DISSERTATION

MACHINE LEARNING FOR COMPUTER AIDED PROGRAMMING: FROM STOCHASTIC PROGRAM REPAIR TO VERIFIABLE PROGRAM EQUIVALENCE

Submitted by

Steve Kommrusch

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2022

Doctoral Committee:

Advisor: Louis-Noël Pouchet
Co-Advisor: Charles Anderson

Ross Beveridge
Mahmood Azimi-Sadjadi

Copyright by Steve Kommrusch 2022

All Rights Reserved

ABSTRACT

MACHINE LEARNING FOR COMPUTER AIDED PROGRAMMING: FROM STOCHASTIC PROGRAM REPAIR TO VERIFIABLE PROGRAM EQUIVALENCE

Computer programming has benefited from a virtuous cycle of innovation as improvements in computer hardware and software make higher levels of program abstraction and complexity possible. Recent advances in the field of machine learning, including neural network models for translating and answering questions about human language, can also be applied to computer programming itself. This thesis aims to make progress on the problem of using machine learning to improve the quality and robustness of computer programs by contributing new techniques for representation of programming problems, applying neural network models to code, and training procedures to create systems useful for computer aided programming. We first present background and preliminary studies of machine learning concepts. We then present a system that directly produces source code for automatic program repair which advances the state of the art by using a learned copy mechanism during generation. We extend a similar system to tune its learning for security vulnerability repair. We then develop a system for program equivalence which generates deterministically checkable output for equivalent programs. For this work we detail our contribution to the popular OpenNMT-py GitHub project used broadly for neural machine translation. Finally, we show how the deterministically checkable output can provide self-supervised sample selection which improves the performance and generalizability of the system. We develop breadth metrics to demonstrate that the range of problems addressed is representative of the problem space, while demonstrating that our deep neural networks generate proposed solutions which can be verified in linear time. Ultimately, our work provides promising results in multiple areas of computer aided programming which allow human developers to produce quality software more effectively.

ACKNOWLEDGEMENTS

I began my PhD with a goal of making a contribution to the groundbreaking advances now occurring in machine learning and artificial intelligence. Additionally, I wanted to better understand the scientific process and the community that participates in presenting and discussing ideas in scientific venues. Finally, I wanted to prepare to do further research after my PhD in the fields of machine learning and artificial intelligence. Many people have helped me in my pursuit of these goals.

I would first like to thank my mentor, advisor, and friend Louis-Noël Pouchet. Louis-Noël encouraged me to explore new areas for applying machine learning with a focus on reasoning about programs and generating computer code. He shared his experience with the publication process including being an author, reviewer, and session chair with me and provided many fantastic opportunities for me to interact with the broader research community. His enthusiasm for all facets of computer science was a great motivation for me.

In my interest to connect with researchers in Europe, I was able connect with Martin Monperrus who invited me to research in Sweden during the fall of 2018. During this time I worked with Zimin Chen, a PhD student at KTH also researching the use of machine learning on computer code. This was a very fruitful collaboration, resulting in 3 published papers which I authored with Martin and Zimin and one more paper with Martin in review as of April 2022. I deeply appreciate Martin's enthusiasm for open science and his introductions to researchers at KTH but also in London and Cambridge, England. His support helped broaden my experience. My work with Zimin was also a great experience - as two PhD students we were able to divide up technical tasks effectively and made great collaborative contributions to science together. Thank you Zimin and Martin.

My time at CSU let me meet many professors who contributed to my education. Certainly Chuck Anderson, my co-advisor for this thesis, helped me in many ways. I met with Chuck before applying as a PhD student and his energy and expertise in the field of machine learning attracted me to CSU. His insights over the years improved my research path and helped me explore the

field more deeply. Ross Beveridge was not only on my thesis committee but helped expand my knowledge of human-machine interaction using AI and machine learning on videos and agents acting on the world, which helped prepare me for the industry position I have taken after my PhD. I also thank Mahmood Azimi-Sadjadi for insightful questions and comments on this thesis, and professors Krishnaswamy, Blanchard, Ray, Draper, Ortega, and Pallickara for discussions and teachings that helped make my time at CSU enlightening.

Certainly my friends and family have been instrumental in helping me choose and complete this PhD adventure. Our discussions have been very helpful as we collectively explored career and life options. The full list would be too long but Dan, Yi-Fan, Kerri, John, and Andy have been friends for years and were a great source of support on my path. My parents and brother were constantly interested in and supportive of my efforts. And my wonderful wife Linda helped keep our home and social life fun while I worked diligently on this thesis. Indeed, my wife and father joined my CSU experience as they attended classes on art history and climate science, and we had many great lunches on campus after class. Thank you so much - I love you all.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii

Chapter 1 Introduction 1

I Machine Learning on Programs: Background and Our Contributions 9

Chapter 2 Background on Machine Learning for Computer Aided Programming 10
2.1 Machine Learning for Syntax Understanding and Sequence Generation . . 11
2.1.1 Neural machine translation with sequence-to-sequence learning 11
2.1.2 Gated recurrent unit 12
2.1.3 Get to the Point: Summarization with Pointer-Generator Networks . . . 13
2.1.4 Graph neural networks 16
2.1.5 Graph-to-Sequence Learning using Gated Graph Neural Networks . . . 17
2.1.6 Transformer Networks Use Attention Layers 19
2.2 Computer Aided Programming 22
2.2.1 Staged Program Repair with Condition Synthesis 22
2.2.2 PHOG: Probabilistic Model for Code 24
2.2.3 Abstract Syntax Trees 26
2.2.4 code2vec: Learning Distributed Representations of Code 26
2.2.5 Program repair 31
2.2.6 Program equivalence 31
2.2.7 Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection 32
2.3 Preliminary Studies on Machine Learning Concepts and Limitations 33
2.3.1 Multilayer Perceptron (Vanilla) Neural Network Models And the Challenge of Learning Hash Functions 35
2.3.2 Generalizing to a Target Distribution With Proper Network Sizing . . . 38
2.3.3 Using Hindsight Learning to Improve Goal Achievement In Multigoal Environment With Sparse Rewards 43
2.4 Complexity Classes for Program Equivalence 46
2.5 Limitations of Prior Research 52

Chapter 3 Contributions to Machine Learning on Computer Aided Programming 54
3.1 Repairing Functional Bugs in Java Programs 55
3.2 Repairing Security Vulnerabilities in C Language Programs 58
3.3 Proving Equivalence of Complex Linear Algebra Expressions 60
3.4 Proving Equivalence of Basic Block Expressions in C Code 62

II Machine Learning for Repairing Software Defects 65

Chapter 4 SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair 66
4.1 Introduction . . . 66
4.2 Background on Neural Machine Translation with Seq-to-Seq Learning . . 67
4.3 Approach to Using Seq-to-Seq Learning for Repair . . . 69
4.3.1 Problem Definition . . . 71
4.3.2 Abstract Buggy Context . . . 71
4.3.3 Sequence-to-Sequence Network . . . 73
4.3.4 Patch Inference . . . 75
4.3.5 Patch Preparation . . . 76
4.3.6 Implementation Details & Parameter Settings . . . 77
4.4 Evaluation . . . 78
4.4.1 Research Questions . . . 79
4.4.2 Experimental Methodology . . . 79
4.4.3 Training Data . . . 81
4.4.4 Experimental Results . . . 84
4.4.5 Defects4J Evaluation . . . 88
4.4.6 Qualitative Case Studies . . . 92
4.5 Ablation Study . . . 94
4.6 Related Work . . . 97
4.7 SEQUENCER Summation . . . 100
4.8 Continuous Integration with SEQUENCER . . . 101

Chapter 5 Neural Transfer Learning for Repairing Security Vulnerabilities in C Code . . 104
5.1 Introduction . . . 104
5.2 Background on Software Vulnerabilities and Related Machine Learning . . 105
5.2.1 Software Vulnerabilities . . . 105
5.2.2 Transformers . . . 106
5.2.3 Transfer Learning . . . 106
5.3 VRepair: Repairing Software Vulnerabilities with Transfer Learning 108
5.3.1 Overview . . . 108
5.3.2 Code Representation . . . 109
5.3.3 Tokenization . . . 112
5.3.4 Source Domain Training . . . 113
5.3.5 Target Domain Training . . . 113
5.3.6 Inference for Patch Synthesis . . . 114
5.3.7 Neural Network Architecture . . . 115
5.3.8 Usage of VRepair . . . 116
5.4 Experimental Protocol . . . 117
5.4.1 Research Questions . . . 117
5.4.2 Datasets . . . 118
5.4.3 Methodology for training with either source or target domain samples . 120
5.4.4 Methodology for transfer learning with source and target domain data . 122
5.4.5 Methodology for pre-training with denoising samples . . . 122

5.4.6 Methodology for data split strategies . . . 123
5.5 Experimental Results . . . 124
5.5.1 Results for training with either source or target domain samples . . . 124
5.5.2 Results for transfer learning with source and target domain samples . . . 127
5.5.3 Results for pre-training with denoising samples . . . 130
5.5.4 Results for data split strategies . . . 133
5.6 Ablation Study . . . 134
5.7 Related Work . . . 137
5.7.1 Vulnerability Fixing with Learning . . . 137
5.7.2 Vulnerability Fixing without Learning . . . 138
5.7.3 Vulnerability Datasets . . . 140
5.7.4 Machine Learning on Code . . . 141
5.7.5 Transfer Learning in Software Engineering . . . 143
5.8 Conclusion . . . 144

III Producing Verifiable Program Equivalence Proofs Using Machine Learning 146

Chapter 6 Proving Equivalence Between Complex Linear Algebra Expressions With Graph-to-Sequence Neural Models . . . 147
6.1 Introduction . . . 147
6.2 Motivation and Overview for Machine Learning Applied to Program Equivalence . . . 148
6.3 Framework for Program Equivalence . . . 152
6.3.1 Input Representation . . . 152
6.3.2 Axioms of Equivalence . . . 153
6.3.3 Space of Equivalences . . . 156
6.4 Samples Generation . . . 157
6.5 Deep Neural Networks for Program Equivalence . . . 160
6.6 Experimental Results . . . 162
6.6.1 WholeProof Models: Language Complexity and Performance . . . 165
6.7 Related Work . . . 167
6.8 Conclusion . . . 169
6.9 Supplementary Materials . . . 170
6.9.1 Dataset generation . . . 170
6.9.2 Language and Axioms for Complex Linear Algebra Expressions . . . 176
6.9.3 Details on neural network model . . . 177
6.9.4 Details on Experimental Results . . . 182

Chapter 7 Self-Supervised Learning to Prove Equivalence Between Programs via Semantics-Preserving Rewrite Rules . . . 193
7.1 Introduction . . . 193
7.2 Problem Statement . . . 195
7.2.1 Scope . . . 195

7.2.2 Encoding of Programs and Rewrite Rules 196
7.2.3 Pathfinding Rewrite Rule Sequences 198
7.3 S4Eq: Deep Learning to Find Rewrite Rule Sequences 199
7.3.1 Overview of S4Eq 199
7.3.2 Transformer Model 200
7.3.3 Beam Search to Explore Multiple Possible Proofs 202
7.3.4 Training Process 204
7.4 Experimentation 208
7.4.1 Dataset Generation 208
7.4.2 Experimental setup of incremental training 212
7.4.3 Research Questions 213
7.4.4 Experimental Results 216
7.4.5 Ablation Study 224
7.4.6 Execution time 226
7.5 Related Work 227
7.5.1 Static Program Equivalence 227
7.5.2 Incremental and Transfer Learning 227
7.5.3 Symbolic Mathematics Using Machine Learning 228
7.5.4 Program Analysis using Machine Learning 229
7.6 Conclusion 230

Chapter 8 Conclusion 231
8.1 Contributions 232
8.2 Avenues for Future Research 233

Bibliography 261

Chapter 1

Introduction

Machine learning and artificial intelligence have been advancing for decades. Originally named in the 1950's [156], the field of artificial intelligence has been constantly advanced by researchers throughout the world. For almost as long, the field of computer aided programming has pursued the goal of having computers handle the bulk of the work required to build and maintain code [26]. In parallel with these software efforts, the field of computer hardware was exponentially growing the number of transistors available on a computer chip year after year following Moore's law [162]. Now is the time when these ongoing trends have enabled the opportunity for computers to learn to repair and analyze programs directly - to use machine learning running on modern computing hardware to enable computers to not merely provide useful development tools and compilers to software engineers, but to directly understand and improve programs themselves. Providing such capability will allow human programmers to work at higher levels of abstraction, creating quality software with fewer bugs and fewer security vulnerabilities.

The field of program synthesis studies the goal of generating code to meet a given set of requirements, *i.e.,* programming. Given recent advances in machine learning and artificial intelligence, researchers have begun to pursue the dream of having computers aid in their own programming in order to improve the quality of programs and reduce the effort needed to create and maintain them. Advances in hardware capability and software algorithms have enabled deep learning to apply neural network models to a variety of problems. Typically, a deep learning model is learning how to map information from an input distribution (in the sense of a statistical or probabilistic distribution) to an output distribution. The mapping rarely yields a 100% accuracy in the result, whether the output is a class label for image recognition [62], a word identifier for language generation [208], or the optimal action in reinforcement learning [83]. In some applications, such as image identification on a web application, imperfect outputs can be tolerated so long as a given quality metric is met. When processing or generating computer code, however, some form of automated

verification of the output becomes necessary to ensure the output is usable in the highly structured world of computer programming.

The problem area addressed in this thesis is built upon prior work in program synthesis and its intersection with machine learning. Prior work on computer aided programming has explored directed search and probabilistic code synthesis [143, 34], various machine learning techniques for generating language tokens and syntax [197, 29], and learned representations of code for machine learning generalization [12]. Recognizing that computer code is a form of human communication, techniques from natural language processing have been successfully applied to computer code understanding and generation. Recognizing machine learning outputs may be imprecise, various software engineering techniques for verification can be combined to help improve the output quality of a system. By advancing the state-of-the art for machine learning on computer code, we hope to improve the efficiency of software engineers and computer scientists and support their efforts to continue innovating on more and more complex tasks.

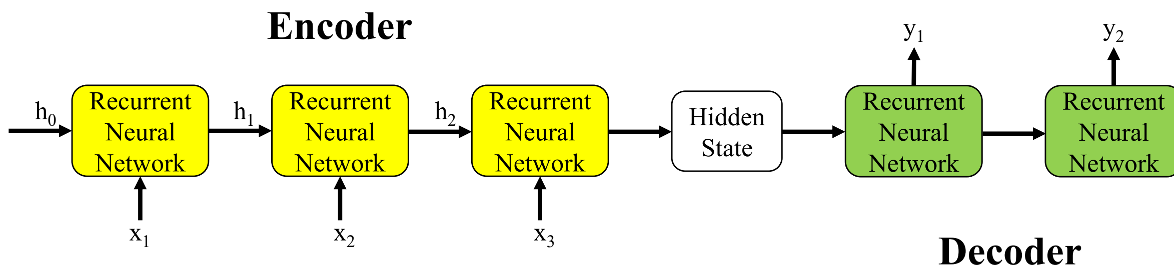


Figure 1.1: Sequence-to-Sequence model showing encoder, decoder, and hidden state

Computer programs are typically thought of by both programmers and the computers processing the program as a sequence of computer language tokens. For example, `if ( x < 0 )` is a sequence of 6 tokens which checks whether the variable x is a negative number. The types of machine learning models most commonly used for token sequence analysis are called sequence-to-sequence models [208, 202]. Figure 1.1 diagrams an early type of sequence-to-sequence model using a recurrent neural network (RNN) to encode an input sequence into a hidden state which can then be decoded to produce an output sequence. The recurrent neural network provides a method

for the model to learn a function f which encodes the sequence of tokens provided by x_t into a hidden state h that the model uses to solve a given problem. The model output uses a learned function g to create output tokens y_t based on the problem the model has been designed to solve and the hidden state created by encoding the input tokens.

$$h_t = f(x_t, h_{t-1})$$
$$y_t = g(h_t)$$

The functions f and g are learned by a neural network model by providing it thousands or even millions of example input and output sequences. After being trained on these sequences, the model can be used to produce output for previously unseen input samples. In this way, we can use a sequence-to-sequence model to process computer code and generate output targeting a given programming task. For example, as we shall discuss in Chapter 4, the input could be a buggy program and the output could be a patch which repairs the bug. Alternately, as we shall discuss in Chapter 6, the input could be 2 programs and the output could be a sequence of steps which prove the 2 programs are identical. A challenge of using machine learning models on code is that their output is stochastic - it is the 'best guess' by the model at the correct output based on the training data it has received. This thesis aims to address the problem of using stochastic machine learning models for computer aided programming.

Given this background, we next investigate and evaluate techniques for validating neural network outputs in an automated way. The general approach is to construct a system which involves a machine learning generator and an automated validation component of some kind. Using these techniques we show contributions which advance the state of the art in multiple areas and demonstrate these contributions with full implementations of 4 computer aided programming systems.

Contributions to Machine Learning Processes

In this section we summarize 4 key contributions we make which have general benefit for machine learning as applied to computer code. These contributions could be applied to a wide

array of problems and cover advances for machine learning models for code, code representation for machine learning, and novel techniques for training neural network models.

Self-Supervised Sample Selection For sequence generation problems in which the output can be automatically checked, we can leverage the aforementioned stochastic nature of machine learning to create new samples on which to iteratively train and improve a model [124]. Because of the stochastic nature of the model, the outputs can be ranked in likelihood based on what the model has learned from previous training examples. For example, the output sequence `a=0` might be ranked higher than `b=1`. When the highest ranked output from the model is indeed correct, then the problem given as input is already well represented by the current model. However, when the highest ranked output is wrong, but a lower ranked output is correct, then the model can be trained on a new input/output example based on this result so that its learned functions improve for future use. Training sample created using our method are inherently biased towards the more challenging problems in the input distribution. During initial model training it is important to have target output provided for a given input so that the model can learn the correct representation, but with self-supervised sample selection, input problems with no known output can be provided to a model and useful training samples can be created for improving the model. While we show how to apply this approach to analyzing computer code, any machine learning system which allows for automated sample generation combined with robust output verification could benefit from it.

Transfer Learning From Bug Fix to Security Vulnerability Fix The specific problem of repairing security vulnerabilities is critical to software engineers today, but before-and-after examples of such fixes are limited as their creation requires careful attention by developers. This data limitation is a problem for machine learning which typically relies on large sample counts on which to train. To address this, we train a model on abundant examples of generic bug fixes and then tune the model for the problem of vulnerability repair [47]. We show that our process exceeds the performance of training on the smaller vulnerability dataset alone, training only on the generic dataset, and is also better than pretraining with a state-of-the-art noising model. While we apply

this approach to security vulnerabilities, many types of targeted program modifications could be initially trained using a similar technique.

Input and output representations for applying machine learning to computer aided programming We develop and analyze multiple ways of presenting computer aided programming problems to machine learning models in formats which allow the model to learn a mapping between the input and output which is useful for addressing a given problem. For the task of generating code patches to address bugs or vulnerabilities in code, we present the construction of a novel *abstract buggy context* which leverages Java code context as input to a model for patch generation [48]. For succinctly specifying a multi-line patch to code, we design a novel code representation for the program repair task which we call the *token context diff* because it identifies the location of a code change using the token sequences which occur before and after the change [47]. For precisely presenting programs for symbolic reasoning and proof generation by a neural network, we demonstrate using *abstract syntax trees (AST)* as neural network model inputs and *rewrite rules* as model outputs to specify transformations on the AST [123, 124].

Token Copy Mechanism to Address Large Vocabulary of Computer Code Prior works for machine learning on code tended to rename functions and variables to reduce the size of language vocabularies necessary to represent a program. We introduce and study the use of a copy mechanism which can be used by a machine learning system to learn when an arbitrary input token should be copied for use in the output sequence [48]. This output behavior is particularly valuable in conjunction with *abstract buggy context* which provides a concise presentation of tokens which might be useful for output. The copy mechanism is also useful when used with *token context diff* which identifies code change locations specifically by copying context tokens from the input to the output.

Applying Machine Learning to Computer Aided Programming

In this section we summarize the contribution of 4 systems to solve problems in computer aided programming with machine learning. For each system, we highlight the machine learning process contributions that are used and analyzed empirically in each work. All 4 of these systems are provided to the research community through open source contributions.

SequenceR SEQUENCER is a system which uses a sequence-to-sequence model with *copy mechanism* to generate proposed Java code to patch bugs in the input Java methods [48]. The full class surrounding the method is provided as input using our *abstract buggy context* format. The validation component is the test suite from the project which allowed for automated detection of the bug and is used for automatic evaluation of a fix before asking a human to review the proposal. The full process of automatically detecting, localizing, repairing, and validating patches is automated by our R-Hero repair bot. Our full datasets, models, and analysis scripts are provided to the community on GitHub[1].

VRepair VREPAIR is a system which uses a sequence-to-sequence model to generate proposed C code to repair security vulnerabilities in functions [47]. Because vetted examples of before-and-after vulnerability fixes are limited, we improve the model result using *Transfer Learning From Bug Fixes to Security Vulnerability Fixes* and demonstrate its benefit over a state-of-the-art pre-training system based on denoising. We show the ability of the system to generate multi-line changes to repair a vulnerable function using our *token context diff*. Our full datasets, models, and analysis scripts are provided to the community on GitHub[2].

pe-graph2axiom We next create a system which uses a graph-to-sequence machine learning system to output a sequence of rewrite rules which prove two programs represented as complex linear algebra expressions to be semantically equal [123]. The system input uses an *AST representation*

[1] `https://github.com/KTH/chai`

[2] `https://github.com/SteveKommrusch/VRepair`

for model input and produces a *Rewrite Rule Format* for the proof steps which is automatically verifiable. The validation component in this case applies the proposed axioms and checks that the input is correctly transformed, allowing for a system which creates 0% false positives by construction. We implement our system by contributing the gated-graph neural network encoder model to OpenNMT-py; details on how to use the model are provided to the community on GitHub[3].

S4Eq Our final contribution is system which is capable of proving computational basic blocks (straight-line programs) equivalent [124]. For this problem we use an *AST representation* as input to a transformer model which produces verifiable *Rewrite Rule* proof steps. We incrementally train the model using *self-supervised sample selection* and show how this approach improves performance and generalization of the system. We generalize from synthetic programs used for initial training to program sequences derived from GitHub in which we find cases where humans created lexically distinct but semantically equivalent programs. Our full datasets, models, and analysis scripts are provided to the community on GitHub[4].

Outline

The remaining chapters of this dissertation are organized as follows. The background Chapter 2 provides a foundation to the reader on the topics of machine learning, program repair, program equivalence, and discusses original background research on machine learning topics. Chapter 2 also includes formal definitions and a proof of the complexity class for program equivalence in a given language. Chapter 3 dives deeply into the key computer aided programming areas addressed by my research work including introducing and integrating the next 4 chapters, which are all based on published work. Chapter 4 discusses SEQUENCER in detail and shows how a sequence-to-sequence model can be used to generate Java code fixes given buggy Java code as input. Chapter 5 discusses VREPAIR and shows how to train a model which can repair C language

[3]`https://opennmt.net/OpenNMT-py/examples/GGNN.html`

[4]`https://github.com/SteveKommrusch/PrgEq`

security vulnerabilities despite limited data available for training. Chapter 6 details a system using a graph-to-sequence generator which produces a verifiable proof of equivalence between two linear algebra expressions. Chapter 7 builds on the work which we discuss in Chapter 6 to include multi-statement programs mined from GitHub C repositories. Chapter 7 details how the verifiable nature of the program equivalence output from a neural network may itself be used to create new samples to improve the model incrementally. Chapter 8 concludes this thesis and summarizes multiple avenues of future research based upon our contributions.

Part I

Machine Learning on Programs: Background and Our Contributions

Chapter 2

Background on Machine Learning for Computer Aided Programming

As our focus is using machine learning for computer aided programming, this chapter provides background and discusses key prior contributions that lay the foundation on which this thesis builds. As this background will show, many successful approaches to applying machine learning to human language have been shown to work well for generating computer languages as well. Allamanis, *et al.* provide a broad summary of the research in this area in their survey paper [9]. In their survey, the authors note that computer language creates a bridge between humans and computers. Hence, while computer language syntax is more precisely defined than human language, humans writing code tend to certain coding patterns and styles. This creates meaningful statistical distributions at token, loop, function, method, and class levels that can be discovered through machine learning techniques.

This chapter is organized as follows. In Section 2.1 we discuss machine learning background for the contributions made in this thesis with particular focus on sequence generation models. In Section 2.2 we cover topics in computer aided programming including summarizing the problems of program repair and program equivalence. In Section 2.3 we present our original research on 3 machine learning problems which, although not directly computer aided programming systems, provide insights useful for the contributions of this thesis. In Section 2.4 we present formal definitions and an overview of complexity classes for problems of proving program equivalence. Section 2.5 summarizes limitations of prior work which motivates the contributions made by this thesis.

2.1 Machine Learning for Syntax Understanding and Sequence Generation

Machine learning models learn a complex function which maps data from the input domain to the target output domain. The following subsections will describe various models useful for analyzing token sequences along with the mathematics used to produce the output given the input sequence. In all of the cases, training occurs through backpropagation of the error term through the network [189]. That is, the inputs are processed through the mathematical model as described in the section and a result is obtained. During the training process, if the output does not match the expected result, then the internal parameters of the model are adjusted slightly based on the gradient of the error relative to the parameter, *i.e.,* for a given neural network weight parameter w_i and an output error E, the weight is updated as $w_i = w_i - \alpha \frac{\partial E}{\partial w_i}$. The α term is referred to as the learning rate, which may vary as more samples are processed to improve stability [116].

2.1.1 Neural machine translation with sequence-to-sequence learning

Neural machine translation (NMT) evolved from statistical machine translation (SMT). SMT made use of smoothed n-gram models to predict the probabilities of words in a destination language given a source language and neighboring words. NMT, by use of examples and back propagation, uses a neural network to learn the most likely translation for a given input [208].

An early example of a sequence-to-sequence network uses an RNN (recurrent neural network) to read in tokens and generate an output sequence, as shown in Figure 2.1 [208]. In this network, outputs are created from the same neurons that received the inputs. The input tokens are denoted x_t, and after receiving all of the input tokens and a special <EOS> token, the output tokens are fed into the network to aid in proper generation of the next token. The output tokens are denoted y_t. In the following equations, h_t is the hidden state of a recurrent neural network, W^{hx}, W^{hh}, and W^{yh} are weights learnable with supervised learning and backpropagation.

$$h_t = \sigma(W^{hx} x_t + W^{hh} h_{t-1})$$

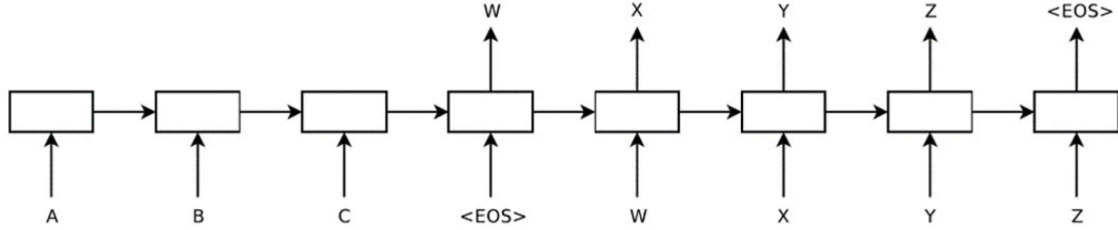


Figure 2.1: Figure from Sequence-to-Sequence paper [208] showing example of early model

$$y_t = W^{yh} h_t$$

A softmax function is then used to turn the y_t values in the preceding equation into probabilities to choose the most likely token from a learned vocabulary. In this early example, one can see how the weight matrices can mimic the learning of n-gram data used in SMT; after processing the input sequence, the hidden state $h_{<eos>}$ encodes the most likely initial token to begin the output and each subsequent h_t can use the W matrices to predict the most likely next token given the input as well as preceding tokens produced on the output. The W matrices can, thus, encode n-gram-like information, but can also learn when to encode longer range likelihoods based on the information in the training data.

2.1.2 Gated recurrent unit

Early neural machine translation architectures made use of Long Short Term Memories (LSTMs) [208], but gated recurrent units were found to train more effectively and produce improved accuracy in some cases [51]. A gated recurrent unit is slightly simpler than an LSTM as it has only 2 gates to learn instead of 3. The feedback is shown in figure Figure 2.2. In the equations below, the notation $[\cdot]_j$ represents the jth element of a vector. x is the input vector to the GRU layer; r is the reset gate; and z is the update gate. W_r, U_r, W_z, U_z, W, and U are all matrices with learnable weights. h_j^t is the hidden state of the jth unit after t iterations of the recurrent equations.

$$r_j = \sigma([W_r x]_j + [U_r h_{t-1}]_j)$$

$$z_j = \sigma([W_z x]_j + [U_z h_{t-1}]_j)$$

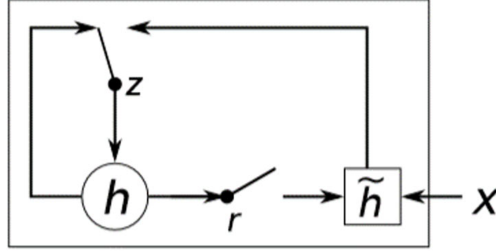


Figure 2.2: Figure from initial GRU paper [51] showing GRU functionality. The update gate z selects whether the hidden state is to be updated. The reset gate r decides whether the previous hidden state is ignored.

$$\tilde{h_j^t} = tanh([Wx]_j + [U(r \odot h_{t-1})]_j)$$

$$h_j^t = z_j h_j^{t-1} + (1 - z_j)\tilde{h_j^t}$$

Papers published recently still may use LSTM or GRU, but the GRU was developed specifically to aid in the problem of learning for neural machine translation.

2.1.3 Get to the Point: Summarization with Pointer-Generator Networks

See *et al.* introduce a new approach to copying information from an input sequence to an output sequence when using a sequence-to-sequence model for natural language processing [197]. A pointer-generator in a neural network model allows the model to 'point' to a specific token on the input sequence that should be copied to the output sequence. In early versions of sequence-to-sequence learning, only tokens that were learned as part of the training vocabulary were available for producing the output [208]. The paper details how a pointer-generator network is useful for text summarization by allowing accurate use in the output of out-of-vocabulary words such as person or place names. In particular, the field of automatically summarizing natural language processing includes both *extractive* approaches where certain key sentences and phrases are copied in full from the source, and *abstractive* approaches which can involve rewording ideas in the input and sound more natural to most readers. The paper notes that abstractive approaches benefit from their copy mechanism, which can use token embedding and encoding information to point to specific tokens that aren't in the language vocabulary but are in the input sequence and should be used at specific

points in the output sequence. In addition to improving natural language models, copying tokens from the input directly to the output also has key advantages for program repair. A known challenge to using sequence-to-sequence models for program repair is the issue that the full vocabulary for source code is nearly infinite due to specific identifier names, numbers, strings, *etc.* [90]. Similar to the summarization problem, in program repair new tokens from the computer language vocabulary may be needed for a bug fix, and copying rare tokens can be used to solve the unlimited vocabulary problem.

See *et al.* did not introduce the copy mechanism for NLP, but their paper has been received by the NLP community as a base on which to build further work. For example, their paper is the basis for the copy mechanism implementation in OpenNMT, a popular open framework for neural machine translation. The main difference between this work and previous work is explicitly computing p_{gen}, the probability for copying a token from the input versus using a token from the vocabulary. The next paragraphs will briefly build up the model to show how p_{gen} is computed and used.

Encoder The encoder is a recurrent neural network using LSTM gates [97] to process the input. It is a bidirectional encoder [196] that allows the encoding for a token to incorporate information from tokens both before and after its occurrence in the input data. The encoder converts the source sequence $X = [x_1, ..., x_n]$ into a sequence of encoder hidden states h_i using a learnable recurrence function f_e. After reading the last token, the last hidden state, h_n^e is used as the context vector c for initializing the decoder [51]:

$$h_i^e = f_e(x_i, h_{i-1}^e); \tag{2.1}$$

Decoder The decoder is also a recurrent neural network using LSTM gates. When initialized by the encoder, it begins production of the output sequence by receiving the special *start* token as input y_0. For each previous output token y_{j-1}, the decoder updates its hidden state h_j^d using the learnable recurrence function f_d [51]:

$$h_j^d = f_d(y_{j-1}, h_{j-1}^d, c) \tag{2.2}$$

The decoder states h_j^d are used for token generation by the attention and copy mechanisms in Equation 2.4 and Equation 2.5. The model stops updating decoder hidden states and generating new tokens when the last token generated by the model is a special end-of-sequence token.

Attention The attention mechanism provides a way to create a context vector c_j for each decoder output token y_j using a linear combination of the hidden encoder states h_i^e [21]:

$$c_j = \sum_{i=1}^{n} \alpha_i^j h_i^e \tag{2.3}$$

Where α_i^j represents attention weights computed with a learnable function based on the relation between the encoder hidden state and the decoder hidden state: $\alpha_i^j = \text{softmax}(f_\alpha(h_i^e, h_{j-1}^d))$. This context vector c_j is used by a learnable function f_a to allow each output token y_j to pay "*attention*" to different encoder hidden states when predicting a token from the vocabulary V:

$$P_V(y_j \mid y_{j-1}, y_{j-2}, ..., y_0, c_j) = f_a(h_j^d, y_{j-1}, c_j) \tag{2.4}$$

Copy The copy mechanism further adjusts Equation 2.4 to produce a token candidate by introducing p_{gen}, the probability that the decoder generates a token from its initial vocabulary. Hence, $1 - p_{gen}$ is the probability to copy a token from input tokens depending on the attention vector α^j in Equation 2.3 [197]:

$$p_{gen} = f_c(h_j^d, y_{t-1}, c_j) \tag{2.5}$$

$$P(y_j) = p_{gen} P_V(y_j) + (1 - p_{gen}) \sum_{i:x_i = y_j} a_i^j \tag{2.6}$$

f_c in Equation 2.5 is learnable function. Using Equation 2.6, the output token y_j for the current decoder state is selected from the set of all tokens that are either: 1. tokens in the training vocabulary (including the <unk> token) or 2. tokens in the input sequence.

In addition to the copy mechanism, See *et al.* introduce a technique for limiting repetition of output sequences. This technique is important for the text summarization use case they envi-

sion, but is less relevant for source code. Reusing the same variable multiple times in a line may sometimes be appropriate (i.e. "if (x < 8) && (x > 2)").

2.1.4 Graph neural networks

Researchers have applied graph analysis to human language generation models, and graphical representations of code are common in compiler research. The key initial paper relating to graph neural networks was written in 2009 [193].

In the 2009 paper, the graph neural network is described as an iterative encoding network. The network uses as input labels for nodes and edges, the labels having dimensions d_N and d_E respectively. The labels attached to node n are denoted $l_n \in \mathbb{R}^{d_N}$; the labels attached to edge (n_1, n_2) are denoted $l_{n_1,n_2} \in \mathbb{R}^{d_E}$. Additionally, $l_{co[n]}$ and $l_{ne[n]}$ are the labels for edges connected to node n and labels of nodes connected by edges to node n, respectively.

Figure 2.3 shows diagramatically how the input information is used in a GNN. x_n is the hidden state for node n, and o_n is the output of node n. These are computed by iterating on the equations below:

$$x_n(t+1) = f_w(l_n, l_{co[n]}, x_{ne[n]}(t), l_{ne[n]})$$

$$\forall n \in N, o_n(t) = g_w(x_n(t), l_n)$$

The outputs o_n of the network allow for training samples to be used for setting the weights in the functions f_w and g_w. Typically $x_n(0)$ is initialized to 0 for all nodes. The 2009 paper discusses the constraints on the learnable function f_W that ensures $x_n(t)$ converges over some finite number of steps. The function f_W is implemented as a different matrix for each edge type, so the number of parameters to learn in a GNN grows linearly with the number of edge types. The hidden state of all connected nodes is processed through each edge matrix to create the final output of f_W. As can be surmised, the number of iteration steps is equal to the hop distance to the furthest node in the graph that can affect another node.

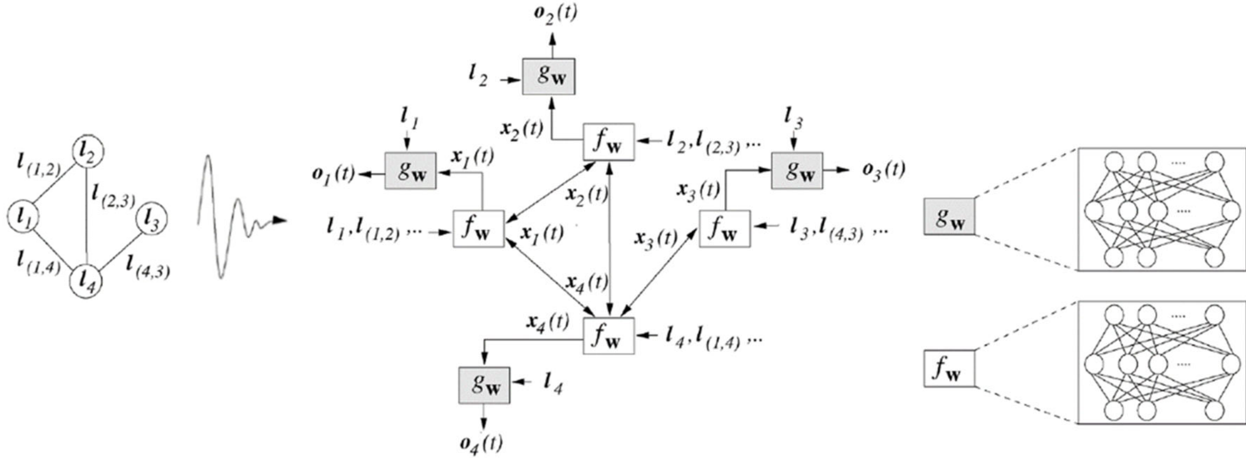


Figure 2.3: Figure from initial graph neural network paper [193]. Using label values for nodes and edges, a learned function f_W is iterated on at each node and ultimately used to produce $o_n(t)$.

As presented in Figure 2.3, the GNN is producing one output per node. Another use case for a GNN is to do a softmax function on o_n, which allows a node to be selected as an output. Alternately, summing all o_n together can produce a usable single output for a graph.

2.1.5 Graph-to-Sequence Learning using Gated Graph Neural Networks

Sequence-to-sequence models for code generation have been augmented to include some AST information [43], but the rich information available statically during program analysis would benefit from a graph neural network. Beck *et al.*, like the Pointer paper, discuss natural language processing, and the concept introduced invites use for code analysis [29]. Beck *et al.* use a gated graph neural network to analyze an input string after the string is transformed into an Abstract Meaning Representation (AMR) graph. To minimize the number parameters in the graph neural network model due to edge types, they discuss transforming the edges in the graph to be extra vertices through a Levi graph transformation. For code analysis, a Levi transformation could be valuable - it allows for a wider variety of edge types to be represented with a reasonable number of parameters.

Figure 2.4 shows an overview of their approach. In a sequence-to-sequence model with attention, as described in Section 2.1.3, the attention is computed using the encoder hidden states for each received token. The encoder state includes information from the embedding of the input

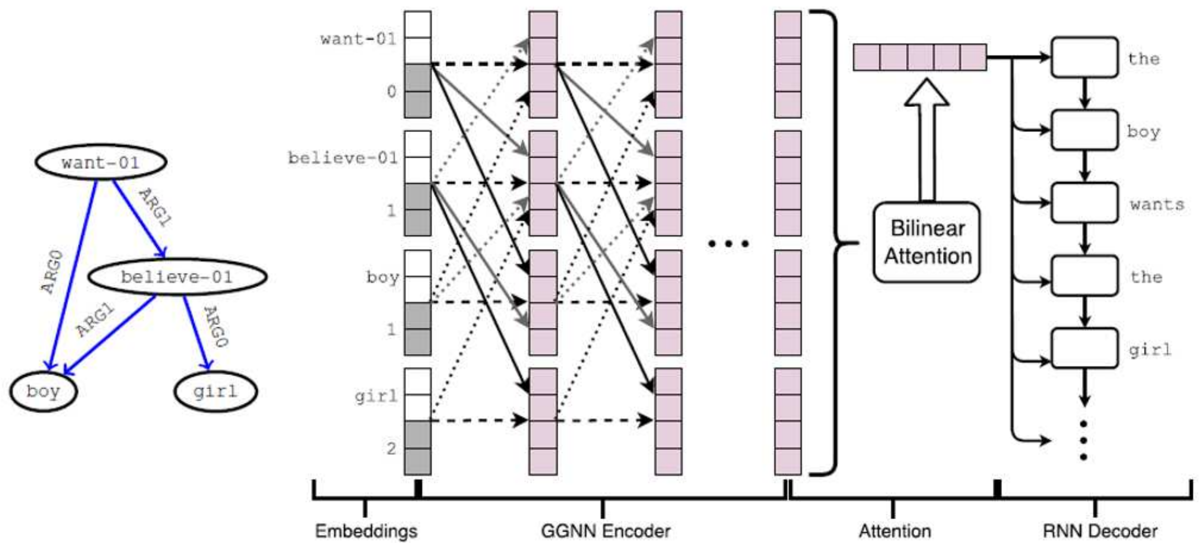


Figure 2.4: Figure from Graph-to-Sequence paper [29] showing encoder and attention mechanism feeding into sequence generation

token, as well as a function of the encoder states before and after this token. In a graph neural network, the attention is computed using the node hidden states after the network has iterated on the structure. Since the nodes are initialized with the token inputs, the node hidden states can include information from an embedding of the input, as well as any other nodes that can be reached during network iteration. An aggregation of the final node states can be used to initialize the hidden state of the sequence decoder.

