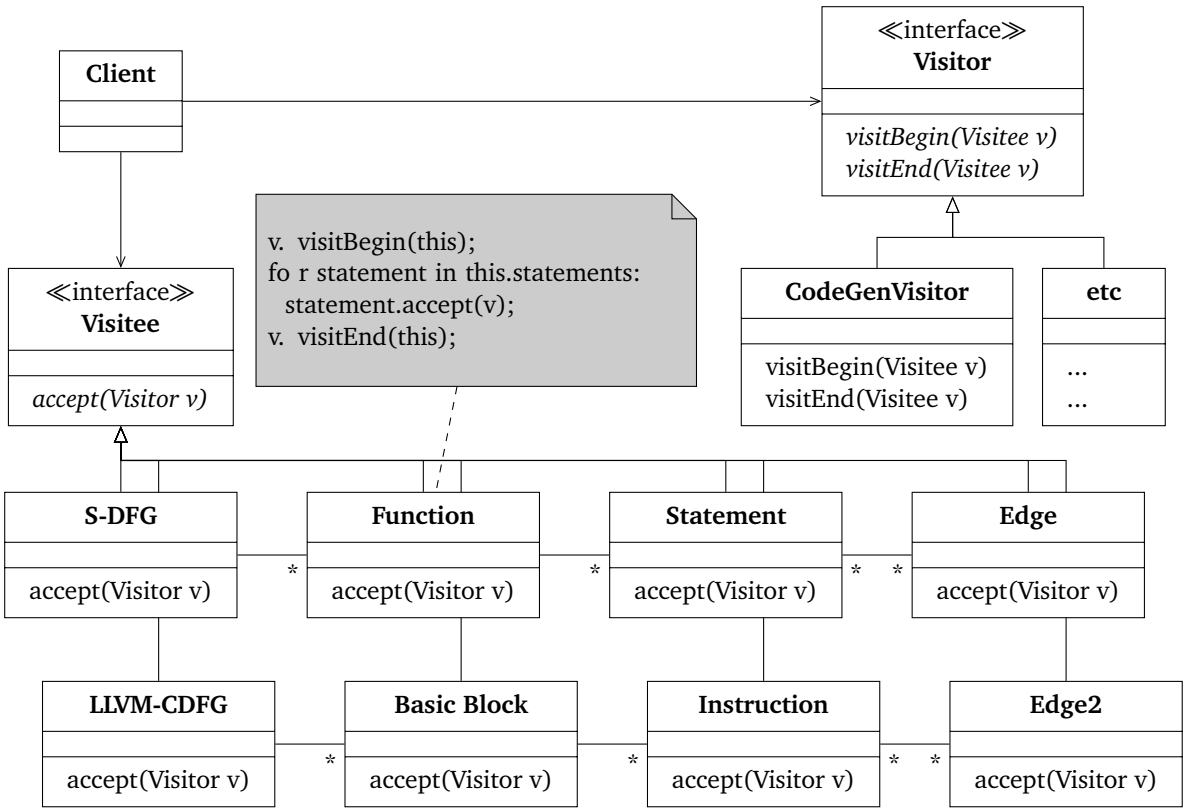


Figure 6.1: UML Class design of the code representation component.[1]

[1] Please note that only the most relevant attributes and functions with visitor- and object recursion functionality are shown here. Low-level implementation details are omitted for more clarity reasons.

6.2 Framework Design

We decide to implement a generic framework that later can be used by the experiments. The framework is naturally divided into two components: The code representation and the model. In the following, we will describe the design features of both of these components.

Code representation In order to comply with requirements R4, R5, R6, and R7 that demand interaction with the graph representation, we design a data structure that represents the code graphs and is easily traversable.

To construct this data structure, we use a pipeline design with components that can be reused across the framework. We start with parsing the source code with Clang and LLVM tools, which capture the information about the Clang AST and LLVM IR. For each source file, the tools produce a stream in JSON format that we capture and parse in our framework into a class hierarchy. This fulfills R1, R2, and R3. Once this data structure is constructed, it is traversed and manipulated by further components.

Figure 6.1 shows the class design of this data structure in UML notation, along with classes that implement concrete algorithms. With the derivates of `Visitee` , each of the representations has its own class hierarchy of entities that are constructed while parsing. After

Representation	Name	Functionality
S-DFG, LLVM-CDFG	MetricsVisitor	Gathering of graph metrics for statistics and evaluation
S-DFG, LLVM-CDFG	DotGraphVisitor	Visualization of the representation
S-DFG, LLVM-CDFG	NodeMappingCreation-Visitor	Creation of node mappings
S-DFG	ActionSequenceCreation-Visitor	Creation of a graph-generating action sequence
S-DFG	NodeEliminationVisitor	Elimination of nodes by merging
S-DFG	CodeGenVisitor	Generation of C code

Table 6.2: Overview of the visitors and their functionality.

construction of the entity model, higher entity objects have associations to corresponding lower entity objects. In the `S-DFG` hierarchy for instance, the `S-DFG` object has associations to the functions that it consists of. Each `Function` object has associations to corresponding `Statement` objects, which have associations to `Edge` objects. Finally, each `Edge` object has an association to a `Statement` object, representing its destination. The derivates of `Visitor` on the other hand implement concrete algorithms that operate on the entity class hierarchies.

The key to the design of the representation component is the use of the *Visitor design pattern* (Gamma (1995)) in conjunction with *Object Recursion*. Algorithms are implemented in accordance with the `Visitor` interface and be applied by passing them to the top-level element of a hierarchy that consists of classes that implement the `Visitee` interface. The `Visitor` object then recursively traverses the hierarchy structure. Every time the object traverses a `Visitee` object, the `visitBegin` method is called. On finishing the traversal, the `visitEnd` method is called. Inside these hook methods, an algorithm can be defined and the concrete `Visitor` object can be used to hold the algorithms' state. Table 6.2 gives an overview of the concrete Visitors that were implemented in this work.

The use of the Visitor design pattern and object recursion has several advantages in our use case: First, we require to implement various tree traversal algorithms and object recursion is a natural way of formulating such algorithms. Second, the Visitor pattern separates the entity class structure from the concrete algorithms. This way, new algorithms can be added without changing the entity classes and therefore, won't break existing functionality. Another advantage is reusability: An algorithm defined as a Visitor can be applied to both representations if implemented in a generic way.

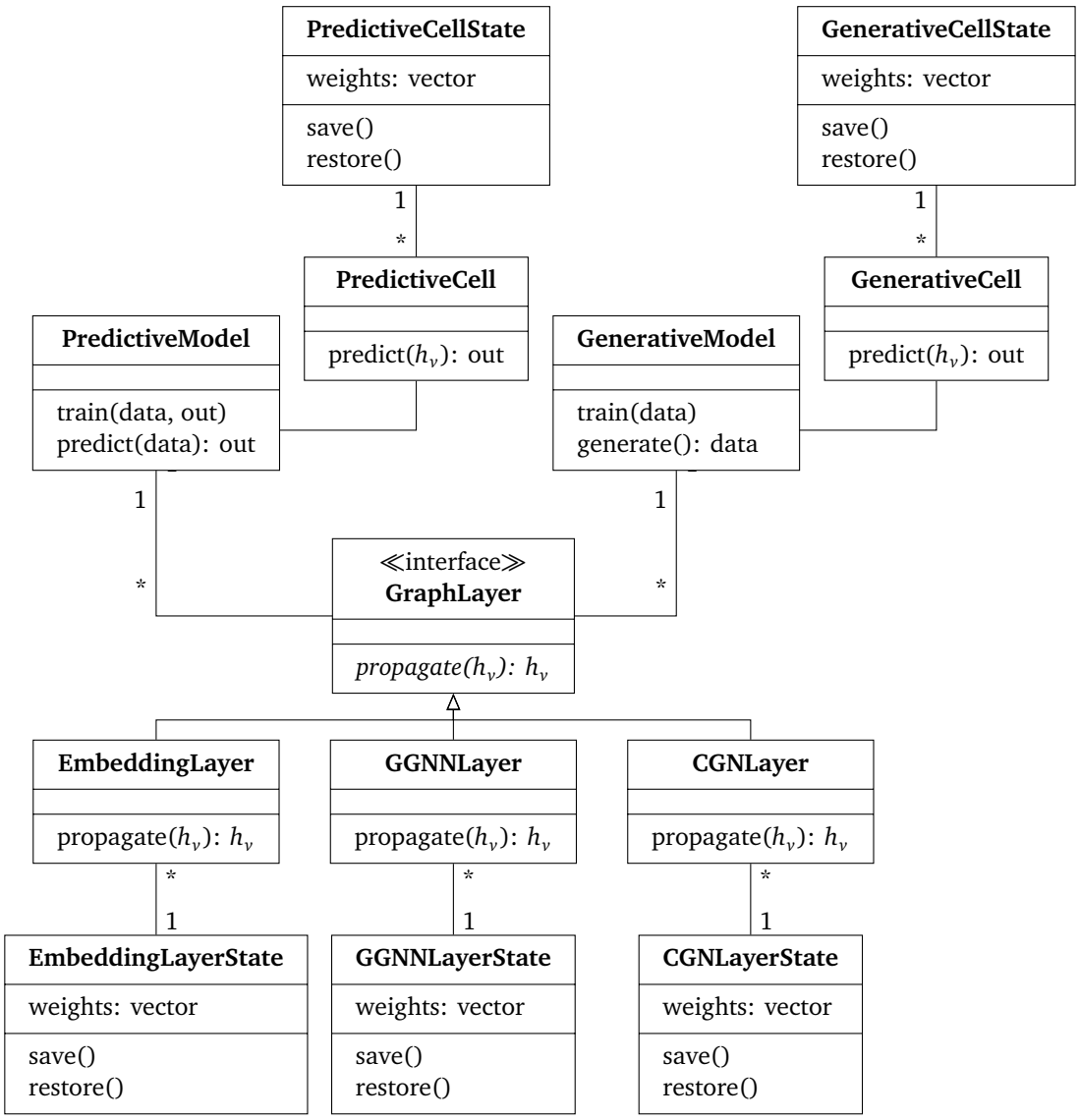


Figure 6.2: UML Class design of the model component.[1]

[1] Please note that only the most relevant attributes and functions of the model organization are shown here. Low-level implementation details are omitted for more clarity reasons.

Model When designing the model, we need to make considerations about software engineering on the one hand (M1, M3, M4), scalability of graph neural network models on the other hand (M2).

Figure 6.2 shows the UML class design of the model component. The most high-level classes are `PredictiveModel` and `GenerativeModel` . They encapsulate the model's functionality and offer a clean interface to the user that allows for training the model with labeled data, making predictions on with a trained model, or for generating new data samples. The model classes are responsible for constructing the model according to a specification that is passed in the constructor. After construction, they hold associated objects of the shown class hierarchy. Furthermore, the classes are partitioned into `GraphLayer` and `Cell` classes. `GraphLayer` derivates represent initial embedding and GNN models described in chapter 4. `Cell` classes represent architectures described in chapter 5. This decoupling is an important design decision towards flexibility and reusability: We can base

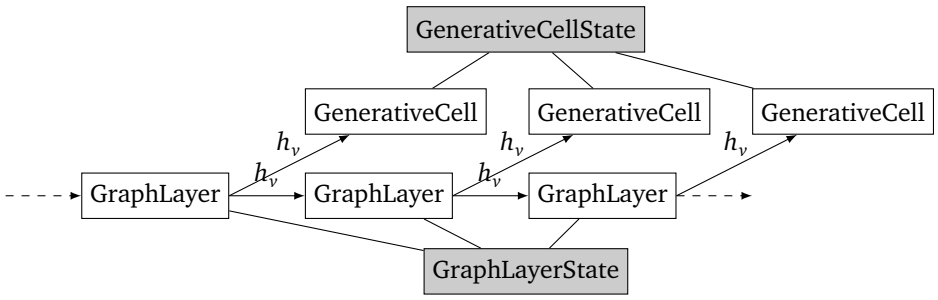


Figure 6.3: Dynamic view of the states in the context of the model component.

the design of multiple architectures on the `GraphLayer` classes that serve as their building blocks. Additionally, this design allows for arbitrary combinations of `GraphLayer` implementations. Another critical design decision is the decoupling of the state from the functional classes. This enables us to build recurrent architectures such as the one defined in the `GenerativeModel` class that requires unrolling the model for a number of timesteps. Figure 6.3 visualizes this relationship. By constructing an unrolled network that is interconnected with `GraphLayer` and `GenerativeCell` objects that share one set of `State` objects, a recurrent model can be trained with BPTT.

Considerations regarding scalability (M2) are mainly towards the representation of the graph structure within the model: Edges in a graph are commonly represented in two ways: First, with an adjacency matrix, whereas a binary setting in the matrix at memory location (x, y) represents an edge. Second, with an adjacency list where each element (x, y) represents an edge.[1] Depending on the density of the graph, either an adjacency matrix or an adjacency list allows for better time and space performance. Because traversing the whole graph to compute a propagation step in the GNN model is a common operation, time and space complexities are highly relevant for the model's applicability and runtime performance. Considering the limits of the time and space complexities[2], the worst-case runtime and space requirement[3] of an adjacency matrix and an adjacency list is $O(V^2)$. The best-case runtimes and space requirements[4] however are $O(V^2)$ for the adjacency matrix and $O(V+E)$ for the adjacency list. Figure 6.4 shows the number of nodes, edges, and the density factor of the device mapping dataset, consisting of 256 OpenCL functions. The density factor is the proportion of the edges over the total possible amount of edges in a graph of a given size. More precisely: $D = \frac{E}{V(V-1)}$. We conclude that a sparse graph model implementation of the S-DFG, LLVM-CDFG, and LLVM-SSA-CDFG representations is more efficient because the S-DFG has a median graph density of 1.65%, the LLVM-CDFG of 1.85%, and the LLVM-SSA-CDFG of 2.17%. Another argument against a dense graph model implementation is that some of the graphs have over 10000 nodes. Constructing a dense model of this size has an impractical memory footprint, even on high-performance hardware.