The transformation of the input AMR graph into a Levi graph allowed for fewer edge types in the model. The paper shows that for optimal performance, the Levi network had these types of edges: self edges, forward and reverse token sequence edges, and forward and reverse tree connection edges. This results in five edge types, each of which has a weight matrix to compute how the gated recurrent unites (GRUs) update each iteration step.

As is common for reporting machine translation results, the authors evaluate results using BLEU scores (BiLingual Evaluation Understudy). The BLEU score was proposed by Kishore Papineni *et al.* in their 2002 paper "BLEU: a Method for Automatic Evaluation of Machine Translation" [172] and scores candidate translations based on a reference translation. Their test set contains both english-to-german and english-to-czeck translation tasks. In their results section they

show the highest BLEU scores for the models tested, including the same test set tested with recent state-of-the-art approaches. Interestingly, they also include results for the ChrF++ scoring method, for which their approach does not score the best. They note that ChrF++ scores have been found to align better with human translation scoring than BLEU and leave the challenge of improving ChrF++ scores to future work. The merits of BLUE versus ChrF++ methods for language output scoring are not directly related to machine learning for code sequence generation. The scores can help alert code generation researchers to new improvements in sequence generators, but the methods tend to relate best to human languages.

2.1.6 Transformer Networks Use Attention Layers

Sequence-to-sequence (seq2seq) learning has become widely successful on many applications such as automated translation [236], text summarization [167] and other tasks related to natural language. The transformer model [220] is a powerful and versatile sequence-to-sequence model, used by GPT-3 [36] as well as showing strong results for source code summarization [3]. As an extension of the original seq2seq model which we introduce in Section 2.1.1, modern seq2seq models generally consist of two parts, an encoder and a decoder. The encoder maps the input sequence $X = (x_0, x_1, ..., x_n)$ to an intermediate continuous representation $H = (h_0, h_1, ..., h_n)$. Then, given H, the decoder generates the output sequence $Y = (y_0, y_1, ..., y_m)$. Note that the size of the input and output sequences, n and m, can be different. A seq2seq model is optimized on a training dataset to maximize the conditional probability of $p(Y \mid X)$, which is equivalent to:

$$\begin{aligned} p(Y \mid X) &= p(y_0, y_1, ..., y_m \mid x_0, x_1, ..., x_n) \\ &= \prod_{i=0}^{m} p(y_i \mid H, y_0, y_1, ..., y_{i-1}) \end{aligned}$$

Prior work has shown that source code has token and phrase patterns that are as natural as human language [95], and thus techniques used in natural language processing can work on source code as well, including seq2seq learning [48].

In our work, we use a variant of a seq2seq model called "Transformer" [220], which is considered the state-of-the-art architecture for seq2seq learning. The main improvement introduced by Transformers is the usage of self-attention. The number of operations required to propagate information between two tokens in seq2seq learning based on recurrent neural networks grows linearly with the distance, while in the Transformer architecture it is a constant time operation with the help of self-attention. Self-attention updates all hidden states simultaneously by allowing all tokens to attend to all previous hidden states. Since there are no dependencies between tokens, this operation can be done in parallel without recurrence. This also helps to mitigate the issue of long term dependencies which can be a problem for recurrent neural networks. Self attention can be described as the process of mapping a query and key-value pairs to an output. Query, key, and value are concepts borrowed from information retrieval systems, where the search engine maps a query against keys and returns values. In a Transformer model, Query (Q), key (K), and value (V) are vectors of the same dimension d_k computed for each input. Mathematically, the matching of Query to Key is done with a dot product and the result is used to scale the Value selected:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

The Transformer model is trained with multiple attention functions (called multi-head attention) which allows each attention function to attend to different learned representations of the input. Since the Transformer model computes all hidden states in parallel, it has no information about the relative or absolute position of the input. Therefore the Transformer adds positional embedding to the input embeddings, which is a vector representing the position.

The encoder of a Transformer has several layers, each layer having two sub-layers. The first sub-layer is a multi-head self-attention layer, and the second sub-layer is a feed forward neural network. The outputs from both sub-layers are normalized using layer normalization together with residual connections around each sub-layer.

The decoder also has several layers, each layer has three sub-layers. Two of the decoder sub-layers are similar to the two sub-layers in the encoder layer, but there is one more sub-layer which

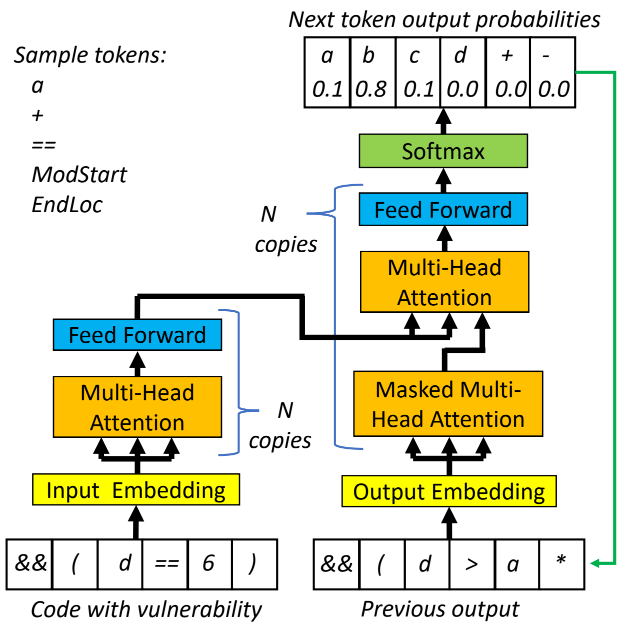


Figure 2.5: Code Generation Neural Architecture based on Transformers

is a multi-head self-attention over the output of the encoder. A copy attention mechanism, as discussed in Section 2.1.3, can be used in conjunction with a Transformer in order to support data copies from input to output. The Transformer architecture we utilize for our work in this thesis is shown in Figure 2.5 and discussed in further detail in subsection 5.3.7 and subsection 7.3.2.

The Transformer model shown in Figure 2.5 includes example input and output tokens which represent computer code. The Transformer for learns to generate outputs (in this example, corrected computer code) by first receiving as input the code with a defect. Multiple copies of multi-head attention layers learn hidden representations of the input data. These representations are then used by multiple copies of a second set of multi-head attention layers to produce a table of probabilities for the most likely token to output. The first token to output is based solely on the hidden representations the model has learned to produce from the input code. As tokens are output they are available as input to the model so it can adjust the next token probabilities correctly. For example, in Figure 2.5, after the sequence of tokens '`&& ( d > a *`' has been output, the model predicts that the next token should be '`b`' with a probability of 0.8.

Libraries The Transformer model we use in our research is implemented in Python using state-of-the-art tools. Once the input code is processed, OpenNMT-py is used to train the core Transformer model [117].

2.2 Computer Aided Programming

In this section we present background work related to computer analysis and generation of computer programs. Works describing attempts to address these concerns without machine learning are presented as well as some initial works which do use machine learning but have weaknesses which we aim to address with the work presented in this thesis.

2.2.1 Staged Program Repair with Condition Synthesis

Long *et al.* introduce staged program repair (SPR), which creates a search space of potential bug fixes in C code [143]. The repair attempts are based on pre-defined parameterized transformation schemas that combine with multiple ways to synthesize changes to conditional statements. The approach to evaluating the schemas allows for a relatively efficient pruning of the search space that improved performance over prior work. There are 6 specific schemas discussed in the paper, which can be useful as a baseline for evaluating machine learning techniques for program repair and synthesis work to see which types of repairs can be found. This paper from 2015 is particularly interesting because it discusses the algorithmic implementation of many program ideas that are today starting to be explored using machine learning techniques.

Before exploring repair schemas, SPR does error localization by finding blocks of code that are often executed for failing test cases but rarely executed for passing test cases. Given the suspected faulty code blocks, SPR will then search for a successful repair by performing code transformations and rerunning the passing and failing tests. The order SPR searches the repair space is a key contribution of this work as it decreases the search time. For example, the first schema evaluated is to change only a branch condition (*e.g.,* tighten and loosen a condition). Further details are in the paper, but a summary of the 6 schemas that SPR uses to explore code transformations are:

- **Condition Refinement:** Given a target *if* statement, SPR transforms the condition of the *if* statement by conjoining or disjoining an abstract condition to the original *if* condition.

- **Condition Introduction:** Given a target statement, SPR transforms the program so that the statement executes only if an abstract condition is true.

- **Conditional Control Flow Introduction:** SPR inserts a new control flow statement (return, break, or goto an existing label) that executes only if an abstract condition is true.

- **Insert Initialization:** For each identified statement, SPR generates repairs that insert a memory initialization statement before the identified statement.

- **Value Replacement:** For each identified statement, SPR generates repairs that replace either 1) one variable with another, 2) an invoked function with another, or 3) a constant with another constant.

- **Copy and Replace:** For each identified statement, SPR generates repairs that copy an existing statement to the program point before the identified statement and then apply a Value Replacement transformation.

Three of the six transformations involve adding an abstract condition. An abstract condition *abstract_cond()* can be added to an existing *if* statement by adding '*&& abstract_cond()*' or '|| *abstract_cond()*' to the statement. The condition to add is generated by creating traces of passing and failing tests that track values of different variables for each case and a new condition is searched for that causes the failing cases to pass.

Listing 2.1 is an example of a one-line patch found by SPR. The failure results because the body of the *if* statement did not execute when it should have. An *abstract_cond()* was added to the condition, then concretized to produce the correct fix.

```
- if( isostr_len ) {
+ if( isostr_len  || ( isostr  != 0)) {
```

Listing 2.1: Patch that uses **Condition Refinement** schema to correct *if* statement

A strength of the SPR paper is its discussion on the balance between plausible (passing all tests) and correct patches. Out of 38 plausible patches, 11 are correct, which is a 29% correct/plausible ratio. Other papers have cited lower ratios for correct/plausible; a careful analysis of prior techniques (GenProg, RSRepair, and AE) shows that they have correct/plausible ratios of less than 12% [181]. Given that the authors compare directly against GenProg and AE, their relatively high ratio implies that the repair schemas they use represent reasonable transformations.

Ultimately, the authors compare their results to two other well-cited repair programs (GenProg and AE). On the same benchmark set, SPR was able to fix five times as many defects as the prior art, representing a significant advance.

A weakness of the approach in the SPR paper is that it cannot effectively apply more than one transformation to attempt a patch. As Chapter 4 will show, this is an area that can be addressed by machine learning for patch generation.

2.2.2 PHOG: Probabilistic Model for Code

Beilik *et al.* create a statistical model of code that can be used for code generation (including code completion, patch generation, and programming language translation) [34]. The model is based on their domain-specific TCOND language which allows for grammar production rules to be context dependent. The probabilities for each contextualized production rule are computed from the training data and evaluated based on how well AST nodes in the test data could be predicted.

The paper builds up ideas based on context-free grammars (CFGs) [15], which include production rules that define how non-terminals can be transformed. Some examples to show the format are:

- $\langle expr \rangle \rightarrow number$
- $\langle expr \rangle \rightarrow \langle expr \rangle + \langle expr \rangle$
- $\langle expr \rangle \rightarrow \langle expr \rangle - \langle expr \rangle$

Table 2.1: Evaluation of prediction for AST nodes in JavaScript

Model	Error Rate
Non-Terminals	
PCFG	48.5%
3-Gram	30.8%
10-Gram	35.6%
PHOG	25.9%
Terminals	
PCFG	49.9%
3-Gram	28.7%
10-Gram	29.0%
PHOG	18.5%

After discussing CFGs, the concept of a high order grammar (HOG) is introduced, which allows for production rules to be based on contexts such that $\alpha[\gamma] \rightarrow \beta$ represents a rule transforming the non-terminal α with context γ into β, where β can be a terminal or non-terminal of the grammar. A context is created by analyzing the program up to the production rule use point and might include statement types or local variable names. For example: $\langle expr\rangle[return] \rightarrow True$ could represent the rule that $\langle expr\rangle$ expands to $True$ when the expression is in a return statement.

The paper defines a PHOG as: a probabilistic high order grammar is a tuple (G, q) where G is a HOG (high order grammar) and $q : R \rightarrow \mathbb{R}^+$ scores rules such that they form a probability distribution. G includes a set of non-terminals N and a conditioning set C (*i.e.,* the contexts). The probability distribution is computed as:

$$\forall \alpha \in N, \gamma \in C : \sum_{\beta : \alpha[\gamma] \rightarrow \beta \in R} q(\alpha[\gamma] \rightarrow \beta) = 1$$

The function q is learned by counting rule expansions observed in a set of training data, and the authors use smoothing techniques to address sparseness in the training data. This is a straightforward technique to learn probabilities. Machine learning approaches, which can learn similar distributional information, are not as easily understood an the technique used for PHOG.

To evaluate PHOG, the authors predict JavaScript elements using PHOG and 3 alternate techniques. The prediction test is done by deleting a node from an AST (and its subtree and all nodes to the right) and querying the model to identify the missing node. Their alternate techniques are a PCFG (probabilistic context-free grammar) that only conditions on the parent non-terminal (no context used), and a 3-gram and 10-gram model (an n-gram model conditions on the n-1 previous symbols in the AST traversal as used in Allamanis *et al.* [10]). Table 2.1 shows their results on predicting both terminal and non-terminal elements. Their results are very strong in this analysis relative to previous techniques; but Chapter 4 will show a more flexible approach using a machine learning technique for patch generation.

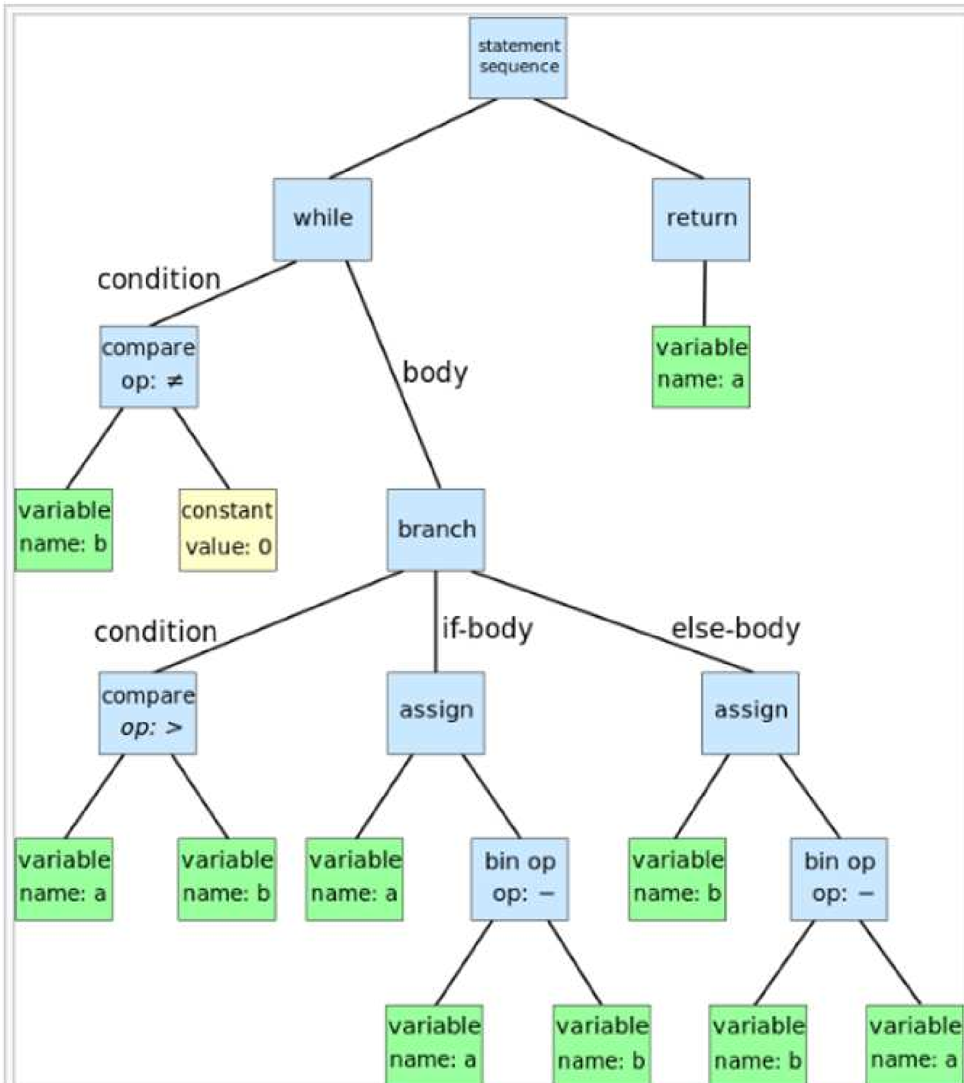
2.2.3 Abstract Syntax Trees

Human languages can be parsed and broken down into clauses and parts of speech, but this process is not mathematically precise due to exceptions and nuances in human language [63]. Computer languages, on the other hand, are designed to be automatically and predictably parsed.

Figure 2.6 shows a typical abstract syntax tree (AST) for a short code snippet [233]. The tree is abstract in that certain syntax (such as parenthesis) are not necessary, but given the AST for a code snippet, equivalent code can be reconstructed. The tree provides a structure for identifying the way in which control statements and variables are used in the program, and Section 2.2.4 will show how it can be useful as an input to machine learning approaches on code.

2.2.4 code2vec: Learning Distributed Representations of Code

Alon *et al.* describe an approach to create an embedding vector for entire Java methods [12] in a way similar to the widely successful word2vec approach used in NLP [159]. The example use case is to predict method names, but the paper aims to produce an embedding that can be used for a variety of cases. Indeed, this approach can be directly applied to the problem of program equivalence. The paper presents cases where similar methods have similar embeddings and 'adding' the embedding from one program can have meaningful results in the method name predicted. In this way, a possible future application for code2vec would be test two programs for equivalence



An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b ≠ 0
    if a > b
        a := a − b
    else
        b := b − a
return a
```

Figure 2.6: Wikipedia image of an abstract syntax tree for a short code snippet

by subtracting their embeddings and the resulting vector could be used to seed a sequence decoder and create a description of the program differences.

The paper builds up d-dimensional embedding for a piece of code (in their example use case they are looking at methods). The embedding is built up using weighted summations of embeddings for path-contexts. A path-context is a sample from the AST for the code that includes a start terminal, the non-terminals from the AST, and the end terminal. In theory, a snippet of code with n terminals in the AST has n^2 path-contexts, but the authors set AST distance constraints on the path-contexts allowed and also have found that an upper bound of 200 path-contexts is sufficient to represent most code correctly. Like the popular word2vec, embeddings are learned during model training for the terminals and AST paths. The embeddings for the set of all terminal symbols are collected into $value_vocab$, and the embeddings for all of the paths seen in the training set are collected into $path_vocab$. Hence, given a path-context that starts at terminal x_s and terminates at x_t following a path p_j through the AST, the path-context is mathematically represented as:

$$c_i = embedding(\langle x_s, p_j, x_t \rangle) = [value_vocab_s; path_vocab_j; value_vocab_t] \in \mathbb{R}^{3d}$$

Figure 2.7 shows an example of path-contexts. For example, the path-context for the red path labeled ① in the figure is the embedding for:

⟨elements,(Name ↑ FieldAccess ↑ Foreach ↓ Block ↓ IfStmt ↓ Block ↓ Return ↓ BooleanExpr),true⟩

Using the path-context c_i and a trainable weight matrix $W \in \mathbb{R}^{d \times 3d}$, the *combined context vector* $\tilde{c_i}$ is:

$$\tilde{c_i} = tanh(W \cdot c_i)$$

A trainable global attention vector $a \in \mathbb{R}^d$ is used in a softmax function to compute attention weights α_i based on the path context embeddings:

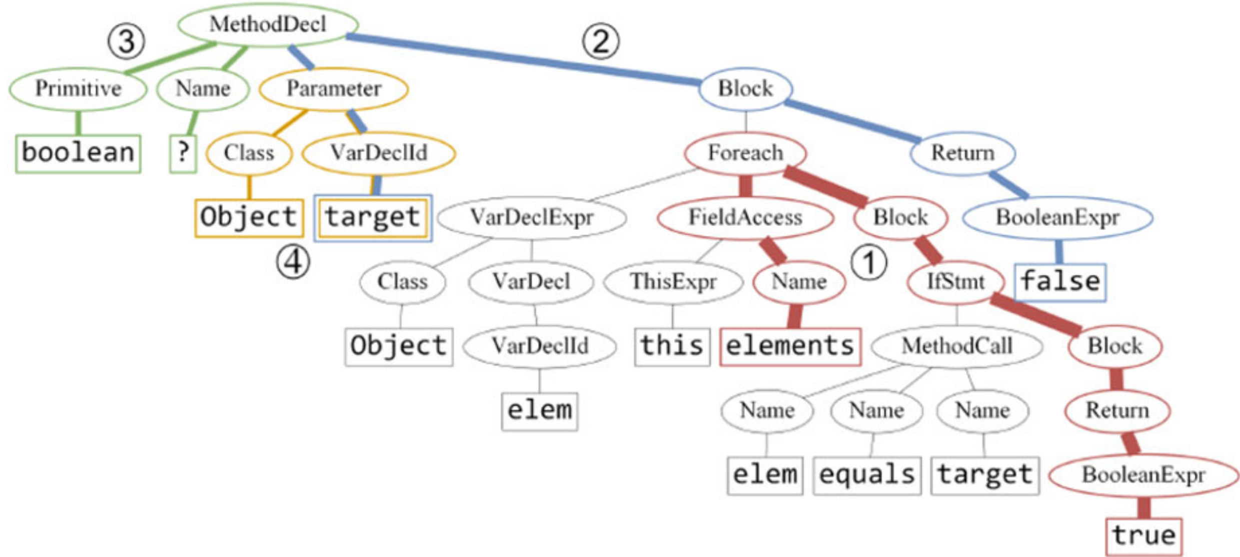


Figure 2.7: Figure from code2vec paper [12] showing different paths through AST. The top-4 attended paths learned by the model are shown in red, blue, green, and yellow. Path width is proportional to the attention given.

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

Ultimately, the whole code snippet is represented by an aggregated code vector $v \in \mathbb{R}^d$:

$$v = \sum_{i=1}^{n} \alpha_i \tilde{c_i}$$

A strength of the code2vec paper is that it teaches a way to visualize the parts of the program that are being used to determine the method name, which is a helpful way to improve understanding of the neural network. Since understanding what a network has learned is a known problem in machine learning, this attention visualization is valuable. Figure 2.7 shows the various weights of the top 4 paths used in computing v for the snippet based on the thickness of lines. Figure 2.8 diagrams the full use model of the code2vec paper. The original code, with '?' for the method name is shown, along with the top 4 weighted paths used to predict the method name 'count'.

A weakness of the code2vec paper is that it learns on full tokens (not subtokens) and has a vocabulary limitation. It cannot learn embeddings for novel variable names. Also, the embeddings it does learn are based on the training set provided, hence the embedding may not be appropriate for

```
int f(String target, ArrayList<String> array) {
    int count = 0;
    for (String str : array) {
        if (target.equals(str)) {
            count++;
        }
    }
    return count;
}
```

Predictions:

count		42.77%
countOccurrences		33.74%
indexOf		8.86%

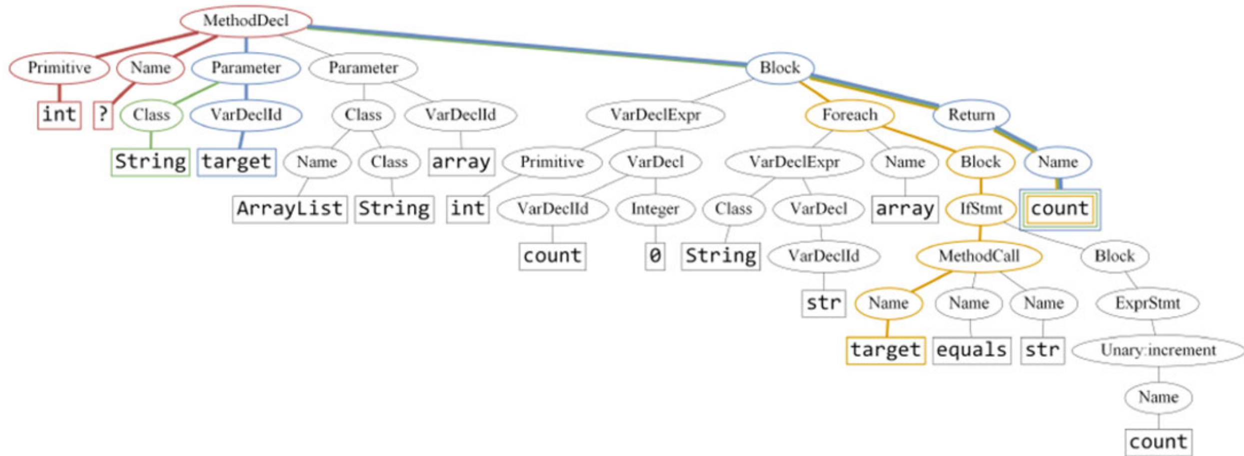


Figure 2.8: Figure from code2vec paper [12] showing full use model from code snippet to method name prediction

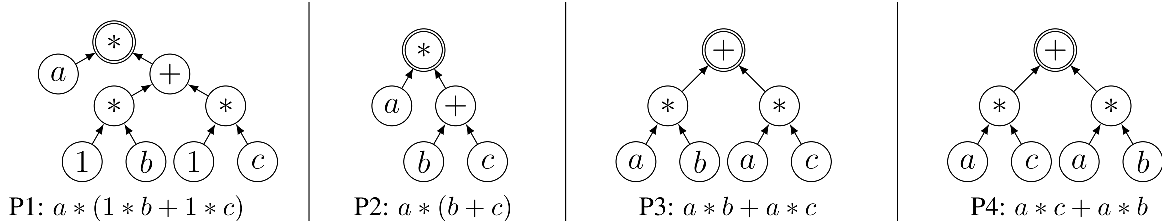


Figure 2.9: Examples of Computations Shown as Symbolic Expressions and Dataflow Graphs

how a variable is used in a given method. For example, a method that iterates over an array using i and accumulates the sum of array elements into sum is semantically equivalent to a method that uses j for the iterator and i for the accumulation. But the somewhat odd use of i for accumulation would result in a shifted code vector for the method.

2.2.5 Program repair

Program repair is the problem of finding a patch to a buggy input program in order to address a bug. In the survey paper "Automatic Software Repair: a Bibliography" [161], Monperrus discusses several kinds of repair. First, the paper discusses behavioral repair where test-suites, contracts, models, or crashing inputs are taken as test oracles to determine software quality. Second, it discusses state repair, also known as runtime repair or runtime recovery, with techniques such as checkpoint and restart, reconfiguration, and invariant restoration. The paper spans the research communities that contribute to this body of knowledge: software engineering, dependability, operating systems, programming languages, and security. It provides a novel and structured overview of the diversity of bug oracles and repair operators used in the literature.

2.2.6 Program equivalence

The problem of proving program equivalence is one of the earliest problems in computer science [101]. The problem is to determine when two programs with different textual representations have the same semantics. In particular, proving that for all program inputs the two programs produce identical output. The applications for program equivalence checking include: 1. validation of any algorithm that does program transformations (such as compilers) 2. plagiarism detection

(useful in MOOCs *etc.*) 3. formal verification when refactoring code (for readability, security automation, *etc.*) 4. virus and other malware detection by detecting similar code sequences.. Since the problem of program equivalence can range from undecidable [79], to trivial in the case of testing the equivalence of a program with itself, new techniques for program equivalence can improve software quality and reduce development time.

In order to reason precisely about programs, symbolic expressions and other program constructs can be represented as abstract syntax trees, as we discuss in Section 2.2.3. Figure 2.9 shows an example of 4 semantically equivalent computations along with their dataflow graphs. A challenge addressed by this thesis is how to formally prove such programs equivalent.

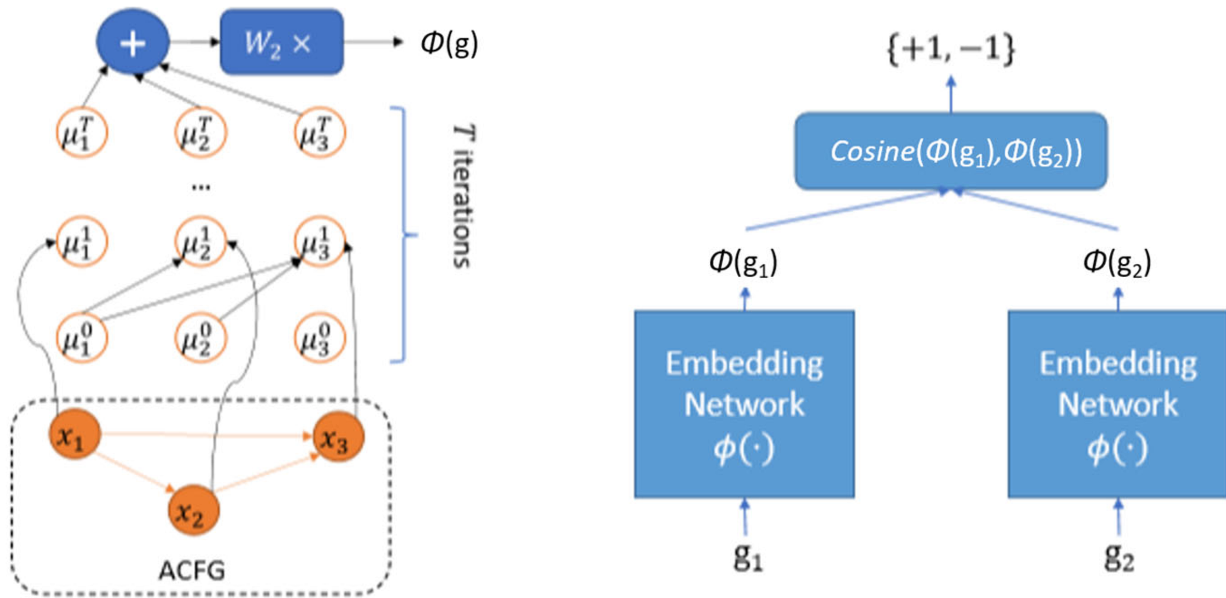
2.2.7 Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection

Xu *et al.* use graph neural networks to detect binary code similarity [241]. Like the code2vec paper (which is more recent), Xu *et al.* learn and embedding function on code and the use model is to compare embedding functions to detect code similarity for vulnerability detection. They refer to their approach as Structure2vec.

The Structure2vec approach starts with an ACFG (*attributed control flow graph*), in which each node represents a basic block and includes attributes such as 'number of calls', 'numeric constants', 'number of offspring', and five other easily computed attributes. After initializing and iterating the graph neural network for T iterations, the embedding for the graph $\phi(g)$ is computed using a learned matrix W to combine the final hidden state $\mu_v^{(T)}$ of each vertex:

$$\phi(g) = W \cdot \sum_{v \in V} \mu_v^{(T)}$$

Figure 2.10 diagrams the approach for detecting code similarity. Subfigure (a) shows how after T iterations of the graph neural network, $\phi(g)$ is created. The Siamese architecture shows how the



(a) Graph Embedding Network Overview **(b) Siamese Architecture**

Figure 2.10: Figures from Code Similarity paper [241] showing generation and use of $\phi(g)$

difference between two code embeddings is computed using the cosine of two multidimensional vectors.

A disadvantage of this approach is that the cosine function tends to limit each dimension of the embedding to a gradient representation of some feature. If the cosine function were replaced by a 2-layer neural network, then mappings from the program embedding to the equivalence value could account for disjoint but similar areas in the embedding spaces.

2.3 Preliminary Studies on Machine Learning Concepts and Limitations

In this section we will explore 3 topics in machine learning with our original research. These 3 subsections are outside the field of program analysis, and hence are not necessary reading for understanding the work presented in this thesis. But they provide some insights into machine learning concepts which inform certain decisions taken in our primary research area.

In Section 2.3.1 we explore the limits of neural networks when attempting to learn pseudo-random datasets such as this study on finding a hash function on binary data. As neural networks excel at finding patterns in data and generalizing an output to these patterns, when no discernible pattern is available a neural network is likely not the best approach. As shown in Parts II and III of this thesis, the general problem of machine learning on code does have patterns and relations which a machine learning system can help address.

In Section 2.3.2 we explore the effects of network size (number of neurons and number of layers) on the ability of a model to avoid overfitting and generalize well on the dataset. We study the problem of generalizing on 3D lung nodule data using a novel scoring metric based on distributional differences of the shape characteristics. We find that a network that is too small may not have sufficient learnable parameters to fully generalize to the data, while a network that is too large may overfit the training data that is used to learn the parameters but then may generalize poorly beyond the training data.

In Section 2.3.3 we use a well-defined problem to explore the ability of a system to learn to solve multiple goals with sparse reward information. The system studied is a hand/eye coordination task within a 2D grid world in which hand and eye movements can be used to accomplish a goal. The goal is defined as getting the hand or eye to a given position, or visually seeing the hand at a given position in the visual field. This problem relates to our problem in this thesis of proving program equivalence. In the case of program equivalence, rewrite rules are actions instead of hand/eye movements and a target equivalent program is the goal instead of a hand/eye position. We find that the concept of hindsight experience replay is very valuable for solving this hand/eye problem and we leverage that finding into our work on program equivalence which we present in Chapter 7.

2.3.1 Multilayer Perceptron (Vanilla) Neural Network Models And the Challenge of Learning Hash Functions

The simplest form of neural network is a fully-connected multilayer perceptron, sometimes called a 'vanilla' nueral network. We now explore an attempted use case of such a network which introduces neural network concepts and demonstrates some limitations of neural networks. This section will also cover techniques which could be used to transform a neural network which has learned a symbolic equation back into such equation.

The problem we explore is to reverse-engineer the Cache/Home Address (CHA) mapping for the L3 cache of the Intel Xeon-Phi. Successfully reverse-engineering the mapping could allow software to map data in the large L3 such that they are close to the CPU core executing code which needs the data. The neural network sample data for this mapping has as input 29 bits of the address which are used to map an address location to one of 36 CHA locations.

Table 2.2 shows 3 fully connected neural network models of varying sizes which were trained on the CHA mapping problem. We found in earlier experiments that classifying each 29 bit address input into one of 36 CHA classes did not work well, so we surmised that training a different model per CHA bit may be productive (6 CHA bits can represent 36 possible CHA identifiers). We also found that having the network produce 2 classes per bit (the bit 'high' class and the bit 'low' class) yielded better results than a single binary output. Our expectation here is that having a 2 bit output approximates one more layer in the network and hence allowed some more learning opportunities for the network. We trained on 384,000 samples for 500,000 iteration steps and tested on a separate 128,000 samples. As the table shows, all 3 network sizes learned CHA bit 0 well as shown by the Test Error scores of 0.04% or less for this bit, but none of the networks learned CHA bit 4 well as shown by the Test Error scores of over 48% on this bit. Of course random prediction for any bit will result in 50% error rate, so 48% error is quite poor. From this observation we surmise that the function for CHA bit 4 is relatively chaotic in the same sense as functions which Bahi *et al.* demonstrated are difficult for a neural network to learn in their work studying neural networks and chaos [22].

Table 2.2: Training and test results as vanilla network size varies

Input	L1	L2	L3	L4	L5	Output	CHA bit	Training Error	Test Error
	Number of elements in layer								
29	256	256	64	64	64	2	0	0.00%	0.03%
							1	43.26%	49.98%
							2	37.20%	49.67%
							3	6.35%	17.36%
							4	38.86%	48.94%
							5	13.84%	23.13%
29	512	256	64	64	64	2	0	0.00%	0.04%
							1	0.00%	0.04%
							2	0.05%	5.01%
							3	2.03%	16.85%
							4	13.07%	48.94%
							5	6.26%	25.68%
29	1024	512	256	128	64	2	0	0.00%	0.04%
							1	0.88%	49.77%
							2	0.00%	2.96%
							3	0.25%	48.82%
							4	0.80%	49.33%
							5	0.15%	22.37%

Outside the scope of this thesis, we developed and published a process which did not use machine learning to decode the CHA mapping [125]. The binary functions which compute CHA bits 0 and 4 are shown in Figure 2.11. As we can see, CHA bit 0 has several layers of exclusive-or functions and a few logical and, or, and invert functions. Despite this moderate Boolean complexity, all 3 models in Table 2.2 were able to learn this function well. However, CHA bit 4 uses a complex function g to select one of 2 other complex functions f and h to produce the final output. This layering of complex functions was not learned well by any of our 3 models.

Had our machine learning model succeeded in learning the CHA decoding, we can see that the number of neurons and layers in our models would make efficient computation challenging. It would be of interest for practical usage to successfully transfer learned neural network models into more direct discrete computations. Figure 2.12 illustrates how this process could be attempted for a neural network trained on a Boolean function. In this figure, training has settled the weights

$$\text{CHA}_0 = a_6 \oplus a_8 \oplus a_9 \oplus a_{10} \oplus a_{14} \oplus a_{15} \oplus a_{17} \oplus a_{18} \oplus a_{20} \oplus a_{23} \oplus a_{27} \oplus ((a_{30}a_{31})|(\overline{a_{30}a_{31}}(a_{32}|a_{33})))$$

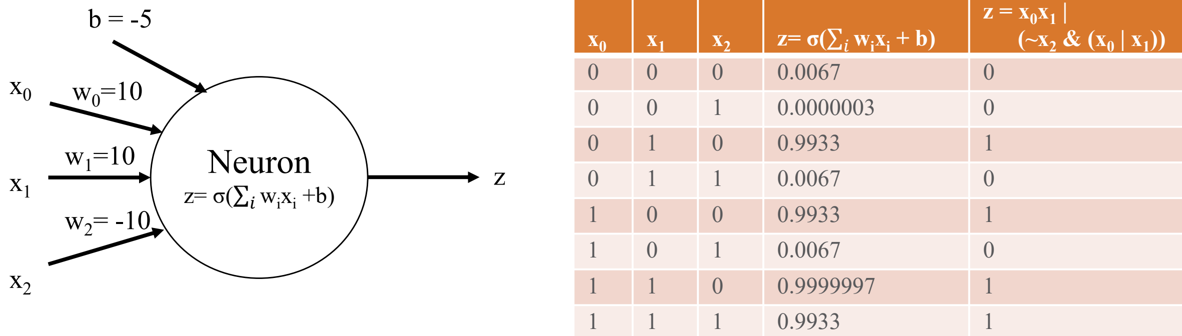
$\text{CHA}_4 = f\overline{g}|gh$, where:

$$f = a_6 \oplus a_{11} \oplus a_{12} \oplus a_{16} \oplus a_{18} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{26} \oplus a_{30} \oplus a_{31} \oplus a_{32}$$

$$\begin{aligned} g = {} & ((a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31}) \ | \\ & \quad (a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34})) \\ & (\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus a_{31} \oplus a_{32} \oplus a_{33}) \\ & (a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34}) \end{aligned}$$

$$\begin{aligned} h = {} & (\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus a_{31} \oplus a_{32} \oplus a_{33}) \\ & (\overline{a_6} \oplus a_{12} \oplus a_{13} \oplus a_{14} \oplus a_{18} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{26} \oplus a_{29} \oplus a_{31} \oplus a_{32} \oplus a_{33}) \\ & (\overline{a_8} \oplus a_{12} \oplus a_{14} \oplus a_{16} \oplus a_{18} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus \overline{a_{30}} \oplus a_{33}) \\ & (a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34}) \\ & (\overline{a_{10}} \oplus a_{11} \oplus a_{13} \oplus a_{16} \oplus a_{17} \oplus a_{18} \oplus a_{19} \oplus a_{20} \oplus \\ & \quad a_{21} \oplus a_{22} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{31} \oplus a_{33} \oplus a_{34}) \end{aligned}$$

Figure 2.11: Reverse-engineered mapping function between memory blocks and Caching/Home Agent (CHA) bits 0 and 4.



x_0	x_1	x_2	$z = \sigma(\sum_i w_i x_i + b)$	$z = x_0 x_1 \mid (\sim x_2 \,\&\, (x_0 \mid x_1))$
0	0	0	0.0067	0
0	0	1	0.0000003	0
0	1	0	0.9933	1
0	1	1	0.0067	0
1	0	0	0.9933	1
1	0	1	0.0067	0
1	1	0	0.9999997	1
1	1	1	0.9933	1

Figure 2.12: Discrete functions can be extracted from a trained neural network [227]

such that $w_0 = w_1 = 10$, $w_2 = -10$, and $b = -5$ for the single neuron shown, which uses a sigmoid function to clamp the z result between floating point values of 0.0 and 1.0. We can see from the z columns in the figure that the values in the floating point result closely approximate the binary computation of $z = x_0 x_1 | \overline{x_2}(x_0|x_1))$. Other researchers have explored the problem of extracting grammar rules from recurrent neural networks [227], but in our study of CHA the neural network did not learn the problem well enough to attempt this effort. Additionally, as we discuss in Parts II and III of this thesis, the stochastic nature of neural network outputs can be useful when a mechanism for verifying the outputs exists. A stochastic result can be used to create multiple proposals from a network, which can increase the likelihood of a useful result being produced.

Section key points: We confirmed that neural networks do not learn hash functions well. We also found that larger networks may often learn training data well but may not generalize well (they can overfit).

2.3.2 Generalizing to a Target Distribution With Proper Network Sizing

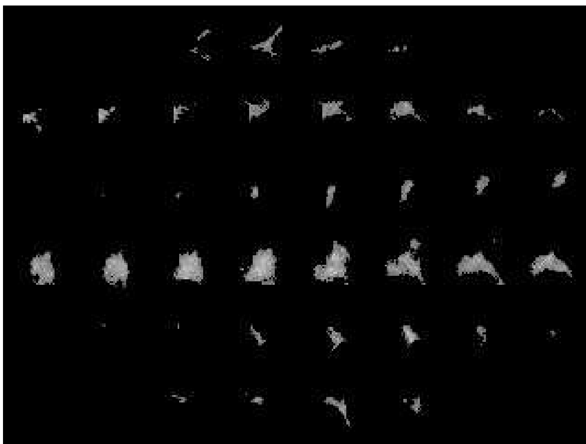


Figure 2.13: Six of the 51 seed nodules showing the middle 8 out of 20 2D slices

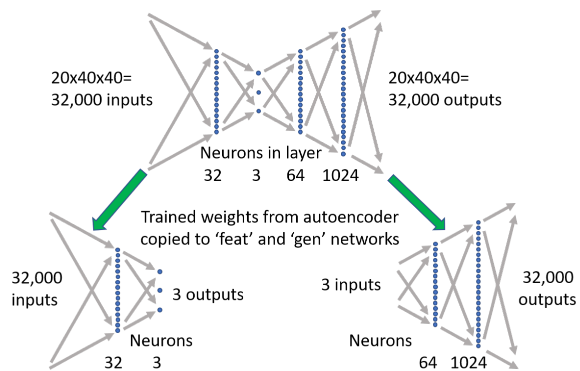


Figure 2.14: Autoencoder and derived feature and generator networks for nodules

In work outside the scope of this thesis, we created a method to score autoencoder output distributions for use in generating artificial examples of 3D lung nodule images to aid in lung cancer diagnosis. One of the challenges of using machine learning techniques with medical data is the

frequent dearth of source image data on which to train. A representative example is automated lung cancer diagnosis, where nodule images need to be classified as suspicious or benign. In our published work [120], we propose an automatic synthetic lung nodule image generator. Our 3D shape generator is designed to augment the variety of 3D images. Our proposed system takes root in autoencoder techniques, and we provide extensive experimental characterization that demonstrates its ability to produce quality synthetic images.

Figure 2.13 shows 6 of the 51 CT scan images which were available for training our model. Each of the images is centered in the $20 \times 40 \times 40$ training size. One of our nodules was slightly too wide and 21 out of 1290 total voxels were clipped; all other nodules fit within the training size. From an original set of 51 images, 816 are generated: 8 copies of each nodule are the 8 possible reflections in X,Y, and Z of the original; and 8 copies are the X,Y, and Z reflections of the original shifted by 0.5 pixels in X and Y. The reflections are still representative of legal nodule shapes to the analyzer, so it improves the generality of the autoencoder to have them included. The 0.5-pixel shift also aids generalization of the network by training it to tolerate fuzzy edges and less precise pixel values. We do not do any resizing of the images as we found through early testing that utilizing the full voxel data resulted in better generated images than resizing the input and output of the autoencoder.

Our autoencoder is trained initially with the 816 images in our base set. We use Adam [116] for stochastic optimization to minimize the mean squared error of the generated 32,000 voxel 3D images. As shown in Figure 2.14, the trained autoencoder network can be split into feature and generator networks. The feature network can be used to map actual nodules into a latent feature space so that novel images similar to the actual input nodule can be created using the generator network. If stepping is done between the latent feature values of nodule suspected as cancerous and another suspected to be non-cancerous, a skilled neurologist could identify the shape at which the original suspicious nodule would not be considered suspicious to help train and improve an automated classifier. The generator network can also be used to generate fully random images for

improving a classifier. For our random generation experiments we use uniform values from -1 to 1 as inputs for the 3 latent feature dimensions.

The autoencoder structure which yielded the best results is not symmetric in that there are fewer layers before the bottleneck layer than after. Like the seminal work by Hinton and Salakhutdinov [96], we explored various autoencoder sizes for our problem, but we added in an exploration of non-symmetric autoencoders. We found during hyperparameter testing that a 2-layer feature network (encoder) performed better than a 1-layer or 3-layer network. We suspect that a single layer for the feature network was not optimal due to limiting the feature encoding of the input images to linear combinations of principle components [88]. We suspect that 3 layers for our feature network was less optimal than 2 layers due to overfitting the model to our limited training set. Given our goal of generating novel nodule shapes, overfitting is a particular concern and we address this using a network scoring metric.

Metrics for scoring the accepted image set The composite score that we use to evaluate networks for LuNG is comprised of 4 metrics used to combine key goals for our work. We compute the percentage of nodule images randomly generated by the generator that are accepted by the analyzer. For assessing the variation of output images relative to the seed images, we compute a feature distance $FtDist$ based on the 12 3D image features used in the analyzer. To track how well the distribution of output images matches the seed image variation, we compute a $FtMMSE$ based on the image feature means. The ability of the network to reproduce a given seed image is tracked with the mean squared error of the image output voxels, as is typical for autoencoder image training.

Our metric of variation, $FtDist$, is the average distance over all accepted images to the closest seed image in the 12-dimensional analyzer feature space and is scaled in a way similar to Mahalanobis distances. As $FtDist$ increases, the network is generating images that are less similar to specific samples in the seed images, hence it is a metric we want to increase with LuNG. Given an accepted set of n images Y and a set of 51 seed images S, and given y_i denotes the value of feature i for an image and σ_{Si} denotes the standard deviation of feature i within S:

$$FtDist = 1/n \sum_{y \in Y} \min_{s \in S} \sqrt{\sum_{i=1}^{12} (\frac{y_i - s_i}{\sigma_{Si}})^2}$$

$FtMMSE$ tracks how closely LuNG is generating images that are within the same analyzer feature distribution as the seed images. It is the difference between the means of the images in Y and S for the 12 3D features. As $FtMMSE$ increases, the network is generating images that are increasingly outside the seed image distribution, hence we want smaller values for LuNG. Given μ_{Si} is the mean of feature i in the set of seed images and μ_{Yi} is the mean of feature i in the final set of accepted images:

$$FtMMSE = 1/12 \sum_{i=1}^{12} (\frac{\mu_{Yi} - \mu_{Si}}{\sigma_{Si}})^2$$

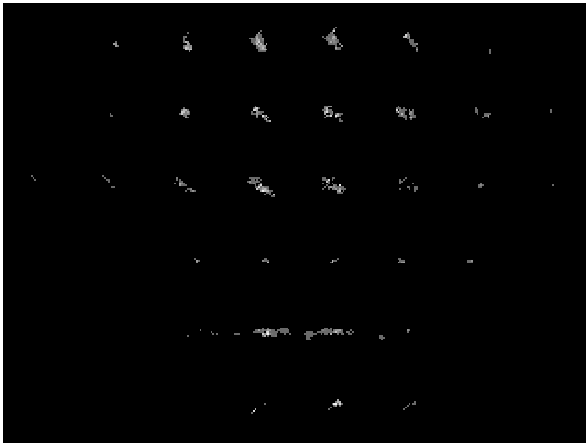


Figure 2.15: 6 images generated using uniform distribution as inputs to generator network

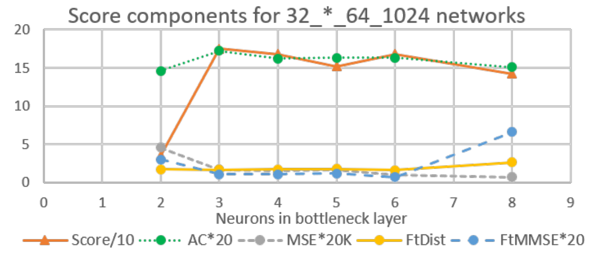


Figure 2.16: There are 4 components used to compute the network score. The component values are scaled as shown so that they can all be plotted on the same scale.

$Score$ is our composite network scoring metric used to compare different networks, hyperparameters, feedback options, and reconnection options. In addition to $FtDist$ and $FtMMSE$, we use AC, which is the fraction of generated images which the analyzer accepted, and MSE which is the mean squared error that results when the autoencoder is used to regenerate the 51 seed nodule images.

$$Score = \frac{FtDist - 1}{(FtMMSE + 0.1) * (MSE + 0.1) * (1 - AC)}$$

$Score$ increases with $FtDist$ and AC and decreases with $FtMMSE$ and MSE. The constants in the equation are based on qualitative assessments of network results; for example, using $MSE + 0.1$ means that MSE values below 0.1 don't override the contribution of other components and aligns with the qualitative statement that an MSE of 0.1 yielded visually acceptable images in comparison with the seed images.

Figure 2.15 shows 6 example lung nodules generated randomly by our generator network. Results using $Score$ to evaluate networks and LuNG interface features are shown in Figure 2.16. Note that the MSE metric (mean squared error of the network on training set) continues to decrease with larger networks, but $Score$ is optimal with 3 bottleneck latent feature neurons. Our intuition is that limiting our network to 3 bottleneck neurons results in most of the available degrees of freedom being required for proper image encoding. As such, using a -1 to 1 uniform random distribution as the prior distribution for our generative network creates a variety of acceptable images. The $Score$ metric helps us to tune the system such that we do not require VAE techniques to constrain our random image generation process, although such techniques may be a valuable path for future research.

Our use of $Score$ to evaluate the entire nodule generation process rates the quality of the random input distribution, the generator network, the reconnection algorithm, the analyzer acceptance, and the interaction of these components into a system. Our use of the analyzer acceptance rate is similar in some functional respects to the discriminator network in a GAN as both techniques are used to identify network outputs as being inside or outside an acceptable distribution.

Section key points: We developed an autoencoder problem which can be used to explore how network and feature sizes can generalize on a limited dataset.

2.3.3 Using Hindsight Learning to Improve Goal Achievement In Multigoal Environment With Sparse Rewards

For the problem of program understanding, some researchers have used reinforcement learning to explore ways to repair code [83] as well as to prove symbolic statements [69, 23, 171]. One challenge with reinforcement learning on code is that a reward function to train on may be quite sparse since there is no easy distance metric from buggy code to fixed code. Additionally, for the case of proving programs equivalent, the goal to be reached is itself part of the input (we want to prove a first program equal to a second program). Hence, we would like to study a reinforcement learning system which has multiple goals and a sparse reward. We researched the ability of reinforcement learning to solve this challenge by creating a problem simpler than program analysis for study of learning techniques: a multigoal grid world [122].

We based our reinforcement learning model on the Deep Q Network (DQN) tutorial for PyTorch [174], which includes a replay buffer [141, 139], and a target network [160] to help with stability. As we will show, the base network performs poorly on the multi-goal learning problem due in part to a sparse reward, so we added hindsight experience replay [14] to create a system which is able to learn more complex behaviors. In our previously published work, we show how a symbolic learner performs on this same task [122], but for this thesis we are interested in results applicable to neural networks.

Model inputs For the DQN model, the input is the current world state, including the desired goal, and the output is the action the model recommends taking. Figure 2.17 shows the full set of inputs. The DQN receives data about the current world state through hand and eye proprioception (a sense of where the hand and eye are in space) as well as a visual field which is the same dimension as the grid world. These 3 input forms are provided as 3 N×N input values. A fourth N×N matrix encodes the goal state:

- `IN[3][0][x]` specifies the X position of the goal
- `IN[3][1][y]` specifies the Y position of the goal

- `IN[3][2:N-1][0]` set to 1 for goal to move hand to a given proprioceptive position
- `IN[3][2:N-1][1]` set to 1 for goal to move eye to a given proprioceptive position
- `IN[3][2:N-1][2]` set to 1 for goal to see hand at a given position in visual field

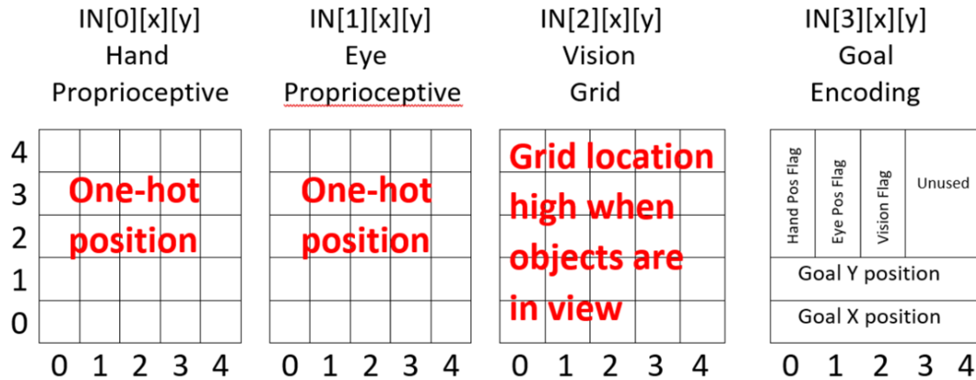


Figure 2.17: Input tensor has four 2D grids. Three grids represent sensory input, and one grid represents the goal to achieve.

The network output is the 8 possible actions: move hand forward, backward, left, right, and move eye forward, backward, left, or right.

During reinforcement learning, the model is trained by using back-propagation as the model optimizes an action policy given the input state [14]. In typical reinforcement learning, an action policy determines which action to apply in which state: $\pi(s) : \mathbb{S} \rightarrow \mathbb{A}$. Reinforcement learning works by learning a Q-function $Q(s,a)$ which represents the reward from the environment achieved by taking action a in state s. For any given state s, the action a with the highest $Q(s,a)$ value is the recommended action to take, as it achieves the highest reward. The optimal Q-function $Q^*(s,a)$ is given by the Bellman equation, shown in Equation 2.7. In this equation, $Q^*(s,a)$ is the expected value over all possible next states of the immediate reward $r(s,a)$ achieved by taking action a in state s plus the best Q-value of the possible future states discounted by a depreciation factor λ.

$$Q^*(s,a) = \mathbb{E}_{s' \sim p(\cdot|s,a)}[r(s,a) + \gamma \max_{a' \in A} Q^*(s',a')] \tag{2.7}$$

From the Bellman equation, we can see that having frequent non-zero rewards will help a system learn the correct Q-function and, hence, the correct action to take in a given state. But our goal in this study is to model a system which is only rewarded when the goal is achieved, which models situations in which a distance to goal metric for an incremental reward may be difficult to evaluate. As we will show, sparse rewards can create training convergence challenges to traditional reinforcement learning models. In order to learn with sparse rewards, the model needs to get sufficiently lucky to take the correct action when it happens to be right next to a goal, then the system can learn how to take actions that get it one action away from the goal, and so on. But with enough random actions occurring, the ability to learn the Q function can become intractable. Andrychowicz *et al.* address the issue of learning with sparse rewards in a multi-goal system by introducing hindsight learning [14]. In hindsight learning, the current state and goal state are given and the system chooses an action given it's currently poorly performing Q-function. This action is not likely to have a reward, but the training can be adjusted to reward the state achieved. Hindsight works by adding a training sample as if the state reached was the goal. For example, if the model is asked to achieve goal g from current state s and it takes action a resulting in state s', then a training sample is created which teaches the model that if the goal had been to reach s' from s then the correct action was a. In this way, every step taken by the model provides some teachable information.

Figures 2.18 and 2.19 show the importance of using hindsight to enhance the ability of the DQN network to learn from sparse rewards; the DQN needs reward information to learn well. The blue lines in the graphs are the path length to reach the goal each episode, the gold line is the average path length over 200 episodes. For example, when there are no blue lines reaching to the max path length (4 times the length of a grid side) then for that period the network solved all the goal challenges presented. Hindsight learning learns 2 datum per action step - it trains on the benefit of the action taken with respect to the actual goal requested, and it trains on the benefit of the given action as if the next state reached was the goal. The gold lines in the figures show that a model with hindsight learns to solve multiple goals in a 7x7 grid faster than a model without

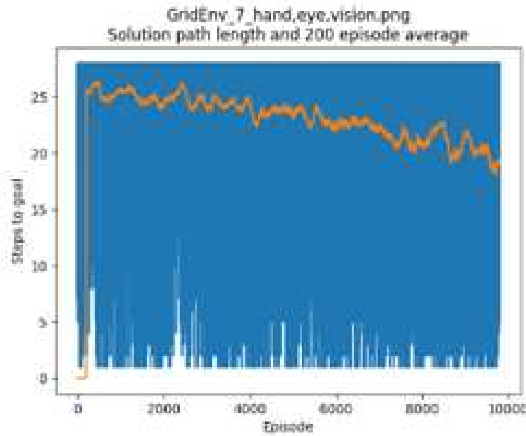


Figure 2.18: Gold line shows the average steps to reach a random goal (hand, eye, or vision) on a 7x7 grid. Learning progresses slower without hindsight.

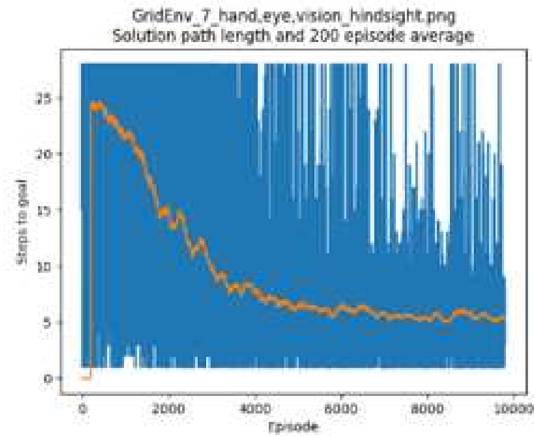


Figure 2.19: The benefit of hindsight. Multigoal learning on a 7x7 grid converges to nearly optimal relatively quickly.

hindsight. Without hindsight, learning the multi-goal problem on a 9x9 grid did not converge with our model.

> **Section key points:** We developed a system to explore multigoal problems with sparse rewards and found hindsight learning to be highly valuable in adding training information from every action taken.

2.4 Complexity Classes for Program Equivalence

Computational complexity classifies problems based on the computation resources required to solve a problem [234]. The famous P vs NP problem, one of the seven unsolved mathematical problems for which the Millennium Prize would be awarded, is based on computational complexity theory [59]. Figure 2.20 shows some of the complexity classes related to the problem of program equivalence. The classes are defined in relation to using a Turing machine [218] to solve the problems. A Turing machine is a mathematical model of computation useful for precise discussions of algorithmic complexity and we shall use Turing machines to define the complexity classes in Definitions 2.4.1 through 2.4.6. The 'simplest' problems for a computer to solve in Figure 2.20 would be in complexity class P. The next most difficult are NP, then PP, and finally 'Decidable'

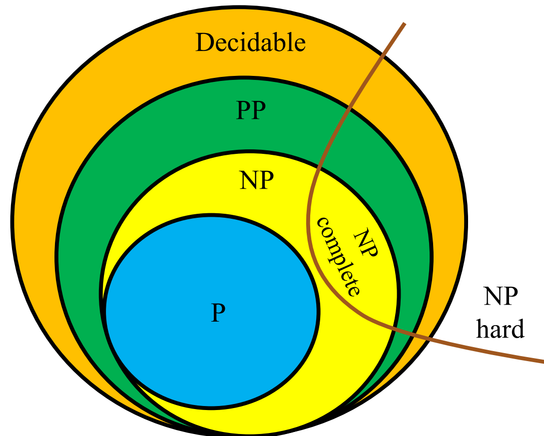


Figure 2.20: Complexity classes indicate how hard a problem is for a Turing machine to solve.

problems. Some problems, including some program equivalence problems as we shall show, are undecidable.

Definition 2.4.1. *Complexity class P* is the class of problems which can be solved by a deterministic Turing machine in polynomial time. Effectively this means that for a problem of size N, a computer algorithm which can be run on a typical physical computer today can solve the problem in time less than $k * N^p$. Where k and p are positive constants related to the details of the algorithm.

Definition 2.4.2. *Complexity class NP* is the class of problems which can be solved by a non-deterministic Turing machine in polynomial time. A non-deterministic Turing machine describes a system which can have multiple next states given a certain input and current computation state; *i.e.,* the next state is non-deterministic.

A non-deterministic Turing machine can be conceptualized as being allowed to pick the 'best' next state for the goal of solving the problem, which is not currently realizable by physical computers (although quantum computers are expected to be able to address some problems that are part of NP but not in P [1]). It has not been proven that P is not equal to NP; *i.e.,* it is not known if a non-deterministic Turing machine can actually solve problems in polynomial time which could not be solved by a deterministic machine in polynomial time, but it is expected by most computer science researchers that P is not equal to NP. Many problems that are in NP currently take expo-

nential time to solve on modern computers. Indeed, many modern security protocols rely on this exponential time for securing critical data.

Definition 2.4.3. *NP-complete* is a subset of problems in NP which can be used to simulate every other problem in NP [60]. Because we require that mapping an NP problem to an NP-complete problem must be possible in polynomial time on a deterministic Turing machine, if any problem that is NP-complete is found to be in complexity class P, then any NP problem can be solved in polynomial time on a deterministic Turing machine, in which case $P = NP$.

Definition 2.4.4. *NP-hard* is a class of problems which have proven to be "at least as hard" as NP-complete problems. Proving that a problem is NP-complete requires proving that the given problem can indeed be solved by a non-deterministic Turing machine in polynomial time (*i.e.,* the problem is in NP). But proving a problem is NP-hard is easier; one must only prove that an NP-complete problem can be mapped into the given problem such that a solution for the given problem could also be used to solve the NP-complete problem. Hence, the NP-hard classification contains all NP-complete problems, but also includes some problems which are undecidable.

Definition 2.4.5. *Complexity class PP* is the class of problems which can be solved by a probabilistic Turing machine in polynomial time. If a binary yes/no decision problem is in PP, then there is an algorithm which is allowed to flip coins to make random decisions and will answer the problem correctly more than 1/2 of the time. All of the problems in NP are also in PP, and all of the problems in PP are decidable.

Definition 2.4.6. *Decidable* problems are all problems which can be computed by a Turing machine with no bound on the time or space required.

One famously undecidable problem is the halting problem [219]. This is the problem of determining whether an arbitrary computer program with a given input will finish running or run forever. This yes/no problem is not itself computable by a computer program.

Definition 2.4.7. *Program equivalence* Programs P_1 and P_2 are considered equivalent if, for all inputs in the input domain D the programs produce the same output(s).

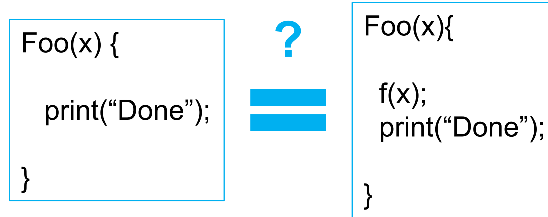


Figure 2.21: The general case of determining program equivalence requires solving the halting problem on $f(x)$, and hence is undecidable.

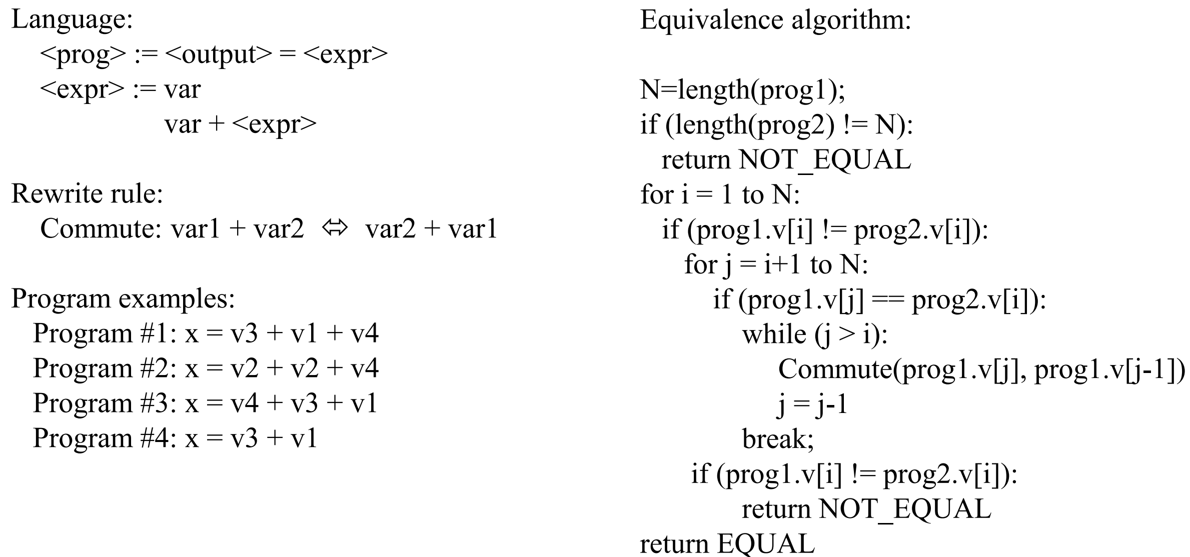


Figure 2.22: A rather simple language with only a single operator (+) and only a single program rewrite rule (Commute) can be checked for equivalence with a polynomial-time algorithm requiring fewer than $k \times N^2$ steps, where N is the program length and k is a constant.

It can be quickly shown that the general case of program equivalence is undecidable with reference to Figure 2.21. The only difference between these 2 programs is that the one on the right will only print "Done" if the function $f(x)$ halts. Hence, proving these 2 programs equivalent is synonymous with solving the halting problem, which is undecidable.

One the other end of the complexity spectrum, consider the language defined in Figure 2.22. In this language a program is only allowed to contain a single statement which sums up one or more variables. Two programs will be equivalent if they contain the same variable list (in any order) in the summation expression.

Table 2.3: Mapping Boolean functions to scalar mathematics

Boolean expression	Scalar expression
$\overline{a}$	$1 - a$
$a \vee b$	$1 - (1 - a)(1 - b)$
$a \wedge b$	$a * b$