[1] For simplicity, we consider untyped edges in this description. It can be easily extended to typed edges however by replacing the binary memory in the case of the adjacency matrix, or by considering triples $(x, y,$type$)$ in the adjacency list.
[2] With V being the number of nodes, E the number of edges
[3] Which is a fully-connected graph
[4] Which is a sparse graph

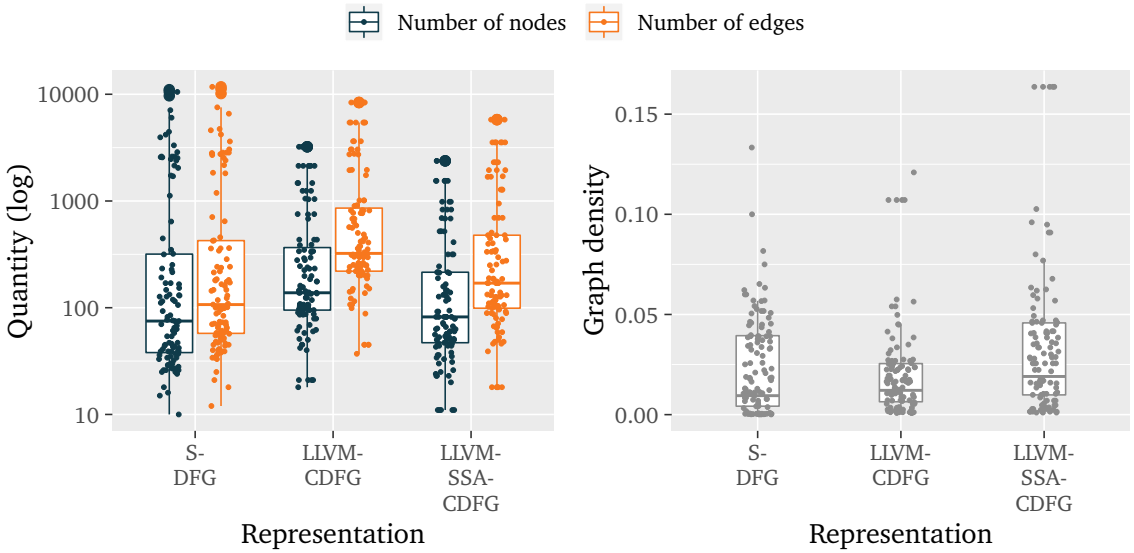


Figure 6.4: Node and edge quantities of the device mapping dataset in different graph representations (left) and the graph densities (right).

Experiments Experiments use the code representation and model components of our framework. Because these components are designed with a generic and non-task specific interface, experiment implementation is straightforward. Multiple experiment steps are supported as different run modes of the experiment executables (E1).

For running the experiments in different environments, we design the experiment component that uses a worker queue design, as shown in Figure 6.5. Users of this component, which are the experiments, can create Task objects that represent concrete experiment executions that are added to the ProcessingQueue. An Executor requests a configurable amount of Tasks from the ProcessingQueue if it has free workers and executes them. On completion of all Tasks, a feedback mechanism reports the results back to the experiment, which then adds new Tasks to the ProcessingQueue. This design stands out in allowing resource sharing, as well as automatic experiment execution. Multiple tasks can be collectively and concurrently executed on a single worker. Automatic experiment execution is especially critical in the case of the hyperparameter tuning experiments because the configurations of the experiments depend on the results of the last execution.

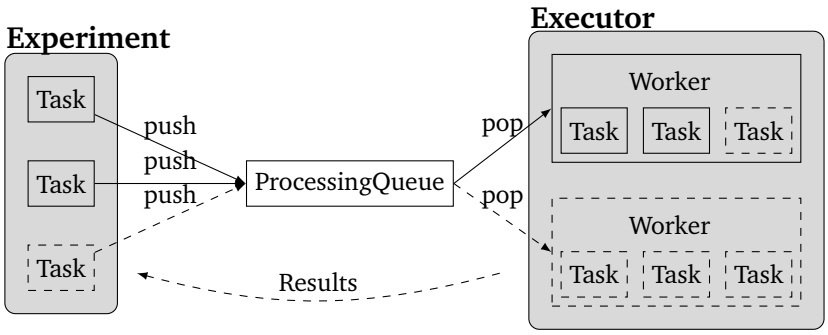


Figure 6.5: Processing queue of the experiment module.

Practically, we have two implementations of Executors, fulfilling requirement E2: A workstation executor for running the experiments locally and a SLURM-cluster executor that enables running the experiments across a larger number of workers.

6.3 Framework Implementation

The implementation of the framework requires at least one programming language supporting the *C++ Application Binary Interface (C++ ABI)* because the Clang/LLVM compiler framework provides native libraries conforming to this. Because the Clang/LLVM provides a maintained CMake module that enables seamless cross-platform builds, we decide to use this programming language to extract the code representations. Additionally, source-level debugging within the Clang/LLVM source code has been used in the development of the extraction tools, which works seamlessly and out-of-the-box within the exclusive *C++* toolchain.

Implementing further functionality of the code representations and the model requires the usage of several algorithms and third-party libraries. The *Python* programming language offers a rich third-party library ecosystem, which is why we deemed it suitable for the task of implementing the required functionality in a productive manner. While the C++ representation extractors only do the mandatory tasks at a minimum level, the majority of the framework is implemented in Python. Both components interact with a data structure that is encoded in the *JSON* format and passed via *IO streams*.

For the model, we decide to implement it using the *Tensorflow* library (Abadi et al. (2016)). Using array operations, we implement a dataflow graph according to the equations described in chapter 4 in a vectorized way, allowing for arbitrary parallel execution to exploit the capabilities of the computation devices.

7

Evaluation

In this chapter, we show an evaluation of the different representations and models of code across different tasks. For evaluating the graph-based representations and models, we use the framework described in chapter 6. We will start by giving a definition of common performance metrics that are shared across the experiments. Then, we will describe two predictive experiments and one generative experiment, while giving more details on the metrics, experimental setup, and a discussion of the results.

7.1 Performance Metrics

Accuracy The *accuracy* in the context of classification is a statistical measure for a model's performance. In the case of binary classification, it is defined as the proportion of the correctly predicted samples (TP) over the sum of the TP and the wrongly predicted samples (FP), whereas a higher accuracy value is better:

$$\text{Accuracy} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{7.1.1}$$

Speedup The *speedup* is a measure for a method's performance in the domain of computation. Considering the same experiment and the objective is to reduce latency, it defined as the proportion of the old runtime t_{old} over the new runtime t_{new}, whereas a higher speedup value is better:

$$\text{Speedup} = \frac{t_{\text{old}}}{t_{\text{new}}} \tag{7.1.2}$$

Kullback Leibler divergence (KL) The *KL* is a measure of similarity between two discrete probability distributions P and Q, whereas a lower KL value means more similarity between the distributions. P is the expectation probability distribution, Q the approximation probability distribution:

$$D_{\text{KL}}(P\|Q) = \sum_{x \in X} P(x)\ log(\frac{P(x)}{Q(x)}) \tag{7.1.3}$$

Cross validation Essential to a predictive model is the ability to generalize to new unseen data.

Therefore, the dataset is commonly split into two disjunct sets: The *training set* and *test set*. The model is fit to the training set and evaluated on the test set, which has been held-out during training. A typical train/test split proportion is 80/20 or 90/10.

Using a typical training and test split on small data sets has a disadvantage: Because the test set samples are few, high variability can occur, which can be reduced by using a *k-fold cross validation* scheme. The dataset is split into k parts, which are used to construct many pairs of disjunct training and test sets. The training sets consist of $k-1$ parts, the test sets of k parts. For each of the $k-1,1$ partitions, the model is retrained on the training set and evaluated on the test set. Upon completion of this iterative scheme, the results of the splits are aggregated using e.g. the *arithmetic mean* or *geometric mean* function.

7.2 Heterogeneous Device Mapping Task

The problem of heterogeneous device mapping has the goal of classifying OpenCL kernel functions to the computation device where they run faster. This problem has been studied extensively, and several approaches have been proposed to solve it. Initially, Grewe et al. (2013) proposed a heuristic method based on *manually-defined features and a decision tree* as decision model. Cummins et al. (2017a) used a deep learning approach based on an *RNN model and an S-TS representation*, which resulted in major performance improvements. More recently, Ben-Nun et al. (2018) used an *RNN model based on an LLVM-TS representation* in combination with word2vec embeddings (Mikolov et al. (2013)) to improve upon this further. Throughout the evaluation, we will use these sequential-based methods and refer to them as S-TS, and LLVM-TS respectively.

7.2.1 Metrics

For evaluating the representations and models, we want to analyze their generalization performance and inference times.

Generalization Performance For measuring generalization performance, we use the *accuracy* as a metric, which in this context is the ratio of the correctly predicted device mappings over all predicted device mappings. As an additional metric for the performance benefit that the methods bring in this specific compiler task, we use the *speedup*, which is the sample's runtime on the predicted compute device over the runtime of a sample on a statically chosen device. Because a correctly predicted mapping of a sample yields a faster execution, it results in a speedup. In the static mapping, a single platform (CPU or GPU) is selected and all kernels are mapped to this platform. For selecting this static mapping, the platform which is fastest in most of the samples of the whole dataset is chosen.

Type	Vendor	Model	Frequency
CPU	Intel	i7-7700k	4.2GHz
GPU	NVIDIA	GTX 1080 Ti	1.5GHz
Mem	GSKILL	Ripjaws V	3.2GHz

(a) Hardware specification.

Type	Software	Version
Operating System	Ubuntu	16.04
Application	Python	3.6
Application	Tensorflow	1.12

(b) Software specification.

Table 7.1: Measurement environment for inference times.

Inference Time Additionally, we compare the models in their inference time, which is the time required to make a prediction. Low inference time is a rather practical goal but nevertheless very important towards applicability in compiler-related tasks, as it translates to a faster compilation time for an end-user. A low compile time is desirable in software engineering because it increases the development process efficiency. For taking the measurements, we first construct the models on the system described in table 7.1, then measure the time to infer on 1 sample of the dataset.

7.2.2 Experimental Setup

Dataset The dataset consists of a total of 256 OpenCL kernel functions of the seven benchmark suites AMD SDK, NVIDIA SDK, NPB (Seo et al. (2011)), Parboil (Stratton et al. (2012)), Polybench (Jia et al. (2014)), Rodinia (Che et al. (2009)), and SHOC (Danalis et al. (2010)), along with the execution times for both CPU and GPU on two different heterogeneous systems. The output of the classification problem is obtained by selecting the device that yields the minimum execution time. Both of the heterogeneous systems on which the execution times have been measured have an Intel Core i7-3820 CPU. In terms of GPU, one system features an AMD Tahi 7970, the other system an NVIDIA GTX 970.

While the S-TS and LLVM-TS representations of the dataset already exist, we construct the graph-based representations for the whole dataset. This results in graphs with 92 node types for the S-DFG graphs and 140 node types for the LLVM-CDFG graphs.

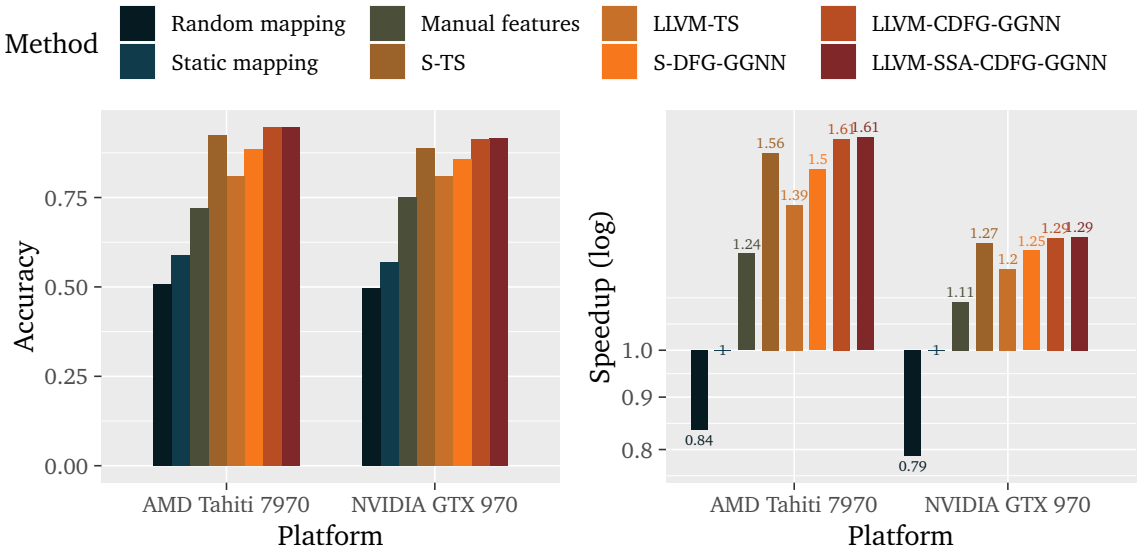
Hyperparameters To compare the graph model with the S-TS and LLVM-TS representations, we construct a model, whereas we manually choose the hyperparameters.[1] To compute the initial node embeddings, we map the one-hot encoded vectors with a MLP f_{init} consisting of 2 hidden layers of size 64 each to vectors of size 32. As propagation size T, we choose 4 iterations, which yields embeddings that include a 4-neighborhood of the individual nodes. The graph embedding vector h_G of size 64 is created by aggregating the node embeddings using the two MLPs f_m and g_m, which are dimensioned with 2 hidden layers of size 64 each. The graph embedding is mapped with a MLP with two hidden layers of size 32 to a dimension of 32, then serves as input to the prediction model, which is a MLP with 1 hidden layer of size 32.

Training We train the GNN models for 1500 epochs on the S-DFG, the LLVM-CDFG, and the LLVM-SSA-CDFG graphs with the objective to fit the training set using the stochastic gradient descent algorithm. Additionally, we train the S-TS model of Cummins et al. (2017a), the LLVM-TS model of Ben-Nun et al. (2018), as well as the decision-tree-based model of Grewe et al. (2013). We compare the results to the static mapping model, and to a model choosing the CPU/GPU mapping at random.

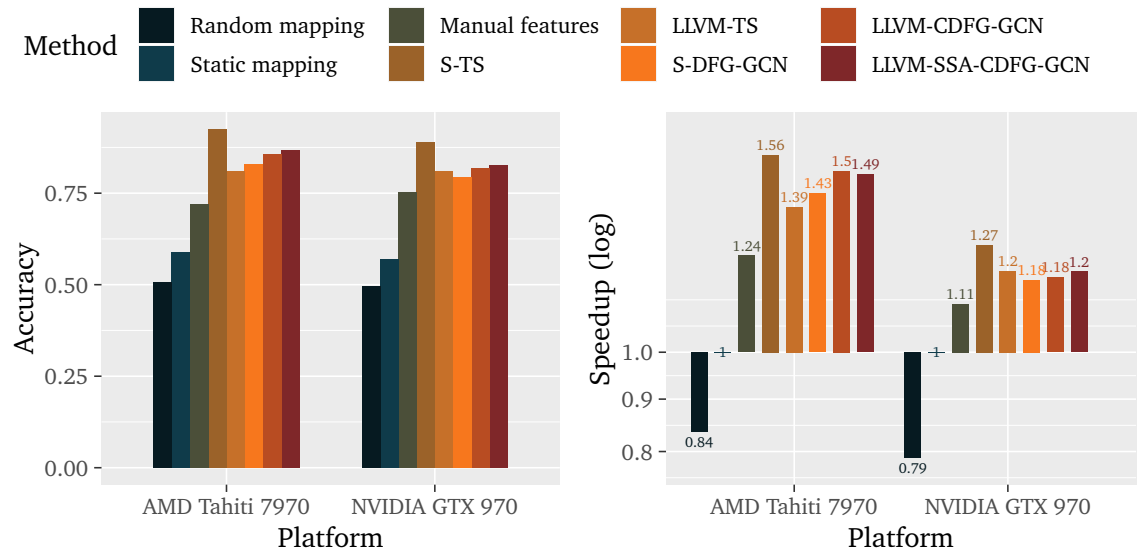
In all methods, we use the same training-test-data split as in Cummins et al. (2017a) and Ben-Nun et al. (2018) in a k-fold cross-validation scheme in a first experiment. In an additional experiment, we split according to the benchmark suites, as described in the next chapter.

[1] An automatic hyperparameter search based on *Bayesian Optimization* is reported in Appendix B.

7.2.3 Results



(a) GGNN model.

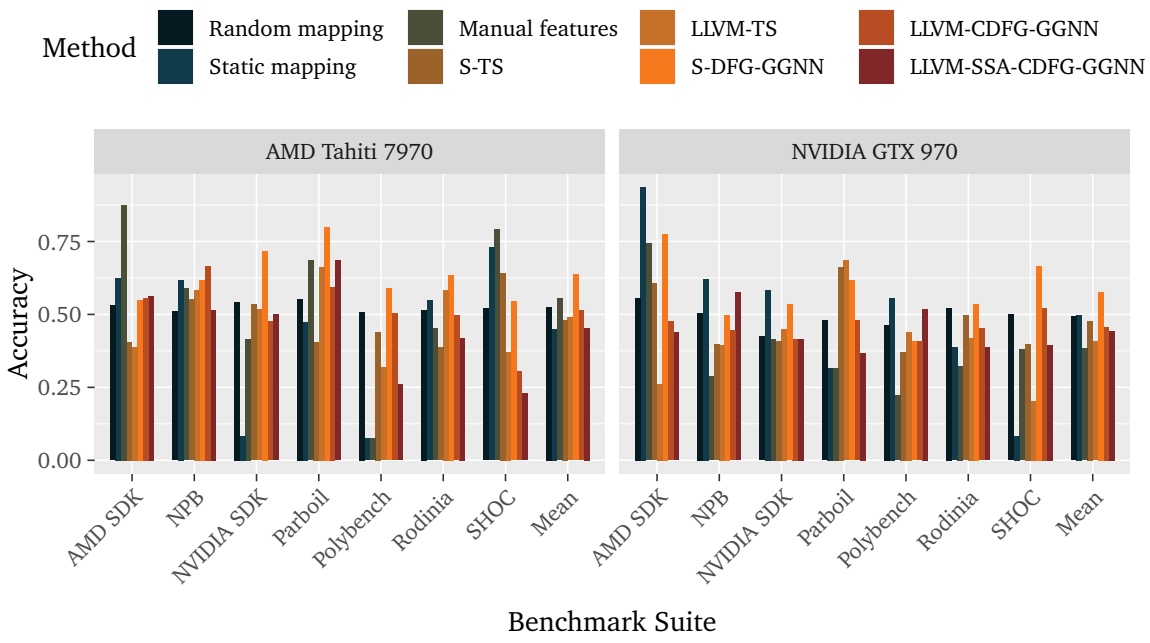


(b) GCN model.

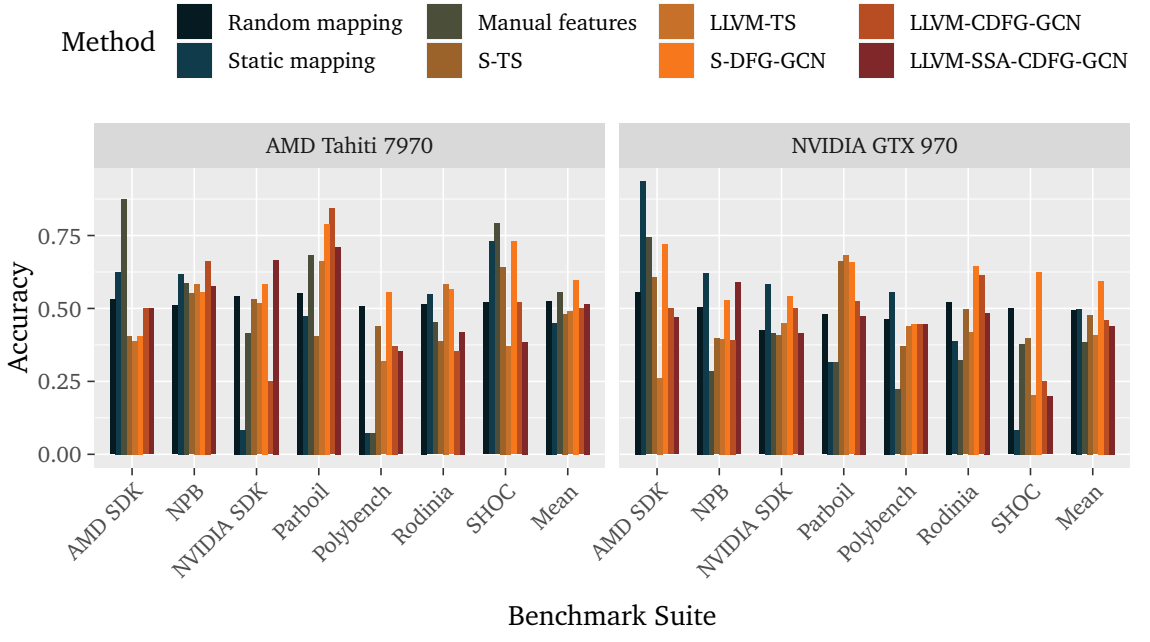
Figure 7.1: Accuracies and Speedups of the heuristic methods in the device mapping task with random k-fold splits.

Figure 7.1 shows the overall accuracy results and speedups of the heuristic methods in the device mapping task. We see that the graph-based methods (S-DFG, LLVM-CDFG, and LLVM-SSA-CDFG in combination with the GGNN and GCN models) perform better than LLVM-TS. While the S-TS method produces a slightly better accuracy than S-DFG, LLVM-CDFG and LLVM-SSA-CDFG yield the highest overall accuracy when used with the GGNN model. The GCN models on the other hand, are not able to cope with the S-TS method. Further, we see that this described trend is comparable to the speedup results.

While useful for comparison with the state-of-the-art methods used in the original work of the S-TS and LLVM-TS methods, their cross-validation setup trains the model with kernels



(a) GGNN model.



(b) GCN model.

Figure 7.2: Accuracies and Speedups of the heuristic methods in the device mapping task with grouped k-fold splits.

from the same benchmarks that it then uses to evaluate the heuristic. Having 7 different benchmark suites, we believe that using these as groups in a k-groups-split methodology yields more insights about the generalization capabilities of the models. This way, they are tested on kernels from a benchmark suite they have not been trained on. To this end, we split the dataset into 7 parts, each of the 7 parts being the different benchmarks suites, instead of 10 randomly-chosen parts out of the set of all kernels of the benchmark suites. This way, the

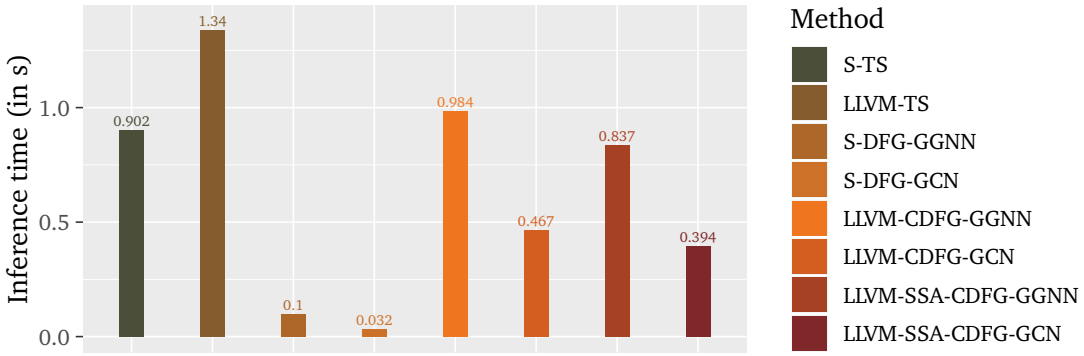


Figure 7.3: Inference times of the heuristic methods in the device mapping task.

model is tested on data that is fundamentally different from the benchmarks it was trained on. Figure 7.2 shows the results of the experiment with this alternative setup. Since we split the train and test sets by benchmarks, we can see how the models fare on every benchmark after being trained on the other six. It is notable how the different methods can have vastly different results on the different benchmarks. The final entries for each platform show an aggregated result (arithmetic mean) over all benchmarks.

In the results, we can see that S-DFG is not only the one with the best overall results but also the most consistent ones. It had an overall accuracy of 60.6% when combined with the GGNN model, which is over 12 percentage points better than that of the S-TS model at 47.9% and even better than the 44.9% overall accuracy obtained by the LLVM-TS model. In this case, LLVM-CDFG had a similar performance, beating the S-TS method by less than a percentage point in accuracy at 48.5%. Similar results can be observed for the combinations with the GCN model. In fact, we see how when tested on different benchmark suites than they were trained on, all state-of-the-art methods we compared to here perform worse than a coin-toss (50.9% accuracy). This indicates that the models probably do not learn the relationship between the code's semantics and the optimal compute device in a way that is generalizable when the code becomes different enough.

Figure 7.3 shows the inference times of the heuristic methods. We can see that the manually-defined features method is considerably faster in training and inference. The deep learning models are much slower in inference than the decision tree based on manually-defined features. It is important to note however, that the graph-based models are up to an order of magnitude faster in inference than the sequence-based deep learning-based models.

7.3 Thread Coarsening Task

In parallel architectures, faster execution times can sometimes be achieved by merging multiple parallel threads to fewer threads. The thread coarsening factor is a parameter that controls this behavior in OpenCL implementations. Various predictive models have been proposed to solve this task, such as that by Magni et al. (2014), who used an *MLP based on static code features*, such as the quantities of certain LLVM IR instructions. Similarly to the heterogeneous device mapping task, Cummins et al. (2017a) and Ben-Nun et al. (2018) used deep learning approaches based on a *RNN models and S-TS and LLVM-TS* representations to improve upon this.

7.3.1 Metrics

For evaluating the representations and models, we want to asses them in terms of generalization performance and inference times.

Generalization Performance For measuring the generalization performance in this task, we use the *speedup*, which is the ratio of the runtime of a kernel with a correctly predicted thread coarsening factor over the runtime of a kernel with no thread coarsening applied.

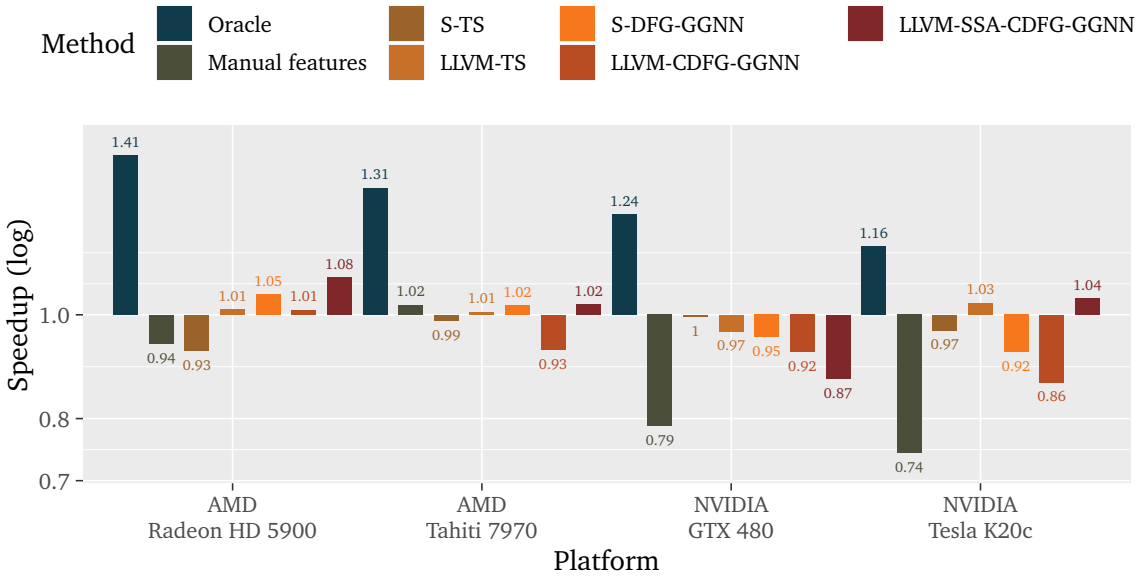
Inference Time Again, we compare the different models in their inference time because the resulting fast compilation time when integrating such a heuristic model into a compiler method is a desirable property. For this, we use the measurement environment described in table 7.1. After constructing the models, we measure the time to infer on 1 sample in the dataset.