```
Language:
  <prog> := o = <expr>                 <const> := 0
  <expr> := <const>                               1
            <var>                      <var>   := i_j
            <expr> <op> <expr>         <op>    := +
            (<expr>)                              -
                                                  *
```

Zero-function:
o = 0

Example SAT problem: $(i_1 \vee \overline{i_2} \vee i_3) \wedge (i_2 \vee \overline{i_3} \vee i_4)$

Program based on example SAT problem:
o = (1-(1-i_1) * i_2 * (1-i_3)) * (1-(1-i_2) * i_3 * (1-i_4))

Figure 2.23: A simple computer language into which the Boolean satisfiability problem can be mapped.

Definition 2.4.8. *Rewrite rule* is defined as a rule which can be applied to a program which adjusts the program lexically but not semantically. Applying a rewrite rule does not change the computation performed by a program and creates a program equivalent to the original.

By applying the rewrite rule shown in Figure 2.22 multiple times we can see that example program #1 is equivalent to program #3. Starting with program #1, we first apply Commute such that $v1 + v4 \leftrightarrow v4 + v1$ and then again such that $v3 + v4 \leftrightarrow v4 + v3$, which results in a program lexically equal to program #3.

The algorithm shown in Figure 2.22 demonstrates how to determine program equivalence given this constrained language. The algorithm uses the Commute rewrite rule to transform one program into the other if it is possible. Given programs of length N, since the i and j loops may both grow by N, this algorithm will run in time less than $k \times N^2$ where k is some constant. Hence, this algorithm is in complexity class P.

Given the synthetic language for straight-line programs we detail in Chapter 7, we can demonstrate that solving the program equivalence problem in this language is at least as difficult as solving the Boolean satisfiability problem, which is itself NP-complete [60]. Our proof is based on the ability to map a Boolean SAT problem into our scalar expression language as shown in Table 2.3.

Definition 2.4.9. *Boolean satisfiability problem (SAT)* This is the problem of determining whether, given a Boolean function S with Boolean input values $b_{1..n}$, there is a set of Boolean input values for which the function returns "True". The Boolean function may include logical AND, OR, and INVERT operations on the variables, which may be operated on in any order.

The Boolean satisfiability problem (SAT) has been proven to be NP-complete [60]. Hence, if we can show that a program equivalence problem could be used to solve SAT then that program equivalence problem is NP-hard.

Definition 2.4.10. *Binary Domain Program* is defined for discussion to be the language shown in Figure 2.23. This language supports the normal $+, -,$ and $\times$ operations on real numbers, but the inputs and constants in the language are limited to the values 0 and 1.

Theorem 2.4.1. Binary Domain Program equivalence is NP-hard.

Proof. Consider a specific Boolean satisfiability problem S in which the task is to determine if there is a configuration of inputs to S such that S will return "true". Using Table 2.3, map S into our scalar expression language to create program P. This mapping is shown with an example SAT problem in Figure 2.23. If, given inputs from the domain {0,1}, we can prove P is equal to a program which simply returns 0 (the zero-function), then there is no input to P which returns 1. Additionally, if there is a set of inputs for which P returns 1, that set of inputs maps to a set of logical inputs which would satisfy S. Hence, P computes the zero-function if and only if S is not satisfiable and hence the general case of proving a program in our language equal to the zero-function is at least as hard as NP-complete, which means it is in NP-hard. □

In their paper "Probabilistic Algorithms for Deciding Equivalence of Straight-Line Programs" [102], Ibarra and Moran further prove that when the the input to two straight-line programs is limited to a finite set of integers of cardanality ≥ 2 then the equivalence problem is NP-hard, and when the input can be an infinite field (such as all rational numbers) then the equivalence problem for straight-line programs in a language which includes only $+, -, *,$ and $/$ is probabilistically decidable in polynomial time. In other words our full program equivalence problem as covered

in chapter Chapter 7 is in complexity class PP, a class which contains NP. (Gurari and Ibarra also study the complexity of varying program equivalence problems [85]).

2.5 Limitations of Prior Research

While the prior research we discuss in Sections 2.1, 2.2, and 2.3 provide a solid foundation for machine learning as applied to computer aided programming, they have several weaknesses which this thesis aims to address.

Program repair research is very active and dominated by techniques based on static analysis (*e.g.,* SPR [143] and Angelix [157]) and dynamic analysis (*e.g.,* CapGen [230]). While great progress has been achieved, the current state of automated program repair is limited to simple small fixes, mostly one line patches [191, 230]. These techniques are heavily top-down, based on intelligent design and domain-specific knowledge about bug fixing in a given language or a specific application domain. Another weakness of prior work is vocabulary limitations [12]. Such works cannot learn representations for novel variable names. Also, the representations which are learned are based on the training set provided, hence they may not be appropriate for how a variable is used. For example, a Java method that iterates over an array using i and accumulates the sum of array elements into sum is semantically equivalent to a method that uses j for the iterator and i for the accumulation. But the somewhat odd use of i for accumulation would result in a shifted internal representation for the method. Given continually advancing machine learning techniques [208, 197, 220], a framework for utilizing machine learning to automatically repair bugs is needed.

Machine learning models benefit from having millions of training samples on which to generalize [207]. For concrete problems in computer science, the problem domain may suffer from a limited set of labeled data. For example, security vulnerability repair has a small number of reviewed examples for some vulnerability categories [68, 32], and linear algebra equivalence also has limited examples on which to learn [114]. Systems to provide samples to machine learning models are needed in order to train such models effectively.

The general problem of program equivalence is undecidable by modern computers, as we detail in Section 2.4. This invites the use of machine learning as it may be able to learn equivalence proofs beyond what static algorithms can solve. The primary weakness with applying machine learning to program equivalence is the imprecision in the results when determining equivalent semantics [12, 241]. If machine learning is to be useful in situations where a guarantee of correctness for program equivalence is needed then we must develop a new approach.

Traditional machine learning systems often use early stopping [178] to detect when the training has plateaued for a given training dataset and further training will not improve the model. Developing a system which can create additional samples which specifically target areas where a model needs improvement could increase both performance and generalization.

In Chapter 3 we further detail the problems addressed in this thesis and summarize the contributions we make to the field of machine learning for computer aided programming.

Chapter 3

Contributions to Machine Learning on Computer Aided Programming

Machine learning techniques have produced strong and useful results in areas where imperfect outputs are acceptable. Achievements of image classification and language translation that exceed human capabilities are used in a wide range of commercial applications [106, 209, 36]. In parallel, the importance of having a verification of correctness comes up in many environments. Safety systems such as self-driving cars would benefit from techniques that can demonstrate how to create robust systems which can generate verifiable output. Systems which generate sequences constrained in some technical way - be it computer code generation, manufacturing sequence optimizations, or formal proof generation - would benefit from understanding how training datasets can improve the learning of rigid constraints in a given target environment.

Verifiable correctness can help address problems recognized by the broader research community. The problem of robustness to distributional shift is one of 5 failure modes for AI noted in 'Concrete problems in AI safety' [13]. Techniques which can help prove that a system has learned to generate correct output could help address this key challenge for AI. The White House has posted a memorandum to executive agencies titled "Guidance for Regulation of Artificial Intelligence Applications" [225]; three of its 10 principles for the stewardship of AI applications are related to the effort of verifying machine learning outputs well: "scientific integrity and information quality", "risk assessment and management", and "safety and security".

In the following sections, we will introduce problems in machine learning which will be explored in depth in the remaining chapters of this thesis. In addition to addressing specific problems with machine learning, these applications also demonstrate concepts for machine learning on computer aided programming which can be leveraged to solve other problems.

3.1 Repairing Functional Bugs in Java Programs

People have long dreamed of machines capable of writing computer programs by themselves. Having machines writing a full software system is science-fiction but teaching machines to modify an existing program to fix a bug is within the reach of current software technology; this is called automated program repair [161].

Program repair research is very active and dominated by techniques based on static analysis (*e.g.,* Angelix [157]) and dynamic analysis (*e.g.,* CapGen [230]). While great progress has been achieved, the current state of automated program repair is limited to simple small fixes, mostly one line patches [191, 230]. These techniques are heavily top-down, based on intelligent design and domain-specific knowledge about bug fixing in a given language or a specific application domain. In Chapter 4, we also focus on one line patches, but we aim at doing program repair in a language-agnostic generic manner, fully relying on machine learning to capture syntax and grammar rules and produce well-formed, compilable programs. By taking this approach, we aim to provide a foundation for connecting program repair and machine learning, allowing the program repair community to benefit from training with more complete bug datasets and continued improvements to machine learning algorithms and libraries.

As the foundation for our model, we apply sequence-to-sequence learning [208] to the problem of program repair. Sequence-to-sequence learning is a branch of statistical machine learning, mostly used for machine translation: the algorithm learns to translate text from one language (say French) to another language (say Swedish) by generalizing over large amounts of sentence pairs from French to Swedish. The training data comes from the large amount of text already translated by humans, starting with the Rosetta stone written in 196 BC [204]. The name of the technique is explicit: it is about learning to translate from one sequence of words to another sequence of words.

Now let us come back to the problem of programming: we want to learn to 'translate' from one sequence of program tokens (a buggy program) to a different sequence of program tokens (a fixed program). The training data is readily available: we have millions of commits in open-source code repositories. Yet, we still have major challenges to overcome when it comes to using sequence-to-

sequence learning on code: 1. the raw (unfiltered) data is rather noisy; one must deploy significant effort to identify and curate commits that focus on a clear task; 2. contrary to natural language, misuse of rare words (identifiers, numbers, etc) is often fatal in programming languages [91]; in natural language some errors may be tolerable because of the intelligence of the human reader while in programming languages the compiler (or interpreter) is strict 3. in natural language, the dependencies are often in the same sentence ("it" refers to "dog" just before) , or within a couple of sentences, while in programming, the dependencies have a longer range: one may use a variable that has been declared dozens of lines before.

We are now at a tipping point to address those challenges. First, sequence-to-sequence learning has reached a maturity level, both conceptually and from an implementation point of view, that it can be fed with sequences whose characteristics significantly differ from natural language. Second, there has been great recent progress on using various types of language models on source code [9]. Based on this great body of work, we present our approach to using sequence-to-learning for program repair, which we created to repair real bugs from large open-source projects written in the Java programming language.

Our end-to-end program repair approach is called SEQUENCER and it works as follows. First, we focus on one-line fixes: we predict the fixed version of a buggy programming line. For this, we create a carefully curated training and testing dataset of one-line commits. Second, we devise a sequence-to-sequence network architecture that is specifically designed to address the two main aforementioned challenges. To address the unlimited vocabulary problem, we use the copy mechanism [197]; this allows SEQUENCER to predict the fixed line, even if the fix contains a token that was too rare (*i.e.,* an API call that appears only in few cases, or a rare identifier used only in one class) to be considered in the vocabulary. This copy mechanism works even if the fixed line should contain tokens which were not in the training set. To address the dependency problem, we construct *abstract buggy context* from the buggy class, which captures the most important context around the buggy source code and reduces the complexity of the input sequence. This enables us to capture long range dependencies that are required for the fix.

We evaluate SEQUENCER in two ways. First, we compute accuracy over 4,711 real one-line commits, curated from three open-source projects. The accuracy is measured by the ability of the system to predict the fixed line exactly as originally crafted by the developer, given as input the buggy file and the buggy line number. Our golden configuration is able to perfectly predict the fix for 950/4,711 (20%) of the testing samples. This sets up a baseline for future research in the field. Second, we apply SEQUENCER to the mainstream evaluation benchmark for program repair, Defects4J. Of the 395 total bugs in Defects4J, 75 have one-line replacement repairs; SEQUENCER generates patches which pass the test suite for 19 bugs and patches which are semantically equivalent to the human-generated patch for 14 bugs. To our knowledge, this is the first report ever on using sequence-to-sequence learning for end-to-end program repair, including validation with test cases.

Overall, the novelty of this work is as follows. First, we create and share a unique dataset for evaluating learning techniques on one-line program repair. Second, we report on using the copy mechanism on seq-to-seq learning on source code. Third, on the same buggy input dataset, SEQUENCER is able to produce the correct patch for 119% more samples than the closest related work [216].

To sum up:

- Our key contribution is an approach for fixing bugs based on sequence-to-sequence learning on token sequences. This approach uses the copy mechanism to overcome the unlimited vocabulary problem in source code.

- We present the construction of an *abstract buggy context* that leverages code context for patch generation. The input program token sequences are at the level of full classes and capture long-range dependencies in the fix to be written. We implement our approach in a publicly-available program repair tool called SEQUENCER.

- We evaluate our approach on 4,711 real bug fixing tasks. Contrary to the closest related work [216], we do not assume bugs to be in small methods only. Our golden trained model

is able to perfectly fix 950/4,711 testing samples. To the best-of-our knowledge, this is the best result reported on such a task at the time of writing this thesis [216, 182, 155].

- We evaluate our approach on the 75 one-line bugs of Defects4J, which is the most widely used benchmark for evaluating programming repair contributions. SEQUENCER is able to find 2,321 patches for these bugs, 761 compile successfully, 61 are plausible (they pass the full test suite) and 18 are semantically equivalent to the patch written by the human developer.
- We provide a qualitative analysis of 8 interesting repair operators captured by sequence-to-sequence learning on the considered training dataset.

3.2 Repairing Security Vulnerabilities in C Language Programs

On the code hosting platform GitHub, the number of newly created code repositories has increased by 35% in 2020 compared to 2019, reaching 60 million new repositories during 2020 [213]. This is a concern to security since the number of software security vulnerabilities is correlated with the size of the software [184]. Perhaps worryingly, the number of software vulnerabilities is indeed constantly growing [98]. Manually fixing all these vulnerabilities is a time-consuming task; the GitHub 2020 security report finds that it takes 4.4 weeks to release a fix after a vulnerability is identified [75]. Therefore, researchers have proposed approaches to automatically fix these vulnerabilities [86, 49].

In the realm of automatic vulnerability fixing [104], there are only a few works on using neural networks and deep learning techniques. One of the reasons is that deep learning models depend on acquiring a massive amount of training data [207], while the amount of confirmed and curated vulnerabilities remains small. Consider the recent related work Vurle [152], where the model is trained and evaluated on a dataset of 279 manually identified vulnerabilities. SeqTrans is another recent effort, trained and evaluated on a dataset with 1282 vulnerabilities [49]. On the other hand, training neural models for a translation task (English to French) has been done using over 41

million sentence pairs [35]. Another example is the popular summarization dataset CNN-DM [94] that contains 300 thousand text pairs. Li *et al.* showed that the knowledge learned from a small dataset is unreliable and imprecise [134]. Schmidt *et al.* found that the error of a model predicting the thermodynamic stability of solids decreases with the size of training data [194]. We argue that learning from a small dataset of vulnerabilities suffers from the same problems (and will provide empirical evidence later).

In Chapter 5, we address the problem that vulnerability fix datasets are too small to be meaningfully used in a deep-learning model. Our key intuition to mitigate the problem is to use transfer learning. Transfer learning is a technique to transfer knowledge learned from one domain to solve problems in related domains, and it is often used to mitigate the problem of small datasets [2]. We leverage the similarity of two related software development tasks: bug fixing and vulnerability fixing. In this context, transfer learning means acquiring generic knowledge from a large bug fixing dataset and then transferring the learned knowledge from the bug fixing task to the vulnerability fixing task by tuning it on a smaller vulnerability fixing dataset. We realize this vision in a novel system for automatically repairing C vulnerabilities called VRepair.

To train VRepair, we create a sizeable bug fixing dataset, large enough to be amenable to deep learning. We create this dataset by collecting and analyzing all 892 million GitHub events that happened between 2017-01-01 and 2018-12-31 and using a heuristic technique to extract all bug fix commits. In this way, we obtain a novel dataset consisting of over 21 million bug fixing commits in C code. We use this data to first train VRepair on the task of bug fixing. Next, we use two datasets of vulnerability fixes from previous research, called Big-Vul [68] and CVEfixes [32]. We tune VRepair on the vulnerability fixing task based on both datasets. Our experimental results show that the bug fixing task can be used to train a model meant for vulnerability fixing; the model only trained on the collected bug fix corpus achieves 14.77% accuracy on Big-Vul and 15.1% on CVEfixes, which validates our initial intuition that these tasks are profoundly similar. Next, we show that by using transfer learning, *i.e.,* by first training on the bug fix corpus and then by tuning the model on the small vulnerability fix dataset, VRepair increases its accuracy

to 17.77% on Big-Vul and 19.94% on CVEfixes, demonstrating the power of transfer learning. Additionally, we compare transfer learning with a denoising pre-training followed by fine-tuning on the vulnerability fixing task and show that VRepair's process is better than pre-training with a generic denoising task. We also show that transfer learning improves the stability of the final model.

In summary, our contributions are:

- We introduce VRepair, a Transformer Neural Network Model which targets the problem of vulnerability repair. The core novelty of VRepair is to employ transfer learning as follows: it is first trained on a bug fix corpus and then tuned on a vulnerability fix dataset.
- We design a novel code representation for the program repair task with neural networks. Our output code representation is a token difference instead of the entire fixed source code used in recent research [216].
- We empirically demonstrate that on the vulnerability fixing task, the transfer learning VRepair model performs better than the alternatives: 1) VRepair is better than a model trained only on the small vulnerability fix dataset; 2) VRepair is better than a model trained on a large generic bug fix corpus; 3) VRepair is better than a model pre-trained with a denoising task. In addition, we present evidence that the performance of the model trained with transfer learning is stable.
- We share all our code and data for facilitating replication and fostering future research on this topic.

3.3 Proving Equivalence of Complex Linear Algebra Expressions

Deep neural network systems have excelled at a variety of classification and reinforcement learning tasks [81]. However, their stochastic nature tends to hinder their deployment for auto-

mated program analysis: ensuring the correctness of the solution produced is often required, e.g., when determining the semantics equivalence between two programs (or symbolic expressions).

In Chapter 6 we target the problem of automatically computing whether two input symbolic expressions are semantically equivalent [110], under a well-defined axiomatic system for equivalence using semantics-preserving rewrite rules [64]. Program equivalence is summarized as determining whether two programs would always produce the same outputs for all possible inputs, and is a central problem in computing [110, 77, 221]. The problem ranges from undecidable, e.g. [79], to trivial in cases of testing the equivalence of a program with itself. Our work directly studies the subset of programs represented by symbolic linear algebra expressions which include scalar, vector, and matrix types for both constants and variables, and 16 different operators with 147 distinct axioms of equivalence. For example, the expression using matrices, scalars, and a vector: $(A + B)I((a + (b - b))/a)\vec{v} - A\vec{v}$ can be proven equivalent to $B\vec{v}$ by applying 10 axioms in sequence; our work generates the proof steps between these expressions.

While prior work has shown promise for deep networks to compute some forms of program equivalence [241, 12], the system typically outputs only a probability of equivalence, without any reasoning or insight that can be verified easily: false positive can be produced. Programs can be represented as a tree (or graph) of symbols, and deep networks for symbolic reasoning have been studied, e.g. to compute the derivative of a symbolic expression [130]. In this work, we take a significantly different approach to the problem of symbolic program reasoning with deep networks: we make the system produce the sequence of steps that lead to rewriting one program into another, that is the *reasoning* for (or proof of) equivalence between the two programs, instead of producing directly the result of this reasoning (e.g., a probability of equivalence, without explanation about the reasoning). In a nutshell, we approach expression equivalence as a theorem proving problem, in which all the axioms as well as tactics to compute a proof are all learned by example in a deep learning system, without any human insight.

We propose a method for generating training samples using probabilistic applications of production rules within a formal grammar, and then develop a graph-to-sequence [137, 29] neural

network system for program equivalence, trained to learn and combine rewrite rules to rewrite one program into another. It can *deterministically* prove equivalence, entirely avoids false positives, and quickly invalidates incorrect answers produced by the network (no deterministic answer is provided in this case, only a probability of non-equivalence). In a nutshell, we develop the first graph-to-sequence neural network system to accelerate the search in the space of possible combinations of transformation rules (i.e., axioms of equivalence in the input language) to make two graphs representing symbolic expressions structurally identical without violating their original semantics. We propose a machine learning system for program equivalence which ensures correctness for all non-equivalent programs input (specificity = 100%) , and a deterministically checkable output for equivalent programs (no false positives). We make the following contributions:

1. We design, implement and evaluate two competing approaches using graph-to-sequence neural network systems to generate proofs of equivalence. We provide the first implementation of such graph-to-sequence systems in the popular OpenNMT-py framework [117].

2. We present a complete implementation of our system operating on a rich language for multi-type linear algebra expressions. Our system provides a correct rewrite rule sequence between two equivalent programs for 93% of the 10,000 test cases. The correctness of the rewrite rule is deterministically checkable in all cases in negligible time.

3.4 Proving Equivalence of Basic Block Expressions in C Code

Deep neural networks have excelled at a variety of classification, reinforcement learning, and sequence generation tasks [81]. However, their stochastic nature complicates the use of such networks in formal settings where one requires a *guarantee* that the result produced is provably correct, such as to assess semantic equivalence between programs.

Proving program equivalence means determining whether two programs always produce identical output if given the same input, for all possible inputs, and is a central problem in computing [110, 77, 221]. The problem ranges from undecidable [79], to trivial in the case of testing the equivalence of a program with itself. Proving program equivalence is useful for e.g. verifying

compiler correctness [169], replacing code fragments by more optimized ones [74], malicious software detection [146] or automated student feedback [57]. In Chapter 7, we propose a machine learning framework for proving program equivalence, named S4Eq.

S4Eq takes as input two programs and generates a sequence of rewrite rules under a well-defined system for equivalence using semantics-preserving rewrite rules [64]. Our work studies programs represented as a list of statements with straight-line control-flow, using multiple variable types and complex mathematical expressions to compute values. S4Eq outputs a verifiable sequence of rewrite rules, meaning that it guarantees no false positives (no programs are stated equivalent if they are not) by design.

The problem domain at hand, generating a provably correct sequence of rewrite rules, requires a specific training procedure. We devise a novel self-supervised learning technique for proving equivalence. We initially train a model in a supervised manner with synthetic data which has a broad distribution on the use of rewrite rules. Then we propose a self-supervised technique based on comparing results between broad and narrow proof searches to incrementally train our model. Rewrite rule sequences demonstrating equivalence found by a quick, narrow search are not considered interesting for further training; while sequences found by a broad search indicate samples for which the model's rewrite rule selections could be improved. We name this procedure *self-supervised sample selection*. We fully implement our learning and inference models in the popular OpenNMT-py framework [117], based on the transformer model.

We demonstrate S4Eq by proving equivalence between unrelated program blocks derived from C functions mined on the popular code hosting platform GitHub [213]. We process GitHub C functions to find 13,215 unique multi-statement program blocks which can be analyzed for equivalence. We show that S4Eq can prove that various compilation and optimization steps, such as Common Sub-expression Elimination [58] produce equivalent and correct code; and we search for equivalent codes between the GitHub samples themselves.

To sum up, we make the following contributions:

- We present S4Eq, an end-to-end deep learning framework to find equivalence proofs between two complex program blocks. S4Eq produces a sequence of semantics-preserving rewrite rules that can be built to construct one program from the other, via successive rewrites. We consider rewrites which support complex program transformations such as Common Subexpression Elimination and computational strength reduction. S4Eq emits a verifiable sequence of rewrites, leading to no false positives by design.

- We devise an original training technique, tailored to our problem domain, called self-supervised sample selection. This incremental training approach further improves the quality, generalizability and extensibility of the deep learning system.

- We present extensive experimental results to validate our approach, demonstrating our system can successfully prove equivalence on both synthetic programs and programs derived from GitHub with up to 97% success, making the system ready for automated and unsupervised deployment to check equivalence between programs.

- We provide all our datasets to the community including synthetic generation techniques for the problem of program equivalence via rewrite rules, as well as sequences mined from GitHub [121].

Part II

Machine Learning for Repairing Software Defects

Chapter 4

SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair

4.1 Introduction

The problem of program repair is the challenge of finding the correct modification to an existing program which will resolve a fault of some kind. Using machine learning to automate this task would significantly improve the productivity of software developers while improving the quality of code created.

As we introduce in Section 3.1, we develop SEQUENCER to learn to 'translate' a buggy program into a fixed program using machine learning. To efficiently utilize machine learning on this problem, we must adjust the input presented to the model such that sufficient information needed to fix the program is available as input. We accomplish this by introducing the *abstract buggy context* to represent the input code. Also, when creating a fix for a program it may be necessary to draw from a large vocabulary of identifiers (such as variable names). We demonstrate that using a *token copy mechanism* can allow the system to propose program fixes which use any token from the input at an appropriate point in the output.

Our contributions detailed in this chapter are:

- Our key contribution is an approach for fixing bugs based on sequence-to-sequence learning on token sequences. This approach uses a *token copy mechanism* to overcome the unlimited vocabulary problem in source code.

- We present the construction of an *abstract buggy context* that leverages code context for patch generation. The input program token sequences are at the level of full classes and capture long-range dependencies in the fix to be written. We implement our approach in a publicly-available program repair tool called SEQUENCER.

- We evaluate our approach on 4,711 real bug fixing tasks. Contrary to the closest related work [216], we do not assume bugs to be in small methods only. Our golden trained model is able to perfectly fix 950/4,711 testing samples. To the best-of-our knowledge, this is the best result reported on such a task at the time of writing this chapter [216, 182, 155].
- We evaluate our approach on the 75 one-line bugs of Defects4J, which is the most widely used benchmark for evaluating programming repair contributions. SEQUENCER is able to find 2,321 patches for these bugs, 761 compile successfully, 61 are plausible (they pass the full test suite) and 18 are semantically equivalent to the patch written by the human developer.
- We provide a qualitative analysis of 8 interesting repair operators captured by sequence-to-sequence learning on the considered training dataset.

4.2 Background on Neural Machine Translation with Seq-to-Seq Learning

SEQUENCER is based on the idea of receiving buggy code as input and producing fixed code as output. The concept is similar to neural machine translation where the input is a sequence of words in one language and the output is a sequence in another language. In this section, we provide a brief introduction to neural machine translation (NMT).

In neural machine translation, the dominant technique is called "sequence-to-sequence learning", where "sequence" refers to the sequence of words in a sentence. An early example of a sequence-to-sequence network is summarized in Section 2.1.1 and diagrammed in Figure 2.1. A problem with the sequence generation described in Section 2.1.1 is that the output vocabulary that can be produced is limited: only tokens which are in the training set are available for output as y_t. In the case of natural human language, words such as proper names (*e.g.,* Chicago, Stockholm) may be so rare that they do not appear in the training vocabulary, but those words may be necessary for proper output. One successful approach to overcome the vocabulary problem is to use a

copy mechanism [197]. The basic intuition behind this approach is that rare words not available in the vocabulary (*i.e.,* unknown words, referred as `<unk>`), may be directly copied from the input sentence over to the output translated sentence. This relatively simple idea can be successful in many cases - especially when translating sentences containing proper names - where these tokens can be easily copied over.

For example, let's consider the task of translating the following English sentence *"The car is in Chicago"* to French. Let's also assume that all the tokens in the sentence are in the vocabulary, except *"Chicago"*. An NMT model might output the following sentence: *"La voiture est à* `<unk>`*"*. With a copy mechanism, the model would be able to automatically replace the unknown token with one of the tokens from the input sentence, in this case, *"Chicago"*.

The copy mechanism can be particularly relevant for source code, where the size of the vocabulary can be several times the size of a natural language corpus [217]. This results from the fact that developers are not constrained by any vocabulary (*e.g.,* English dictionary) when defining names for variables or methods. This leads to an extremely large vocabulary containing many rare tokens, used infrequently only in specific contexts. Thus, the copy mechanism applied to source code allows a system to generate rare out-of-vocabulary identifier names and numeric values as long as they are somewhere in the input. Furthermore, in natural language, a human recipient may be able to use context to cope with one missing word in an automatically translated sentence. In a programming language, the compiler does not make any semantic inference, and the generation has to be complete. For example, if the code to predict is "if (i < num_cars)", then generating "if (i < int)" is not going to work at all. The prior research on the copy mechanism is presented in Section 2.1.3 and we discuss the mathematics of the copy mechanism in the context of SequenceR in Section 4.3.3.

Tufano *et al.* [216] proposed using NMT with the goal of learning bug-fixing patches by translating the entire buggy method into the corresponding fixed method. Before the translation, the authors perform a code abstraction process which transforms the source code into an abstracted version, which contains: (i) Java keywords and identifiers; (ii) frequent identifiers and literals

(a selection of 300 idioms); (iii) typified IDs (*e.g.,* `METHOD_1`, `VAR_2`) that replace identifiers and literals in the code. In Section 4.6 we highlight differences and improvements introduced in SEQUENCER.

Another approach to addressing the vocabulary size problem in code is to use byte pair encoding (BPE), which has been widely used in NLP and also applied to source code [112]. For SEQUENCER, we did preliminary experiments with BPE to solve the unlimited vocabulary problem, but our early results showed that it is less effective than the copy mechanism.

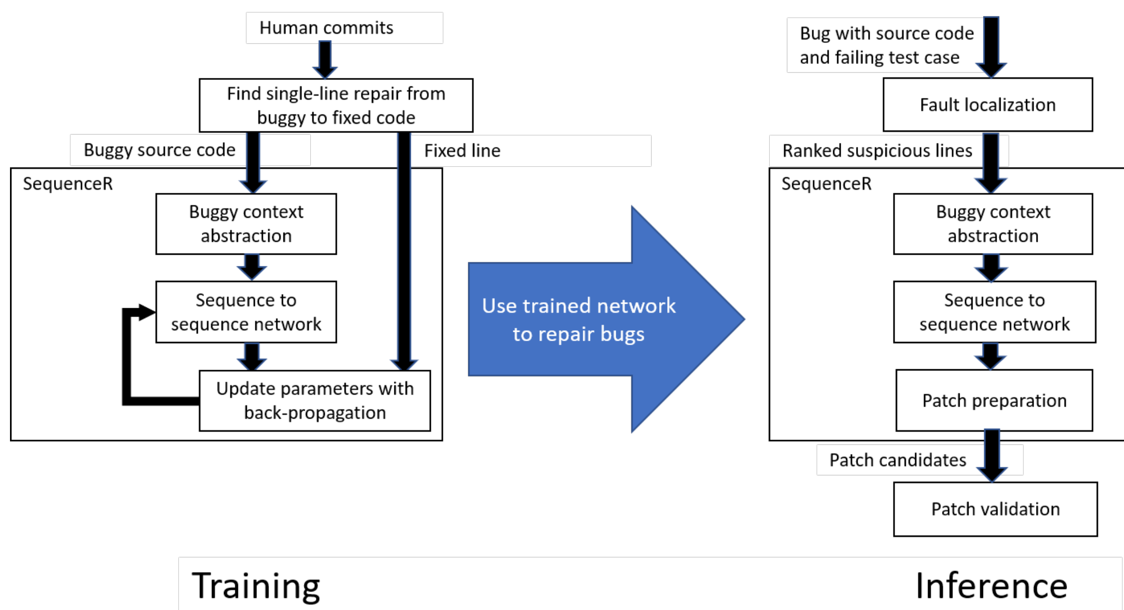


Figure 4.1: Overview of our approach using sequence-to-sequence learning for program repair.

4.3 Approach to Using Seq-to-Seq Learning for Repair

SEQUENCER is a sequence-to-sequence deep learning model that aims at automatically fixing bugs by generating one-line patches (*i.e.,* the bug can be fixed by replacing a single buggy line with a single fixed line). We do not consider line deletion because: 1) it does not require a method for token generation (and is thus less interesting to our research) and 2) if desired, SEQUENCER could be combined with the lightweight Kali [182] to include line deletion. We do not consider line

```
1   class Foo {
2     int i = 0;
3     int bar;
4     Foo(int bar){
5       this.bar = bar;
6     }
7     int decrement(){
8       return bar-1;
9     }
10    int increment(){
11      return bar-1;
12    }
13  }
```

Listing 4.1: Original code

```
1   class Foo {
2     int i = 0;
3     int bar;
4     Foo(int bar){
5     }
6     int decrement(){
7     }
8     int increment(){
9       <START_BUG>
10      return bar-1;
11      <END_BUG>
12    }
13  }
```

Listing 4.2: *abstract buggy context*

```
1   class <unk> {
2     int i = 0;
3     int <unk>;
4     <unk>(int <unk>){
5     }
6     int <unk>(){
7     }
8     int increment(){
9       <START_BUG>
10      return <unk>-1;
11      <END_BUG>
12    }
13  }
```

Listing 4.3: Context with <unk>

Figure 4.2: Illustration of the *abstract buggy context* step in SEQUENCER. b^c is highlighted in yellow, b^m is highlighted in orange and b^l is highlighted in red.

addition because spectrum based fault localization, used in most of the related work, is not effective for line addition patches [249]. We note that in 64% of all 395 bugs in Defects4J are fixed by replacing existing source code [107]. Given a Software System with a faulty behavior (*i.e.,* failing test case), state-of-the-art fault localization techniques are used to identify the buggy method and the suspicious buggy lines. Such techniques have been shown to predict the correct buggy line as one of the top 10 candidates in 44% of the time [249]. SEQUENCER then performs a novel ***Buggy Context Abstraction*** (Section 4.3.2) process which intelligently organizes the fault localization data (*i.e.,* buggy classes, methods, and lines) into a representation that is concise and suitable for the deep learning model yet able to preserve valuable information regarding the context of the bug, which will be used to predict the fix. The representation is then fed to a trained sequence-to-sequence model (Section 4.3.3) which performs ***Patch Inference*** (Section 4.3.4) and is capable of generating multiple single-lines of code that represent the potential one-line patches for the bug. Finally, SEQUENCER in the ***Patch Preparation*** (Section 4.3.5) step generates the concrete patches by formatting the code and replacing the suspicious line with the proposed lines. Figure 4.1 shows the aforementioned steps both for the training phase (left) and inference phase (right). In the

remainder of this section we will discuss the common steps as well as those specific for training and inference.

4.3.1 Problem Definition

Given a buggy system b^s, and test suite t, we assume a fault localization technique, FL, which identifies an ordered set of potential bug locations $l = \{l_1, l_2, ...\}$, where each location l_i consists of the buggy class b_i^c, buggy method b_i^m, and the buggy line b_i^l:

$$l = \{loc \mid loc \in FL(b^s, t)\}$$

$$\forall l_i \in l, l_i = \{b_i^c, b_i^m, b_i^l\} \quad \text{and} \quad b_i^l \subset b_i^m \subset b_i^c$$

The problem is to predict (*i.e.,* generate) a fixed line f_i^l, where l_i is the true bug location, such that by replacing b_i^l with f_i^l in b_i^m, the resulting system f^s passes the test suite and the bug is considered fixed. SEQUENCER tackles this problem by taking as input the fault localization data (*i.e.,* $l = \{l_1, l_2, ...\}$) of a buggy system and attempts to generate fixed line f_i^l for each l_i in order. The b^s, t, l, l_i, b_i^c, b_i^m, b_i^l, f_i^l and f^s notations are used throughout this work.

4.3.2 Abstract Buggy Context

The *context* of a bug plays a fundamental role in understanding the faulty behavior and reasoning about the possible fix. During bug-fixing activities, developers usually identify the buggy lines, then analyze how they interact with the rest of the method's execution, and observe the context (*e.g.,* variables and other methods) in order to reason about the possible fix and possibly select several tokens in the context to build the fixed line [118].

SEQUENCER mimics this process by constructing the *abstract buggy context* and organizing the fault localization data into a representation that is concise yet retains the necessary context that allows the model to predict the possible fix. During this process SEQUENCER needs to balance two contrasting goals: (i) reduce the buggy context into a reasonably concise sequence of tokens (since sequence-to-sequence models suffer from long sentences [50]), (ii) while at the same time

retaining as much information as possible to allow the model to have enough context to predict a possible fix.

Given the bug locations $l = \{l_1, l_2, ...\}$, for each $l_i \in l, l_i = \{b_i^c, b_i^m, b_i^l\}$, SEQUENCER performs the following steps:

Buggy Line `<START_BUG>` is inserted before the first token in the buggy line b_i^l and `<END_BUG>` is inserted after the last token. The rationale is that we would like to propagate the information extracted by the fault localization technique and indicate to the model what is a buggy line. In doing so, we mimic developers who focus on the buggy lines during their bug-fixing activities.

Buggy Method The remainder of the buggy method b_i^m is kept in the representation. The rationale is that the method provides crucial information on where the buggy line is placed and its interaction with the rest of the method.

Buggy Class From the buggy class b_i^c we keep all the instance variables and initializers, along with the signature of the constructor and non-buggy methods even if they are not called in the buggy method. The body of the non-suspcious methods is stripped out. The rationale for this choice is that the model could use variables and method signatures as potential sources when building the fixed line f_i^l.

After these steps, SEQUENCER performs tokenization and truncation to create the *abstract buggy context*. Truncation is used to limit the *abstract buggy context* to a predetermined size in cases where the input sequence is too long. This allows SEQUENCER to process input files of arbitrary size without running out of memory. The truncation process can be summarized as: 1. the truncation size will be chosen such that most input files do not require truncation 2. if the buggy line itself is over the truncation limit, as many tokens as possible from the start of the line are included up to the limit 3. otherwise, the buggy line is included in *abstract buggy context* and twice as many tokens are included before the line as after the line. For example, if the truncation limit is 1,000 tokens and a 5,000 token file has a buggy line with 100 tokens (including the START_BUG

and END_BUG tokens) in the middle of the file, then *abstract buggy context* will consist of 600 tokens before the buggy line, then 100 tokens of the buggy line, then 300 tokens after the buggy line. Generally, truncation will delete the actual class definition from the input, but context near the buggy line is preserved to aid in patch generation.

The *abstract buggy context* represents the input to the sequence-to-sequence network which will be used to predict the fixed line. Internally, *abstract buggy context* is represented as a sequence of tokens belonging to a vocabulary V. The out-of-vocabulary tokens ($token \notin V$) are replaced with the unknown token `<unk>`. In Section 4.3.6 we describe how we empirically derive the vocabulary V and in Section 4.3.3 we explain how the copy mechanism helps in overcoming the unknown tokens problem.

Figure 4.2 shows the output of this process. The original class is presented in Listing 4.1 and Listing 4.2 displays the buggy class after Buggy Context Abstraction. Listing 4.3 illustrates the class when tokens that are out of vocabulary are replaced with the unknown token <unk>. Programming language tokens such as `class` and `int` are not replaced with <unk> because they are part of the vocabulary. Other in-vocabulary tokens include common variable names such as `i`. Our sequence-to-sequence network receives Listing 4.2 as input.

4.3.3 Sequence-to-Sequence Network

In this phase we train SEQUENCER to learn how to generate a fix for a given bug. Specifically, we train a Sequence-to-Sequence Network with Encoder-Decoder model (with attention and copy mechanism) to translate the *abstract buggy context* of a bug to the corresponding target fixed line f_l. To train such a network we rely on a large dataset of bug fixes mined from different sources, explained in Section 4.4.3. The bug fixes are divided into training and testing data, which are used to train and evaluate the Sequence-to-Sequence Network described in Section 4.3.3.

Model

Figure 4.3 shows our model for sequence-to-sequence learning to create Java source code patches. The basis of our model is a recurrent neural network similar to a natural language process-

Neural Network

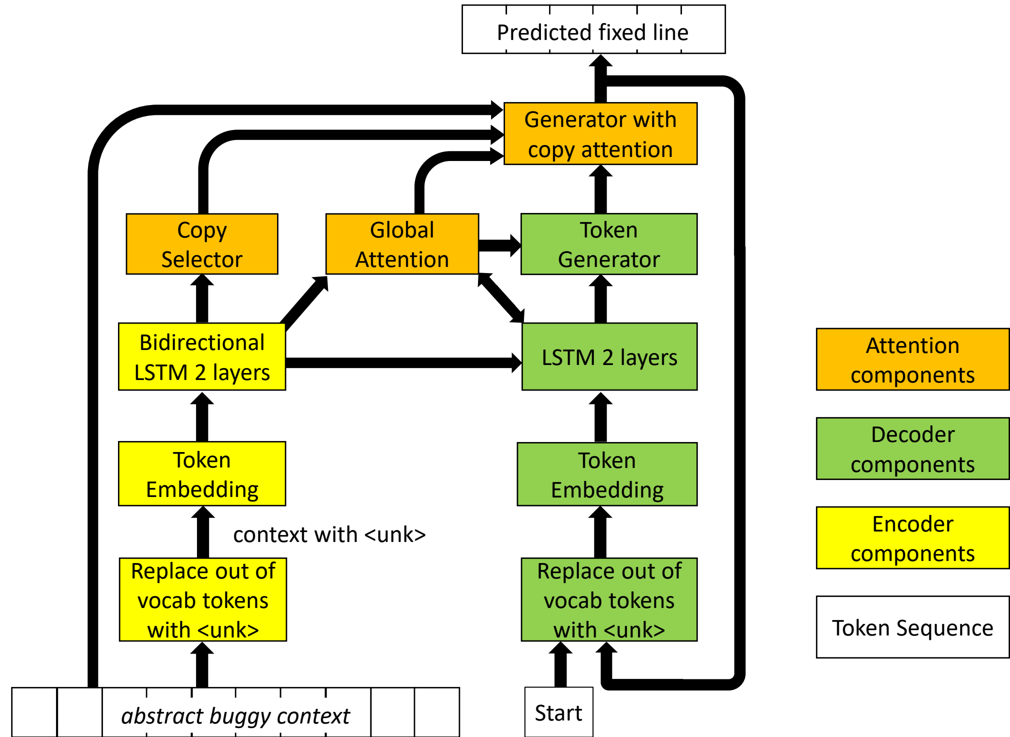


Figure 4.3: Sequence-to-sequence model used in SEQUENCER.

ing architecture [208]. During training, the source token sequence $X = [x_1, ..., x_n]$ (*i.e., abstract buggy context*) is provided to the encoder, where n is the token length of *abstract buggy context*. Then, the decoder produces the target sequence $Y = [y_1, ..., y_m]$ (*i.e.,* the fixed line), where m is the token length of the fixed line. Back propagation is used to update the parameters in the network with stochastic gradient decent during training [115]. The trained parameters are unchanged during inference (patch generation in our case).

Encoder, Decoder, Attention, and Copy Mechanism

We are the first to use the copy mechanism from natural language processing models (which we detail in Section 2.1.3) to solve the unlimited vocabulary problem on source code. The final probability of output token y_j being produced by the network is given in Equation 2.6 and we repeat it here for discussion:

$$P(y_j) = p_{gen} P_V(y_j) + (1 - p_{gen}) \sum_{i:x_i = y_j} a_i^j \tag{4.1}$$

In this equation, the encoder, decoder, and attention networks contribute to $P_V(y_j)$ as tokens are processed during inference. The encoder receives tokens from the *abstract buggy context*. When initialized by the encoder, it begins production of the patch candidate by receiving the special *start* token as input y_0. For each previous output token y_{j-1}, the decoder updates its hidden state as we summarize in Section 2.1.3.

In Section 4.2 we presented the intuition behind the copy mechanism, while in this section we describe how it operates during patch generation. The copy mechanism can significantly improve the performance of the system by allowing the model to select a token from any of the tokens provided in the *abstract buggy context*, even when the tokens are not contained in the training vocabulary. We empirically show the improvements offered by this approach by comparing it to the vanilla sequence-to-sequence model without a copy mechanism in Section 4.4.4. The copy mechanism contributes to Equation 4.1 to produce a token candidate. The copy mechanism calculates p_{gen}, the probability that the decoder generates a token from its initial vocabulary. And $1 - p_{gen}$ is the probability to copy a token from the input tokens depending on the attention vector α^j. Using Equation 4.1, the output token y_j for the current decoder state is selected from the set of all tokens that are either: 1. tokens in the training vocabulary (including the <unk> token) or 2. tokens in the *abstract buggy context*. Although there are no <unk> targets in the training set for patches, if the P_V computation is very uncertain which token is correct, it may happen to have a high likelihood for <unk>. If at the same time, p_{gen} is high then a <unk> token will be produced as the copy mechanism did not replace it. Such outputs are discarded as discussed in Section 4.3.5.

4.3.4 Patch Inference

Once the sequence-to-sequence network is trained, it can be used to generate patches for projects outside of the training dataset. During patch inference, we still generate *abstract buggy context* for the bug, as described in Section 4.3.2. But we will use beam search to generate multiple likely patches for the same buggy line, as done in related work [216, 5]. Beam search works by keeping the n best sequences up to the current decoder state. The successors of these states are

```
1 return 1 ;
2 return i ;
3 return <unk> ;
4 return <unk> + 1 ;
5 return <unk> . <unk>;
```

Listing 4.4: Without copy mechanism

```
return 1 ;
return i ;
return <unk> ;
return bar + 1 ;
return Foo . bar ;
```

Listing 4.5: Network output