7.3.2 Experimental Setup

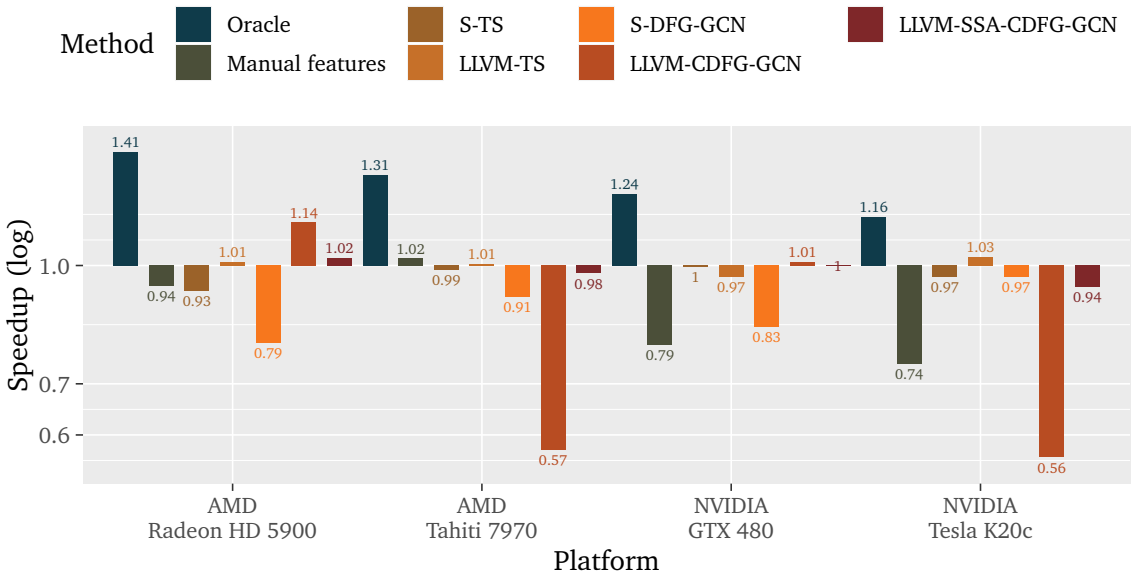
Dataset The dataset for this problem consists of a total of 17 selected kernels from the AMD SDK, NVIDIA SDK, and Parboil (Stratton et al. (2012)) benchmark suites. As output, the model predicts among 6 classes for every kernel, corresponding to the coarsening factors of 1, 2, 4, 8, 16, 32. Furthermore, the dataset consists of runtime measurements for each of the kernels for each thread coarsening factor on 4 different GPUs.

Hyperparameters For this dataset, we obtain a total of 46 distinct node types for the S-DFG and 54 node types for LLVM-CDFG graph representations. In this task, we keep the dimensions of the model at a minimum, as the amount of training data is quite slim. We apply one-hot encoding to the nodes represented as node types and map the resulting vectors with the MLP f_{init} to a size of 4. This MLP only contains the input and output layers and no hidden layers. After 4 propagation time steps T, we aggregate the node embedding vectors

to a graph embedding vector h_G of size 8 by using the two MLPs f_m and f_g with no hidden layers.



(a) GGNN model.



(b) GCN model.

Figure 7.4: Speedups of the heuristic methods in the thread coarsening task.

Training We train the GNN models for 1500 epochs on the dataset on the S-DFG, the LLVM-CDFG, and the LLVM-SSA-CDFG graphs with the objective to fit the training set best using the stochastic gradient descent algorithm. Additionally, we train the S-TS model of Cummins et al. (2017a), the LLVM-TS model of Ben-Nun et al. (2018), as well as the MLP-based model of Magni et al. (2014).

In all methods, we use the same training-test-data split as in Cummins et al. (2017a) and

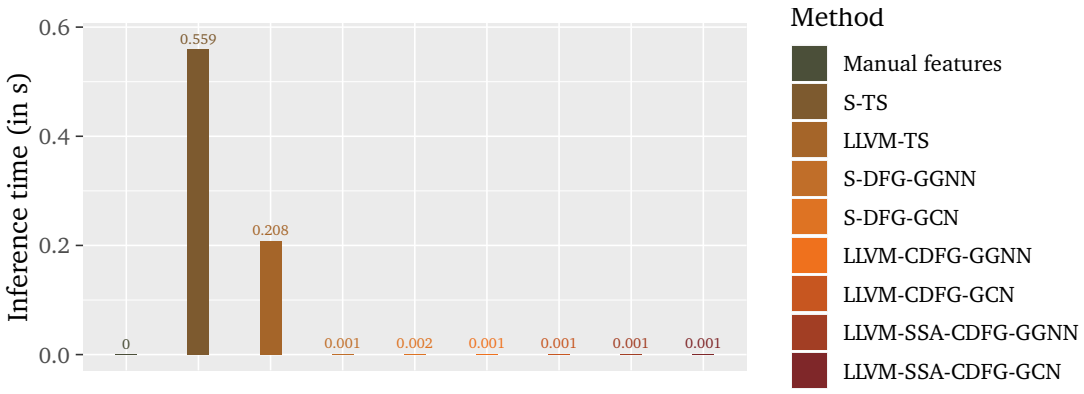


Figure 7.5: Inference times of the heuristic methods in the thread coarsening task.

Ben-Nun et al. (2018) in a k-fold cross validation scheme.

7.3.3 Results

Figure 7.4 shows the results of the thread coarsening experiment. In the figure, we add an “Oracle” value for reference, which depicts the best possible speedups. It becomes obvious that in many cases, the predicted thread-coarsening factors yield an overall slowdown. Overall, the S-TS model yields an overall speedup (geometric mean) of around 0.97 across all platforms. In combination with the GGNN model, the S-DFG is slightly better at 0.98, and LLVM-CDFG is slightly worse at 0.93. With the GCN model, the S-DFG achieves a speedup of 0.87, the LLVM-CDFG 0.78, and the LLVM-SSA-CDFG 0.98, which is mostly worse than the combinations with the GGNN model. The MLP relying on the manually-defined features of Magni et al. (2014) yields an overall speedup of 0.87. The best results in this task are achieved by the LLVM-TS and the LLVM-SSA-CDFG-GGNN models, which have an overall speedup of 1.00, i.e. just as good as doing nothing. In general, all predictive fare comparably bad at this task. This might be explained in part by the modest possible maximal speedups, which are limited to the oracles that are 1.28 overall. Also, the sparsity of training data contributes to the poor performances of the methods.

Figure 7.5 shows the inference times of the heuristic methods. Similarly to the heterogeneous device mapping task, we can see that the MLP based on the manually-defined features method is considerably faster in training and inference. While the other models relying on more complex deep learning architectures are generally slower in inference than the MLP, it is important to note that the graph-based model is an order of magnitude faster in inference than the sequence-based deep learning models. The faster inference times of the GNNs over the RNNs can be explained by the different natures of computation: While the dataflow in the RNN model is sequentially dependent on the results of the previous computation step, the GNN models perform computations with very little dependencies. Therefore, the computation of the GNN model can be massively parallelized, which is in contrast to the RNN models where the computation is strongly sequential.

7.4 Code Generation Task

Generative models of code can be used in two ways: First, to tackle the problem of dataset sparsity. As statistical methods, such as the thread coarsening factor prediction we have evaluated in the previous paragraph, may suffer from enough training data, generative models of source code can learn to re-generate the distribution of source code they have been trained with and beyond - to generate new samples that have similar properties as the training samples. However, current deep learning-based methods are biased towards shorter and little complex samples, as shown by Goens et al. (2019). A second use case of generative models of code could be to tackle one of the major problems of compiler research: The lack of representative benchmarks. As pointed out by Hall et al. (2009), many widely used programs are proprietary, rendering them unavailable to the compiler research community that requires them to design and experimentally validate new methods. Generative models could bridge this gap by learning the distribution of closed-source programs in a closed, e.g. company-internal environment. Sharing the trained models with compiler researchers enables them to generate programs with properties that match the distribution of the properties of the closed-source programs, while the actual programs remain secret.

Considering these two use cases, they share a similar objective: Re-generating the distribution of the training data. For the use case of dataset augmentation, re-modeling the original distribution is a needed objective before considering to generate new data samples. For the use case of generating benchmarks for compiler research, re-generating the distribution is the main goal.

Throughout this experiment, we will evaluate two generative models: One based on an S-TS representation, one on an S-DFG representation. A generative method based on an S-TS representation in conjunction with an RNN deep learning model has been proposed by Cummins et al. (2017b), which we will re-use and re-evaluate in our experiments. The S-DFG representation, on the other hand, is implemented in our framework.

7.4.1 Metrics

For evaluating the code-generative models towards the objective of re-generating the distribution of the training data, we use the *KL between the probability distributions of different code properties* that we extract from the training data and the generated samples. In this metric, the expectation probability distribution P represents the probability distribution of the properties of the training samples, the approximation probability distribution Q the probability distribution of the generated samples.

While the code properties are a useful metric for evaluating the main objective, we evaluate on an additional metric for the use case of dataset augmentation: We extract the code features of the compiler heuristic of Grewe et al. (2013) and perform a *principal component analysis* on them, in order to analyze the KL between the main principal components of the training set and the generated sets. This will indicate whether the generative models could be useful in this specific task.

7.4.2 Experimental Setup

Dataset The dataset for this task consists of OpenCL kernel functions that have automatically been obtained from popular open-source projects from the GitHub software development collaboration platform, following the methodology of Cummins et al. (2017b).

For this experiment however, we limit the total dataset consisting of 6800 compilable OpenCL kernels to a subset. This is because the development of a code generator for generating valid C code is a major engineering effort and therefore, we chose to support only the major features of the C programming language. Another reason for limiting the data set to smaller-sized samples is that training of the models requires a large amount of time to converge. The S-TS model for example has been trained for several weeks, as the authors reported in the original work. By limiting the dataset to smaller-sized samples, we can considerably reduce the experiment execution time, while allowing a first comparison of the generative models.

After preprocessing the reduced dataset, which consists of 1423 samples, we obtain a normalized S-TS and a S-DFG representation. The S-DFG representation has a total of 56 node types. The distributions of different code properties of the training dataset can be seen in Figure 7.8. After filtering the original training set with the criteria about the size and the code generator support described above, the training set consists of kernel functions with a maximum AST depth of 13 and AST node quantity of 87, which translate to a maximum amount of 336 sequence-generating and 490 graph-generating actions.

Hyperparameters We dimension the S-DFG model to be of a similar size as the S-TS model: After applying one-hot encoding to the nodes according to their types, we map it to embeddings h_v of size 256 with a MLP f_{init} with no hidden layers. Based on all node embeddings h_v that are propagated for 4 timesteps T, we compute h_G with the MLP f_m and g_m that have one hidden layer each. Based on h_G, we compute the probability distributions of the next action with the MLPs f_{addnode}, f_{addedge}, and f_{nodes} that have one hidden layer each.

Training and Sampling Both, the S-TS and the S-DFG models are trained with the same data set until convergence of the loss function described in chapter 4 using the stochastic gradient descent algorithm.

After training the models, we use them to produce new samples, starting with an empty initial state, which is an empty sequence[2] and an empty graph. We generate a total amount of 1423 valid samples with each model in each configuration we report to match the number of samples of the training set for the later result evaluation. The criterion for a sample being considered as valid is passing the Clang/LLVM compiler pipeline without any errors and obtaining executable code. Please note that this criterion only includes compilability, not executability.

Additionally, we control the sampling process by varying the randomness by introducing a

[2]Please note that this is different from the original S-TS work, which uses a function header with a return type, 4 arguments, and parentheses as initial set in the sampling process.

temperature parameter t, which we implement in both models by scaling the logits, which are the inputs to the final softmax and sigmoid functions that produce the probability distributions over the actions by a function f. This methodology was originally introduced by Hinton et al. (2015).

$$f(a_i) = \frac{e^{a_i/t}}{\sum_{j=1}^{n} e^{a_j/t}} \tag{7.4.1}$$

The effect of the temperature parameter t is as follows: A smaller value of t makes the sampling process more confident, but also conservative because the probability distribution is sharper. This causes the class with the largest logit to be more likely to be chosen, which makes the generative model produce more valid, but fewer novel samples. A smaller value of t on the other hand causes a smoother probability distribution, which makes the sampling process produce more novel, but also less valid samples.

Tuning the t parameter unveils the performance potential of the methods. We produce samples in the configurations $t = 0.6, 0.8, 1.0, 1.2, 1.4, 2.0$.[3] and optimize for the minimal KL.

[3]The temperature configuration of $t = 1$ yields the same results as without the temperature sampling.

7.4.3 Results

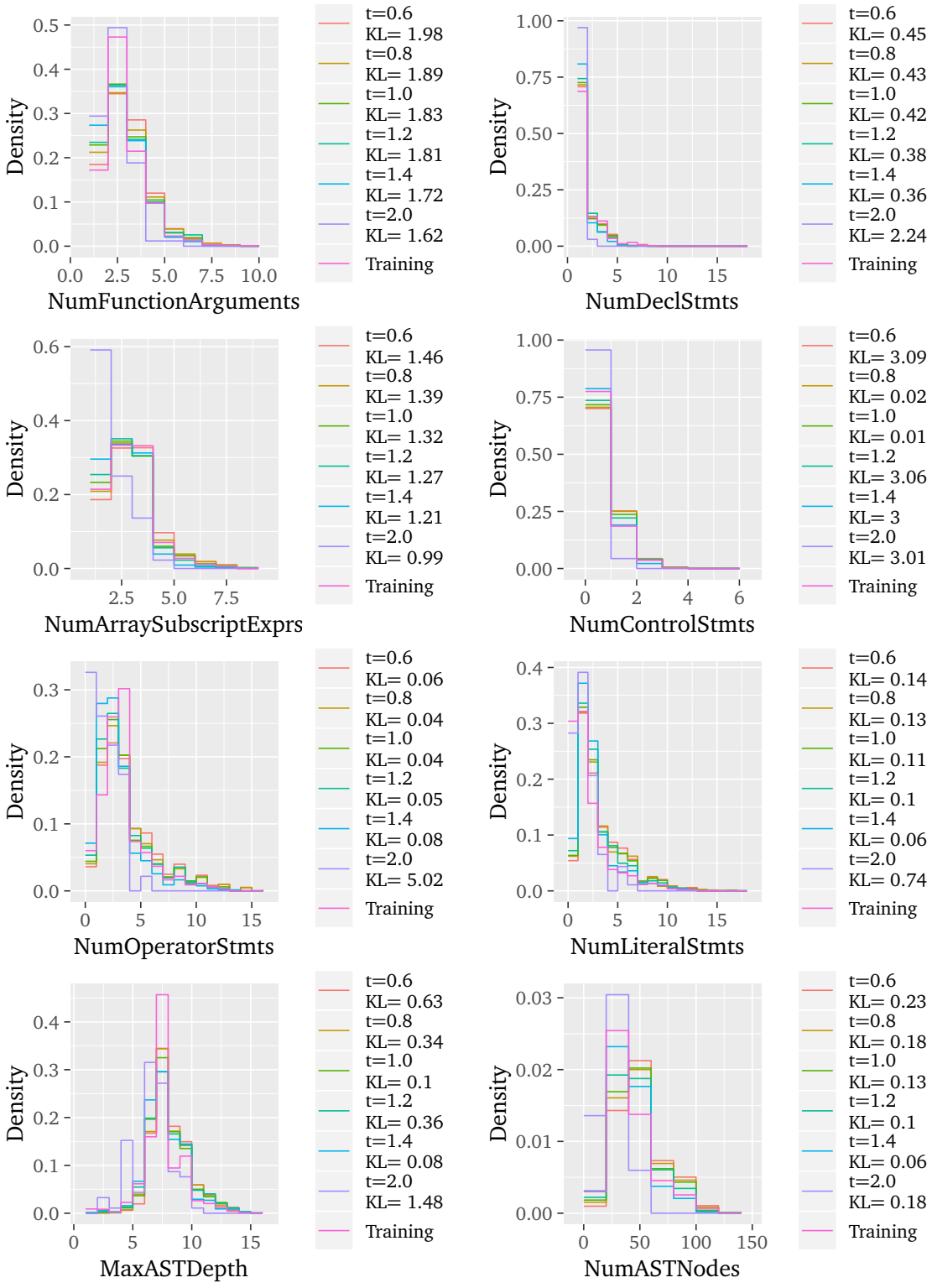


Figure 7.6: Distribution of code properties of S-TS generated samples with their KL-Divergence to the training set.

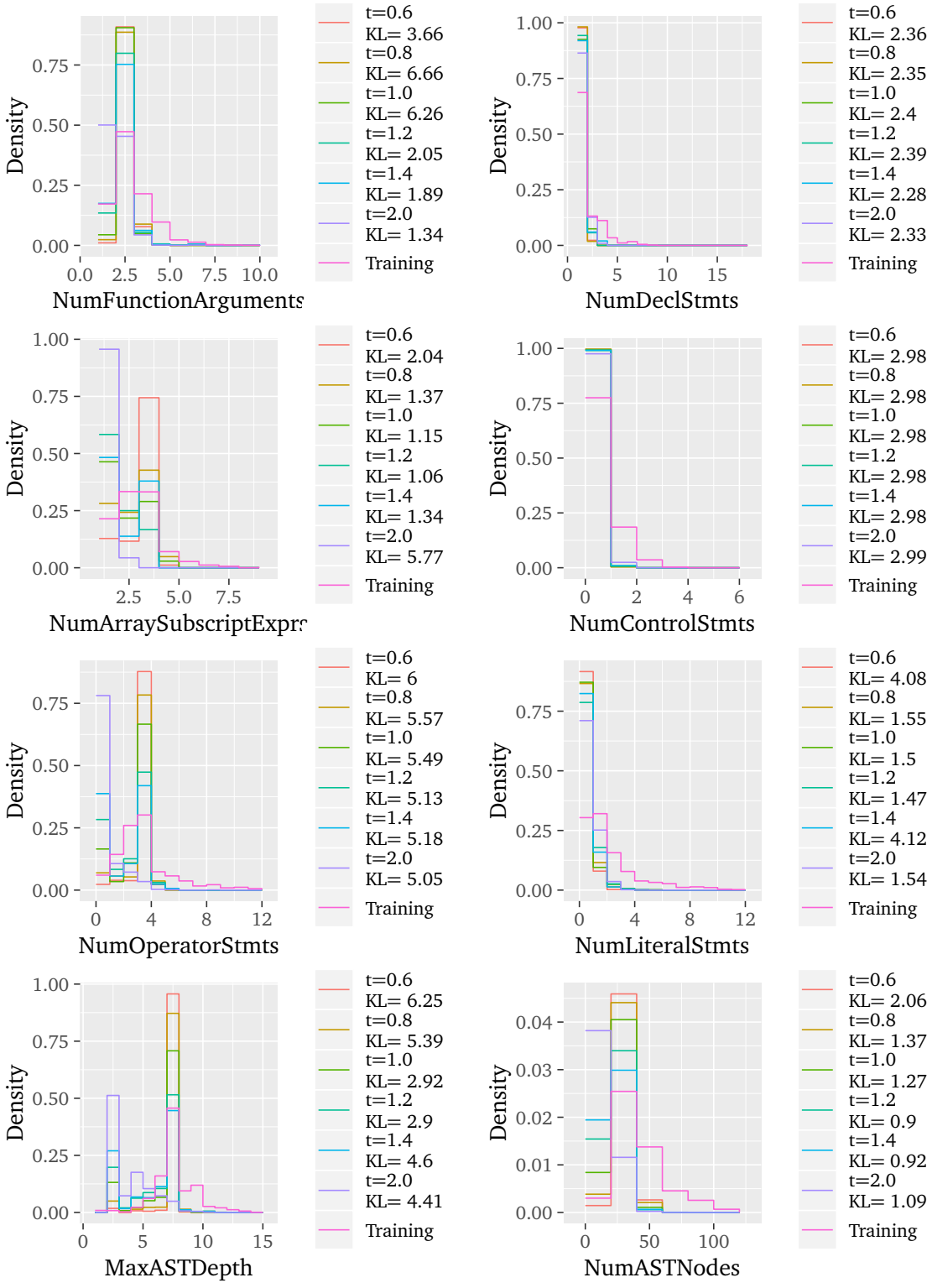


Figure 7.7: Distribution of code properties of S-DFG generated samples with their KL-Divergence to the training set.

Method	Temperature	NumFunction Arguments (KL)	NumDecl Stmts (KL)	NumArraySubscript Exprs (KL)	NumControl Stmts (KL)	NumOperator Stmts (KL)	NumLiteral Stmts (KL)	MaxAST Depth (KL)	NumAST Nodes (KL)	Validity (percentage, max. 1)
S-TS	0.6	1.98	0.45	1.46	3.09	0.06	0.14	0.63	0.23	**0.38**
S-TS	0.8	1.89	0.43	1.39	0.02	0.04	0.13	0.34	0.18	0.37
S-TS	1.0	1.83	0.42	1.32	**0.01**	**0.04**	0.11	0.10	0.13	0.32
S-TS	1.2	1.81	0.38	1.27	3.06	0.05	0.10	0.36	0.10	0.25
S-TS	1.4	**1.72**	**0.36**	1.21	3.00	0.08	**0.06**	**0.08**	**0.06**	0.16
S-TS	2.0	1.62	2.24	**0.99**	3.01	5.02	0.74	1.48	0.18	0.01
S-DFG	0.6	3.66	2.36	2.04	2.98	6.00	4.08	6.25	2.06	**0.88**
S-DFG	0.8	6.66	2.35	1.37	2.98	5.57	1.55	5.39	1.37	0.71
S-DFG	1.0	6.26	2.40	1.15	2.98	5.49	1.50	2.92	1.27	0.56
S-DFG	1.2	2.05	2.39	**1.06**	**2.98**	5.14	**1.47**	**2.90**	**0.90**	0.45
S-DFG	1.4	1.89	**2.28**	1.34	2.98	5.18	4.12	4.60	0.92	0.31
S-DFG	2.0	**1.34**	2.33	5.77	2.99	**5.05**	1.54	4.41	1.09	0.16

Table 7.2: Summary of the sampling results of the generative model regarding KL and Validity.

Figure 7.6 shows the distribution of the code properties across different temperature configurations on the S-TS method, Figure 7.7 on the S-DFG method respectively. The code properties that we analyze are quantities extracted from the original, unmodified Clang AST. Specifically, the number of function arguments (NumFunctionArguments), the number of declaration statements (NumDeclStmts), the number of array accesses (NumSubscriptExprs), the number of control flow statements such as for and while loops and conditionals statements (NumControlStmts), the number of operator statements which include binary and unary operators, the number of literals such as floats and integers. Additionally, we measure the maximum depth of the AST (MaxASTDepth), and the overall quantity of AST nodes (NumASTNodes).

Table 7.2 summarizes the KLs of the code properties, along with the validity in different temperature configurations. A first important observation is that the S-DFG method has a higher overall validity than the S-TS method in all configurations of t. By lowering the t parameter, the validity increases, as well as the KL values for most of the code properties, which means that the probability distributions increasingly diverge. By raising the t parameter, the validity decreases. However, this results in a lower KL until to a certain configuration of t, which means that the similarity between the distributions increases. This minimum point is as follows: For the S-TS method, a temperature configuration of $t = 1.4$ produces the minimum KL for the majority of code properties. For the S-DFG method this configuration is $t = 1.2$ respectively. We consider these parameters as optimal within each of the methods regarding the objective to re-generate the code distribution of the training data. As a first conclusion, we can see that tuning the temperature positively influences the model's abilities to reproduce the input distribution.

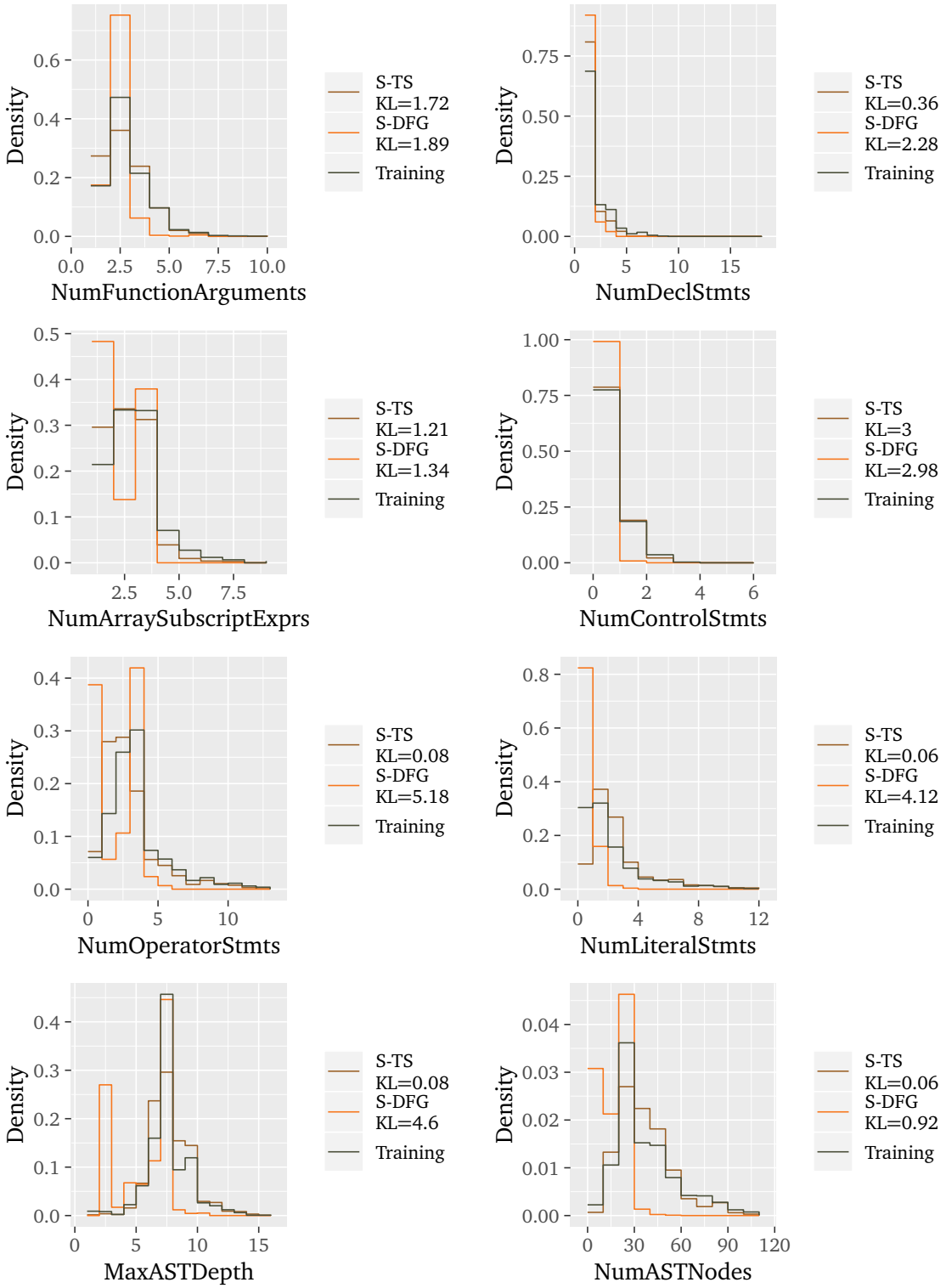


Figure 7.8: Distribution of code properties of samples of the S-TS and S-DFG data sets with their KL-Divergence to the training set.

Having the tuned methods, we will now compare the results between them. Figure 7.8 shows the S-TS method in configuration $t = 1.4$ and the S-DFG method in configuration $t = 1.2$, along with their KL on the code properties. We see that the S-TS method has a lower KL in 7 out of 8 code properties, which means that it more closely reproduces

the probability distribution of the training set. Notable is that the S-DFG method fails to produce samples with higher values in almost all of the code properties. Because most code properties are quantities, we can assume that the S-DFG fails to produce larger-sized samples. This assumption is also supported by the trend of the NumASTNodes property, which is a measure for the size of a sample.

The additional comparison of the methods regarding the manually-defined features defined by Grewe et al. (2013) supports this trend. Figure 7.9 shows the two main principal components of the tuned methods. While the S-TS method is able to generate samples that cover several clusters of the original feature space in terms of these components and beyond, the S-DFG method fails to do so and generates samples that mainly cover one cluster of it. However, the S-TS method also fails to reproduce some of the training sets samples. The probability density function representation along with an analysis of the KL in Figure 7.10 supports this numerically: The first principal component (PC1) of the samples generated with the S-TS method have a considerably lower KL than the ones generated with the S-DFG method.