```
return 1;
return i;
// discarded
return bar+1;
return Foo.bar;
```

Listing 4.6: After patch preparation

Figure 4.4: Patch preparation step using copy mechanism

computed and ranked based on their cumulative probability; and the next n best sequences are passed to next decoder state. n is often called the width or beam size, and beam search with an infinite n corresponds to doing a complete breath-first-search. In Listing 4.5, we have an example of predictions with beam size 5 for the bug presented in Listing 4.2. Each row is one prediction from the model, representing one potential bug fix, and each of them is further processed by the patch preparation step described below.

4.3.5 Patch Preparation

The raw output from the sequence-to-sequence network cannot be used as a patch directly. First, the predictions might still contain <unk> tokens not handled by the copy mechanism. Listing 4.4 illustrates token values before the copy mechanism replaces <unk> for samples 4 and 5. But the copy mechanism may not replace all such tokens as seen in sample 3 of Listing 4.5. Second, the predictions contain a space between every token, which is not well-formed source code in many cases. (For example, a space is not allowed between the dot separator, ".", and a method call, but a space is required between a type and the corresponding identifier name.)

Consequently, we have a final patch preparation step as follows. We discard all line predictions that contain <unk> and we reformulate the remaining predictions into well-formed source code by removing or adding the required spaces. An example is shown between Listing 4.5 and Listing 4.6, whitespaces are adjusted and the third prediction from Listing 4.5 is removed since it contains <unk> token. Each one of the line predictions is used to create a candidate program by replacing

the original buggy line b_i^l (*i.e.,* the <START_BUG>, <END_BUG> and all tokens in between are replaced with the model output).

More formally, the remaining candidate fixed lines, $cand_i = \{pre_i^1, pre_i^2, ..\}$, will replace the buggy line b_i^l in buggy system b^s and generate candidate patches $\{patch_i^1, patch_i^2, ...\}$, which should be verified with any patch validation technique, such as test suite validation. When the test suite is weak to specify the bug, we can have different patches $\{patch_i^1, patch_j^1, ...\}$ for different bug locations $\{l_i, l_j, ...\}$ that passed the test suite. Then, the correctness can be verified, for example, by manual inspection.

4.3.6 Implementation Details & Parameter Settings

Library. We have implemented our Encoder-Decoder model using OpenNMT-py [117], built in the Python programming language and the PyTorch neural network platform [173].

Vocabulary In this chapter, we consider a vocabulary of the 1,000 most common tokens. To the best of our knowledge, this is one of the largest vocabularies considered for machine learning for patch generation: for comparison, DeepFix [84] has a vocabulary size of 129 words, and Tufano *et al.* [216] considered a vocabulary size of 430 words.

Limit for truncation We truncate if the *abstract buggy context* is longer than 1,000 tokens. This is motivated by Figure 4.6, where we can see that *abstract buggy context* is often less than 1,000 tokens long. SEQUENCER truncates by keeping the buggy line but removing statements, class definitions, and method definitions until *abstract buggy context* is 1,000 tokens or less.

Network parameters We explored a variety of settings and network topologies for SEQUENCER. Most major design decisions are verified with ablation experiments that change a single variable at a time as detailed further in Section 4.5. We train our model with a batch size of 32 for 10,000 iterations. To prevent overfitting, we use a dropout of 0.3. In relation to the components shown in Figure 4.3, below are the primary matrix sizes associated with each component along with a reference to the equations in Section 4.3.3 to which they relate:

- Token embedding (our model uses the same embedding for both g_e and g_d): 1,004x256 (1,000 + 4 special tokens)

- Encoder bidirectional LSTM (part of g_e fuction): 256x256x4x2x2

- Decoder LSTM (part of g_d function): 512x256x4x2 + 256x256x4x2

- Token generator (part of g_a function): 256x1004

- Bridge between encoder and decoder (path for h_i^e to initialize h_0^d): 256x256x2

- Global Attention (α_i^j weights): 256x256 + 512x256

- Copy selector (g_c function): 256x1

We use a beam size of 50 during inference, which is the default value used in the literature [216, 5] and which proves to be good empirically.

Input and output summary The input to SequenceR is a Java class of any size. The non-empty faulty line within a method on which to attempt repair has been identified by another technique (usually line-based fault localization). The output is the fixed line which must have fewer than 100 tokens with our current model.

Usage After SEQUENCER is trained, we can use it to predict fixes to a bug. SEQUENCER takes as input the buggy file and a line number indicating where the bug is. The output is a list of patches in the diff format, so that the user can run their own patch validation step, which could either be test validation or manual inspection.

The source code of SEQUENCER is available at `https://github.com/kth/SequenceR`, together with the best model we have identified and the synthesized patches.

4.4 Evaluation

In this section, we describe our evaluation of SEQUENCER.

4.4.1 Research Questions

The two first research questions focus on machine learning:

- RQ1: To what extent can the fixed line be perfectly predicted?
- RQ2: How often does the copy mechanism generate out-of-vocabulary tokens for a patch, and which parts of *abstract buggy context* are referenced for the copy?

The last two research questions look at the system from a domain-specific perspective: we assess the performance of SEQUENCER from the viewpoint of program repair research.

- RQ3: How effective is SEQUENCER's sequence-to-sequence learning in fixing bugs in the well-established Defects4J benchmark?
- RQ4: What repair operators are captured with sequence-to-sequence learning?

4.4.2 Experimental Methodology

Methodology for RQ1

We train SEQUENCER with the parameter settings described in Section 4.3.6. The training and validation accuracy and perplexity will be plotted. Perplexity (ppl) is a measurement of how well a model predicts a sample and is defined as:

$$ppl(X, Y) = exp\left(\frac{-\sum_{i=1}^{|Y|} \log P(y_i \mid y_{i-1}, \ldots, y_1, X)}{\mid Y \mid}\right)$$

where X is the source sequence, Y is the true target sequence and y_i is the *i*-th target token [117]. Luong *et al.* found a strong correlation between a low perplexity value and high translation quality [148].

The resulting model is tested on our testing dataset, CodRep4 (see Section 4.4.3). Next, in order to compare SEQUENCER against the state-of-the-art approach by Tufano *et al.* [216], we created CodRep4Medium. It is a subset of CodRep4 containing 1,116 samples where the buggy method length is limited to 100 tokens.

Methodology for RQ2

To evaluate the effectiveness of the copy mechanism (described in Section 4.3.3), we consider all samples from CodRep4. For each successfully predicted line, we categorize tokens in that line based on whether the token is in the vocabulary or not. And at the same time, for tokens that are out-of-vocabulary but are copied from the input sequence, we try to find the original location of the copied token. By analyzing the original location of out-of-vocabulary tokens, we can measure the importance of the context, in particular of the *abstract buggy context* we define in this chapter. The copy mechanism allows the system to be more powerful by providing more tokens beyond the vocabulary to be used in the patch.

Methodology for RQ3

We evaluate SEQUENCER on Defects4J [107], which is a collection of reproducible Java bugs. Most recent approaches in program repair research on Java use Defects4J as an evaluation benchmark [155, 240, 239, 230, 105].

Since the scope of this chapter is on one-line patches, we first focus on Defects4J bugs that have been fixed by developers by replacing one single line (there are 75 such bugs). In order to study the effectiveness of sequence-to-sequence itself, we isolate the fault localization step as follows: the input to SEQUENCER is the actual buggy file and the buggy line number. SEQUENCER then produces a list of patches (recall that beam search produces several candidate patches). All patches are compiled and then executed against the test suite written by the developer.

Each candidate patch generated by SEQUENCER is then categorized as follows:

- **Compilable patch**: The patch can be compiled.
- **Plausible patch**: The patch is compilable and passes the test suite. The patch may yet be incorrect because of the overfitting problem [203].
- **Correct patch**: The patch passes the test suite, and is semantically equivalent to the human patch. We hand-check for semantic equivalence for this evaluation.

As per the definitions, there is a strict inclusion structure in those categories: correct patches are necessarily plausible and compilable, plausible patches are necessarily compilable.

Methodology for RQ4

For RQ4, we aim at having a qualitative understanding of the cases for which our sequence-to-sequence repair approach works. This research question is motivated by the need to understand what grammatically correct code transformations are captured by SEQUENCER, even though it is purely a token-based approach with no first class AST or grammar knowledge. For gaining this understanding, we use a mixed method combining grounded theory and targeted analysis. The results would be an understanding of the variety of repair operators and programming language syntax captured by SEQUENCER in cases where the model output correctly matches the test data. For the grounded theory, we have been regularly sampling successful cases, *i.e.,* cases in our testing dataset CodRep4 for which SEQUENCER was able to predict the fixed line, for each case, the authors reached a consensus to know whether 1) the case is interesting from a programming perspective (*e.g.,* it represents a common bug fix pattern), and 2) the case highlights a phenomenon that has already been covered in a previously found case. For the targeted analysis, we specifically searched for 2 kinds of results: cases where the copy mechanism was used and cases where a specific programming construct was involved (method call, field reference and string literals).

4.4.3 Training Data

SEQUENCER is trained based on past modifications made to source code, *i.e.,* it is trained on past commits. In our experiments, we combine two sources of past commits, the CodRep dataset [46] and the Bugs2Fix dataset [216], into what appears to be the largest dataset of one-line bug fixes published to date. Both datasets 1) consider Java code and 2) have been built based on the history of open-source projects.

The CodRep dataset focuses solely on one-line source code fixes (aka one-line patches), it contains 5 datasets curated from real commits on open-source projects. The Bugs2Fix dataset contains diffs mined from Github between March 2010 and October 2017 for bug-fixing commits

(based on heuristics to only consider bug-fixing commits). Neither dataset requires the buggy project to have a test suite for exposing the buggy behavior, instead they are focusing on collecting bug fix commits.

Data Preparation

Since CodRep and Bugs2Fix datasets are in different formats, we first unify these two datasets as follows. First, we only keep diffs from Bugs2Fix which are fixes with a single line replacement. Further, we filter out certain diffs if the changes are outside of a method.

Since the Bugs2Fix dataset comes from a generic bug-fix data mining which includes multi-line fixes and fixes outside of methods, we can look at its statistics to help understand the generality of SEQUENCER. Bugs2Fix contains 92,849 commits. 15,548 of these (17%) are one-line patches within a method, and are within the problem domain of SEQUENCER.

After preparing the dataset, we divide it into training and testing data. CodRep is originally split into 5 parts, numbered from 1 to 5, with each part containing commits from different groups of projects. Our training data consists of CodRep datasets 1,2,3 & 5 and the Bugs2Fix dataset. Our testing data is CodRep dataset 4 (or CodRep4 for short). We chose dataset 4 because it is approximately 20% of the entire CodRep data (data set 1 is less than 10% and data set 5 is over 30%) and because CodRep 4 contains a broad and representative set of projects on which to evaluate [46].

Furthermore, we ensure there are no duplicate samples between the training and testing datasets. During the model setup, we use a random subset of 95% of the training data for model training and 5% as our validation dataset.

Descriptive Statistics of the Datasets

In total, we have 35,578 samples in our training set and 4,711 samples in our testing set.

Input Size Figure 4.6 shows the size distribution of the *abstract buggy context* in number of tokens before truncation is done. The CodRep training data has a median token length of 372; the Bugs2Fix dataset has a median length of 340 tokens; and the testing dataset has a median length

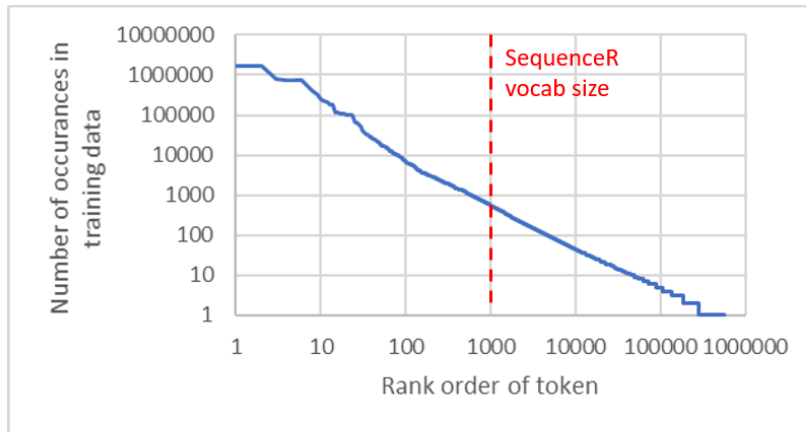


Figure 4.5: Overview of vocabulary: token count occurrences follow a Zipf's law distribution.

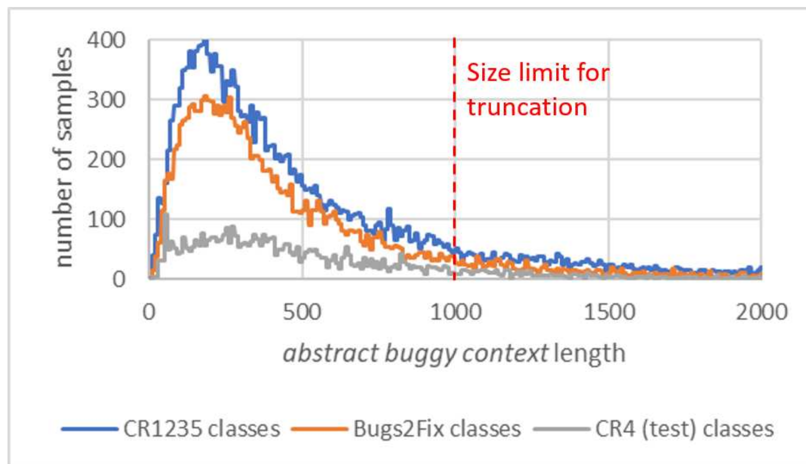


Figure 4.6: Only 14% of samples exceed the 1K token length limit and require truncation.

of 411. These variations are a result of using different Java projects in the datasets, but we observe that the distribution of lengths is similar.

Prediction Size The lines from the *abstract buggy context* samples in our dataset had a median length of 6. 99% of the lines were 30 tokens or fewer, which fits well typical output sizes used for natural language processing. To sum up, the order of magnitude of the sequence-to-sequence prediction receives an input sequence with an average length of 350 tokens and produces an output sequence with an average length of 6 tokens.

Vocabulary Size In our training data, the full vocabulary is 567,304 different tokens. Figure 4.5 shows the distribution of the number of occurrences for the whole vocabulary. It is a typical power-

Table 4.1: Comparison with state-of-the-art approach by Tufano *et al.*

Approach	Prediction Accuracy	
	CodRep4Medium	CodRep4
simple seq2seq line2line, no copy	77/1116 (6.9%)	206/4711 (4.4%)
Tufano *et al.* [216]	157/1116 (14.1%)	N/A
SEQUENCER	344/1116 (30.8%)	950/4711 (20.2%)

law like distribution with a long tail. We limit our training vocabulary to the 1,000 most common tokens.

4.4.4 Experimental Results

Answer to RQ1: Perfect Predictions

We trained our model on a GPU (Nvidia K80) for 1.2 hours. For a typical training run on our golden model, Figure 4.7 shows the training and validation accuracy per token generated (the accuracy for the entire patch would be lower) and Figure 4.8 shows the perplexity (ppl) per token generated over the training and validation datasets. In this particular run, the best results for both the perplexity and accuracy on the validation dataset occur at 10,500 iterations. We chose 10,000 iterations as the standard training time for our model.

CodRep4 On the 4,711 prediction tasks of our best model, SEQUENCER is able to generate the perfect fix in 950 cases (from Table 4.1). In all those cases, the predicted line that replaces the buggy line is exactly the line fix implemented by the developer. The copy mechanism is used in a number of cases, this will be further discussed in section 4.4.4.

Comparison to state-of-the-art To the best of our knowledge, the state-of-the-art approaches are from Tufano *et al.* [216] and Hata *et al.* [87]. We only compare against Tufano *et al.* since their approach has been open sourced while that one of Hata *et al.* was not made available at the time of writing this chapter. The approach used by Tufano *et al.* is limited to fixes only inside small methods, consisting of less than 100 tokens. The limitation is due to the fact that their approach generates the entire fixed source code method as output of the decoder. This means that the decoder may need to generate a long sequence of source code tokens, which is one of the major

challenges for NMT models [119]. SEQUENCER does not make any assumption on the size of the buggy method. In order to compare against [216], we select those 1,116 tasks from CodRep4 where the buggy line resides in a method smaller than 100 tokens. Those 1,116 tasks are called the CodRep4Medium testing dataset.

Our testing accuracy for both CodRep4 and CodRep4Medium are shown in Table 4.1. From the table, we see that the accuracy of SEQUENCER is 344/1,116 (30.8%) while Tufano *et al.* [216] is 157/1,116 (14.1%). This is a clear indicator that SEQUENCER outperforms the current state-of-the-art showing twice as many correct predictions. It shows that our construction of the *abstract buggy context*, together with the copy mechanism, leads to higher accuracy than only having the buggy method as context with a specific encoding for variables. Recent fault localization research [249] indicates that best-in-class techniques can predict the faulty line 44% of the time and the faulty method 68% of the time. If we extrapolate these percentages to our data, SEQUENCER is more likely to find correct one-line patches than the prior work [216] is to find method replacements, and SEQUENCER can process and repair larger methods as demonstrated by the right-hand column of Table 4.1.

We now concentrate on the effectiveness of the approach depending on the buggy method length. Overall, we observe that SEQUENCER has a lower accuracy on longer methods (30.8% accuracy on CodRep4Medium, 20.2% accuracy on CodRep4). This phenomenon is explained by the fact that fixes in long methods are usually more complex and involve more context variables, identifiers and literals that are not easily captured by the learning system. This phenomenon has also been previously observed [216].

Answer to RQ2: Copy Mechanism

We now look at to what extent the copy mechanism is used. Figure 4.9 shows the origin of tokens in successfully predicted lines, per patch size. Let us consider the highest bar, corresponding to all successfully predicted lines consisting of 7 tokens. For those 7-token patches, the black bar means that all tokens are taken from the vocabulary. The non-black bars mean that the copy mechanism has been used to predict the line fix. Overall, there is a minority of patches (216/950,

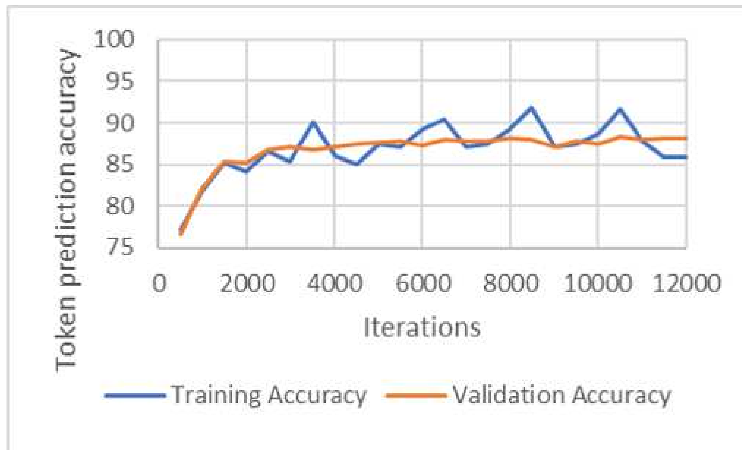


Figure 4.7: Training and validation accuracy

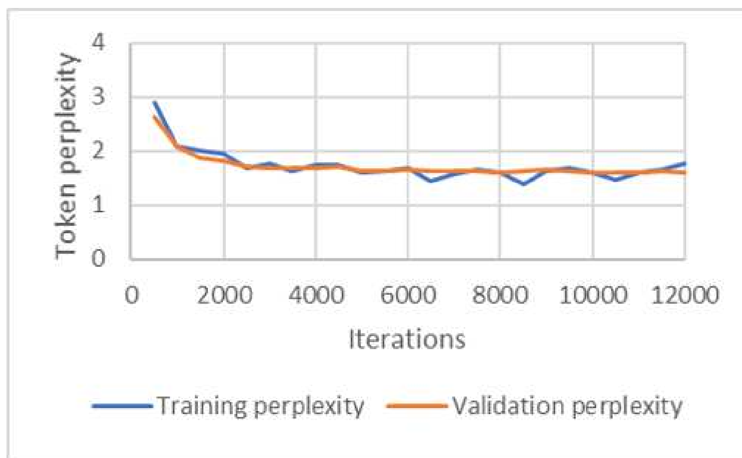


Figure 4.8: Training and validation perplexity

23%) for which all tokens come from the vocabulary. At the extreme, the longest successful patch generated by SEQUENCER was 68 tokens long, but the longest successful patch without the copy mechanism was only 27 tokens long.

Figure 4.9 also lets us analyze the location origin of the copied token. The brown bars represent those patches for which copied tokens all come from the buggy line: this is the majority of cases (641/950, 68%). However, we also observe cases where some copied tokens have been taken from the buggy method (green bars) and cases where the copied tokens has been taken from the buggy class (red bars), *i.e.,* taken from the class context as captured in our encoding.

As an example, Listing 4.7 replaces variable `masterNode` with `nonMasterNode` as in the correct human patch. `nonMasterNode` in the fixed line does not occur in our training data and hence it is not in our 1000 token vocabulary. Therefore, SEQUENCER was able to generate this patch because it copied the out-of-vocabulary token `nonMasterNode` from within the buggy method. As this example is a 4 token long patch, it would contribute to the green bar for patch length 4 in Figure 4.9.

```
while( nonMasterNode == null ) {
  nonMasterNode=randomFrom( internalCluster().getNodeNames());
   if ( nonMasterNode.equals( masterNode ) ) {
-        masterNode = null ;
+        nonMasterNode = null;
   }
}
```

Listing 4.7: Example of the copy mechanism creating a correct patch by incorporating a variable which is not in the vocabulary from the broader context around the buggy line.

Overall, Figure 4.9 shows that the copy mechanism is extensively used (734/950, 77%) and that our class level abstraction enables us to predict difficult cases where only the buggy line or the buggy method would not have been enough.

In order to understand the benefits of context size with the copy mechanism, we measured the distance in tokens to reach a copied token used to generate a patch. In the 87 cases where a copied token was needed from the buggy method b_m, the median distance from the buggy line b_l to the nearest use of the copied token was 9 tokens, 90% of the 87 cases were within 49 tokens of b_l, and 100% were found within a 122 token distance. In the 7 cases when a copied token was needed from the buggy class b_c, the median distance to the copied token from b_m was 25 tokens, and 100% were found within a 241 token distance. In addition to ablation study results discussed is Section 4.5, the preceding data supports our decision to create the *abstract buggy context*.

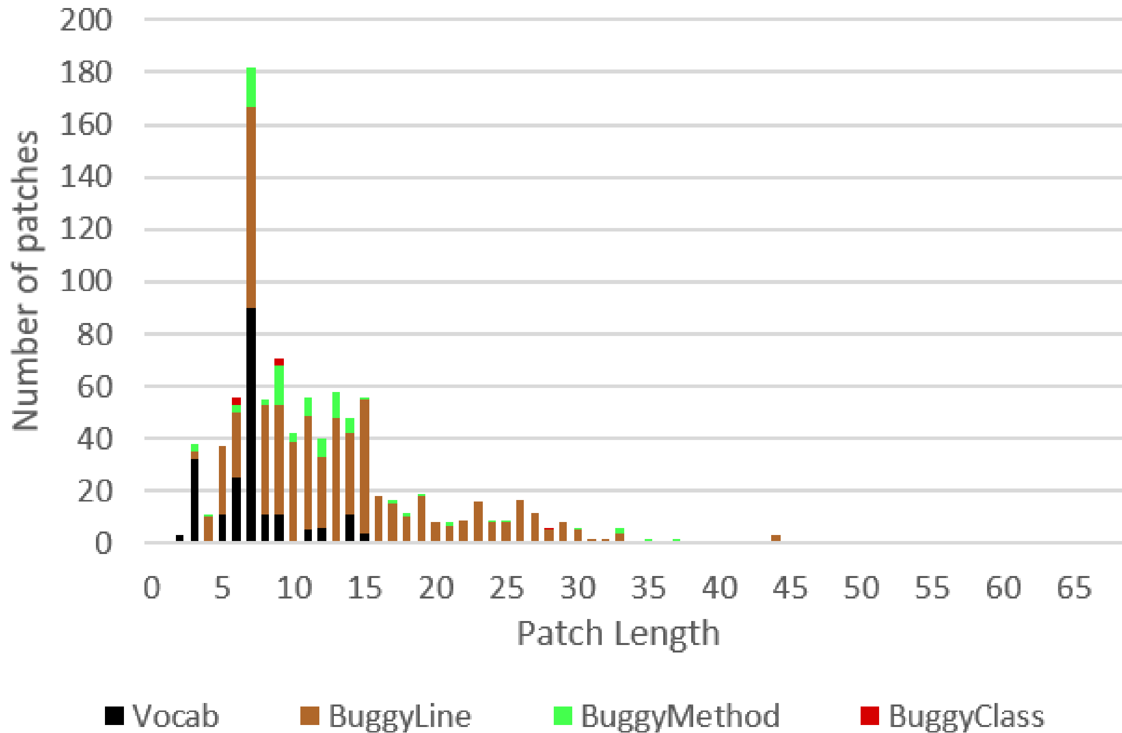


Figure 4.9: Histogram showing correctly generated patches: 1) that only use tokens in our 1,000 token vocabulary, 2) that need to copy tokens from the buggy line, 3) from the buggy method and 4) from the buggy class.

4.4.5 Defects4J Evaluation

As explained in Section 4.4.2, we consider 75 Defects4J bugs that have been fixed with a one-line patch by human developers. In total SEQUENCER finds 2,321 patches for 58 of the 75 bugs. The main reason that we are unable to fix the remaining 17 bugs is due to fact that some bugs are not localized inside a method, which is a requirement for the fault localization step that SEQUENCER assumes as input. Listing 4.8 is one such example where the Defects4j bug is not localized inside a method. We have 2,321 patches instead of 2,900 (58x50) because some predictions are filtered by the patch preparation step (Section 4.3.5), *i.e.,* patches that contain the <unk> token. The statistics about all bugs can be found in Figure 4.10. Out of 75 bugs, SEQUENCER successfully generated at least one patch for 58 bugs, 53 bugs have at least one compilable patch, 19 bugs have at least one patch that passed all the tests (*i.e.,* are plausible) and 14 bugs are considered to be correctly fixed

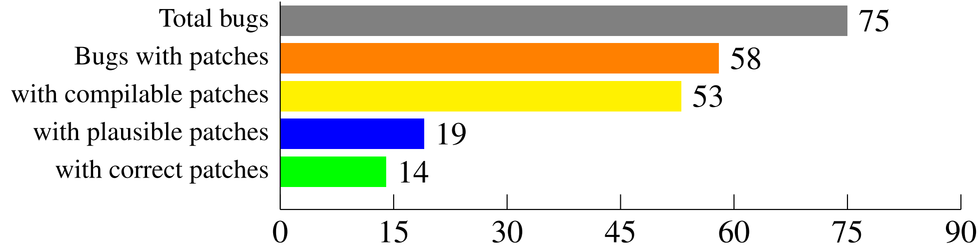


Figure 4.10: SEQUENCER results on the 75 one-line Defects4J bugs.

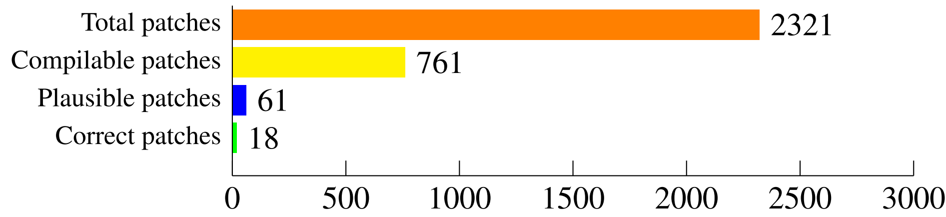


Figure 4.11: Stastistics on patches synthesized by SEQUENCER for the 75 one-line Defects4J bugs.

(semantically identical to the human-written patch). Of these 14 bugs, in 12 cases the plausible patch with the highest ranking in the beam search results was the semantically correct patch.

```
− private static final double DEFAULT_EPSILON = 10e−9;
+ private static final double DEFAULT_EPSILON = 10e−15;
```

Listing 4.8: An example of Defects4J defect (Math 104) where the bug is not localized inside a method. In this case, a class variable is changed.

Figure 4.11 gives a different perspective on this data, focusing on patches (and not bugs). SEQUENCER is able to generate 761 compilable patches (33% of all patches). SEQUENCER finds 61 plausible patches spread over 19 bugs, thus there can be several plausible patches for the same bug, a phenomenon well-known in the program repair field [155]. One reason is that some Defects4J bugs have a weak test suite. To the best of our knowledge, we are the first to report the correctness of patches generated by a sequence-to-sequence model, where correctness means passing the test suite and being semantically equivalent to the human patch. In the end, SEQUENCER is able to generate 18 patches that are semantically equivalent to the correct bug fix.

For SEQUENCER applied to Defects4J bugs, we observe that out of 61 plausible patches, 18 are correct, which is a ratio of 30%. An analysis of prior techniques which used a different benchmarck in C (GenProg [131], RSRepair [180], and AE [229]) shows that they have a correct patch ratio of less than 12% [182]. We did not evaluate SEQUENCER on the same benchmark as this prior work (we target Java not C), but the ratio is evidence that SEQUENCER has learned to produce outputs which represent reasonable patch proposals.

Although we did not directly include fault localization in our evaluation of SEQUENCER, we can estimate the performance of a repair system which includes state-of-the-art fault localization techniques [249] as follows. It has been shown that there is an estimated 44% success of correctly identifying a faulty line in the top 10 candidates. Hence, in order to process 75 total bugs from Defects4J, 750 candidate abstract buggy contexts would need to be prepared for input to our model. We have run fault localization with Gzoltar [41] and found that it successfully localized the faulty line for 9 of the 14 bugs for which SEQUENCER found a correct fix.

Let us now discuss timing. We estimate the machine time required to automatically find patches for 75 bugs with the summation below[5]:

- Estimated time to run fault localization on 75 bugs and identify 10 likely faulty line locations: 112 minutes
- Time to create 750 *abstract buggy contexts* (10 created for each bug): 29 minutes
- Time to create 37,500 patch candidates (50 candidates created from beam size 50 for each *abstract buggy context*): 9 minutes
- Estimated time to prune raw patches down to 23,210 total patches: 2 minutes
- Time to attempt compile on 23,210 patches: 1378 minutes
- Time to run test cases on 7,610 patches: 6287 minutes

[5]Our Defects4J testing was run on an Intel Core i7 at 3.5GHz and our sequence-to-sequence model was run on an Nvidia K80.

- Final result estimated to take 130 total machine hours to find patches which correctly fix 9 bugs.

Listing 4.9 shows the SEQUENCER patch for Math 75, which is semantically equivalent to the human patch. We observe that it contains some unnecessary parentheses, and the same behavior occasionally occurs in other patches found by SEQUENCER. We have observed unnecessary parenthesis in some of the human-generated patches in our training data and SEQUENCER occasionally replicates this human style. In this case, the parentheses do not change the order of evaluation. Therefore the SEQUENCER patch for Math 75 is semantically equivalent to the human patch.

Interestingly, `getPct` is not part of the vocabulary, and it did not appear in the buggy method. The `getPct` method is defined in the same buggy class, as captured by our *abstract buggy context*. In Defects4J, the copy mechanism is also useful to capture the right tokens to add in the patch.

```
- return  getCumPct((Comparable<?>) v);
+ return  getPct ((Comparable<?>) v);  //  Human patch
+ return  getPct (((Comparable<?> )(v))) ;  //  SEQUENCER  patch
```

Listing 4.9: Found patch for Math 75

We now compare those results against the patches found by recent program repair tools that are publicly available. Elixir [191], CapGen [230] and SimFix [105] have reported 26, 22, 34 correctly repaired bugs for all Defects4J bugs, where the patch is identical to the human patch or claimed as correct. Of those correctly repaired bugs, 22, 19 and 17 respectively are for the 75 one-line bugs that we consider for SEQUENCER. We notice that the majority of claimed correct patches are for one-line bugs. We observe that SEQUENCER does not fix more one-line Defects4J bugs.

While Elixir, CapGen, and SimFix are driven with intelligent design and require substantial configuration and handcrafted rules, our goal with SEQUENCER is to be agnostic and to *not* design any repair operator upfront. For example, CapGen implements context-aware operator selection and context-aware ingredient prioritization [230]. The CapGen implementation heavily relies on code transformation tools and carefully selected algorithms/parameters/metrics. In constrast, our SEQUENCER can be considered less heavyweight. We note that the required parameter tuning

in SEQUENCER can easily be performed using grid search or other meta-optimization techniques [30]. To that extent, it is remarkable that such a generic approach is able to learn bug-fixing patterns and synthesizes 18 patches that are semantically equivalent to the human repair, without any static or dynamic analysis. By providing a generic approach, SEQUENCER will improve in the future as machine learning sequence-to-sequence techniques improve, and as more bug fix training data is provided. Also, since SEQUENCER learns repair operators from examples, it could be trained on less common languages (such as COBOL).

We assume perfect fault localization while other related tools ran fault localization to localize the buggy source code. Yet, different papers use different fault localization algorithms, implementations, and granularity (*e.g.,* methods versus line). Liu *et al.* pointed out that because of different assumptions about fault localization, it is hard to compare different repair techniques [142]. By assuming perfect fault localization, we purely focus on the patch generation step of the algorithm.

4.4.6 Qualitative Case Studies

We now answer RQ4 by presenting the diversity of repair operators that are captured by SEQUENCER. These cases are culled from the 950 correct patches SEQUENCER generated for the CodRep4Full test dataset. Both the buggy line that was part of the input is shown and the correct patch which includes examples of repair operators. We also highlight again the effectiveness of the copy mechanism by using a **bold underlined** font for those tokens that were copied (*i.e.,* that are outside the vocabulary of the 1,000 most common tokens).

Case study: method call change

Our training and evaluation data consist of object-oriented Java software. We observe that SEQUENCER captures different kinds of operations related to method calls.

Call change Here a call to method **writeUTF** is replaced by a call to method **writeString**.

```
- out.writeUTF( failure );
+ out.writeString( failure );
```

Listing 4.10: Call change

Call deletion The buggy line chains two method calls; this successful prediction consists of deleting one of them.

```
- FieldMappers x = context.mapperService().smartNameFieldMappers( fieldName );
+ FieldMappers x = context.smartNameFieldMappers( fieldName );
```

Listing 4.11: Call deletion.

Argument addition In this patch, SEQUENCER adds an argument (which in Java, means calling another method).

```
- stage.getViewport().update( width, height );
+ stage.getViewport().update( width, height, true );
```

Listing 4.12: Argument addition

Target change In this successful case, the patch also calls method **isTerminated** but on another target (**scheduledExecutorService** instead of **executorService**, which is copied from the input context).

```
- if( !( executorService.isTerminated() ) ){
+ if( !( scheduledExecutorService.isTerminated() ) ){
```

Listing 4.13: Target change

Case study: if-condition change

SEQUENCER can change if conditions, and in this particular case, removes two clauses from the Boolean formula.

```
- if( ( ( t >= 0 ) && ( t <= 1 ) ) && ( intersection != null ) )
+ if( intersection != null )
```

Listing 4.14: if-condition change

Case study: Java keyword change

SEQUENCER is also able to generate patches involving the replacement of programming language keywords, indicating clues of syntax understanding.

```
– break ;
+ continue ;
```

Listing 4.15: Java keyword change

Case study: change from field access to method call

A good practice of software engineering is to implement encapsulation by calling methods instead of directly accessing fields, this is handled by SEQUENCER as follows (`size` to `size()`)

```
– app . log( "PixmaPackerTest", ( "Number of textures: " + ( atlas . getTextures() . size ) ) );
+ app . log( "PixmaPackerTest", ( "Number of textures: " + ( atlas . getTextures() . size () ) ) );
```

Listing 4.16: change from field access to method call

Case study: off-by-one repair

Finally, SEQUENCER is also able to repair classical off-by-one errors.