It is an interesting observation that the samples produced with the S-DFG method tend to be smaller sized than the ones produced with the S-TS method. To investigate this further, we analyze the number of predictions it takes to generate a sample because in sequential generative models, there is a small probability of producing an error in each prediction of a structure-building action. Long prediction sequences cause this error probability to grow. Relying on a sequence of predictions, both of the evaluated methods suffer from this.

To compare the proneness of methods to this issue, we analyze the distributions of prediction sequence lengths for the different methods on the training samples, as well as the generated samples. Figure 7.11 reveals that the prediction sequences of the S-DFG method are significantly longer than the ones of the S-TS method. For the generated samples, the lengths of the prediction sequences are about equal, which means that both methods are about equally-well fit to the training data. However, because the predictions needed to construct a graph are longer for the S-DFG method and the probability of generating an erroneous sample increases with the number of predictions, it is more biased towards generating smaller-sized samples than the S-TS method.

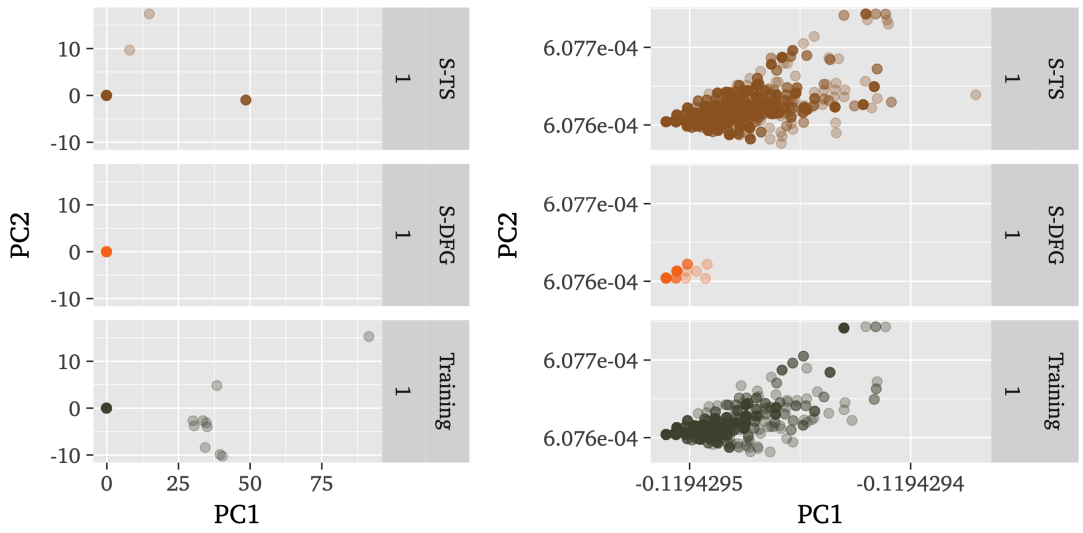


Figure 7.9: A complete view (left) and a view of the densest cluster (right) of the two main principal components (PC1 and PC2) of the training and the generated S-TS and S-DFG data sets.

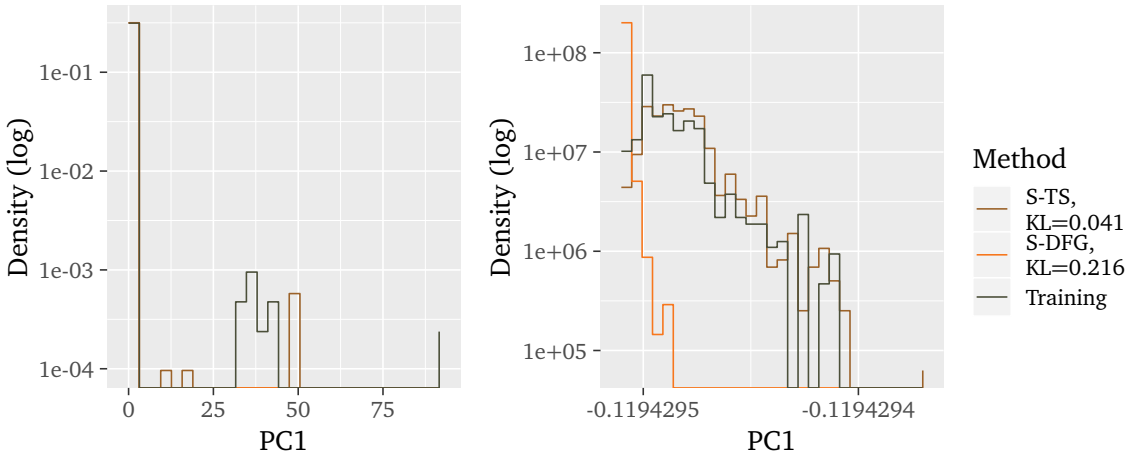


Figure 7.10: A complete view (left) and a view of the most dense cluster (right) of the two main principal components (PC1 and PC2) of the training and the generated S-TS and S-DFG data sets.

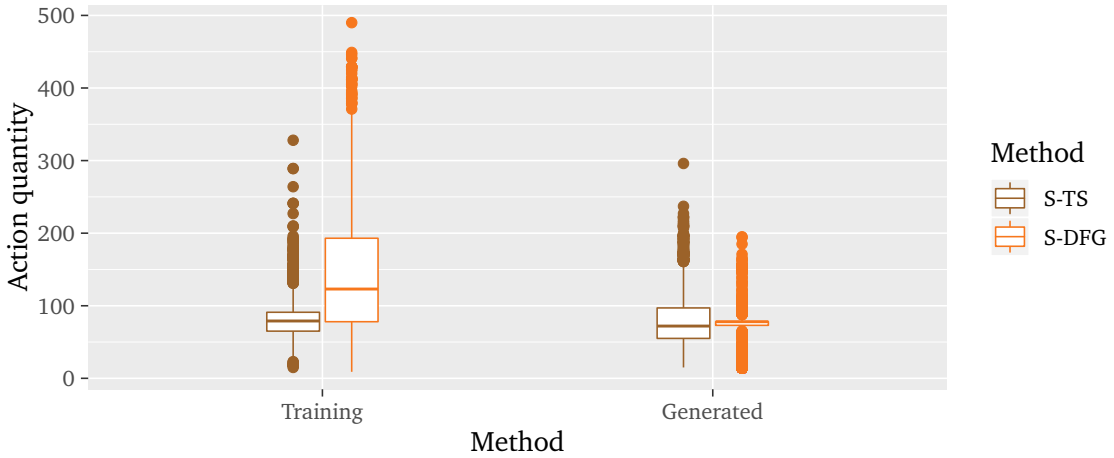


Figure 7.11: Number of generating actions for the S-TS and S-DFG methods in the training and generated data sets.

8

Conclusion and Outlook

8.1 Conclusion

In this thesis, we have successfully defined and implemented compiler-internal representations of programs for different types of tasks. We have shown an empirical evaluation between them in predictive and generative tasks.

In the scope of the evaluated predictive tasks, the results allow the conclusion that the GGNN models in combination with LLVM-SSA-CDFG and S-DFG representations mostly outperform the other methods in terms of generalization performance on unseen data. In inference time, simple models that rely on manually-defined features show the best results. In the class of deep learning methods however, GNNs can have a magnitude faster inference times if the graph representation is compact. Based on the evaluated metrics generalization performance and inference times, which are a tradeoff, we conclude that the GNN-based methods bring the best generalization results while having acceptable inference times.

In the class of generative tasks, we evaluated an RNN-based method based on sequences, as well as a GNN-based method that operates on graphs. The results suggest that the RNN-based architecture outperforms the GNN-based, which is conceptually due to longer prediction sequences of the graph-constructing actions that lower the probability of producing a valid sample of larger size.

8.2 Outlook

8.2.1 Analysis of Learned Features

We have shown an evaluation of the different deep learning models across different tasks. In the scope of future work, it would be interesting to understand the learned features across the models, similarly to the work done by Karpathy (2015), who analyzed the learned features of LSTMs as cell activations on a character-based representation of C code.

8.2.2 Extend Program Graph Semantics

In this work, we limited the evaluation to part of the semantic information that can be extracted from Clang and LLVM. Further semantics that can be added to the program graph representations are for example control-flow or liveness information for the S-DFG or dominator trees in the LLVM-CDFG. Additionally, multiple variants of integrating these semantics into the graph structure are possible, which can be subject to future work.

8.2.3 Domain-Specific Aggregation Schemes

While GNNs are powerful architectures to extract features of a node and its neighborhood as a node embedding, the aggregation to a fixed-sized graph-level embedding is a limiting factor of this architecture. Intuitively, this is because of the compression of many vectors to a single vector of a fixed and limited dimension.

One approach to further improving the aggregation scheme is to use domain-specific knowledge about the graphs, i.e. by limiting the aggregation scheme to specific nodes of the graph only. An extreme example of this is to include only the return statement of a function, or more moderately, to include only dataflow-terminating nodes. Another interesting approach is to exploit known structures, e.g. the instructions affinity to basic-blocks. By aggregating node embeddings to basic block embeddings first, then basic block embeddings to a graph embedding, the graph embedding can be formed hierarchically and information can be reduced in a more structured way.

8.2.4 Ensembles Across Representations and Models

In machine learning model competitions, so-called *ensembles* are successfully used to achieve state-of-the-art results. This is done by combining several state-of-the-art models into a single prediction model. In the context of the representations and models shown in this work, combining them could lead to further improvements in accuracy and speedup.

8.2.5 Domain-Specific Generative Model for S-DFGs

We have shown that the graph-generative model is biased towards generating shorter and structurally simpler samples than the sequence-based model. We investigated that this is because the graph-generative model uses longer generation sequences than the graph model. By using domain-specific knowledge about a programming language, such as its grammar, the number of graph-generating actions can be reduced. A further example of exploiting such domain-specific knowledge is limiting the decisions of certain states, e.g. in binary operators, which need to have one incoming AST and two dataflow edges. Therefore, at least three actions can be saved in this specific case.

8.2.6 Learning Dataflow Analyses

Another exciting direction is to train the GNN model to predict common dataflow-analyses. In this work, we have used the GNN model to make graph-level predictions. However, by skipping the output phase, it can be used for node-level predictions. A first objective could be to investigate the model's capability of learning dataflow analysis on dataflow-labeled graphs. Afterwards, the model could be integrated into a higher-level architecture to solve a compiler task. Analyzing the optimized dataflow features could allow conclusions on the design for future dataflow analyses.

8.2.7 GNN Model Heuristic in a Real-World Compiler

We evaluated the model in the predictive use case as a compiler heuristic in two concrete tasks. These tasks are popular among the research community and are well-suited for a first evaluation of different models' potentials.

A further exciting use case for the GNN models as heuristics is the Milepost GCC project by Fursin et al. (2011) who integrates a heuristic model relying on manually-defined features that are input to a decision tree into the production-grade GCC compiler. By predicting optimizations and their order on a function-level granularity, they achieve speedups over GCC's highest optimization level.

In this work, we have shown that models relying on manually-defined features can be outperformed by far by deep learning-based models while maintaining acceptable inference times. Integrating the graph-based models into Milepost GCC could further improve the speedup and reduce the execution times and energy efficiency of real-world applications.

Bibliography

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4): 81, 2018.

Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

Amir H Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):29, 2017.

Rohan Bavishi, Michael Pradel, and Koushik Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: a learnable representation of code semantics. In *Advances in Neural Information Processing Systems*, pages 3585–3597, 2018.

Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software*

engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 213–222. ACM, 2009.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

Kyunghyun Cho, B van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. In *Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST-8), 2014*, 2014.

Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017a.

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99. IEEE, 2017b.

Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, 2010.

Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272, 2017.

Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. A case study on machine learning for synthesizing benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 38–46. ACM, 2019.

Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.

Mary Hall, David Padua, and Keshav Pingali. Compiler research: the next 50 years. *Communications of the ACM*, 52(2):60–67, 2009.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 272–283. IEEE, 2014.

Daniel D. Johnson. Learning graphical state transitions. In *International Conference on Learning Representations (ICLR)*, 2017.

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184. ACM, 2014.

Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks, May 2015. URL `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *ACM SIGPLAN Notices*, volume 47, pages 147–162. ACM, 2012.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*, 2016.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018a.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. Sysevr: a framework for using deep learning to detect software vulnerabilities. *arXiv preprint arXiv:1807.06756*, 2018b.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018c.

Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 455–466. ACM, 2014.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *International conference on artificial intelligence: methodology, systems, and applications*, pages 41–50. Springer, 2002.

Dana Movshovitz-Attias and William W Cohen. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 35–40, 2013.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 585–596. IEEE, 2015.

Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542. ACM, 2013.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1): 61–80, 2008.

Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE international symposium on workload characterization (IISWC)*, pages 137–148. IEEE, 2011.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

Zheng Wang and Michael OBoyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

Paul J Werbos et al. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

Appendices

APPENDIX A

Further Program Examples

```
int isPrime(int n) {
    if (n <= 1) return 0;
    if (n % 2 == 0 && n > 2) return 0;
    for(int i = 3; i < n / 2; i += 2) {
        if (n % i == 0)
            return 0;
    }
    return 1;
}
```

Figure A.1: Original source code of the `isPrime` function.

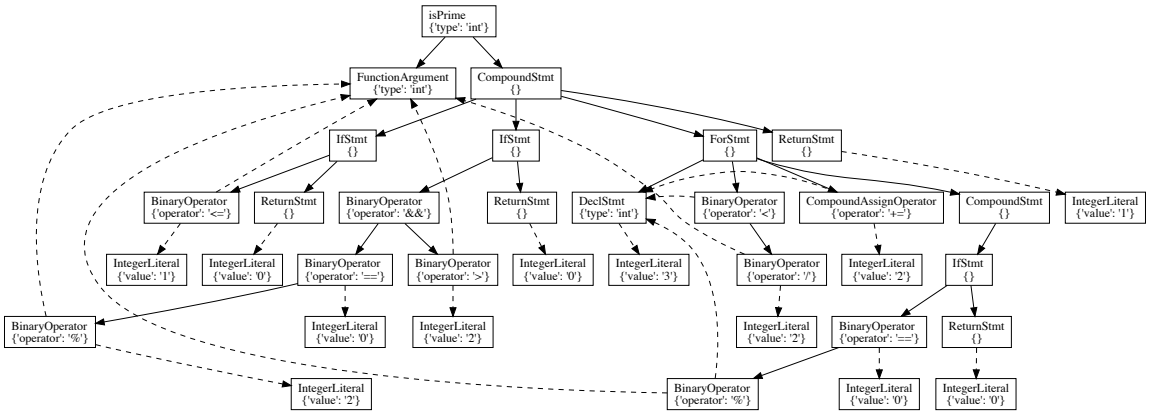
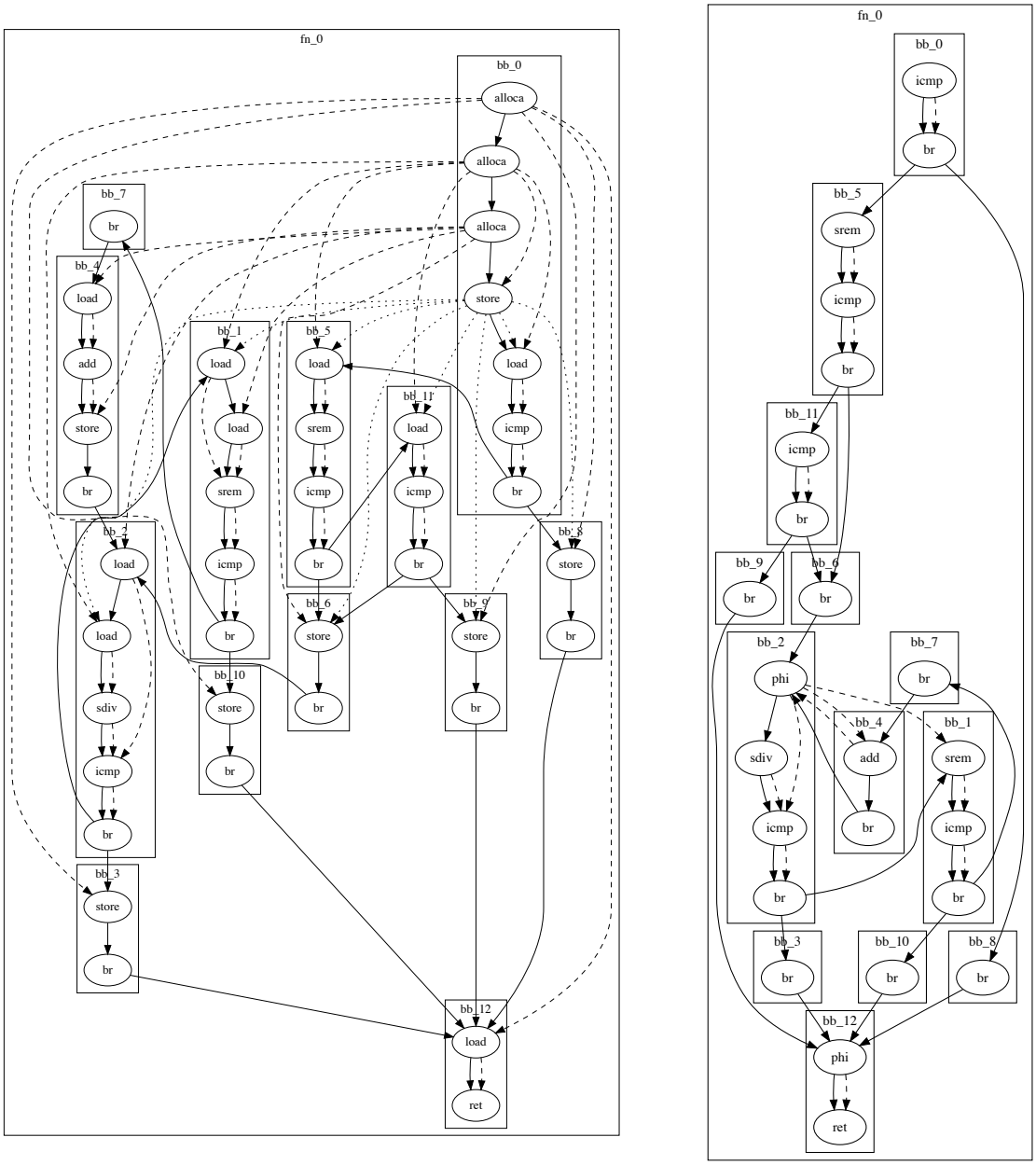


Figure A.2: S-DFG representation of the `isPrime` function.



(a) LLVM-CDFG graph.

(b) LLVM-SSA-CDFG graph.

Figure A.3: LLVM IR-based representations of the `isPrime` function. Control-flow edges are in solid, dataflow edges in dashed, memory access edges in dotted, and call edges in bold style.

```
int fibbonacci(int n) {
   if(n == 0) {
      return 0;
   } else if(n == 1) {
      return 1;
   } else {
      return (fibbonacci(n-1) + fibbonacci(n-2));
   }
}
```

Figure A.4: Original source code of the `fibbonacci` function.

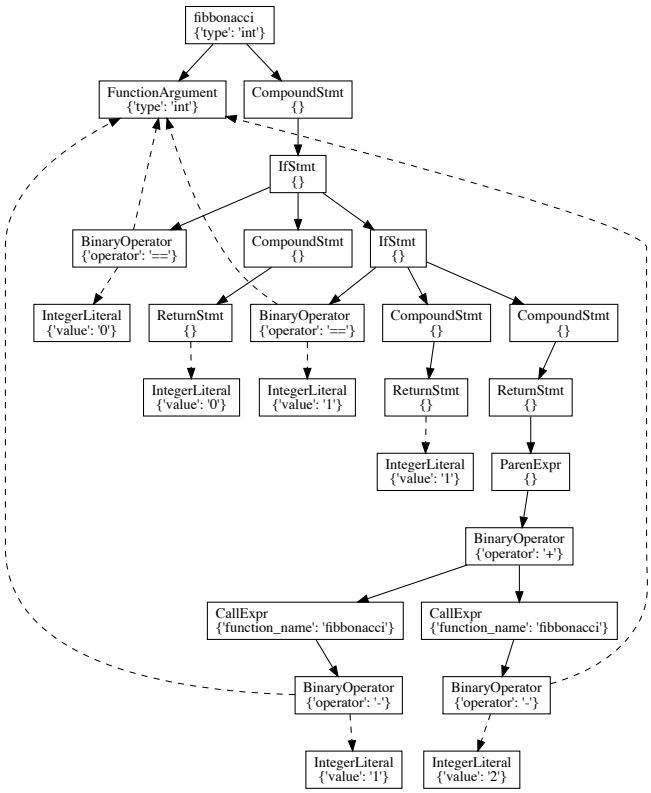
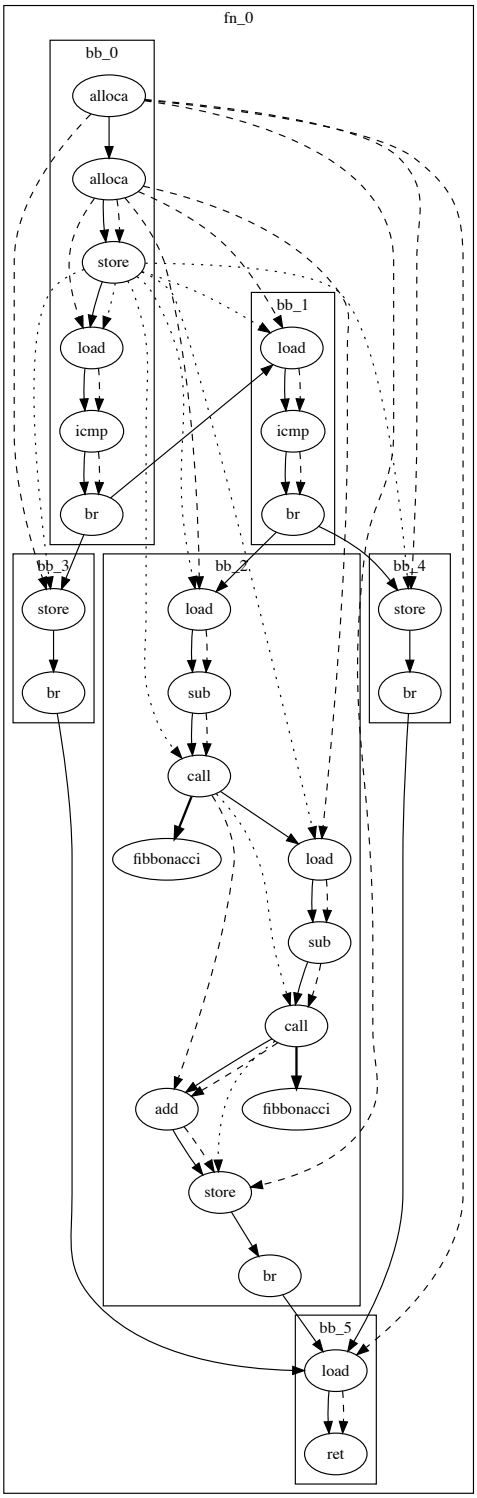
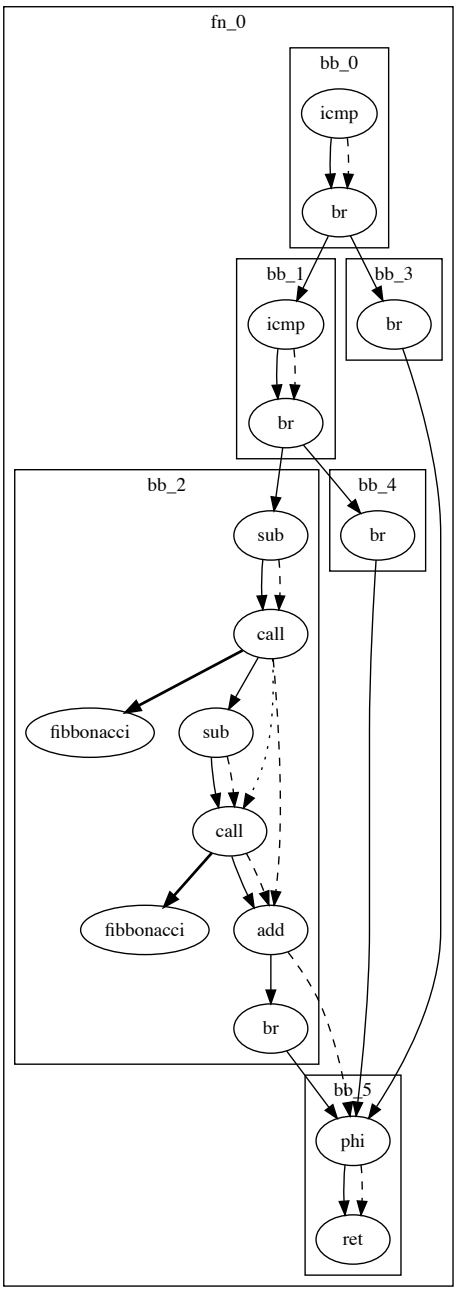


Figure A.5: S-DFG representation of the `fibbonacci` function.



(a) LLVM-CDFG graph.



(b) LLVM-SSA-CDFG graph.

Figure A.6: LLVM IR-based representations of the `fibbonacci` function. Control-flow edges are in solid, dataflow edges in dashed, memory access edges in dotted, and call edges in bold style.

```
int isPalindrome(int* str, int size) {
    int l = 0;
    int h = size;

    while (h > l) {
        if (str[l++] != str[h--]) {
            return 0;
        }
    }
    return 1;
}
```

Figure A.7: Original source code of the `isPalindrome` function.

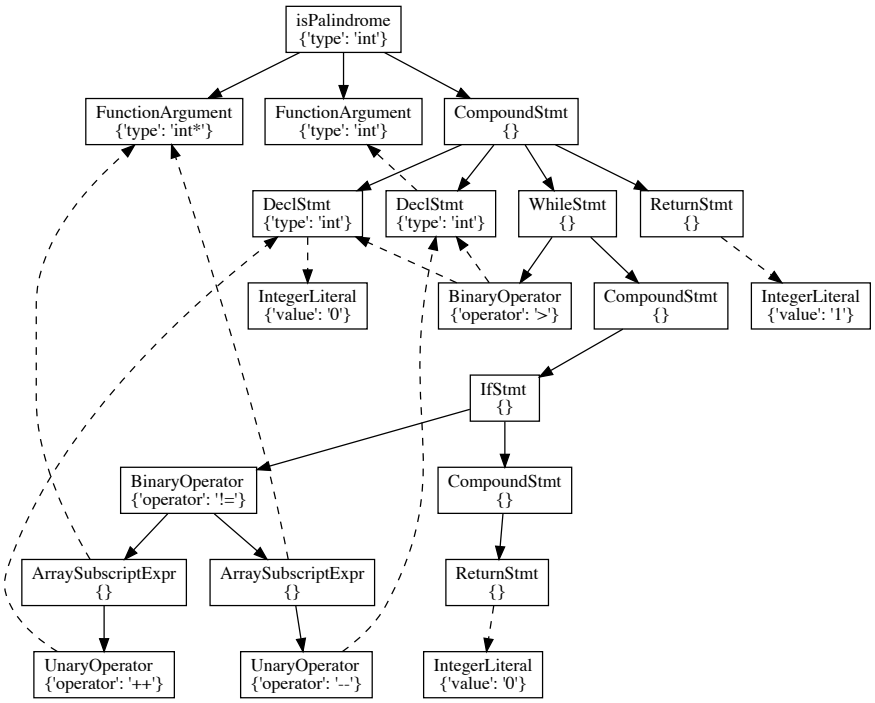
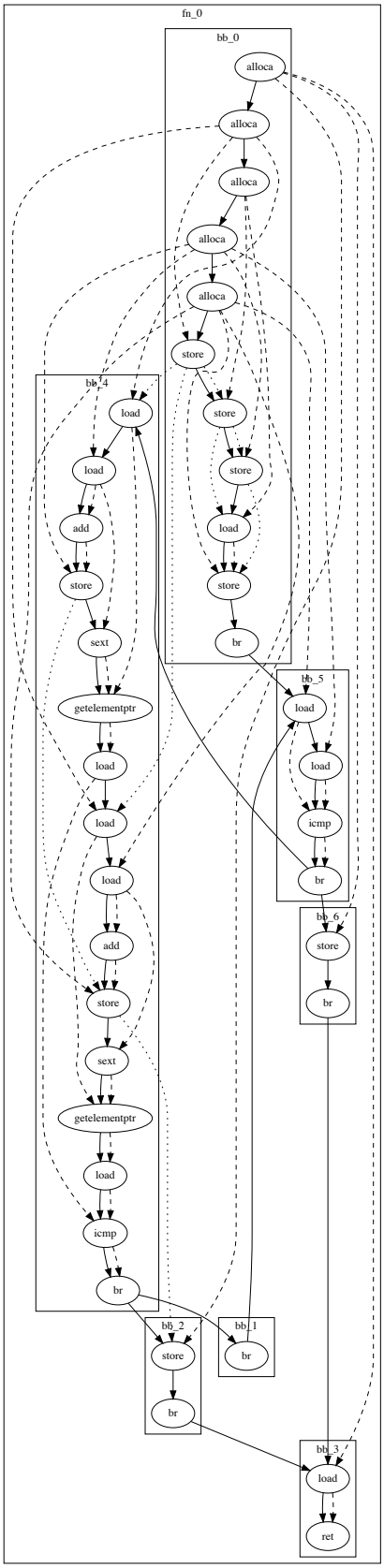
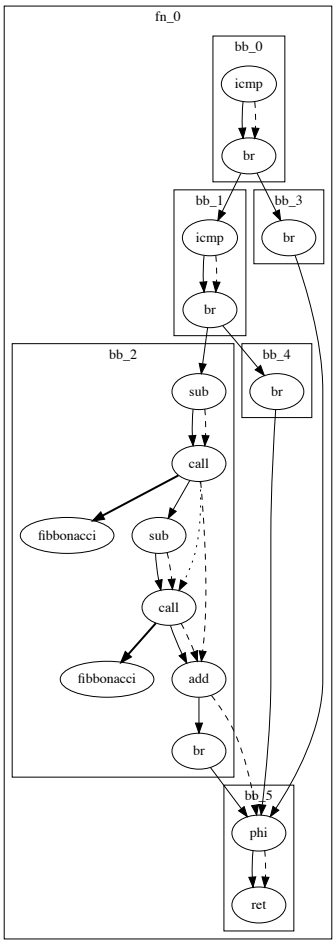