```
– nextIndex = currentIndex;
+ nextIndex = ( currentIndex ) – 1;
```

Listing 4.17: off-by-one repair

Overall, SEQUENCER uses all three kinds of token operations: 1. Token deletion, *e.g.,* Listing 4.11; 2. Token addition, *e.g.,* Listing 4.12; 3. Token replacement, *e.g.,* Listing 4.10.

4.5 Ablation Study

We perform an ablation study to understand the relative importance of each component of our approach. The process is as follows. First, we identify the golden model based on a greedy optimization in the parameter search space. This is the model that we described in section 4.4. Then we change one single parameter to a different reasonable value and report the performance on the same testing dataset. The ablation results demonstrate that parameter selections for the golden model produce the highest acceptance rates for the configurations we tested. The model

parameters we found with our dataset are likely to yield reasonable results when training for other computer languages so long as a form of *abstract buggy context* can be done to provide context related to the buggy line. We provide details on our ablation results to aid future researchers in understanding which variables are most likely to improve their own models.

Due to randomness in learning, for each parameter, we run each configuration multiple times and report the mean and standard deviation for the model as recommended for assessment of random algorithms [16]. As our goal is to select the best model for use in our Defects4J evaluation, we use the test set from CodRep4Full to select the best run of each model, hence we report the percentage decrease of the best run for a given model from the best result found with the golden model. Due to computational constraints, we only run each model 10 times; for the 18 configurations reported, almost 200GB of disk storage was used and 400 machine-hours. When using SEQUENCER to learn new datasets, we would recommend a similar approach where a validation set is used to select the best performing model after multitple training runs.

First, we consider the very coarse grain features. Table 4.2 shows the performance of four models, starting from a simplistic seq-to-seq model that only takes a single buggy line b_l as input when learning to produce the fixed line f_l. Then we show beam search, copy, and the use of the *abstract buggy context* improving the model performance. These results confirm our answer to RQ2 that the copy mechanism is essential to the performance of the system.

Second, Table 4.3 shows the results of our 'Golden model' against the results of single specific, targeted changes made to the model. Ablation ID 1 shows that our 10K training limit is sufficient given our training data. ID 2 shows that a vocabulary smaller than 1K tokens performs worse - likely due to a loss of learned tokens that can be used even if an instance of the token is not in the *abstract buggy context*. ID 3 shows that a vocabulary larger than 1K tokens performs worse - perhaps due to the additional tokens having insufficient training examples for learning a proper embedding. To further understand the effect of vocabulary size, we analyzed the raw output of our model before the patch preparation step. For the golden model (vocab=1000), 38% of the generated patches on CodRep4 have <unk> tokens and would be discarded; with ID 2 (700) it is

43%, and with ID 3 (1400) it is 37%. Hence, although a larger vocabulary had fewer raw <unk> tokens, the 1000 token vocabulary was able to produce better optimized models.

ID 4 is about pretraining; in order to provide more opportunities to learn a quality embedding, we created unsupervised pretraining data for the encoder/decoder. Using this unsupervised data did not improve the model, it worsened it.

ID 5 a and b show the value of combining the CodRep and Bugs2Fix data sets to improve the generalization of the model. ID 6 demonstrates the effect of removing the bridge between the encoder and decoder, which improved the mean for the model but tightened the standard deviation and hence produced a lower best result that the golden model. This is perhaps due to the bridge layer allowing for more variation in the encoder hidden state embedding and decoder hidden state embedding.

IDs 7 through 10 demonstrate that our LSTM network is sized correctly; presumably a smaller network cannot generalize on the model data well enough whereas a larger network has too many degrees of freedom. Our speculation is that a 2 layer encoder/decoder network allows the layer connected directly to the token embedding to 'focus' the weight matrix on input syntax while the layer connected to the attention/copy mechanism 'focuses' on output generation. ID 11 shows the loss in accuracy when *abstract buggy context* is reduced to just the buggy line.

ID 12 shows that truncation is necessary otherwise an out-of-memory error crashes the system, due to too many time steps being stored in memory per token in the sequence. ID 13 shows that if we truncated to 4,000 tokens then the system passes, but the increased context size (4,000 vs the golden model 1,000) did not improve accuracy of the model. ID 14 shows that using a 500 token limit for *abstract buggy context* hurts accuracy presumably because there are less opportunities for token copy. We also speculate that a possible advantage of 1K truncation instead of 500 could be that 1K provides a type of unsupervised learning for the encoder hidden states, the global attention, and the copy mechanism.

ID 15 removes the <START_BUG> and <END_BUG> tokens from the *abstract buggy context* input. The target output is still the correct single-line patch. Without these labels, SEQUENCER

Table 4.2: Performance impact of the key features of beam size, copy, and context.

Model description	CR4Full	ratio
50K vocab, no copy, beam size 1, no context	55	baseline
50K vocab, no copy, beam size 50, no context	206	3.7x
1K vocab, copy, beam size 50, no context	826	15.0x
Golden Model (with *abstract buggy context*)	950/4711	17.3x

must learn line break positions and learn a type of fault localization in order to create a valid patch. Because *abstact buggy context* does not include test coverage data or other information useful for fault localization, there is a significant accuracy loss for this ID, but the network was still able to create 356 correct patches.

Our primary use case modeled in this chapter is to use our golden model for SEQUENCER on projects for which it was not trained. This allows for a simpler use model than retraining the model periodically on an ongoing project. ID 16 explores the use case where SEQUENCER is trained with samples from the same projects that the buggy test cases come from. CodRep4 is added to the training set data and then 4,711 random samples are removed for testing (these samples may be from CodRep or Bugs2Fix project files). When the training data includes bugs from the same projects as the test data, we see a 12% improvement in the best model. This use model is viable, but it does require more complete integration of SEQUENCER into a project regression system.

4.6 Related Work

The work presented here is on built on top of two big and active research fields: program repair and machine learning on code. We refer to recent surveys for getting a good overview on them: [161] for program repair and Allamanis *et al.*'s [9] for the latter. In the following, we focus on those works that are about learning and automatic repair.

sk_p is a program repair technique for syntactic and semantic errors in student programs submitted to MOOCs [179]. First, it uses the previous and next statement to predict the statement in the middle, *i.e.,* to replace the current statement. The probability of a patch is the product of

Table 4.3: Results with selected configurations in the parameter neighborhood of the golden model. For ID 0 through 15, results are total exact matches when model is tested on 4,711 testcases from CR4Full. ID16 results selected 4,711 testcases after merging CR1,2,3,4, and 5 with Bugs2Fix.

ID	Model description	mean	SD	max	chng
0	Golden Model	859	61	950	—
1	more training iterations (20K vs 10K)	832	78	901	-5%
2	smaller token vocabulary (700 vs 1000)	824	70	886	-7%
3	larger token vocabulary (1400 vs 1000)	868	32	907	-5%
4	with unsupervised pretraining	821	65	922	-3%
5a	less training data (CR vs CR+Bugs2Fix)	742	47	810	-15%
5b	less training data (Bugs2Fix vs CR+Bugs2Fix)	748	24	785	-17%
6	no bridge layer from encoder to decoder	887	34	942	-1%
7	fewer LSTM layers on enc/dec (1 vs 2)	281	203	513	-46%
8	more LSTM layers on enc/dec (3 vs 2)	833	49	914	-4%
9	fewer LSTMs per layer (128 vs 256)	848	40	888	-11%
10	more LSTMs per layer (512 vs 256)	812	89	907	-5%
11	without context (input only buggy line)	738	63	826	-13%
12	no truncation of *abstract buggy context*	crash			
13	truncate to larger context (4K vs 1K)	848	79	950	-0%
14	truncate to smaller context (500 vs 1K)	826	54	890	-6%
15	remove START_BUG & END_BUG	331	33	412	-57%
16	Intraproject training (4,711 testcases from CR+Bugs2Fix)	984	47	1068	+12%

the probabilities for all chosen statements. As we do, sk_p uses beam search to produce the top n predictions.

Another paper on MOOCs [33] repairs student submissions in Python by combining learning and sketch-based synthesis. The approach by Wang *et al.* [226] considers MOOC but the technique itself is completely different: [226] does deep learning on program traces in order to predict the kind of bug affecting a student submission. The main differences between those works and ours are that 1) we consider a larger context (the buggy class) and 2) we consider real programs for training and testing that are bigger and more complex than student's submissions. Shin *et al.* [201] consider simple programs in the educational programming language Karel. As SEQUENCER, their system predicts to delete, insert or replace tokens. Henkel *et al.* [92] compute an embedding for symbolic

traces and perform a pilot experiment for fixing error-handling code, which is very different from concrete bug fixing as we do here.

DeepFix is a program repair tool for fixing compiler errors in introductory programming courses [84]. The input is the whole program, (100 to 400 tokens long for their data), and the output is a single line fix. The vocabulary size is set to 129, which was enough to map every distinct token type to a unique word in the vocabulary. TRACER is another program repair tool for fixing compiler errors which outperforms DeepFix in terms of success rate [5]. Santos *et al.*'s [192] further refines the idea and evaluates it with an even larger dataset. The focus of those three works and ours is very different, they focus on compiler errors, we focus on logical bugs. For compiler errors, one does not need to consider the whole vocabulary, but only token types. On the contrary, we have to address this problem and we do so by using the copy mechanism.

DeepRepair [232] is an early attempt to integrate machine learning in a program repair loop. DeepRepair leverages learned code similarities, captured with recursive autoencoders [231], to select repair ingredients from code fragments that are similar to the buggy code. Our usage of learning is different, DeepRepair uses machine learning to select interesting code, SEQUENCER uses machine learning to generate the actual patch.

Tufano *et al.* investigated the feasibility of using neural machine translation for learning bug-fixing patches via NMT [216]. The authors first perform a source code abstraction process that relies on a combination of Lexer+Parser which replaces identifiers and literals in the code. The goal of this abstraction is it reduce the vocabulary while keeping the most frequent identifiers/literals. In their work the authors analyzed small methods (no longer than 50 tokens) and medium methods (no longer than 100 tokens) and observed a drop in performance for longer methods. Since their approach takes a buggy method as input and generates the entire fixed method as output, the maximum method length Tufano *et al.* considered is only 100 tokens. Their work addressed the vocabulary problem by renaming rare identifiers through a custom abstraction process. SEQUENCER is different in the following ways. First, we consider the entire context of the buggy class, rather than only the buggy method, in order for the model to access more tokens when predicting the fix.

Second, our abstraction process uniquely utilizes the copy mechanism (which they do not), which allows SEQUENCER to utilize a larger set of tokens when generating the fix and to include information about the context within the *abstract buggy context* in which a token is used. Beyond those two major qualitative differences, a quantitative one is that they only consider small methods, no longer than 100 tokens, while we have no such restriction; SEQUENCER can potentially generate a one-line patch within a method of any size.

Parallel work by Hata *et al.* [87] discusses a similar network architecture, also applied to one-line diffs. The major differences between [87] and our work are the following: First, they do project-specific training, which means that their approach is only evaluated on testing data coming from the same project. On the contrary, we do global training and we show that SEQUENCER captures repair operators applicable to any project. Our qualitative case studies are unique with that respect. Second, they only look at wellformedness of the output, while we also compile and execute the predicted patch. Our work is an end-to-end test-suite based repair approach. Third, their input is limited to the precise buggy code to replace, while SEQUENCER uses *abstract buggy context*, which allows for a broader set of tokens for the copy mechanism to select from.

4.7 SEQUENCER Summation

In this chapter, we have presented a novel approach to program repair, called SEQUENCER, based on sequence-to-sequence learning. Our approach uniquely combines an encoder/decoder architecture with the copy mechanism to overcome the problem of large vocabulary in source code. On a testing dataset of 4,711 tasks taken from projects which were not in the training set, SEQUENCER is able to successfully predict 950 changes. On Defects4J one-line bugs, SEQUENCER produces 61 plausible, test-suite adequate patches. To our knowledge, this work is the first ever to show the effectiveness of the copy mechanism for program repair, which provides a mechanism to alleviate the unlimited vocabulary problem.

This work opens promising research directions. First, we aim to improve and adapt SEQUENCER with the goal of addressing multi-line patches. We believe there are different ways

we can tackle this: (i) for fixes modifying contiguous lines of code (*i.e.,* hunk) we can extend SEQUENCER to learn to generate multiple lines of code as output, with the special tokens (*i.e.,* `<START_BUG>` and `<END_BUG>`) surrounding the entire hunk; (ii) for fixes modifying multiple lines in different locations, we could envision SEQUENCER generating a finite set of combinations of the program containing a predicted fixed line for each of the suspicious locations. Second, there is some preliminary work on tree-to-tree transformation learning [43], which conceptually is very appropriate for code viewed as parse trees. Such techniques may augment or supersede sequence-to-sequence approaches. Finally, the originality of our context abstraction is to capture class-level, long range dependencies: we will study whether such a network architecture is able to capture dependencies beyond that, at the package or application level.

4.8 Continuous Integration with SEQUENCER

Working with a broad team of experienced researchers, we have integrated SEQUENCER into a continuous learning system which we have named R-HERO [27] which is capable of automatically finding and repairing bugs on the GitHub platform.

Figure 4.12 shows the six main building blocks of R-HERO: Continuous integration, Fault localization, Patch generation, Compilation & Test execution, Overfitting prevention, and Pull-request creation. R-HERO stores its knowledge in two databases respectively composed of human-written and machine-synthesized patches.

R-HERO receives and analyzes the events from a continuous integration (CI) system such as Travis CI. It collects commits that result in a passing build as determined by CI. The changes from a commit may or may not have been a bug fix, but the fact that the change passes all tests hints that it is useful training data. The extraction of single-line changes works as follows: for each commit, R-HERO goes over the corresponding *diff* and iterates over each *hunk*. It extracts training data only from hunks that describe single-line changes. This may produce several useful training data points per commit. In other words, R-HERO uses the before and after commit code to train its machine learning model for patch generation. R-HERO currently relies on the SequenceR ML-

based patch generator [48], a sequence-to-sequence neural network model trained to receive buggy code as input and to generate patch proposals as output. At each training step, SequenceR updates its model's weights to determine the tokens that should be output in the proposed patches.

Next, we detail the repair process, shown with blue arrows in Figure 4.12. The repair process is triggered by R-HERO monitoring the continuous integration to detect failing builds, manifested by at least one failing test case. For a given failing build, R-HERO checks out the version of the project that produces the failing build. Then, the fault localization component models the program under repair and pinpoints the locations that could be buggy (file names and line numbers). R-HERO passes the collected locations to SequenceR which generates one or more potential patches for each location. Because the patch generation was trained on any kind of one-line change that results in a passing build, R-HERO repairs both compilation errors and test failures.

R-HERO then validates each candidate patch. It first compiles the patches and executes all the tests to verify if, after applying the patch, the compilation and test execution do not fail anymore. The patches that pass both validations are known as *plausible* patches. Once R-HERO finds *plausible* patches, it assesses them, in order to avoid annoying developers with *overfitting patches*. This check is based on the overfitting detection system *ODS* [245]. ODS is a probabilistic model trained using supervised learning on both human patches (which are assumed to be positive examples), and machine patches (labelled as correct or incorrect), collected from previous program repair research [243].

Finally, when a high-quality patch is identified by ODS, R-HERO submits a pull-request to the corresponding GitHub project which has the failing build. The pull-request message to the developer describes the build failure and the patch, https://bit.ly/3fRIHhd is an example of such a pull-request.

R-HERO demonstrates how a machine learning system such as SEQUENCER can be integrated into a fully automated repair system. Detailed analysis of the R-HERO system is outside the scope of this thesis, but is available in our published work [27].

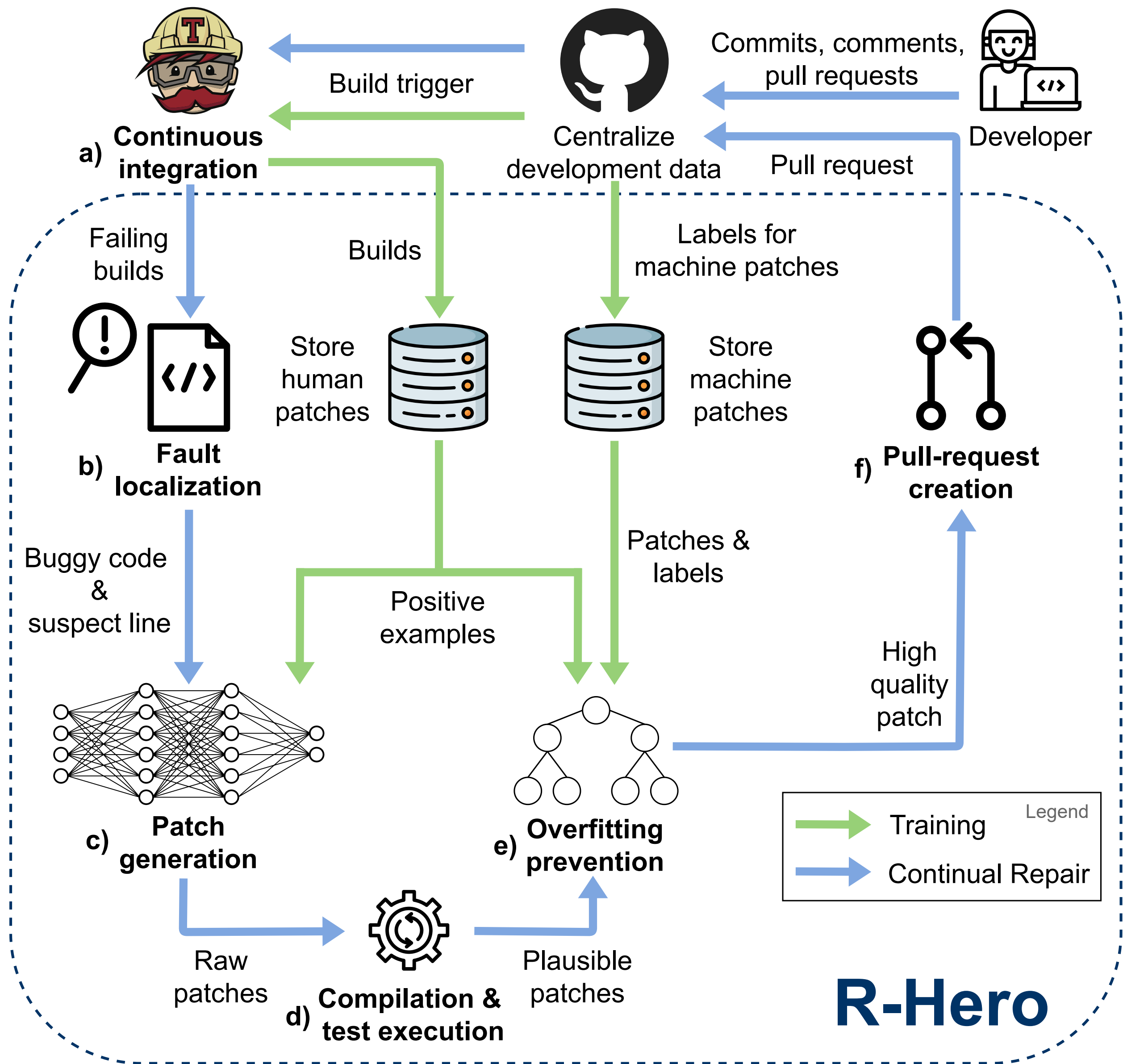


Figure 4.12: Overview of R-HERO, a software bot that learns to generate patches, build after build.

Chapter 5

Neural Transfer Learning for Repairing Security Vulnerabilities in C Code

5.1 Introduction

In the realm of automatic vulnerability fixing [104], there are only a few works on using neural networks and deep learning techniques. One of the reasons is that deep learning models depend on acquiring a massive amount of training data [207], while the amount of confirmed and curated vulnerabilities remains small.

As we introduce in Section 3.2, we address the problem that vulnerability fix datasets are too small to be meaningfully used in a deep-learning model. Our key intuition to mitigate the problem is to use *transfer learning from bug fixing to vulnerability fixing*. Transfer learning is a technique to transfer knowledge learned from one domain to solve problems in related domains, and it is often used to mitigate the problem of small datasets [2]. We leverage the similarity of two related software development tasks: bug fixing and vulnerability fixing. In this context, transfer learning means acquiring generic knowledge from a large bug fixing dataset and then transferring the learned knowledge from the bug fixing task to the vulnerability fixing task by tuning it on a smaller vulnerability fixing dataset. We realize this vision in a novel system for automatically repairing C vulnerabilities called VRepair.

In addition to transfer learning, we develop a change representation technique that allows for multi-line fixes to be encoded efficiently by a machine learning system. Our *token context diff* allows large functions to have changes defined compactly, which has been shown in prior work to improve performance. In contrast to prior change techniques, the *token context diff* makes use of the copy mechanism introduced in Section 2.1.3 which is an efficient mechanism for machine

learning to use. By supporting multi-line changes, we double the number of real fixes which can be attempted by VRepair.

In summary, our contributions are:

- We introduce VRepair, a Transformer Neural Network Model which targets the problem of vulnerability repair. The core novelty of VRepair is to employ transfer learning as follows: it is first trained on a bug fix corpus and then tuned on a vulnerability fix dataset.
- We design a novel code representation for the program repair task with neural networks. Our output code representation supports specifying multi-line patches but is a token difference instead of the entire fixed source code used in recent research [216].
- We empirically demonstrate that on the vulnerability fixing task, the transfer learning VRepair model performs better than the alternatives: 1) VRepair is better than a model trained only on the small vulnerability fix dataset; 2) VRepair is better than a model trained on a large generic bug fix corpus; 3) VRepair is better than a model pre-trained with a denoising task. In addition, we present evidence that the performance of the model trained with transfer learning is stable.
- We share all our code and data for facilitating replication and fostering future research on this topic.

5.2 Background on Software Vulnerabilities and Related Machine Learning

5.2.1 Software Vulnerabilities

A software vulnerability is a weakness in code that can be exploited by an attacker to perform unauthorized actions. For example, one common kind of vulnerability is a buffer overflow, which allows an attacker to overwrite a buffer's boundary and inject malicious code. Another example is an SQL injection, where malicious SQL statements are inserted into executable queries. The

exploitation of vulnerabilities contributes to the hundreds of billions of dollars that cybercrime costs the world economy each year [144].

Each month, thousands of such vulnerabilities are reported to the Common Vulnerabilities and Exposures (CVE) database. Each one of them is assigned a unique identifier. Each vulnerability with a CVE ID is also assigned to a Common Weakness Enumeration (CWE) category representing the generic type of problem.

Definition a *CVE ID* identifies a vulnerability within the Common Vulnerabilities and Exposures database. It is a unique alphanumeric assigned to a specific vulnerability.

For instance, the entry identified as CVE-2019-9208 is a vulnerability in Wireshark due to a null pointer exception. 2019 is the year in which the CVE ID was assigned or the vulnerability was made public; 9208 uniquely identifies the vulnerability within the year.

Definition a *CWE ID* is a Common Weakness Enumeration and identifies the category that a CVE ID is a part of. CWE categories represent general software weaknesses.

For example CVE-2019-9208 is categorized into CWE-476, which is the 'NULL Pointer Dereference' category. In 2020, CWE-79 (Cross-site Scripting), CWE-787 (Out-of-bounds Write) and CWE-20 (Improper Input Validation) are the 3 most common vulnerability categories[6].

Each vulnerability represents a threat until a patch is written by the developers.

5.2.2 Transformers

In VRepair, we use a variant of a seq2seq model called the "Transformer" [220], which is considered the state-of-the-art architecture for seq2seq learning, and is presented in detail in Section 2.1.6 and diagrammed in Figure 2.5.

5.2.3 Transfer Learning

Traditional machine learning approaches learn from examples in one domain and solve the problem in the same domain (*e.g.,* learning to recognize cats from a database of cat images).

[6] `https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html`

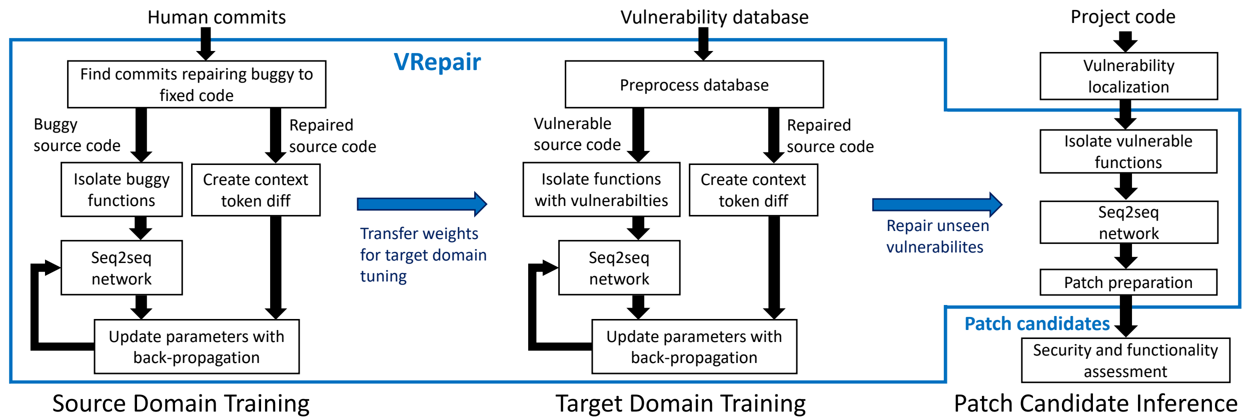


Figure 5.1: The VRepair Workflow: Two phases of training for transfer learning to repair vulnerabilities.

However, as humans, we do better: we are able to apply existing knowledge from other relevant domains to solve new tasks. The approach works well when two tasks share some commonalities, and we are able to solve the new problem faster by starting at a point using previously learned insights. Transfer learning is the concept of transferring knowledge learned in one source task to improve learning in a related target task [215]. The former is called the *source domain* and the latter the *target domain*. Transfer learning is commonly used to mitigate the problem of insufficient training data [2]. If the training data collection is complex and hard for a target task, we can seek a similar source task where massive amounts of training data are available. Then, we can train a machine learning model on the source task with sufficient training data, and tune the trained model on the target task with the limited training data that we have.

Tan *et al.* divide transfer learning approaches into four categories: 1) Instance-based 2) Mapping-based 3) Adversarial-based 4) Network-based [211]. Instance-based transfer learning is about reusing similar examples in the source domain with specific weights as training data in the target domain. Mapping-based transfer learning refers to creating a new data space by transforming inputs from both the source and target domains into a new representation. Adversarial-based transfer learning refers to using techniques similar to generative adversarial networks to find a representation that is suitable on both the source and target domain. Network-based transfer learning is where we reuse the network structure and parameters trained on the source domain and transfer the knowledge to the target domain; this is what we do in this chapter.

5.3 VRepair: Repairing Software Vulnerabilities with Transfer Learning

In this section, we present a novel neural network approach to automatically repair software vulnerabilities.

5.3.1 Overview

VRepair is a neural model to fix software vulnerabilities, based on the state-of-the-art Transformer architecture. The prototype implementation targets C code and is able to repair intraprocedural vulnerabilities in C functions. VRepair uses transfer learning (see subsection 5.2.3) and thus is composed of three phases: source domain training, target domain training, and inference, as shown in Figure 5.1.

Source domain training is our first training phase. We train VRepair using a bug fix corpus because it is relatively easy to collect very large numbers of bug fixes by collecting commits (*e.g.,* on GitHub). While this corpus is not specific to vulnerabilities, per the vision of transfer learning, VRepair will be able to build some knowledge that would turn valuable for fixing vulnerabilities. From a technical perspective, training on this corpus sets the neural network weights of VRepair to some reasonable values with respect to manipulating code and generating source code patches in the considered programming language.

Target domain training is the second phase after the source domain training. In this second phase, we used a high quality dataset of vulnerability fixes. While this dataset only contains vulnerability fixes, its main limitation is its size because vulnerability fixes are notoriously scarce. Based on this dataset, we further tune the weights in the neural network of VRepair. In this phase, VRepair transfers the knowledge learned from fixing bugs to fixing vulnerabilities. As we will demonstrate later in subsection 5.5.2, VRepair performs better with transfer learning than with just training on the small vulnerability fixes dataset.

Inference is the final phase, where VRepair predicts vulnerability fixes on previously unseen functions known to be vulnerable according to a given vulnerability localization technique. VRe-

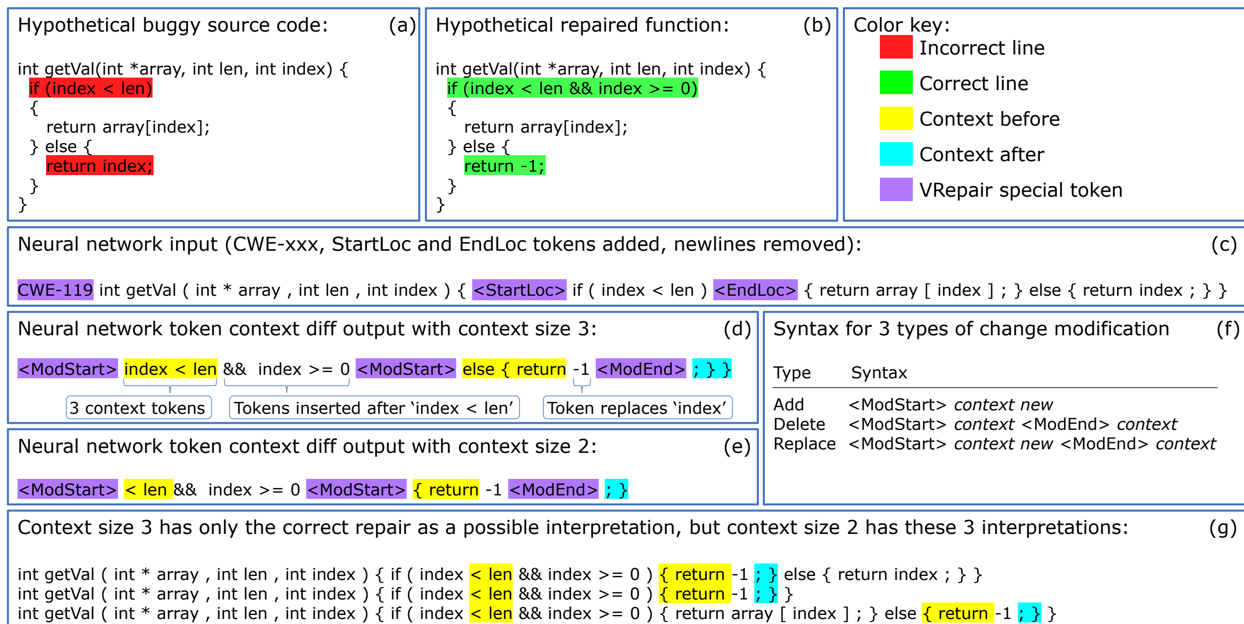


Figure 5.2: The VRepair code representation, based on a token diff script. Outputs are shown for a network trained to produce context size 3 and another network trained to produce context size 2.

pair is agnostic with respect to vulnerability localization, it could be for example a human security expert or a static analyzer dedicated to some vulnerability types. For each potentially vulnerable location, VRepair may output multiple predictions which may be transformable into multiple source code patches. Those tentative patches are meant to be verified in order to check whether the vulnerability has disappeared. This means that, like localization, verification is independent of VRepair. It would typically be another human security expert, or the same static analyzer used before.

5.3.2 Code Representation

Early works on neural repair [216] simply output the full function for the proposed patched code, but the performance of such a system was seen to diminish as the function length increased [216]. Recently, other output representations have been proposed, including outputting a single line [48], and outputting transformations as repair instructions [158, 212, 248]. Inspired by this recent related work, we develop a token-based change description for repairs. It allows for sequences shorter than a full function to be generated, it is easily applied to code, and it can represent any

token change necessary for code repair. To sum up, in VRepair, the core idea is to represent the patch as an edit script at the token level.

The full overview of the input and output formats for VRepair are shown in Figure 5.2. As shown in box (c), Similar to SequenceR which we present in Chapter 4, VRepair adjusts the input by removing all newline characters and identifying the first suspicious code line with the special tokens `<StartLoc>` and `<EndLoc>`. However, because VRepair is acting on C functions and SequenceR acts on Java code, VRepair does not use the *abstract buggy context* from Chapter 4 as input. Those localization tokens come from an external entity localizing the vulnerable line, such as any vulnerability detection system from the corresponding prolific literature [140, 138, 190] or a human security expert. For instance, a static analyzer such as Infer [67] outputs a suspicious line that causes a vulnerability. The input function is also labeled with the vulnerability type suspected by adding a vulnerability type token at the start of the Transformer model input as detailed in subsection 5.3.4 (prefixed by CWE by convention).

Regarding the neural network output, it is known that a key challenge for using neural nets on code is that many functions are made of long token sequences [216], and the performance of seq2seq models decreases with the size of the sequence [52]. In VRepair, the core idea is that the network only outputs the changed source code tokens, and not the whole function, *i.e.,* not the whole token sequence. By doing this: 1) the representation allows for representing multiple changes to a function 2) the representation decreases the size of the output sequence to be generated. As seen in box (d) on Figure 5.2, our system uses a special token to identify a change, `<ModStart>`, followed by $n_{context}$ tokens which locate where the change should start by specifying the tokens before a change. `<ModEnd>` is used when tokens from the input program are being replaced or deleted, and it is followed by $n_{context}$ tokens to specify the completion of a change.

Definition a *token context diff* is a full change description for a function, identifying multiple change locations on multiple lines, using two special tokens `<ModStart>` and `<ModEnd>` to identify the start and end contexts for change locations.

Definition The *context size* is the number of tokens from the source code used to identify the location where a change should be made. The context after the <ModStart> special token indicates where a change should begin, and the context after a <ModEnd> special token indicates where a change that removes or replaces tokens should end.

For example, Figure 5.2 boxes (a) and (b) show a buggy function in which 2 lines are changed to repair the vulnerability. For this repair, the change encoded with a *context size* of 3 results in a 17-token sequence show in box (d). Box (e) shows that the same change can be represented with a *context size* of 2, this would result in a 14 token sequence.

There are 3 types of changes used to patch code: new tokens may be added, tokens may be deleted, or tokens may be replaced. Our novel code representation supports them all, as shown in box (f) of Figure 5.2. Technically, only add and delete would be sufficient, but the replacement change simplifies the specification and allows a *token context diff* to be processed sequentially without backtracking.

A shorter context size minimizes the output length to specify a code change, hence potentially facilitates the learning task. Yet, the issue that arises is that shorter *context sizes* risk having multiple interpretations. For example, in box (d) of Figure 5.2, the context size 3 specification uniquely identifies the start and end locations for the token modifications in the example function. The 3 token start contexts 'index < len' and 'else { return' and the end context '; } }' each only occur once in the buggy source code, so there is no ambiguity about what the resulting repaired function should be. But as shown in box (g), the context size 2 specification has multiple interpretations for the token replacement change. This is because '{ return' and '; }' both occur 2 times in the program resulting in 3 possible ways the output specification can be interpreted. For example, the first time the tokens '{ return' occur in the buggy source code is directly before 'arrax[index]', which is not the correct location to begin the modification to produce the correct repaired function. In this example, we see that a 2-token context is ambiguous whereas 3 tokens are sufficient to uniquely identify a single possible patch. Our pilot experiments

showed that a context size of 3 successfully represents most patches without ambiguity, and hence we use this context size to further develop VRepair.

In addition to confirming a context size of 3, our initial experiments have shown that the *token context diff* approach supports multi-line fixes, which is an improvement on prior work only attempting to repair single lines [48]. We note that 49% of our source domain dataset which we will use in the experiment (see subsection 5.4.2) contains multi-line fixes, demonstrating the importance of the *token context diff*. We also present in subsection 5.5.3 a case study of such a multi-line patch.

In summary, VRepair introduces a novel context-based representation for code patches, usable by a neural network to predict the location where specific tokens need to be added, deleted, or modified. Compared to other works which limit changes to single lines [48] or try to output the entire code of repaired functions [84, 216], our approach allows for complex multi-line changes to be specified with a small number of output tokens.

5.3.3 Tokenization

In this work we use a programming language tokenizer with no subtokenization. We use Clang as the tokenizer because it is the most powerful tokenizer we are aware of, able to tokenize un-preprocessed C source code. We use the copy mechanism to deal with the out-of-vocabulary problem per [48]. We do not use sub-tokenization such as BPE [198] because it increases the input and output length too much (per the literature [52], confirmed in our pilot experiments). Variable renaming is a technique that renames function names, integers, string literals, *etc.* to a pool of pre-defined tokens. For example function names *GetResult* and *UpdateCounter* can be replaced with FUNC_1 and FUNC_2. We do not use variable renaming because it hides valuable information about the variable that can be learned by word embeddings. For example, *GetResult* should be a function that returns a result.

5.3.4 Source Domain Training

In the source domain training phase, we use a corpus of generic bug fixes to train the model on the task of bug fixing. In this phase, the network learns about the core syntax and the essential primitives for patching code. This works as follows. From the bug fix corpus, we extract all functions that are changed. Each function is used as a training sample: the function before the change is seen as buggy, and the function after the change is the fixed version.

We follow the procedure described in subsection 5.3.2 to process the buggy and fixed functions and extract the VRepair representation to be given to the network. All token sequences are preceded with a special token 'CWE-xxx', indicating what type of CWE category this vulnerability belongs to. We add this special token because we believe that vulnerabilities with the same CWE category are fixed in a similar way. For the bug fix corpus where we don't have this information, we use the 'CWE-000' token meaning "generic fix". This special token is mandatory for target domain training and inference as well, as we shall see later.

In machine learning, overfitting can occur when a model has begun to learn the training dataset too well and does not generalize well to unseen data samples. To combat this issue we use the common practice of early stopping [178] during source domain training. For early stopping, a subset of our generic bug fix dataset is withheld for model validation during training. If the validation accuracy does not improve after two evaluations, we stop the source domain training phase and use the model with the highest validation accuracy for target domain training.

5.3.5 Target Domain Training

Next, we use the source domain validation dataset to select the best model produced by source domain training, and tune it on a vulnerability fixes dataset. The intuition is that the knowledge from bug fixing can be transferred to the vulnerability fix domain. We follow the same procedure mentioned in the subsection 5.3.4 to extract the buggy and fixed functions in the VRepair representation from all vulnerability fixes in the vulnerability dataset. We precede all inputs with a special token 'CWE-xxx' identifying the CWE category to which the vulnerability belongs. To ensure

sufficient training data for each 'CWE-xxx' special token, we compute the most common CWE IDs and only keep the CWE IDs with sufficient examples in the vocabulary. The top CWE IDs that we keep cover 80% of all vulnerabilities and the CWE IDs that are not kept in the vocabulary are replaced with 'CWE-000' instead. Early stopping is also used here, and the model with the highest validation accuracy is used for inference, *i.e.,* it is used to fix unforeseen vulnerabilities.

5.3.6 Inference for Patch Synthesis

After the source and target domain training, VRepair is ready to be used as a vulnerability fixing tool. The input provided to VRepair at inference time is the token sequence of a potentially vulnerable function with the first vulnerable line identified with special tokens `<StartLoc>` and `<EndLoc>`, as described in subsection 5.3.2. The input should also be preceded by a special token 'CWE-xxx', representing the type of vulnerability. If the vulnerability was found by a static analyzer, we use a one-to-one mapping from each static analyzer warning to a CWE ID. VRepair then uses the Transformer model to create multiple *token context diff* proposals for a given input. For each prediction of the neural network, VRepair finds the context to apply the patch and applies the predicted patch to create a patched function.

As we discuss in Section 5.3.7, the Transformer model learns to produce outputs that are likely to be correct based on the training data it has processed. Beam search is a well-known method [216] in which outputs from the model can be ordered and the n most likely outputs can be considered by the system. Beam search is an important part of inference in VRepair. We introduce in VRepair a novel kind of beam, which is specific to the code representation introduced in subsection 5.3.2, defined as follows.

Definition The *Neural beam* is the set of predictions created by the neural network only, starting with the most likely one and continuing until the maximum number of proposals configured has been output [216]. Neural beam search works by keeping the n best sequences up to the current decoder state. The successors of these states are computed and ranked based on their cumulative probability, and the next n best sequences are passed to the next decoder state. n is often called

the width or beam size. When increasing the beam size, the benefit of having more proposals is weighed against the cost of evaluating their correctness (such as compilation, running tests, and executing a process to confirm the vulnerability is repaired).

Definition The *Interpretation beam* is the number of programs that can be created from a given input function given a *token context diff* change specification. The interpretation beam is specific to our change representation. For example, Figure 5.2 shows a case where a 3 token context size has only 1 possible application, while the 2 token context can be interpreted in 3 different ways. Hence, the interpretation beam is 3 for context size 2, and 1 for context size 3.

Definition The *VRepair beam* is the combination of the neural beam and the interpretation beam, it is the cartesian product of all programs that can be created from both the Neural beam and the Interpretation beam.

Once VRepair outputs a patch, the patched function is meant to be verified by a human, a test suit, or a static analyzer, depending on the software process. For example, if the vulnerability was found by a static analyzer, the patched program can be verified again by the same static analyzer to confirm that the vulnerability has been fixed by VRepair. As this evaluation may consume time and resources, we evaluate the output of the VRepair system by setting the limit on *VRepair beam* which represents the number of programs proposed by VRepair for evaluation.

5.3.7 Neural Network Architecture

VRepair uses the Transformer architecture [220] as sketched in Figure 2.5. The Transformer for VRepair learns to repair vulnerabilities by first receiving as input the code with a vulnerability encoded in our data representation (subsection 5.3.2). Multiple copies of multi-head attention layers learn hidden representations of the input data. These representations are then used by a second set of multi-head attention layers to produce a table of probabilities for the most likely token to output. In addition, we use a copy mechanism that trains an alternate neural network to learn to copy input tokens to the output in order to reduce the vocabulary size required [82, 48].

The first token to output is based solely on the hidden representations the model has learned to produce from the input code. As tokens are output they are available as input to the model so it can adjust the next token probabilities correctly. For example, in Figure 2.5, after the sequence of tokens '`&& ( d > a *`' has been output, the model predicts that the next token should be '`b`' with a probability of 0.8.

Libraries VRepair is implemented in Python using state-of-the-art tools. We download all GitHub events using GH Archive [73]. For processing the source code we use GCC and Clang. Once the source code is processed, OpenNMT-py is used to train the core Transformer model [117].

Hyperparameters Hyperparameters define the particular model and dataset configuration for VRepair and our primary hyperparameters are presented in Table 5.1. The context size is a hyperparameter specific to VRepair and is discussed in more detail in subsection 5.3.2. We include the most common 80% of CWE IDs among our 2000 word vocabulary. For both learning rate and the hidden and embedding size, we do a hyperparameter search for finding the best model among the values specified in Table 5.1. The learning rate decay is 0.9, meaning that our model decays the learning rate by 0.9 for every $10{,}000$ training steps, starting from step $50{,}000$.

Beam Width An important parameter during inference is the beam width. Since the majority of papers on code generation use a value of 50 for neural beam size [48, 216, 5], we also select this number. Recall that the interpretation beam combines with the neural beam (see subsection 5.3.6), which may increase the number of proposals. Hence, we also set the *VRepair beam* width to 50 (50 possible predictions per input). Our ablation study in section 5.6 provides data demonstrating that a beam width of 50 produces more accurate results than smaller beam widths we tested.

The scripts and data that we use are available at https://github.com/SteveKommrusch/VRepair.

5.3.8 Usage of VRepair

As indicated by Figure 5.1, VRepair is intended to be used for proposing vulnerability fixes within an environment that provides vulnerability detection. Recall that vulnerability detection

Table 5.1: Training Hyperparameters in VRepair.

Hyperparameter	Value
Context size	3
Batch size	32
Vocabulary size	2000
Layers (N copies of attention)	6
Learning rate	[0.005, 0.001, 0.0005, 0.0001, 0.00005]
Hidden and embedding size	[128, 256, 512, 1024]
Feed forward size	2048
Dropout [205]	0.1
Label smoothing [210]	0.1
Attention heads	8
Learning rate decay	0.9

is not in the scope of VRepair per se, tools such as Infer can be used to statically analyze code and indicate a suspicious line containing a vulnerability (see subsection 5.3.2). Once a patch is generated, the expected use case is to recompile the function with the proposed vulnerability fix, to pass a functional test suite if such a test suite exists, and then to pass the vulnerability checker under consideration (such as Infer). As the last step, having the patch reviewed by a human is likely to be done in a practical setting. The human review would occur after a patch has been shown to compile and pass the test suite and vulnerability oracle, in order to save human effort. It is also possible to integrate VRepair in continuous integration, based on the blueprint architecture of R-HERO [27].

5.4 Experimental Protocol

In this section, we describe our methodology for evaluating our approach by defining 4 research questions and how we propose to answer them.

5.4.1 Research Questions

1. RQ1: What is the accuracy of only source or only target domain training on the vulnerability fixing task?

2. RQ2: What is the accuracy of transfer learning with source and target domain training on the vulnerability fixing task?

3. RQ3: What is the accuracy of transfer learning compared to denoising pre-training?

4. RQ4: How do different data split strategies impact the accuracy of models trained with transfer learning and target domain training?

RQ1 will explore the performance of the neural model when only trained with the small dataset available from the target domain (vulnerability fixing), or only trained with a larger dataset from the bug fixing source domain; both being called ‘single domain training’. After exploring single domain training, RQ2 will study the effectiveness of using transfer learning to mitigate the issue of the small vulnerability dataset size demonstrated in RQ1. For RQ2, we study whether a model trained on the source domain (bug fixing) and then tuned with the target domain (vulnerability repair), produces a better result than either source or target domain training alone. Once we have explored transfer learning, in RQ3, we will investigate the possibility of using an unsupervised denoising pre-training technique to alleviate the small dataset problem. In RQ4, we would like to understand how transfer learning’s measurement is affected when we split the evaluation data with different strategies.

5.4.2 Datasets

We create a bug fixing dataset for the purpose of the experiment (see section 5.4.2). For vulnerabilities, we use two existing vulnerability fix datasets from the literature, called Big-Vul [68] and CVEfixes [32] that both consist of confirmed vulnerabilities with CVE IDs.

Bug Fix Corpus

We create a bug fix corpus by mining the GitHub development platform. We follow the procedures in related works [216, 151] and collect our bug fixing dataset by filtering GitHub commits to C code projects based on keywords such as ’bug’ or ’vulnerability’ in the commit message. Filtering commits based on commit messages can be imprecise and generate false positives. However,

this imprecision has minimal effect for our source domain dataset - any code change will likely help train the model on how to propose code patches.

We download 892 million GitHub events from the GH Archive data [73] which happened between 2017-01-01 and 2018-12-31. These events have been triggered by a Github issue creation, an opened pull request, and other development activities. In our case, we focus on push events, which are triggered when at least one commit is pushed to a repository branch. In total there were 478 million push events between 2017-01-01 and 2018-12-31.

Next, we filter bug fix commits as follows. Per the related work [216, 151], we adopt a keyword-based heuristic: if the commit message contains keywords (*fix* OR *solve* OR *repair*) AND (*bug* OR *issue* OR *problem* OR *error* OR *fault* OR *vulnerability*), we consider it a bug fix commit and add it to our corpus. In total, we have analyzed 729 million (728,916,054) commits and selected 21 million (20,568,128) commits identified as bug fix commits. This is a dataset size that goes beyond prior work by Tufano *et al.* [216], who uses 10 million (10,056,052) bug-fixing commits.

In our experiment, we focus on C code as the target programming language for automatic repair. Therefore we further filter the bug fix commits based on the file extension and we remove commits that did not fix any file that ends with '.c', resulting in 910,000 buggy C commits.

For each changed file in each commit, we compare all functions before and after the change to extract functions that changed; we call these function pairs. To identify these function pair changes, we use the GNU compiler preprocessor to remove all comments, and we extract functions with the same function signature in order to compare them. Then, we used Clang [54] to parse and tokenize the function's source code. Within a '.c' file we ensure that only full functions (not prototypes) are considered.

In the end, we obtain 1,838,740 function-level changes, reduced to 655,741 after removing duplicate functions. The large number of duplicates can be explained by code clones, where the same function is implemented in multiple GitHub projects. As detailed in Section 5.3.2, our preferred

context size is 3 tokens; when duplicate functions plus change specifications are removed using this representation, we have $650{,}499$ unique function plus specification samples.

For all of our experiments, we limit the input length of functions to 1000 tokens and the token context diff output to 100 tokens in order to limit the memory needs of the model. A 100 token output limit has been seen to produce quality results for machine learning on code [216]. For all research questions, we partition this dataset into B_{train} (for model training, $534{,}858$ samples) and B_{val} (for model validation, $10{,}000$ samples).

Vulnerability Fix Corpus

We use two existing datasets called Big-Vul [68] and CVEfixes [32] for tuning the model trained on the bug fixing examples. The Big-Vul dataset has been created by crawling CVE databases and extracting vulnerability related information such as CWE ID and CVE ID. Then, depending on the project, the authors developed distinct crawlers for each project's pages to obtain the git commit link that fixes the vulnerability. In total, Big-Vul contains 3754 different vulnerabilities across 348 projects categorized into 91 different CWE IDs, with a time frame spanning from 2002 to 2019.

The CVEfixes dataset is collected in a way similar to the Big-Vul dataset. This dataset contains 5365 vulnerabilities across 1754 projects categorized into 180 different CWE IDs, with a time frame spanning from 1999 to 2021. By having two datasets collected in two independent papers, we can have higher confidence about the generalizability of our conclusions. All the research questions will be done on both vulnerability fix datasets.

5.4.3 Methodology for training with either source or target domain samples

For RQ1, we would like to understand the performance of a model which is trained with source domain only or target domain only. For our source domain dataset, we train on B_{train} and validate with B_{val}, as detailed in section 5.4.2. Next, we randomly divide the vulnerable and fixed function pairs from Big-Vul into training Big-Vul$^{rand}_{train}$, validation Big-Vul$^{rand}_{val}$ and testing Big-Vul$^{rand}_{test}$ sets, with 2226 (70%), 318 (10%) and 636 (20%) examples in each respective set. Similarity, we also

Table 5.2: Number of top-10 CWE examples from the Big-Vul$_{test}^{rand}$ in Big-Vul$_{train}^{rand}$, Big-Vul$_{val}^{rand}$ and Big-Vul$_{test}^{rand}$

CWE ID	Examples in train/valid/test
CWE-119	698/108/187
CWE-20	152/25/51
CWE-125	164/15/45
CWE-264	129/16/32
CWE-399	95/19/29
CWE-200	103/19/28
CWE-476	87/12/26
CWE-284	66/10/26
CWE-189	58/8/26
CWE-362	76/2/26

Table 5.3: Number of top-10 CWE examples from the CVEfixes$_{test}^{rand}$ in CVEfixes$_{train}^{rand}$, CVEfixes$_{val}^{rand}$ and CVEfixesl$_{test}^{rand}$

CWE ID	Examples in train/valid/test
CWE-119	322/37/102
CWE-20	216/26/55
CWE-125	244/34/55
CWE-476	118/23/39
CWE-362	110/9/30
CWE-190	98/17/29
CWE-399	75/8/26
CWE-264	105/18/26
CWE-787	97/15/24
CWE-200	113/12/22

randomly divide the vulnerable and fixed function pairs from CVEfixes into training CVEfixes$_{train}^{rand}$, validation CVEfixes$_{val}^{rand}$ and testing CVEfixes$_{test}^{rand}$ sets, with 2383 (70%), 340 (10%) and 681 (20%) examples in each respective set. Big-Vul$_{test}^{rand}$ and CVEfixes$_{test}^{rand}$ are used to evaluate the models trained with source and target domain training. For all of the data splits in each dataset, we make sure that all of the examples are mutually exclusive, as recommended by Allamanis [7]. The number of examples for the top-10 CWE values from Big-Vul$_{test}^{rand}$ and CVEfixes$_{test}^{rand}$ are given in Table 5.2 and Table 5.3.

For the model with only source domain training, we train on B_{train} and apply early stopping with B_{val}. The model is then evaluated by using a VRepair beam size of 50 on each example in Big-Vul$_{test}^{rand}$ and CVEfixes$_{test}^{rand}$, and the sequence accuracy is used as the performance metric. The sequence accuracy is 1 if any prediction sequence among the 50 outputs matches the ground truth sequence, and it is 0 otherwise. We compute the average test sequence accuracy over all examples in Big-Vul$_{test}^{rand}$ and CVEfixes$_{test}^{rand}$.

For the model with only target domain training, we train on Big-Vul$_{train}^{rand}$ (or CVEfixes$_{train}^{rand}$) and apply early stopping on Big-Vul$_{val}^{rand}$ (or CVEfixes$_{val}^{rand}$). The model is then evaluated by using VRepair beam size 50 on each example in Big-Vul$_{test}^{rand}$ (or CVEfixes$_{test}^{rand}$), and the predictions are used to calculate the sequence accuracy. We hypothesize that the test sequence accuracy with

source domain training will be lower than target domain training. This is because in source domain training, the training dataset is from a different domain, *i.e.,* bug fixes. But the result will show if a model trained on a large number of bug fixes can perform as well as a model trained on a small number of vulnerability repairs on the target task of vulnerability repair.

5.4.4 Methodology for transfer learning with source and target domain data

The state-of-the-art vulnerability fixing models are usually trained on a relatively small vulnerability fixing dataset [49], or generated from synthesized code examples [86]. Based on prior studies showing the effectiveness of large datasets for machine learning [207], we hypothesize that it is hard for a deep learning model to generalize well on such a small dataset. For RQ2, we compare the performance between the transfer learning model and the model only trained on the small vulnerability fix dataset.

We first train the models with source domain training, *i.e.,* we train them on B_{train} and apply early stopping on B_{val}. The models from source domain training are used in the target domain training phase. We continue training the models using Big-Vul$_{train}^{rand}$ (or CVEfixes$_{train}^{rand}$) and the new model is selected based on Big-Vul$_{val}^{rand}$ (or CVEfixes$_{val}^{rand}$) with early stopping. During the target domain training phase, per the standard practice of lowering the learning rate when learning in the target domain [200], we use a learning rate that is one tenth of the learning rate used in the source domain training phase. Finally, the final model is evaluated with Big-Vul$_{test}^{rand}$ (or CVEfixes$_{test}^{rand}$) by using VRepair beam size 50 on each example, which we will use to calculate the sequence accuracy.

5.4.5 Methodology for pre-training with denoising samples

For RQ3, we compare our source domain training with the state of the art pre-training technique of PLBART [4], which is based on denoising. Specifically, we select full functions from our bug fix corpus and apply PLBART's noise function techniques for token masking, token deletion, and token infilling. This results in artificial 'buggy' functions whose target repair is the original function. As per PLBART (and the original BART paper addressing natural language noise func-

Table 5.4: Number of top-10 CWE examples from the Big-Vul$^{year}_{test}$ in Big-Vul$^{year}_{train}$, Big-Vul$^{year}_{val}$ and Big-Vul$^{year}_{test}$

CWE ID	Examples in train/valid/test
CWE-119	835/41/117
CWE-125	71/78/75
CWE-416	48/19/47
CWE-476	64/15/46
CWE-190	48/3/43
CWE-787	14/13/27
CWE-20	165/37/26
CWE-200	103/21/26
CWE-362	68/11/25
CWE-415	9/0/13

Table 5.5: Number of top-10 CWE examples from the CVEfixes$^{year}_{test}$ in CVEfixes$^{year}_{train}$, CVEfixes$^{year}_{val}$ and CVEfixesl$^{year}_{test}$

CWE ID	Examples in train/valid/test
CWE-125	172/35/126
CWE-20	185/11/101
CWE-787	37/44/55
CWE-476	107/25/48
CWE-119	396/20/45
CWE-416	48/24/30
CWE-190	92/24/28
CWE-362	117/11/21
CWE-200	123/14/10
CWE-415	12/17/9

tions [133]): token masking replaces tokens with a special token `<MASK>` in the 'buggy' function; token deletion removes tokens; and token infilling replaces multiple consecutive tokens with a single `<MASK>` token. As in PLBART, we sample from a Poisson distribution to determine the length for the token infilling noise function.

Following this methodology, we generate the pre-training dataset from the bug fix corpus divided into Pre$_{train}$ (728,739 samples) and Pre$_{val}$ (10,000 samples). For the pre-trained model, similar to subsection 5.4.4, we first train the models on Pre$_{train}$ and apply early stopping on Pre$_{val}$. We continue training the models using Big-Vul$^{rand}_{train}$ (or CVEfixes$^{rand}_{train}$) and the new model is selected based on Big-Vul$^{rand}_{val}$ (or CVEfixes$^{rand}_{val}$) with early stopping. Finally, the final model is evaluated with Big-Vul$^{rand}_{test}$ (or CVEfixes$^{rand}_{test}$), and compared against the transfer learning models trained in subsection 5.4.4.

5.4.6 Methodology for data split strategies

In RQ4, we wish to study how the performance of a vulnerability fixing model varies with different data split strategies compared to only target domain training. For this, we divide the Big-Vul and CVEfixes dataset using two strategies: 1) random, as done previously in subsection 5.4.3, subsection 5.4.4, and subsection 5.4.5; 2) time-based;

Time-based splitting First, we sort the Big-Vul dataset based on CVE publication dates. Recall that Big-Vul contains vulnerabilities collected from 2002 to 2019. We create a testing set Big-Vul$_{test}^{year}$ of 603 data points, containing all buggy and fixed function pairs with a published date between 2018-01-01 to 2019-12-31. The validation set Big-Vul$_{val}^{year}$ (302 examples) contains all buggy and fixed function pairs with a published date between 2017-06-01 to 2017-12-31, and the rest are in the training set Big-Vul$_{train}^{year}$ (2272 examples).

Similarly, for the CVEfixes dataset, we create a testing set CVEfixes$_{test}^{year}$ of 794 data points, containing all buggy and fixed function pairs with a published date between 2019-06-01 to 2021-06-09. The validation set CVEfixes$_{val}^{year}$ (324 examples) contains all buggy and fixed function pairs with a published date between 2018-06-01 to 2019-06-01, and the rest are in the training set CVEfixes$_{train}^{year}$ (2286 examples). The dates are chosen so that we have roughly 70% data in the training set, 10% data in the validation set and 20% data in the test set. This data split strategy simulates a vulnerability fixing system that is trained on past vulnerability fixes, and that is used to repair vulnerabilities in the future. The number of examples of the top-10 CWE from Big-Vul$_{test}^{year}$ and CVEfixes$_{test}^{year}$ are given in Table 5.4 and Table 5.5.

For all strategies, we train two different models with source+target domain training (transfer learning) and only target domain training, following the same protocol in subsection 5.4.3, subsection 5.4.4, and subsection 5.4.5, but with different data splits. We report the test sequence accuracy on all data splits.

5.5 Experimental Results

We now present the results of our large scale empirical evaluation of VRepair, per the experimental protocol presented in section 5.4.

5.5.1 Results for training with either source or target domain samples

We study the test sequence accuracy of models trained only with source or only with target domain training. Given our datasets, source domain training means training the model only with bug

Table 5.6: RQ1: Test sequence accuracy on Big-Vul$_{test}^{rand}$ with source/target domain training, we also present the accuracy on the top-10 most common CWE IDs in Big-Vul$_{test}^{rand}$. The absolute numbers represent # of C functions in test dataset.

CWE ID	Source domain training	Target domain training
CWE-119	11.23% (21/187)	8.56% (16/187)
CWE-20	17.64% (9/51)	11.76% (6/51)
CWE-125	20% (9/45)	8.89% (4/45)
CWE-264	9.38% (3/32)	3.13% (1/32)
CWE-399	20.69% (6/29)	6.9% (2/29)
CWE-200	28.57% (8/28)	3.57% (1/28)
CWE-476	26.92% (7/26)	3.85% (1/26)
CWE-284	0% (0/26)	23.08% (6/26)
CWE-189	15.38% (4/26)	3.85% (1/26)
CWE-362	19.23% (5/26)	0% (0/26)
All	14.77% (94/636)	6.76% (43/636)

fix examples from B_{train}, while target domain training uses only Big-Vul$_{train}^{rand}$ (or CVEfixes$_{train}^{rand}$). Both models are evaluated on the vulnerability fixing examples in Big-Vul$_{test}^{rand}$ (or CVEfixes$_{test}^{rand}$). Table 5.6 gives the results on Big-Vul$_{test}^{rand}$. In the first column we list the top-10 most common CWE IDs in Big-Vul$_{test}^{rand}$. The second column shows the performance of the model only trained with source domain training. The third column shows the performance of the model only trained with target domain training. The last row of the table presents the test sequence accuracy on the whole Big-Vul$_{test}^{rand}$. The result for CVEfixes$_{test}^{rand}$ is in Table 5.7.

Even when the model is trained on the different domain of bug fixing, the model still achieves a 14.77% accuracy on Big-Vul$_{test}^{rand}$ and 15.1% accuracy on CVEfixes$_{test}^{rand}$. This is better than the models that are trained with only target domain training, *i.e.,* training on a small vulnerability fix dataset, that has a performance of 6.76% on Big-Vul$_{test}^{rand}$ and 11.29% on CVEfixes$_{test}^{rand}$. The result shows that training on a small dataset indeed is ineffective and even training on a bug fix corpus, which is a different domain but has a bigger dataset size, will increase the performance.

If we compare the results across the top-10 most common CWE IDs in both Big-Vul$_{test}^{rand}$ and CVEfixes$_{test}^{rand}$, we see that the source domain trained models outperform the target domain trained models over almost all CWE categories. In Big-Vul$_{test}^{rand}$, the only exception is for CWE-284 (Im-

Table 5.7: RQ1: Test sequence accuracy on CVEfixes$_{test}^{rand}$ with source/target domain training.

CWE ID	Source domain training	Target domain training
CWE-119	8.82% (9/102)	9.8% (10/102)
CWE-20	20% (11/55)	18.18% (10/55)
CWE-125	21.82% (12/55)	5.45% (3/55)
CWE-476	12.82% (5/39)	10.26% (4/39)
CWE-362	3.33% (1/30)	3.33% (1/30)
CWE-190	24.14% (7/29)	27.59% (8/29)
CWE-399	15.38% (4/26)	3.85% (1/26)
CWE-264	0% (0/26)	3.85% (1/26)
CWE-787	12.5% (3/24)	0% (0/24)
CWE-200	31.82% (7/22)	13.64% (3/22)
All	15.1% (103/682)	11.29% (77/682)

proper Access Control) where the source domain training has 0% accuracy and target domain training has 23.08% accuracy. When inspecting the Big-Vul dataset, we found a vulnerability with CVE ID CVE-2016-3839 and category CWE-284, which changed the vulnerable function across 33 different files. All the vulnerable functions were changed in a similar way, giving enough data for the neural network to learn how to fix this kind of vulnerability. Therefore, the target domain trained model obtains a high test accuracy for CWE-284 on Big-Vul$_{test}^{rand}$.

In Listing 5.1 we show an example of a vulnerability fix that was correctly predicted by the model trained on the target domain only. The vulnerability is CVE-2013-3231 with type CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor) from the Linux project. CVE-2013-3231 can leak sensitive information by a crafted system call[7]. The vulnerability is fixed by setting the msg_namelen to 0, which is correctly predicted by the VRepair model.

[7] https://nvd.nist.gov/vuln/detail/CVE-2013-3231

Table 5.8: RQ2: Test sequence accuracy on Big-Vul$_{test}^{rand}$ with transfer learning. The 'Improvement over source domain training' and 'Improvement over target domain training' columns show the percentage and numerical improvement compared to the result in Table 5.6. Transfer learning achieved better overall performance and on almost all CWE IDs.

CWE ID	Transfer learning	Improvement over source domain training	Improvement over target domain training
CWE-119	18.72% (35/187)	+7.49%/+14	+10.16%/+19
CWE-20	19.61% (10/51)	+1.97%/+1	+7.85%/+4
CWE-125	17.78% (8/45)	-2.22%/-1	+8.89%/+4
CWE-264	6.25% (2/32)	+3.13%/-1	+3.12%/+1
CWE-399	31.03% (9/29)	+10.34%/+3	+24.13%/+7
CWE-200	39.29% (11/28)	+10.72%/+3	+25.72%/+10
CWE-476	26.92% (7/26)	+0%/+0	+23.07%/+6
CWE-284	0% (0/26)	+0%/+0	-23.08%/-6
CWE-189	15.38% (4/26)	+0%/+0	+11.53%/+3
CWE-362	19.23% (5/26)	+0%/+0	+19.23%/+5
All	17.77% (113/636)	+3%/+19	+11.16%/+71

```
 int target; /* Read at least this many bytes */
 long timeo;

+ msg->msg_namelen = 0;
+
 lock_sock(sk);
 copied = -ENOTCONN;
 if (unlikely(sk->sk_type == SOCK_STREAM && sk->sk_state == TCP_LISTEN))
```

Listing 5.1: CVE-2013-3231 is correctly predicted by the model trained with target domain only.

Answer to RQ1: Training a VRepair Transformer on a small vulnerability fix dataset achieves accuracies of 6.76% on Big-Vul and 11.29% on CVEfixes. Surprisingly, by training on a bug fix only, the same neural network achieves better accuracies of 14.77% on Big-Vul and 15.1% on CVEfixes, which shows the ineffectiveness of just training on a small vulnerability dataset.

5.5.2 Results for transfer learning with source and target domain samples

In RQ2, we study the impact of transfer learning, *i.e.,* we measure the performance of a model with target domain training applied on the best model trained with source domain training. The results are given in Table 5.8 and Table 5.9. The first column lists the top-10 most common CWE IDs. The second column presents the performance of the model trained with transfer learning, *i.e.,*

Table 5.9: RQ2: Test sequence accuracy on CVEfixes$_{test}^{rand}$ with transfer learning. The 'Improvement over source domain training' and 'Improvement over target domain training' columns show the percentage and numerical improvement compared to the result in Table 5.7.

CWE ID	Transfer learning	Improvement over source domain training	Improvement over target domain training
CWE-119	18.63% (19/102)	+9.81%/+10	+8.8%/+9
CWE-20	23.64% (13/55)	+3.64%/+2	+5.46%/+3
CWE-125	21.82% (12/55)	+0%/+0	+16.37%/+9
CWE-476	15.38% (6/39)	+2.56%/+1	+5.12%/+2
CWE-362	3.33% (1/30)	+0%/+0	+0%/+0
CWE-190	24.14% (7/29)	+0%/+0	-3.45%/-1
CWE-399	19.23% (5/26)	+3.85%/+1	+15.38%/+4
CWE-264	0% (0/26)	+0%/+0	-3.85%/-1
CWE-787	16.67% (4/24)	+4.17%/+1	+16.67%/+4
CWE-200	40.91% (9/22)	+9.09%/+2	+27.27%/+6
All	19.94% (136/682)	+4.84%/+33	+8.65%/+59

taking the source domain trained model from Table 5.6 (or Table 5.7) and tuning it with target domain training. The third and fourth column gives the performance increase compared to the model with only source or target domain training. The last row of the table presents the test sequence accuracy on the whole Big-Vul$_{test}^{rand}$ (or CVEfixes$_{test}^{rand}$).

The main takeaway is that the transfer learning model achieves the highest test sequence accuracy on both Big-Vul$_{test}^{rand}$ and CVEfixes$_{test}^{rand}$ with an accuracy of 17.77% and 19.94% respectively. Notably, transfer learning is superior to just target domain training on the small vulnerability fix dataset. In addition, transfer learning improves the accuracy over all CWE categories when compared to only source domain training. This shows that the knowledge learned from the bug fix task can indeed be kept and fine-tuned to repair software vulnerabilities and that the previously learned knowledge from source domain training is useful. The result confirms that the bug fixing task and the vulnerability fixing task have similarities and that the bug fixing task can be used to train a model from which knowledge can be transferred.

Now, we compare the results across the top-10 most common CWE IDs in Big-Vul$_{test}^{rand}$ and CVEfixes$_{test}^{rand}$. For all rows except for CWE-125 and CWE-264 in Table 5.8, along with CWE-190 and CWE-264 in Table 5.9, the CWE ID performance is better with the transfer learning model. For CWE-284 (Improper access control), the transfer learning model did not fix a single vulnerability in this category. This is explained by the fact that the transfer learning model prefers generalization

(fixing more CWE types) over specialization. The same phenomenon has been observed in a compiler error fix system, where pre-training lowers the performance of some specific compiler error types [242]. Overall, the best VRepair transfer learning model is able to fix vulnerability types that are both common and rare.

Finally, we discuss a vulnerability where the VRepair transfer learning model is able to predict the exact fix, but not the target domain trained model. Vulnerability CVE-2019-19079, labeled with type CWE-401 (Missing Release of Memory after Effective Lifetime) is shown in Listing 5.2. The vulnerability can cause a denial of service by a memory leak[8]. The vulnerability patch frees the *kbuf* buffer after usage. VRepair with transfer learning successfully predicts the patch, but not the target domain trained model, showing that the source domain training phase helps VRepair to fix a more sophisticated vulnerability.

Listing 5.2 is also a notable case where VRepair successfully predicts a multi line patch. The data representation we described in subsection 5.3.2 allows VRepair to have a concise output representing this multi line patch. The token context diff makes it easier for VRepair to handle multi-line patches, since the output is shorter than the full functions used in related work [216]. Additionally, we note that *kfree* is in our target vocabulary, but *kbuf* is not; so the use of *kbuf* in the patch was done using the copy mechanism.

[8] https://nvd.nist.gov/vuln/detail/CVE-2019-19079

```
if (!kbuf)
   return -ENOMEM;

- if (!copy_from_iter_full(kbuf, len, from))
+ if (!copy_from_iter_full(kbuf, len, from)) {
+   kfree(kbuf);
   return -EFAULT;
+ }

ret = qrtr_endpoint_post(&tun->ep, kbuf, len);