Figure A.8: S-DFG representation of the `isPalindrome` function.



(a) LLVM-CDFG graph.



(b) LLVM-SSA-CDFG graph.

Figure A.9: LLVM IR-based representations of the `isPalindrome` function. Control-flow edges are in solid, dataflow edges in dashed, memory access edges in dotted, and call edges in bold style.

```
void insertionSort(int* arr, int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Figure A.10: Original source code of the `insertionSort` function.

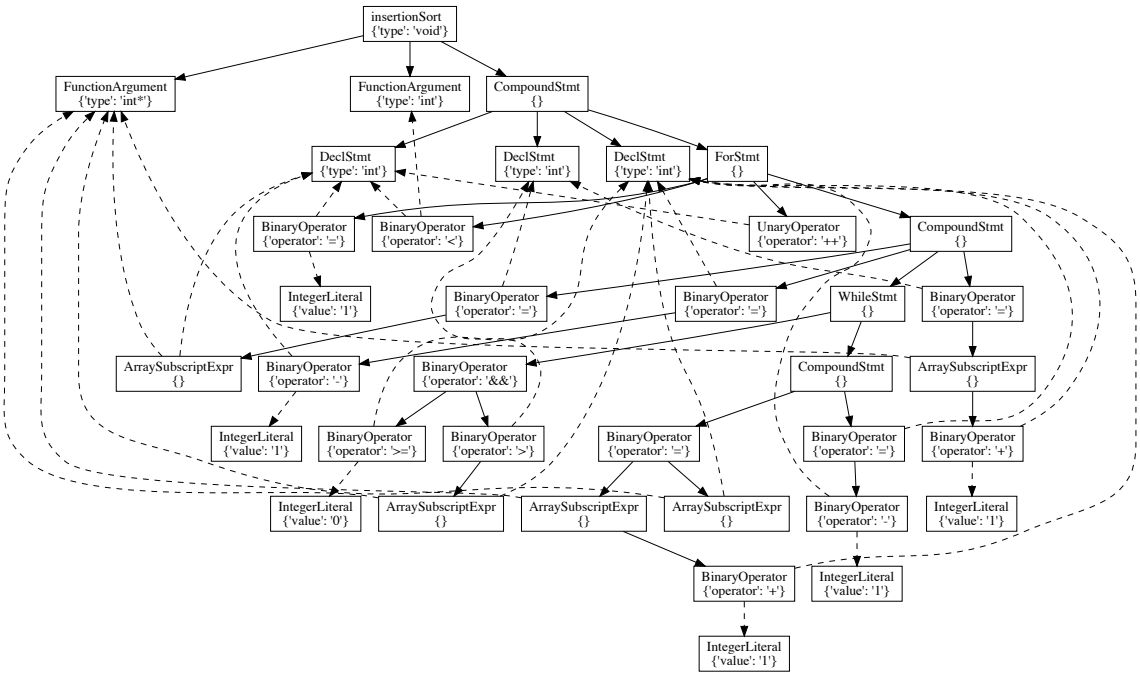
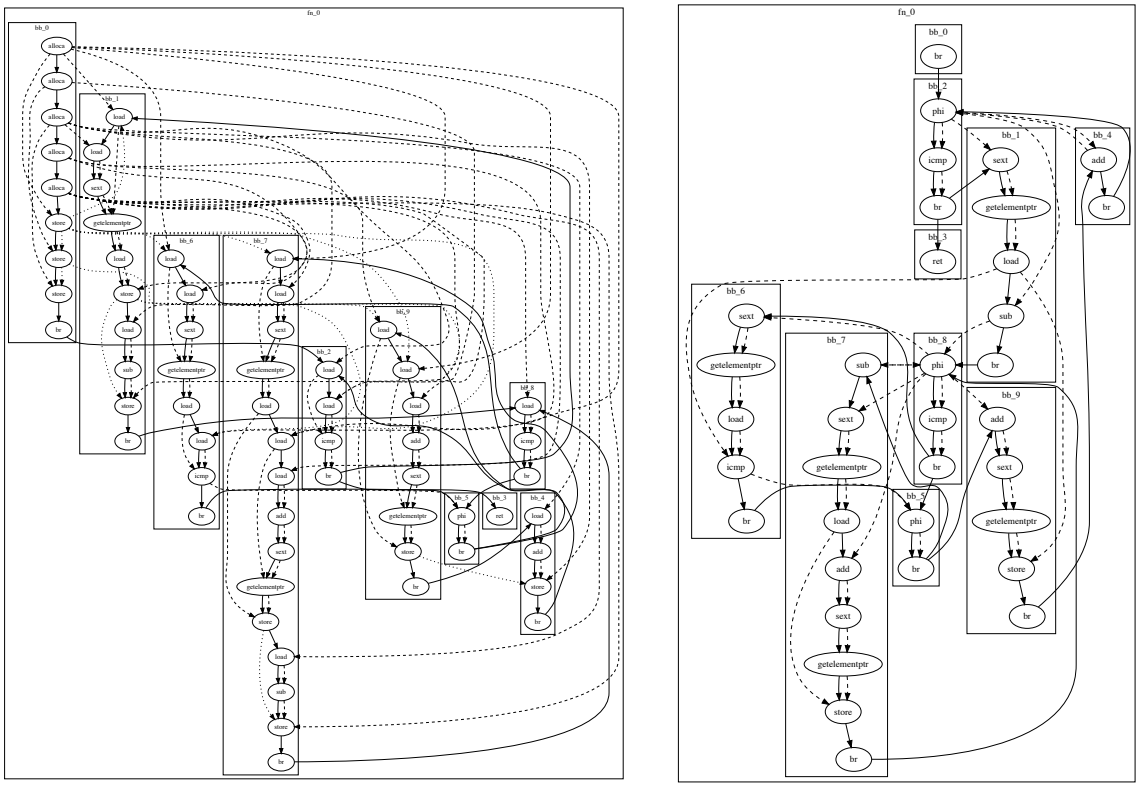


Figure A.11: S-DFG representation of the `insertionSort` function.



(a) LLVM-CDFG graph.

(b) LLVM-SSA-CDFG graph.

Figure A.12: LLVM IR-based representations of the `insertionSort` function. Control-flow edges are in solid, dataflow edges in dashed, memory access edges in dotted, and call edges in bold style.

APPENDIX B

Cross Validation with Training-Test-Validation Splits

As an additional validation of part of the results presented in this thesis, we perform a hyperparameter search for the device mapping task. For this, we perform k-fold cross-validation on the same split as the original work with the modification that we group the dataset into three parts: A *training set* to fit the data, a *validation set* to find the best hyperparameters, and a *test set* to evaluate the performance with the model with the best hyperparameters.

Such an evaluation has not been done before by any of the prior works that compare in this task. Nevertheless insightful as it allows drawing general conclusions the models performances, independently of the hyperparameter sets, that are commonly part of published deep learning-based methods.

Applying the 10-fold cross-validation of the original works with this 3-split results in 90 unique configurations of this split: $\binom{10}{8} = 45$ yields 45 configurations of a training set. Because two configurations of the remaining sets as validation and test sets exist, we finally obtain $2\binom{10}{8} = 90$ configurations.

We use *Bayesian Optimization* (Snoek et al. (2012)) to find the set of best-performing hyperparameters for each of the configurations. Bayesian Optimization is a guided hyperparameter search that treats the model as a black-box function and tries to iteratively minimize its output (the objective) by proposing new values for model estimation, while constructing a model internally. As this objective, we use the negative accuracy on the validation set. In our setup, we use a unique bayesian optimizer for each of of the configurations, because the combination of the model with each of the configurations is a unique function.

Hyperparameter	Model function	Domain
State dimension	2^x	$[1,9]$
Number of LSTM layers	x	$[0,6]$
Number of epochs	$2^{100\ x}$	$[0,7]$
L2 loss	$0.05\ x$	$[0,10]$

(a) Sequence-based method.

Hyperparameter	Model function	Domain
Hidden dimension	2^x	$[4,9]$
State dimension	$2\ x$	$[1,9]$
h_G dimension	2^x	$[1,9]$
Number of timesteps	$2\ x$	$[1,4]$
Number of epochs	$2^{100\ x}$	$[0,7]$
Backedges	x	TRUE, FALSE
L2 loss	$0.05\ x$	$[0,10]$

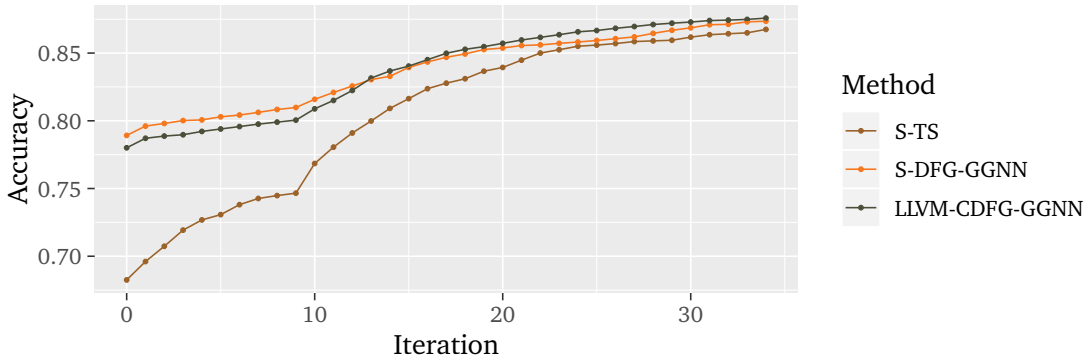
(b) Graph-based methods.

Table B.1: Hyperparameters, model functions, and domains for Bayesian Optimization.

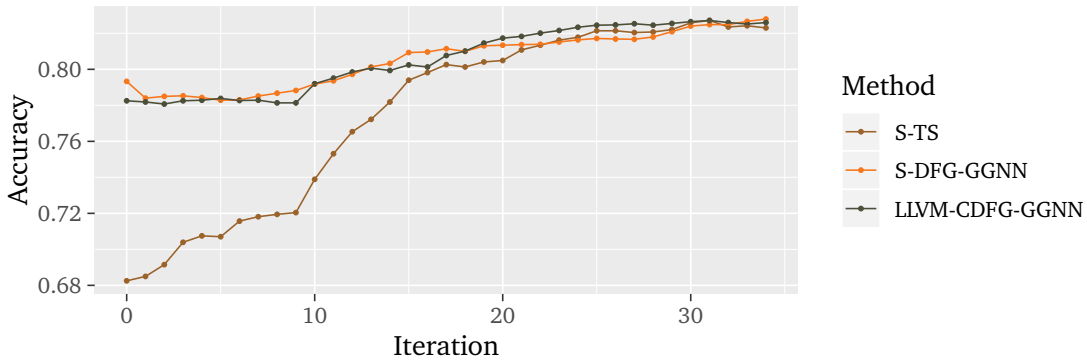
However, the computational cost of a hyperparameter search is enormous, because each iteration requires a model estimation, which is a complete training in this case. We try to evaluate as many representations as we can for a fixed amount of iterations in the scope of the given computational resources. Concretely, we decided for 35 iterations on the device mapping task on the AMD dataset, which results in a total of 3150 trainings. We can report results for the S-TS, the S-DFG-GGNN, and the LLVM-CDFG-GGNN representations.

Figure B.1 shows arithmetic means of the accuracies of the methods over the 90 split configurations per iteration. The hyperparameter search space is summarized in Table B.1, along with the model function that we use to compute the actual hyperparameter with the bayesian optimizations estimation as input x. Please note that we modify the S-TS method of the original work to support more hyperparameters.

The results at iteration 35 generally support the conclusions drawn in our previous experiments obtained with hand-chosen hyperparameters as shown in Figure 7.1: The LLVM-CDFG-GGNN method performs the S-TS method on the test and validation sets, however modestly. Additionally to that, the S-DFG method also outperforms S-TS method in both of the sets.



(a) Validation set.



(b) Test set.

Figure B.1: Accuracy on the test and validation sets in the hyperparameter search.