+ kfree(kbuf);
return ret < 0 ? ret : len;
}
```

Listing 5.2: CVE-2019-19079 is correctly predicted by VRepair. It is an example of a multi line vulnerability fix, and the target domain trained model failed to predict the fix.

> Answer to RQ2: By first learning from a large and generic bug fix dataset, and then tuning the model on the smaller vulnerability fix dataset, VRepair achieves better accuracies than just training on the small vulnerability fix dataset (17.77% versus 6.76% on Big-Vul$_{test}^{rand}$, and 19.94% versus 11.29% on CVEfixes$_{test}^{rand}$). Our experiment also shows that the VRepair transfer learning model is able to fix more rare vulnerability types.

5.5.3 Results for pre-training with denoising samples

RQ3 studies the effect of replacing the source domain training phase of transfer learning with denoising as pre-training in VRepair. As explained in subsection 5.4.5, we consider the state-of-the-art pre-training technique from PLBart [4]. In Table 5.10 and Table 5.11, we see that the transfer learning model dominates both test datasets, with 17.77% versus 10.06% on Big-Vul$_{test}^{rand}$ and 19.94% versus 12.02% on CVEfixes$_{test}^{rand}$. This means that first training on the bug fixing task is better than first training on a denoising task. But when comparing the same result against only training on the vulnerability fix dataset (target domain training) in Table 5.6 and Table 5.7, we can see that denoising pre-training does improve the result (6.76% to 10.06% on Big-Vul$_{test}^{rand}$ and

Table 5.10: RQ3: Comparison between transfer learning and pre-training + target domain training on Big-Vul$_{test}^{rand}$. The result shows the merit of first training on a related task, instead of generic pre-training.

CWE ID	Transfer learning	Pre-training + target domain training
CWE-119	18.72% (35/187)	12.83% (24/187)
CWE-20	19.61% (10/51)	21.57% (11/51)
CWE-125	17.78% (8/45)	11.11% (5/45)
CWE-264	6.25% (2/32))	0% (0/32)
CWE-399	31.03% (9/29)	13.79% (4/29)
CWE-200	39.29% (11/28)	14.29% (4/28)
CWE-476	26.92% (7/26)	15.38% (4/26)
CWE-284	0% (0/26)	0% (0/26)
CWE-189	15.38% (4/26)	7.69% (2/26)
CWE-362	19.23% (5/26)	3.85% (1/26)
All	17.77% (113/636)	10.06% (64/636)

Table 5.11: RQ3: Comparison between transfer learning and pre-training + target domain training on CVEfixes$_{test}^{rand}$.

CWE ID	Transfer learning	Pre-training + target domain training
CWE-119	18.63% (19/102)	6.86% (7/102)
CWE-20	23.64% (13/55)	14.55% (8/55)
CWE-125	21.82% (12/55)	7.27% (4/55)
CWE-476	15.38% (6/39)	12.82% (5/39)
CWE-362	3.33% (1/30)	3.33% (1/30)
CWE-190	24.14% (7/29)	20.69% (6/29)
CWE-399	19.23% (5/26)	11.54% (3/26)
CWE-264	0% (0/26)	0% (0/26)
CWE-787	16.67% (4/24)	16.67% (4/24)
CWE-200	40.91% (9/22)	31.82% (7/22)
All	19.94% (136/682)	12.02% (82/682)

11.29% to 12.02% on CVEfixes$_{test}^{rand}$). This shows that denoising pre-training is an alternative if collecting a large labeled source domain dataset is a hard task.

From a qualitative perspective, denoising pre-training has the advantage of being unsupervised and therefore does not require collecting and curating a source domain dataset. Thanks to this property, CodeBERT [70], CuBERT [109] and PLBART [4] all have millions of examples in the pre-training dataset. On the other hand, He, Girshick, and Dollár found that the performance of a pre-trained model scales poorly with the pre-training dataset size [89]. This result shows that even when the size of the source domain dataset (*i.e.,* B_{train} and B_{val}) is slightly smaller than the size of

Table 5.12: Test sequence accuracy on testing data Big-Vul$_{test}^{rand}$

CWE ID	Transfer learning	Target domain training
CWE-119	18.72% (35/187)	8.56% (16/187)
CWE-20	29.62% (20/51)	11.76% (6/51)
CWE-125	17.78% (8/45)	8.89% (4/45)
CWE-264	6.25% (2/32)	3.13% (1/32)
CWE-399	31.03% (9/29)	6.9% (2/29)
CWE-200	39.29% (11/28)	3.57% (1/28)
CWE-476	26.92% (7/26)	3.85% (1/26)
CWE-284	0% (0/26)	23.08% (6/26)
CWE-189	15.38% (4/26)	3.85% (1/26)
CWE-362	19.23% (5/26)	0% (0/26)
All	17.77% (113/636)	6.76% (43/636)

Table 5.13: Test sequence accuracy on testing data Big-Vul$_{test}^{year}$

CWE ID	Transfer learning	Target domain training
CWE-119	10.26% (23/117)	0% (0/117)
CWE-125	22.67% (17/75)	9.33% (7/75)
CWE-416	31.91% (15/47)	0% (0/47)
CWE-476	26.09% (12/46)	2.17% (1/46)
CWE-190	16.28% (7/43)	0% (0/43)
CWE-787	3.7% (1/27)	0% (0/27)
CWE-20	19.23% (5/26)	0% (0/26)
CWE-200	19.23% (5/26)	0% (0/26)
CWE-362	12% (3/25)	0% (0/25)
CWE-415	23.08% (3/13)	0% (0/13)
All	19.4% (117/603)	1.99% (12/603)

Table 5.14: Test sequence accuracy on testing data CVEfixes$_{test}^{rand}$

CWE ID	Transfer learning	Target domain training
CWE-119	18.63% (19/102)	9.8% (10/102)
CWE-20	23.64% (13/55)	18.18% (10/55)
CWE-125	21.82% (12/55)	5.45% (3/55)
CWE-476	15.38% (6/39)	10.26% (4/39)
CWE-362	3.33% (1/30)	3.33% (1/30)
CWE-190	24.14% (7/29)	27.59% (8/29)
CWE-399	19.23% (5/26)	3.85% (1/26)
CWE-264	0% (0/26)	3.85% (1/26)
CWE-787	16.67% (4/24)	0% (0/24)
CWE-200	40.91% (9/22)	13.64% (3/22)
All	19.94% (136/682)	11.29% (77/682)

Table 5.15: Test sequence accuracy on testing data CVEfixes$_{test}^{year}$

CWE ID	Transfer learning	Target domain training
CWE-125	12.7% (16/126)	0% (0/126)
CWE-20	7.92% (8/101)	0% (0/101)
CWE-787	3.64% (2/55)	0% (0/55)
CWE-476	14.58% (7/48)	0% (0/48)
CWE-119	13.33% (6/45)	0% (0/45)
CWE-416	20% (6/30)	0% (0/30)
CWE-190	42.86% (12/28)	0% (0/28)
CWE-362	0% (0/21)	0% (0/21)
CWE-200	0% (0/10)	0% (0/10)
CWE-415	0% (0/9)	0% (0/9)
All	14.71% (117/795)	0% (0/795)

the pre-training dataset (*i.e.,* Pre_{train} and Pre_{val}), transfer learning clearly outperforms denoising pre-training.

Answer to RQ3: In this experiment, transfer learning outperforms denoising pre-training and fine-tuning with datasets of similar size (17.77% versus 10.06% on Big-Vul$_{test}^{rand}$, and 19.94% versus 12.02% on CVEfixes$_{test}^{rand}$). This result shows that the effort of collecting and curating a source domain dataset, arguably a tedious and consuming task, pays off with respect to performance. Overall, the specific source domain task of bug fixing is better than the generic task of denoising.

5.5.4 Results for data split strategies

In RQ4, we explore different ways of creating test datasets, each of them capturing an important facet of transfer learning. Table 5.12, Table 5.13, Table 5.14 and Table 5.15 show the test sequence accuracies for all considered data splitting strategies. For each table, the first column lists the top-10 most common CWE IDs in each data split of the respective dataset. The second column shows the performance of the transfer learning model, which is a model trained on the large bug fix corpus, and then tuned with the vulnerability fix examples. The third column presents the performance of the model trained with the small dataset only, *i.e.,* only target domain training on each training split of different data splits. The last row of each table presents the test sequence accuracy on the whole test set on each data split.

From all tables, we can clearly see a large difference between the performance of transfer learning and target domain learning confirming the results of RQ2. The models that are only trained with the small vulnerability fix dataset (*i.e.,* target domain training), have a performance of 6.76% on Big-Vul$_{test}^{rand}$, 1.99% on Big-Vul$_{test}^{year}$, 11.29% on CVEfixes$_{test}^{rand}$ and 0% on CVEfixes$_{test}^{year}$. They are all worse than the models trained with transfer learning whose performance are 17.77% on Big-Vul$_{test}^{rand}$, 19.4% on Big-Vul$_{test}^{year}$, 19.94% on CVEfixes$_{test}^{rand}$ and 14.71% on CVEfixes$_{test}^{year}$. For both data split strategies (random and time-based) the transfer learning model outperforms the target domain learning model showing that the result is independent of the data split. In other words, this is an additional piece of evidence about the superiority of transfer learning.

Interestingly, the performance of models trained with target domain training only varies a lot between different data split strategies. It varies from 0% in CVEfixes$_{test}^{year}$ on the CVEfixes dataset to 11.29% in CVEfixes$_{test}^{rand}$. In other words, the performance of a vulnerability fixing system trained on a small dataset is unstable; it is highly dependent on how the data is divided into training, validation, and testing data. This has been observed in prior research as well [134]: the knowledge learned from a small dataset is often unreliable. On the other hand, transfer learning models have stable performance, staying in a high range from 14.71% on CVEfixes$_{test}^{year}$ to 19.94% on CVEfixes$_{test}^{rand}$.

When comparing the test sequence accuracy for each vulnerability type, *i.e.,* different CWE IDs, the transfer learning models also surpassed models trained only on the target domain. The only exceptions are CWE-284 in Big-Vul$_{test}^{rand}$, CWE-190 and CWE-264 in CVEfixes$_{test}^{rand}$. It may be that the nature of the fixes for the CWEs varies over time such that transfer learning was more beneficial for the Big-Vul$_{test}^{year}$ and CVEfixes$_{test}^{year}$ splits. When comparing the overall performance on Big-Vul$_{test}^{rand}$ and CVEfixes$_{test}^{rand}$, target domain training versus transfer learning, the transfer learning model is still to be preferred.

We argue that splitting the vulnerability fix dataset based on time is the most appropriate split for evaluating a vulnerability repair system. By splitting the dataset based on time and having the newest vulnerabilities in the test set, we simulate a scenario where VRepair is trained on past vulnerability fixes and evaluated on future vulnerabilities. To this extent, we suggest that Big-Vul$_{test}^{year}$ and CVEfixes$_{test}^{year}$ are the most representative approximations of the performance of VRepair in practice.

Answer to RQ4: For the two considered data splitting strategies (random and time-based) transfer learning achieves a more stable accuracies (14.71% to 19.94% for transfer learning, and 0% to 11.29% for models only trained on the small vulnerability fix dataset). This validates the core intuition of VRepair: transfer learning overcomes the scarcity of vulnerability data for deep learning and yields reliable effectiveness.

5.6 Ablation Study

During the development of VRepair, we explored alternative architectures and data formats. To validate our explorations, we perform a systematic ablation study. Table 5.16 highlights 8 comparisons that are of particular interest. The description column explains the way the model was varied which produced the results. All the models in the ablation study are evaluated on the Big-Vul dataset. ID 0 is our golden model of VRepair. We include it in the table for easier comparison against other architectures and dataset formats that we have tried. ID 1 summarizes the benefit of using beam search from the neural network model for our problem. In this comparison,

the pass rate on our target dataset Big-Vul$_{test}^{rand}$ increased from 5.66% with a single model output to 17.77% with a beam size of 50. ID 2 indicates the benefit of the Transformer architecture for our problem. Here we see that the Transformer model outperforms the bidirectional RNN model.

ID 3 highlights the importance of fault localization for model performance. When a model was trained and tested on raw vulnerable functions without any identification of the vulnerable line(s), we saw rather poor performance. When all vulnerable lines in the source file are identified, the model was much more likely to predict the correct patch to the function. Also, by localizing where the patch should be applied, there are fewer possible interpretations for the context matching to align with and this improves the viability of smaller context sizes. Given that we rely on fault localization for VRepair, labeling all possible vulnerable lines would be ideal and results in a 22.64% test sequence accuracy. However, most static analysis tools will not provide this information.

If we limit our model to only predict changes for a contiguous block of lines (i.e, one or more lines after the erroneous line is identified), then the test sequence accuracy is 23.96%. We note that single block repairs (which include single-line repairs) form 57.84% of our dataset, so even with 23.96% success, the single block model solves $23.96\% \times 57.84\% = 13.86\%$ of the test data, which is less than the 17.77% our golden model solves. Ultimately, our model identifies only the first vulnerable line for input, but repairs may be done to lines after the identified line also. We consider it a reasonable assumption that a fault localization tool, *e.g.,* static analyzer or human, would tend to identify the first faulty line. In other words, we make no assumption if the first buggy vulnerable line is the only vulnerable line, meaning that VRepair still can fix multi line vulnerabilities, as we have seen in Listing 5.2.

IDs 4 to 7 show our ablation of the key hyperparameters in our golden model. ID 8 shows why our vocabulary size is set to a rather low value of 2000, which is done thanks to using the copy mechanism. We clearly see that by using the copy mechanism, the model can handle the out-of-vocabulary problem well with a low vocabulary size.

Table 5.16: A sample of ablation results over 8 hyperparameters.

ID	Description	Results
0	**VRepair**	**17.77%**
1	Beam width 1	5.66%
	Beam width 10	13.99%
	Beam width 50 (**VRepair**)	**17.77%**
2	RNN seq2seq	15.41%
	Transformer seq2seq (**VRepair**)	**17.77%**
3	No vulnerable line identifier	10.52%
	All vulnerable lines ID'd	22.64%
	Single block ID'd	23.96%
	First vulnerable line ID'd (**VRepair**)	**17.77%**
4	Learn rate 0.0001 (**VRepair**)	**17.77%**
	Learn rate 0.0005	0%
	Learn rate 0.00005	16.67%
5	Hidden size 1024 (**VRepair**)	**17.77%**
	Hidden size 512	16.19%
6	Layers 6 (**VRepair**)	**17.77%**
	Layers 4	17.45%
7	Dropout 0.1 (**VRepair**)	**17.77%**
	Dropout 0.0	15.25%
8	Vocabulary size 2000 (**VRepair**)	**17.77%**
	Vocabulary size 1000	16.82%

5.7 Related Work

5.7.1 Vulnerability Fixing with Learning

We include related work that fixes vulnerabilities with some kind of learning, meaning that the system should learn fix patterns from a dataset, instead of generating repairs from a set of pre-defined repair templates.

Vurle is a template based approach to repair vulnerability by learning from previous examples [152]. They first extract the edit from the AST diff between the buggy and fixed source code. All edits are then categorized into different edit groups to generate repair templates. To generate a patch, Vurle identifies the best matched edit group and applies the repair template. Vurle is an early work that does not use any deep learning techniques and is only trained and evaluated on 279 vulnerabilities. In contrast, VRepair is trained and evaluated on 3754 vulnerabilities and is based on deep learning techniques.

Harer *et al.* [86] proposed using generative adversarial networks (GAN), to repair software vulnerabilities. They employed a traditional neural machine translation (NMT) model as the generator to generate the examples to confuse the discriminator. The discriminator task is to distinguish the NMT generated fixes from the real fixes. The trained GAN model is evaluated on the SATE IV dataset [170] consisting of synthetic vulnerabilities. Although Harer *et al.* trained and evaluated their work on a dataset of 117,738 functions, the main limitation is that the dataset is fully synthetic. In contrast, our results have better external validity, because they are only based on real world code data.

SeqTrans is the closest related work [49]. SeqTrans is a Transformer based seq2seq neural network with attention and copy mechanisms that aims to fix Java vulnerabilities. Similar to VRepair, they also first train on a bug fix corpus, and then fine-tune on a vulnerability fix dataset. Their input representation is the vulnerable statement, and the statements that defined the variables used in the vulnerable statement. To reduce the vocabulary, the variable names in the buggy and fixed methods are renamed and they use BPE to further tokenize the tokens. VRepair is different from SeqTrans in that we target fixing C vulnerabilities instead of Java vulnerabilities. Importantly, we utilize the

copy mechanism to deal with tokens outside the training vocabulary and not BPE. Our evaluation is done based on two independent vulnerability fix datasets to increase the validity. VRepair's code representation is also different, and allows us to represent multi-line fixes in a compact way. We have shown in Listing 5.2 that we are able to fix multi-line vulnerabilities, while SeqTrans focused on single statement vulnerabilities.

5.7.2 Vulnerability Fixing without Learning

We include related work that fixes vulnerabilities without learning. Usually, these works do so by having a pool of pre-defined repair templates or using different program analysis techniques to detect and repair the vulnerability.

Senx is an automatic program repair method that generates vulnerability patches using safety properties [100]. A safety property is an expression that can be mapped to variables in the program, and it corresponds to a vulnerability type. It uses the safety property to identify and localize the vulnerability and then Senx generates the patch. In the implementation, three safety properties are implemented: buffer overflow, bad cast, and integer overflow. For buffer overflow and bad cast, Senx generates a patch where the error handling code is called. But for integer overflow, Senx will generate a patch where the vulnerability is actually fixed.

Mayhem is a cyber reasoning system that won the DARPA Cyber Grand Challenge in 2016 [17]. It is able to generate test cases that expose a vulnerability and to generate the corresponding binary patch. The patches are based on runtime property checking, *i.e.,* assertions that are likely to be true for a correctly behaving program and false for a vulnerable program. To avoid inserting many unnecessary checks, Mayhem uses formal methods to decide which runtime checks to add.

Fuzzbuster is also a cyber reasoning system that has participated in the DARPA Cyber Grand Challenge [166]. It can find security flaws using symbolic execution and fuzz testing, along with generating binary patches to prevent vulnerability. The patches typically shield the function from malicious input, such as a simple filter rule that blocks certain inputs.

ExtractFix is an automated program repair tool that can fix vulnerabilities that can cause a crash [72]. It works by first capturing constraints on inputs that must be satisfied, the constraints capturing the properties for all possible inputs. Then, they find the candidate fix locations by using the crash location as a starting point, and use control/data dependency analysis to find candidate fix locations. The constraints, together with the candidate fix locations, are used to generate a patch such that the constraints cannot be violated at the crash location in the patched program.

MemFix is a static analysis based repair tool for fixing memory deallocation errors in C programs [132], *e.g.,* memory leak, double free, and use-after-free errors. It does so by first generating candidate patches for each allocated object using a static analyzer. The correct patches are the candidate patches that correctly deallocate all object states. It works by reducing the problem into an exact cover problem and using an SAT solver to find the solution.

LeakFix is an automatic program repair tool for fixing memory leaks in C programs [71]. It first builds the control flow graph of the program and uses intra-procedural analysis to detect and fix memory leaks. The generated fix is checked against a set of procedures that ensures the patch does not interrupt the normal execution of the program.

ClearView is an early work that can protect deployed software by automatically patching vulnerabilities [176]. To do so, ClearView first observes the behavior of software during normal execution and learns invariants that are always satisfied. It uses the learned invariant to detect and repair failures. The patch can change register values and memory locations, or change the control flow. It has been able to resist 10 attacks from an external Red Team.

IntRepair is a repair tool that can detect, repair, and validate patches of integer overflows [164]. It uses symbolic execution to detect integer overflows by comparing the execution graphs with three preconditions. Once an integer overflow is detected, a repair pattern is selected and applied. The resulting patched program is executed again with symbolic execution to check if the integer overflow is repaired.

SoupInt is a system for diagnosing and patching integer overflow exploits [228]. Given an attack instance, SoupInt will first determine if the attack instance is an integer overflow exploit

using dynamic data flow analysis. Then, a patch can be generated with different policies. It can either change the control flow or perform a controlled exit.

VRepair is different than all these vulnerability fix systems since it is a learning based system. The major difference is that VRepair is not targeting a specific vulnerability, rather it is able to fix multiple types of vulnerabilities, as seen in Table 5.8 and Table 5.9. VRepair is also designed to be able to learn arbitrary vulnerability fixes, rather than having to manually design a repair strategy for each type of vulnerability.

5.7.3 Vulnerability Datasets

Big-Vul is a C/C++ code vulnerability dataset collected from open sourced GitHub projects [68]. It contains 3754 vulnerabilities with 91 different vulnerability types extracted from 348 GitHub projects. Each vulnerability has a list of attributes, including the CVE ID, CWE ID, commit ID, *etc.* This dataset can be used for vulnerability detection, vulnerability fixing, and analyzing vulnerabilities.

CVEfixes [32] is a vulnerability fix dataset based on CVE records from National Vulnerability Database (NVD) . The vulnerability fixes are automatically gathered from the associated open-source repositories. It contains CVEs up to 9 June 2021, with 5365 CVE records in 1754 projects.

Reis & Abreu collected a dataset of security patches by mining the entire CVE details database [187]. This dataset contains 5942 security patches gathered from 1339 projects with 146 different vulnerability types in 20 languages. They also collected 110k non-security related commits which are useful in training a system for identifying security relevant commits.

A vulnerability detection system, VulDeePecker [138], created the Code Gadget Database. The dataset contains 61,638 C/C++ code gadgets, in which 17,725 of them are labeled as vulnerable and the remaining 43,913 code gadgets are labeled as not vulnerable. The Code Gadget Database only focuses on two types of vulnerability categories: buffer error (CWE-119) and resource management error (CWE-399). By contrast, the Big-Vul that is used in this chapter contains vulnerabilities with 91 different CWE IDs.

Ponta *et al.* created a manually curated dataset of Java vulnerability fixes [177] which has been used to train SeqTrans, a vulnerability fixing system [49] presented above. The dataset has been collected through a vulnerability assessment tool called "project KB", which is open sourced. In total, Ponta's dataset contains 624 vulnerabilities collected from 205 open sourced Java projects.

SATE IV is a vulnerability fix dataset originally used to evaluate static analysis tools on the task of finding security relevant defects [170]. SATE IV consists of $117{,}738$ synthetic C/C++ functions with vulnerabilities spanning across 116 CWE IDs, where $41{,}171$ functions contain a vulnerability and $76{,}567$ do not.

VulinOSS is a vulnerability dataset gathered from open source projects [76]. The dataset is created from the vulnerability reports of the National Vulnerability Database. They manually assessed the dataset to remove any projects that do not have a publicly available repository. In total, the dataset contains $17{,}738$ vulnerabilities from 153 projects across 219 programming languages.

Cao *et al.* collected a C/C++ vulnerability dataset from GitHub and the National Vulnerability Database, consisting of 2149 vulnerabilities [42]. They used the dataset to train a bi-directional graph neural network for a vulnerability detection system.

5.7.4 Machine Learning on Code

Here we present related works that use machine learning on source code. In general, we refer the reader to the survey by Allamanis *et al.* for a comprehensive overview on the field [9].

One of the first papers that used seq2seq learning on source code is DeepFix [84], which is about fixing compiler errors. They encode the erroneous program by replacing variable names with a pre-defined pool of identifiers to reduce the vocabulary. The program is then tokenized, and a line number token is added for each line, so that the output can predict a fix for a single code line. C programs written by students are encoded and used to train DeepFix, and it was able to fix 27% of all compiler errors completely and 19% of them partially.

Tufano *et al.* investigated the possibility of using seq2seq learning to learn bug fixing patches in the wild [216]. Similar to our approach, they collected a bug fix corpus by filtering commits from

GitHub based on the commit message. The input is the buggy function code, where identifiers are replaced with a general name, such as STRING_X, INT_X, and FLOAT_X. Then, they trained a seq2seq model where the output is the fixed function code. They found that seq2seq learning is a viable approach to learn how to fix code, but found that the model's performance decreased with the output length.

SequenceR learned to fix single line Java bugs using seq2seq learning [48]. The input to SequenceR is the buggy class, where unnecessary context is removed, such as the function body of non-buggy functions. The input also encodes the fault localization information by surrounding the suspicious line with <START_BUG> and <END_BUG>. The output is the code line to replace the suspicious line. They used the copy mechanism to deal with the vocabulary problem, instead of renaming identifiers. SequenceR was evaluated on Defects4J [107] and was able to fix 14 out of 75 bugs.

CoCoNut is an approach that combines multiple seq2seq models to repair bugs in Java programs [150]. They used two encoders to encode the buggy program; one encoder generates a representation for the buggy line, and another encoder generates a representation for the context. These two representations are then merged to predict the bug fix. They trained multiple different models and used ensemble learning to combine the predictions from all models.

DLFix is an automated program repair tool for fixing Java bugs [136]. It is different from other approaches in that it uses tree-based RNN and has two layers. The first layer is a tree-based RNN that encodes the AST that surrounds the buggy source code, which is passed to the second layer. The second layer takes the context vector and learns how to transform the buggy sub-tree. It will generate multiple patches for a single bug, and it deploys a character level CNN to rank all the generated patches.

Code2Seq [11] uses AST paths to represent a code snippet for the task of code summarization. An AST path is a path between two leaf nodes in the AST. They sample multiple AST paths and combine them using a neural network to generate sequences such as function name, code caption,

and code documentation. They found that by using AST paths, Code2Seq can achieve better performance than seq2seq neural network and other types of neural networks.

CODIT is a tree-based neural network to predict code changes [44]. The neural network takes the AST as input, and generates the prediction in two steps. The first step is a tree-to-tree model that predicts the structural changes on the AST, and the second step is the generation of code fragments. They evaluate CODIT on real bug fixes and found that it outperforms seq2seq alternatives.

Yin *et al.* worked on the problem of learning distributed representation of edits [246]. The edits they learned are edits on Wikipedia articles and edits on GitHub C# projects. They found that similar edits are grouped together when visualizing the edit representation, and that transferring the neural representation of edits to to a new context is indeed feasible.

The major difference between VRepair and these works is that VRepair is targeting vulnerabilities. In this work, we evaluate VRepair with the most notable vulnerabilities which have a CVE ID; they are all confirmed vulnerabilities reported by security researchers. These vulnerabilities are essentially different from related works which consider functional bugs, for example from Defects4J [107].

5.7.5 Transfer Learning in Software Engineering

To the best of our knowledge, there are only a few works that use transfer learning in the software engineering domain, and none of them use it for generating code fixes. Recently, Ding has done a comprehensive study on applying transfer learning to different software engineering problems [66], such as code documentation generation and source code summarization. He found that transfer learning improves performance on all problems, especially when the dataset is tiny and could be easily overfitted. In our work, we deploy transfer learning for vulnerability fixing and show that it also improves accuracy. Also, we show that our model trained with transfer learning has a more stable and superior performance compared to training on the small dataset.

Huang, Zhou, and Chin used transfer learning to avoid the problem of having a small dataset for the error type classification task, *i.e.,* predict the vulnerability type [99]. They trained a Trans-

former model on the small dataset and achieved 7.1% accuracy. When training first on a bigger source dataset and tuned afterward on the same dataset, Huang *et al.* [99] managed to get 69.1% accuracy. However, their work is not about transfer learning and therefore the transfer learning experiment was relatively simple and short. In our work, we conduct multiple experiments to show the advantages of using transfer learning for vulnerability fixing.

Sharma *et al.* have applied transfer learning on the task of detecting code smell [199]. They train the models on C# code and use them to detect code smells with Java code examples and vice versa. They found that such models achieved similar performance to models directly trained on the same program language. This is different from our work; we observed that transfer learning improved the performance and made the performance more stable.

Ma *et al.* used character, word and sentence-level features from the input text to recognize API uses. They adopted transfer learning to adapt a neural model trained on one API library to another API library. They found that the more similar the API libraries are, the more effective transfer learning is. Overall, this related literature, together with our work presented here, show the applicability and benefits of using transfer learning in software engineering.

5.8 Conclusion

In this chapter, we have proposed VRepair, a novel system for automatically fixing C vulnerabilities using neural networks. To tackle the problem of having only small vulnerability fix datasets, our key insight is to use transfer learning: we first train VRepair with a big bug-fix corpus, and then we fine-tune on a curated vulnerability fix dataset.

We have performed a series of original and large scale experiments. In our experiments, we found that VRepair's transfer learning outperforms a neural network that is only trained on the small vulnerability fix dataset, attaining 17.77% accuracy instead of 6.76% on Big-Vul dataset, and 19.94% accuracy instead of 11.29% on CVEfixes dataset. This result shows that transfer learning is a promising way to address the small dataset problem in the domain of machine learning for vulnerability fixing. Put in another way, our experiments show the knowledge learned from the

bug fixing task can be transferred to the vulnerability fixing task, which has never been studied before to the best of our knowledge.

In the future, we would like to explore the possibility of using an even larger source domain dataset. We believe that one can achieve results similar to those of GPT-3 – a massive, generic natural language model with 175 billion parameters [36]. In the context of software engineering, we envision that we could train a single model on all code changes from GitHub, not just bug fixes, and tune it on tasks such as code comment generation, function name prediction, and vulnerability fixing.

Part III

Producing Verifiable Program Equivalence Proofs Using Machine Learning

Chapter 6

Proving Equivalence Between Complex Linear Algebra Expressions With Graph-to-Sequence Neural Models

6.1 Introduction

Deep neural network systems have excelled at a variety of classification and reinforcement learning tasks [81]. However, their stochastic nature tends to hinder their deployment for automated program analysis: ensuring the correctness of the solution produced is often required, e.g., when determining the semantics equivalence between two programs (or symbolic expressions). In Chapters 4 and 5 we presented techniques for generating bug and vulnerability patches which could be partially checked by test suites or static analyzers but currently still engage a human for final approval of the output.

In this work we target the problem of automatically computing whether two input symbolic expressions are semantically equivalent [110], under a well-defined axiomatic system for equivalence using semantics-preserving rewrite rules [64]. Additionally, *we want a guarantee of correctness for the output of our neural network.* As we introduce in Section 3.3, program equivalence is summarized as determining whether two programs would always produce the same outputs for all possible inputs, and is a central problem in computing [110, 77, 221].

We propose a method for generating training samples using probabilistic applications of production rules within a formal grammar, and then develop a graph-to-sequence [137, 29] neural network system for program equivalence, trained to learn and combine rewrite rules to rewrite one program into another. It can *deterministically* prove equivalence, entirely avoids false positives, and quickly invalidates incorrect answers produced by the network (no deterministic answer is provided in this case, only a probability of non-equivalence). We make the following contributions:

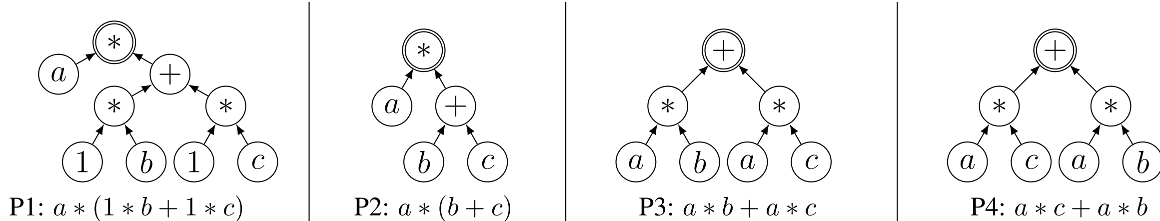


Figure 6.1: Examples of Computations Shown as Symbolic Expressions and Dataflow Graphs

1. We design, implement and evaluate two competing approaches using graph-to-sequence neural network systems to generate proofs of equivalence. We provide the first implementation of such graph-to-sequence systems in the popular OpenNMT-py framework [117].

2. We present a complete implementation of our system operating on a rich language for multi-type linear algebra expressions. Our system provides a correct rewrite rule sequence between two equivalent programs for 93% of the 10,000 test cases. The correctness of the rewrite rule is deterministically checkable in all cases in negligible time.

The rest of this chapter is organized as follows. Sec. 6.2 outlines the program equivalence problem we address, and motivates our proposed approach. Sec. 6.3 formalizes the equivalence problem addressed. Automatic sample generation is discussed in Sec. 6.4 before Sec. 6.5 which introduces our DNN system, its overall design principles and key components. A complete experimental evaluation of our system is detailed in Sec. 6.6. We present related work in Sec. 6.7 before concluding.

6.2 Motivation and Overview for Machine Learning Applied to Program Equivalence

Rewrite rules as axioms of equivalence In this work we represent programs with symbolic expressions made of variables (e.g., a, b, c), operators (e.g., +, $*$) and neutral/absorbing elements (e.g., 1). We consider a rich linear algebra expression language, supporting three variable types (scalars, vectors, and matrices) and 5 different variables per type; 16 operators including operators

mixing different variable types such as vector-matrix product. We represent these programs as dataflow graphs [37] with a single root node, that is to compute a single value.

Figure 6.1 diagrams 4 equivalent programs which we shall discuss with reference to rewrite rules. P1 is equivalent to P2 if we consider the axiom $A1 : 1_{\mathbb{N}} * x = x,\ \forall x \in \mathbb{N}$. This axiom is also a clear rewrite rule: the LHS expression $1_{\mathbb{N}} * x$ (with $x \in \mathbb{N}$) can be matched and replaced by the RHS expression x anywhere in the program without altering its semantics. An axiom, or equivalently here a graph rewrite rule, may be applied repeatedly to different subtrees. When applying $A1$ on a specific location, the node b of $P1$, we obtain an equivalent and yet syntactically different program, we note $P1 \equiv A1(b, P1)$. These equivalences can be composed, incrementally, to form a complex transformation: we have $P1 \equiv A1(c, A1(b, P1))$. The result of these semantics-preserving transformations can be computed in sequence: first implement $A1(b, P1)$ to obtain a new program P', then $A1(c, P')$ to obtain P''. To prove $P1 \equiv P2$, we simply check P'' is structurally identical to $P2$, a linear time process.

To assess the validity of a transformation sequence S where $P2 = S(P1)$, one simply needs to check for S, in sequence, that each axiom is applicable at that program point, apply it to obtain a new temporary program, and repeat the process for each axiom in the complete sequence. If the sequence is verified to be valid, and $S(P1)$ is structurally equivalent to $P2$, then we have proved $P1 \equiv P2$, and S forms the complete proof of equivalence between the two programs. Using $A2 : x * (y + z) = x * y + x * z,\ \forall x, y, z \in \mathbb{N}$ and $A3 : x + y = y + x,\ \forall x, y \in \mathbb{N}$, we have $P1 \equiv P4 \equiv A3(+, A2(*, A1(c, A1(b, P1))))$, a verifiable proof of equivalence under our axioms between the programs $a(1b + 1c)$ and $ac + ab$, which involved structural changes including node deletion, creation and edge modification. Note the bidirectional nature of the process: one can rewrite from $a(1b + 1c)$ to $ac + ab$, or the converse using the same (but reverted) sequence. Note also the non-unicity of a sequence: by possibly many ways a program can be rewritten into another one, for example the sequence $P4 \equiv A3(+, A1(c, A1(b, A2(*, P1))))$ also correctly rewrites $P1$ into $P4$. Conversely, a sequence may not exist: for example no sequence of the 3 above axioms allow to rewrite $a + b$ into $a * b$. We call these non-equivalent in our system, that is precisely if there is no

sequence of axioms that can be applied to rewrite one program into the other. Our approach aims to compute some S for a pair of programs $P1, P2$, so that S is verified correct when $P1 \equiv P2$. Consequently, if $P1 \not\equiv P2$, no sequence S produced can be verified correct: true negatives are trivially detected.

Pathfinding program equivalence proofs Intuitively, we can view the solution space as a graph, where every possible syntactically different program in the language is represented by its own vertex v_i. And $\exists\, e^{(A_k,x)} : v_i \to v_j$ iff $\exists A_k$ an axiom and x a node in v_i such that $v_j = A_k(x, v_i)$. Any two programs connected by a path in this graph are therefore semantically equivalent. Building S for $P1 \equiv S(P2)$ amounts to exposing one path between $P1$ and $P2$ in this graph when it exists, the path forming the proof of equivalence. We build a deep learning graph-to-sequence system to learn a stochastic approximation of an iterative algorithm to construct such feasible path when possible, trained only by randomly sampling pairs of programs and one carefully labeled path between them. This avoids the need to craft smart exploration heuristics to make this path-finding problem practical.

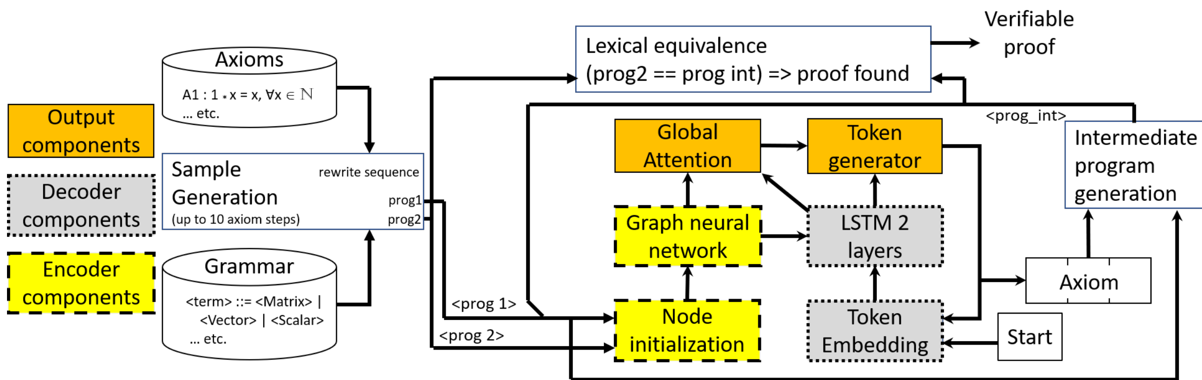


Figure 6.2: `pe-graph2axiom` System Overview

Graph-to-sequence network for pathfinding Instead of crafting path exploration heuristics, we let a neural network learn heuristics automatically; and specifically we implemented a graph neural network to solve this problem [193, 241]. We contributed the graph-to-sequence model to OpenNMT-py based on the concepts we cover in Section 2.1.5. We rely on the network to

suggest a transformation path by inference, and then verify its validity in linear time. To implement our approach, we enumerate randomly valid sentences in a language, and a set of axioms of equivalence expressible as semantics-preserving rewrite rules from one to the other. The system in Fig. 6.2 takes as input two programs represented as symbolic trees, and produces a sequence of axioms along with their position of application (or node) that can be used to rewrite sequentially one input program into the other input program. To train the system, we generate pairs of equivalent programs by iterating the axioms with random probability on one program, thereby generating both a path to equivalence and the target program. Random programs are generated so as to respect the grammar defined. The training set is then appropriately selected from these random samples, as detailed in Sec. 6.6. *Node initialization* initializes the graph neural network, converting the input programs text (e.g., $(a + (b + c))$ into nodes and edges in the *Graph Neural Network* [193, 241]. The details of the network are covered in Sec. 6.5. In a nutshell, the key principle is to combine a memory-based neural network approach, e.g., using Long-Short Term Memory (LSTM) [97] neurons and a graph neural network design (which uses Gated Recurrent Units (GRUs) internally) [29] that matches our program graph representation. *Token embedding* is a neural network layer in which tokens are assigned a learnable multidimensional embedding vector [159]. Each layer in *LSTM 2 layers* has 256 neurons, which support sequence generation. *Token generator* is the final output portion of the network. It learns to output the tokens based on the current LSTM hidden states and the *Global Attention* from the graph neural network. As each token is output, it feeds back into the LSTM layer through the embedding layer to affect its next state. We use a sequence generation principle, using a global attention mechanism [149] to allow observation of program graph node information while generating the axiom and location on which it is applied. As developed below, we specifically study the robustness of our approach to generate proofs of increasingly complex length, contrasting models to output the entire path at once with `pe-graph2axiom` which incrementally builds the sequence one step at a time, as shown in Sec. 6.6.

6.3 Framework for Program Equivalence

We now present the formalism we use in this work to represent symbolic expressions and their equivalences. We carefully co-designed this problem representation and the (graph) neural network approach to make the best use of machine learning via deep networks, as discussed in Sec. 6.5.

6.3.1 Input Representation

A key design aspect is to match the capability of the neural network to model the input as a walkable graph with the actual input program representation to be handled. We therefore model "programs" in a dataflow-like representation (i.e., a directed graph), using a single root/output node. Symbolic expressions computing a single result typically fit this representation. The following definitions are applicable to programs represented as dataflow graphs, albeit we specialize them to symbolic expressions.

Definition 6.3.1 (Expression graph node)**.** A node $n \in N$ in the expression graph models n-ary operations and input operands. A node produces a value which can be consumed by any of its immediate successors in the graph. When a node has no predecessor, it models an input value. The output value for the computation is produced by the unique root node n_{root} of the graph, the only node without successor.

Definition 6.3.2 (Expression graph directed edge)**.** A directed edge $e_{n_1,n_2} : n_1 \rightarrow n_2$ with $n_1, n_2 \in N$ in the expression graph connects the producer of a value (n_1) to a node consuming this value in the computation.

Definition 6.3.3 (Expression graph)**.** A expression graph G is a directed dataflow graph modeling the computation, made of nodes $n_i \in N$ and edges $e_{n_i,n_j} \in E$ as defined in Def. 6.3.1 and Def. 6.3.2. That is, $G = \langle n_{root}, N, E \rangle$. There is no dandling edge nor unconnected node in G.

Language of linear algebra expressions We developed a complex-enough language to evaluate carefully our work, that captures rich linear algebra expressions. Specifically, we support 3 types of data/variables in the expression: scalars, vectors and matrices. We use the standard notation

$a, \vec{a}, A$ for scalars, vectors and matrices. We evaluate using different variable names for each of the 3 types above, along with their identity and absorbing elements.

We also model a rich set of operators, mixing different unary and binary operations for each type. Specifically, we support $*_s, +_s, -_s, /_s$ between scalar operands, and $+_v, -_v, *_v$ between vectors and $+_m, -_m, *_m$ for matrices. For $-, /$ we also support their unary version for all types, e.g. ${}^{-1_s}$ for unary scalar inversion and $-_{um}$ for unary matrix negation. For example a^{-1_s} computes to $1/a$. We also support multi-type operations, such as vector and matrix scaling by a scalar $*_{sv}, *_{sm}$. We support two specific unary matrix operations, transpose ^{t_m} and matrix inversion as ${}^{-1_m}$. Note every operator has a unique name in our language, driven by the type of its operand. This will facilitate the learning of the expression embedding, avoiding the need to learn type propagation.

Examples Expressions of the form $A(BC^tD)E^{-1}$, $\vec{a} + b\vec{c}^{-1} - 0\vec{e}$, $(a + b) + (c(d/e))$, $(aA + bB)C^t$ etc. can be parsed trivially to our representation, one simply needs to be able to provide a unique name for each operand and operator type (possibly via some analysis, or simple language design principles), that is avoiding to overload the semantics of operators and operands. Note the semantics is never explicitly provided to our DNN approach, it is learned by examples. There will be no example of the form e.g. $a + A$, an invalid expression in our language.

We believe a sensible approach is to develop a clean, regular grammar for the language to be handled, as implicitly these are concepts the DNN will need to learn. We did so, using a classical LL(1) grammar description of our linear algebra language. This is not a requirement of our approach, as one can arrive to the desired input expression graph by any means necessary, but we believe making the reasoning on the language structure "easy" is an important design aspect.

6.3.2 Axioms of Equivalence

A central aspect of our approach is to view the problem of expression equivalence as finding a sequence of locally-correct rewrite rules that each preserve the semantics, *thereby making incremental reasoning possible*. We explicitly do not consider non-semantics-preserving axioms. A rich structure of alternate but equivalent ways to rewrite one expression to another makes the problem

easier to sample and more amenable to machine learning. Semantics-preserving axioms enable incremental per-axiom reasoning, and enforce semantics preservation without overly complicated semantics analysis; while still manipulating a very rich space of transformations. To illustrate this we specifically design axioms that perform complex graph modifications, such as node deletion or creation, subtree manipulation, multi-node graph changes, etc.

A graph pattern can be viewed as a pattern-matching rule on graphs and its precise applicability criteria. It can also be viewed as a sentential form of the language grammar, e.g. `ScalarVal PlusOp ScalarVal` is a pattern, if the grammar is well formed.

Definition 6.3.4 (Graph pattern)**.** A graph pattern P is an unambiguous structural description of a (sub-)graph G_P, which can be deterministically matched in any expression graph G. We have $P = \langle G_P, M_n, M_e \rangle$ where for each node $n_i \in N^{G_P}$, $\{n_{match}\} = M_n(n_i)$ returns the set of node values n_{match} accepted to match n_i on a graph G. For $n_i, n_j \in N^{G_P}$, $e_i = M_e(n_i, n_j)$ returns the set of edges between $M(n_i)$ and $M(n_j)$ to be matched in G. A pattern G_P is matched in G if (a) $\forall n_i \in G_p,\ \exists\ n_m = M(n_i) \in N^G$; (b) $\forall e_i \in E^{G_P}, \exists\ e_{M_n(n_i),M_n(n_j)} = M_e(n_i, n_j) \in E^G$; and (c) $\nexists e_{M_n(n_i),M_n(n_j)} \in E^G \neq M_e(n_i, n_j)$.

Note when a graph pattern models a rewrite, M_n and M_e are adjusted accordingly to output the rewrite of a node $n \in N^G$ into its desired value, instead of the set of acceptable nodes from $n \in N^{G_P}$.

Definition 6.3.5 (Axiom of equivalence)**.** An axiom A is a semantics-preserving rewrite rule $G' = A(n, G)$ that can arbitrarily modify a expression graph G, and produces another expression graph G' respecting Def. 6.3.3 with identical semantics to G. We note $A : \langle P_{match}, P_{replace} \rangle$ an axiom, where $P_{match}, P_{replace}$ are graph patterns as per Def. 6.3.4. The application of axiom A to node n in G is written $A(n, G)$.

We can compose axioms to form a complex rewrite sequence.

Definition 6.3.6 (Semantics-preserving axiom composition)**.** Given a sequence of m axiom applications $S :\ A_1(n_1, A_2(n_2, ..., A_m(n_m, G)))$. It is a semantics-preserving composition if for each

$G_j = A_i(n_i, G_i) \in S$, $P^{A_i}_{match}$ succeeds on the subgraph with root n_i in G_i, and G_j is obtained by applying $P^{A_i}_{replace}$ to n_i.

Theorem 6.3.1 (Expression graph equivalence)**.** Given a expression G. If $G' = S(G)$ such that S is a semantics-preserving sequence as per Def. 6.3.6, then $G \equiv G'$, they are equivalent under the axiom system used in S.

This is a direct consequence of using only semantics-preserving axioms, each rewrite cannot individually alter the semantics, so such incremental composition does not. It leads to the formal problem we are addressing:

Corollary 6.3.1.1 (Expression graphs equivalence matching)**.** Given two expressions G, G'. If there exist a semantics-preserving sequence S such that $G' = S(G)$, then $G \equiv G'$.

Note here $=$ means complete structural equivalence between the two graphs: they are identical in structure *and* label/node values. Determining $G = G'$ amounts to visiting both graphs simultaneously e.g. in depth-first search from the root to ensure structural equivalence, and also verifying the same node labels appear in both at the same time. This is trivally implemented in linear time in the graph size.

Language of linear algebra expressions We have implemenented a total of 102 different axioms for our language, made of the multi-type versions of the 13 core restructuring axioms described later in Table 6.1. They all follow established linear algebra properties. Note different data types have different axioms following typical linear algebra rules, e.g., matrix-multiplication does not commute, but scalar and vector multiplications do. Examples of axioms include $x(yz) \rightarrow (xy)z$, $X - X \rightarrow O$, $-(\vec{x} - \vec{y}) \rightarrow \vec{y} - \vec{x}$, or $X^{t^t} \rightarrow X$, an exhaustive list is displayed in the Supplementary Material.

In our experiments, we presume matrix and vector dimensions are appropriate for the given operation. Such dimension compatibility checks are simple to implement by e.g. introducing additional nodes in the prgram representation, but are not considered in our test language.

Examples We illustrate axiom-based rewrites using axioms presented in later Table 6.1. Note axiom names follow the structural changes applied. For example, we have $a{+}b \equiv b{+}a:\ \{a{+}b\} = Commute(\{+\},\{b{+}a\})$. $a{+}b{+}c \equiv b{+}c{+}a:\ \{a{+}b{+}c\} = Commute(\{+_1\}, Commute(\{+_2\}, \{b+c+a\}))$. Note we refer to different nodes with the same symbol (e.g., $+_2$) subscripting them by their order in a DFS traversal of the expression graph, starting from the unique root. We have $0 \equiv a - a:\ \{0\} = Cancel(\{-\},\{a-a\})$. These can be combined in complex paths, e.g., $b{+}c \equiv c{+}b{+}(a{-}a):\ \{b{+}c\} = Commute(\{+\}, Noop(\{+\}, Cancel(\{-\},\{c{+}b{+}(a{-}a)\})))$. Such axioms are developed for scalars, matrices and vectors, and include complex rewrites such as distributivity rules and transpositions. A total of 102 axioms are used in our system.

6.3.3 Space of Equivalences

We now define the search space being explored in this work, i.e., the exact space of solutions on which the DNN system formally operates, and that we sample for training.

Definition 6.3.7 (Graph of the space of equivalences)**.** Given a language $\mathcal{L}$. The directed graph of equivalences between expressions is $G^{equiv} = \langle N^{equiv}, E^{equiv} \rangle$ such that $\forall l \in \mathcal{L}, n_l \in N^{equiv}$, and $e^{A_i,x}_{n_i,n_j} : n_i \to n_j \in E^{equiv}$ iff $n_j \equiv A_i(x, n_i), \forall A_i$ in the axiom system and x a position in n_i where A_i is applicable.

In other words, the graph has one node per possible expression in the language $\mathcal{L}$, and a single axiom application leads to connecting two nodes. We immediately note that G^{equiv} is a (possibly infinite) multigraph, and contains circuits.

Theorem 6.3.2 (Expression equivalence with pathfinding)**.** Given two expressions $n_i, n_j \in N^{equiv}$. If there is any path from n_i to n_j in G^{equiv}, then $n_i \equiv n_j$.

The proof is a direct consequence of Def. 6.3.7. In this work, we randomly sample this exact graph to learn how to build paths between arbitrary expressions. As it is a multigraph, there will be possibly many different sequences modeled to prove the equivalence between two expressions. It is sufficient to expose one to prove equivalence.

Corollary 6.3.2.1 (Semantics-preserving rewrite sequence)**.** Any directed path in G^{equiv} is a semantics-preserving rewrite sequence between the expressions, described by the sequence of axioms and expression position labeling the edges in this path. This sequence forms the proof of equivalence.

We believe that ensuring there are possibly (usually) many ways to compute a proof of equivalence in our specific framework is key to enable the DNN approach to learn automatically the pathfinding algorithm for building such proofs. Other more compact representations of this space of equivalences are clearly possible, including by folding nodes in the equivalence graph for structurally-similar expressions and folding equivalent paths between nodes. When building e.g. a deterministic algorithm for pathfinding, such space size reduction would bring complexity benefits [110, 25]. We believe that for the efficient deployment of graph-to-sequence systems, exposing significant redundancy in the space facilitates the learning process. We also alleviate the need to reason on the properties of this space to find an efficient traversal heuristic.

6.4 Samples Generation

The careful creation of our training dataset is key: as we let the DNN learn *by example only* what the axioms are and when they are applicable in the structure of a program, we must carefully sample the space of equivalences to ensure appropriate distributions of the examples. We produce a final dataset of tuples $(P1, P2, S)$, a pair of input programs and a possible rewrite rule sequence that proves the pair equivalent. Duplicates are removed such that all samples have a unique $P1$. From this dataset, we create 1,000,000 training samples, 10,000 validation samples, and 10,000 test samples. We outline below its generation principles; extensive details and the algorithms used are presented in section 6.9.1.

Random sample generation Deep learning typically requires large training sets to be effectively deployed, hence we developed a process to automate sample generation. We specifically use randomized program generation algorithms that are inspired by a given language grammar. By

randomly choosing between production rulse, one can build random parse trees by simply iterating the grammar. The leaves obtained will form a sentence accepted by the language, i.e., a program [34]. We limit to programs of 50 nodes in the program tree (or AST), with a maximal tree depth of 7. We assert that our random production rule procedure has a non-zero probability of producing any program allowed by the grammar for our datasets.

We produce equivalent program samples by pseudo-randomly applying axioms on one randomly generated program to produce a rewrite sequence and the associated equivalent program. Given a randomly selected node in the program graph, our process checks which axiom(s) can be applied. E.g., the $+_m$ operator may have the Commute axiom category applied, or it may have the Transpose axiom category applied, which affects the operator's children.

Final experimental dataset: AxiomStep10 To train our network to produce one axiom step at a time, as described in Sec. 6.2, AxiomStep10 has a single axiom in each output sequence S. For a complete proof $S : A_1(A_2(...)$ in a $(P1, P2, S)$ we generated made of N axioms, we then create N training examples for the network: $(P1, P2, A_N)$ the first intermediate step by applying the first axiom, then $(A_N(P1), P2, A_{N-1})$, etc. We limit proof length to 10 axioms in our experiments (hence AxiomStep10). Test samples only have the original and target program and the network proposes axioms which create intermediate programs towards the proof, fed back to the system.

Datasets to study generalizability and robustness In order to study our model's ability to generalize, we have created alternate datasets on which to train and test models which are summarized in table 6.2. *WholeProof10* will help us contrast learning approaches. This dataset has the complete proof sequence S made of $N \geq 1$ axioms as reference output for a program pair, while for AxiomStep10, $N = 1$. Models trained on WholeProofX must maintain internal state representing the graph transformations that the axioms create. They are not "iterative": a single inference is expected to produce the complete proof; in contrast to AxiomStep10 for which a single axiom of the sequence is produced at each inference step. Training long output sequences can benefit from com-

Table 6.1: Distribution for the 14 axiom categories in AxiomStep10 test set. Considering scalars (a, b, ...), vectors ($\vec{v}$,$\vec{w}$, ...) and matrices (A, B, ...) types combinations, 147 distinct axioms are represented.

Axiom Category	Example axiom(s)	Samples with
Cancel	(A-A)→O,(b/b)→1	13.8%
NeutralOp	($\vec{v}$ - $\vec{o}$) $\rightarrow \vec{v}$	40.0%
DoubleOp	$A^{t^t} \rightarrow A$, 1/1/x→x	7.3%
AbsorbOp	(A*O)→O, (b*0)→0	30.3%
Commute	(a + b) → (b + a)	48.6%
DistributeLeft	(a + b)c → ac + bc	36.3%
DistributeRight	a(b + c) → ab + ac	27.8%
FactorLeft	ab + ac → a(b+c)	6.1%
FactorRight	ac + bc → (a+b)c	9.0%
AssociativeLeft	a(bc) → (ab)c	46.3%
AssociativeRight	(ab)c → a(bc)	43.1%
FlipLeft	-($\vec{v}$ - $\vec{w}$) $\rightarrow \vec{w} - \vec{v}$	8.4%
FlipRight	a/(b/c) → a(c/b)	26.1%
Transpose	$(AB)^t \rightarrow B^t A^t$,	11.1%

plex training approaches such as Professor forcing [129], but we will show that our AxiomStep10 model generalizes well with our sequence model training approach.

Table 6.2: Datasets used for studies in experiments.

Dataset	AST depth	AST #nodes	Proof length	Iterative
AxiomStep10	2-7	2-50	1-10	Yes
AxiomStep5	2-6	2-25	1-5	Yes
WholeProof10	2-7	2-50	1-10	No
WholeProof5	2-6	2-25	1-5	No

Complexity of equivalence space Figure 6.3 provides a view of the complexity of the equivalence problem we tackle. The distribution of the dataset per proof length is displayed in the right chart; the left chart shows by size of bubble the number of test samples with a given number of *semantics-preserving* axioms that may be implemented as the first step of the proof and the proof length needed.

There is a large number of proofs possible in our system, as detailed in Appendix 18. For example, for proofs of length 5, about 340,000 proofs made only of legal applications of axioms can be performed on the average sample in our dataset. Since many programs have multiple possible

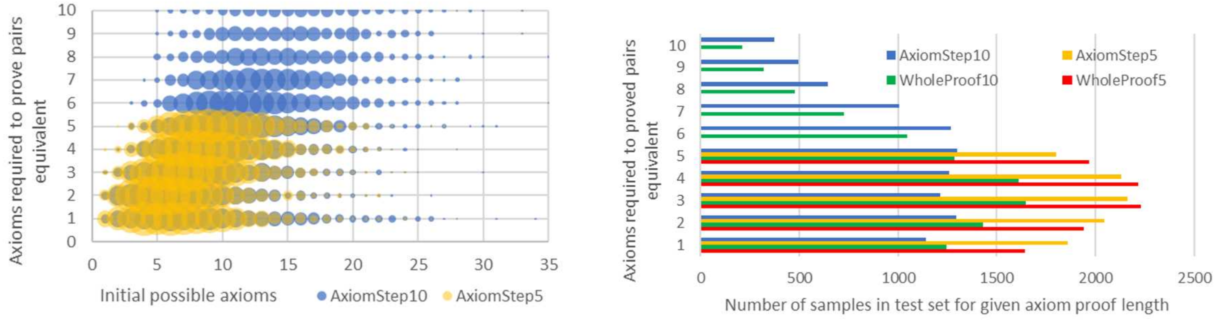


Figure 6.3: Distribution of axiom possibilities and proof complexity for test datasets.

proofs, about 10,000 different programs can be produced, only one of which is the target to prove, i.e., randomly drawing a valid 5 axiom proof on a program known to be 5 axiom steps from the target has roughly a 1 in 10,000 chance of being a correct proof of equivalence between the two programs.

6.5 Deep Neural Networks for Program Equivalence

Fig. 6.2 overviews the entire system architecture including the `pe-graph2axiom` network, sample generation, and the rewrite checker. Key design decisions are presented below.

Graph neural network The sample generation discussed in section 6.4 provides input to the Node Initialization module in Fig. 6.2 to create the initial state of our graph neural network [29]. For each node in the program graph, a node will be initialized in our graph neural network with a value that encodes the AST level and language token of the program node. To interconnect the edges we support 9 edge types and their reverse edges which allows information to move in any direction necessary: 1) left child of binary op, 2) right child of binary op, 3) child of unary op, 4) root node to program 1, 5) root node to program 2, 6-9) there are 4 edge types for the node grandchildren (LL, LR, RL, RR). The node states and edge adjacency matrix represent the initial graph neural network state.

After initialization, the graph neural network iterates 10 times in order to convert the initial node state into the embeddings needed for rewrite rule generation. Given an initial hidden state for node n of $x_n(0)$, $x_n(t+1)$ is computed with a learnable function f which com-

bines the current hidden state $x_n(0)$, the edge types $l_{in[n]}$ of edges entering node n, the edge types $l_{out[n]}$ of edges exiting node n, and the hidden states $x_{ne[n]}$ of the neighbors of node n: $x_n(t+1) = f(x_n(t), l_{in[n]}, x_{ne[n]}(t), l_{out[n]})$.

Each edge type has a different weight matrix for learning, allowing aggregation of information into a given node related to its function in the full graph of the program. The root node's initial state along with the edge types connecting it to the program graph trees allow it to aggregate and transfer specific information regarding rewrite rules as demonstrated by our experimental results. This is a novel feature of our network not used in prior work with GNNs on program analysis [8, 241].

Graph neural network output to decoder After stepping the GGNN, the final node values are used by the decoder in two ways to create rewrite rules. First, the final root node value $x_{root}(10)$ is fed through a learnable bridge function to initialize the LSTMs of the decoder. In this way, the aggregated information of the 2 programs seeds the generation of rewrite rules. The LSTMs update as each output token y_j is generated with a learnable function based on the current decoder hidden state h_j^d at decoder step j and the previous output token y_{j-1} [48]. Second, all nodes in the graph can be used by an attention layer [21]. The attention layer creates a context vector c_j which can be used by a learnable function g when computing the probability for generating the jth output token $P(y_j)$: $P(y_j \mid y_{j-1}, y_{j-2}, ..., y_0, c_j) = g(h_j^d, y_{j-1}, c_j)$. Because `pe-graph2axiom` has a robust output verification, we make use of beam search to track up to 10 likely candidates for proofs of equivalence.

By using the root node only for seeding the initial hidden state h_0^d of the decoder, the weights associated with its connections to the program graphs for $P1$ and $P2$ learn to represent the information necessary for the rewrite rule sequence. In parallel, after the graph neural network iterations complete, the final embedding for all the nodes in the graphs for $P1$ and $P2$ are only used by the attention network, so their final embedding represents information useful during rewrite rule generation.

Intermediate program generation `pe-graph2axiom` applies the axiom and program node chosen by the neural network token generator to the input program to create an intermediate program P' on the path from $P1$ to $P2$. If this program is equal to $P2$, then our axiom path is complete, otherwise the new pair $P', P2$ is inferred to determine the next axiom step.

Incremental versus non-incremental sequence production The models we train on the AxiomStep5, WholeProof10, and WholeProof5 datasets have the same neural network hyperparemeters as the AxiomStep10 data model. However, the models for WholeProof10 and WholeProof5 are trained to output the entire sequence of axioms needed to prove the 2 programs identical, hence these models do not make use of the intermediate program generation and instead have a component which checks whether the full sequence of axioms legally transforms $P1$ into $P2$. We encode the path to the AST node an which to apply an axiom using 'left' and 'right' tokens which specify the path from the current program root node. This encoding is sufficient for the iterative model and necessary to allow the non-iterative model to identify nodes which may not have been in the initial AST for $P1$. The non-iterative models must learn a representation in the LSTM network to allow them to track AST transformations as they are generated.

6.6 Experimental Results

We now present extensive experimental results, and compare the quality of several neural network approaches to address the problem of program equivalence. We have proceeded incrementally for fine-tuning the final system design, and report on several of these design points below.

We focus our experiments below on 4 key questions: 1) Is performance related to input program size? 2) Is performance related to proof length? 3) Is the incremental, per-axiom approach more generalizable than producing the full sequence in a single inference step? And 4) Is performance consistent across a range of datasets, including human-written examples?

Implementation setup We developed the neural network system presented in the OpenNMT-py system [117], adding on a new encoder based on a prior implementation of gated graph neu-

ral networks [137]. Our integration work contributed the yellow encoder components shown in Figurer̃effig:Full to OpenNMT-py. The input to the GGNN encoder involved integrating 'Node initialization' into the existing OpenNMT-py framework including optionally using input token embeddings. The output of the GGNN encoder involved integrating into the bridge layer connecting it to the decoder and providing the proper attention information necessary for the global attention logic in OpenNMT-py. For our training and evaluation experiments, we use systems with Intel Xeon 3.6GHz CPUs and 6GB GeForce GTX 1060 GPUs. During training, we save a model snapshot every 50,000 iterations and score the accuracy the model achieved on the validation dataset. Graphs showing that validation accuracy plateaus at 200,000 to 300,000 iterations are provided in section 6.9.4. We run each model twice and evaluate the test set using the saved model which achieved the highest validation score.

Evaluation procedure and neural network alternatives The benefits of key components of our neural network model are studied in table 6.3. The bidirectional RNN model is similar to state-of-the-art sequence-to-sequence models used for program repair [48]. The results for the graph-to-sequence model without attention show the benefit of providing the node information during the axiom generation process.

Table 6.3: `pe-graph2axiom` mini ablation study.

Model description	Beam width 1	2	5	10
Bidirectional RNN seq-to-seq with attention	48	62	71	75
Graph-to-sequence w/o attention	73	81	87	90
`pe-graph2axiom` model	76	84	90	**93**

Our final design was influenced by explorations we performed on varied models, datasets, and hyperparameters such as LSTM layers and graph neural network parameters. In relation to the model's ability to learn a representation of the proof sequence, we note that our GGNN initialization using the root node connection to the decoder outperforms the embedding learned by a bidirectional RNN model. Also, we found that averaging the embedding of all graph nodes had

about 10% lower accuracy than using the more specific root node information. Numerous additional results are reported in Suppl. material 6.9.4.

Generalizing across different datasets We specifically look at the generalization potential for our models by studying their success rate as a function of the input program complexity, represented as the AST depth, in Table 6.4, and as a function of the output complexity, represented by the proof length in Table 6.5, all using a beam size of 10. We designed our datasets in Sec. 6.4 to study how well `pe-graph2axiom` generalizes and to assess we are not overfitting on training data. Extensive in-depth additional experimental results are presented in Suppl. Material 6.9.4, we summarize key results only below.

Table 6.4: Performance vs. AST size: counts and percentage pass rates.

	Testset Sample Count		Model trained on AxiomStep5		Model trained on AxiomStep10	
AST depth	AS5	AS10	AS5	AS10	AS5	AS10
2-6	10000	6865	99	93	99	94
7	0	3135	n/a	86	n/a	92
All	10000	10000	99	90	99	**93**

Table 6.4 illustrates the ability of a model trained on AxiomStep5 (i.e., limited to proofs of length 5) to perform well when evaluated on the more complex AxiomStep10, which includes proofs of unseen length of up to 10. The robustness to the input program complexity is illustrated with the 86% pass rate on AST depth 7, for the model trained on AxiomStep5 which never saw programs of depth 7 during training.

Table 6.5: Performance vs. proof length: percentage pass rates.

Axiom Count in	Model trained on WholeProof5 (WP5)				Model trained on WholeProof10 (WP10)				Model trained on AxiomStep5 (AS5)				Model trained on AxiomStep10 (AS10)			
Proof	WP5	WP10	AS5	AS10	WP5	WP10	AS5	AS10	WP5	WP10	AS5	AS10	WP5	WP10	AS5	AS10
1-5	95	89	44	44	94	93	44	44	99	97	99	98	99	98	99	**98**
6		14		4		72		5		81		88		90		**93**
7		0		1		63		2		67		81		83		**87**
8		0		0		54		1		54		75		73		**82**
9		0		0		47		0		35		64		63		**74**
10		0		0		34		0		24		57		46		**66**
All	95	66	44	27	94	84	44	27	99	87	99	90	99	93	99	**93**

Table 6.5 compares the results of our 4 models, each trained on one of our 4 datasets, and evaluated with the test set of all 4 datasets. The models all have identical hypermeter settings. We observe the inability of models trained to output the whole proof to generalize to proofs of higher length (WP5 model on AS10/WP10), with near zero success rate. However, per-axiom models (AS5 and AS10) show potential for generalization to proof length: AS5 model performs well when evaluated on AS10, showing the ability to produce proofs of length/complexity unseen in training. Overall, the success rate degrades gracefully with proof length, bottoming at 66% for AS10 for proofs of length 10.

Table 6.6: Results for various language complexities studied, on non-incremental models (WholeProof).

ID	Description	# Operators	# Axioms	# Operands	Program length	Rewrite rules length	Graph2seq (G2S) or seq2seq (S2S)	Training set size	Percent matching with beam width 1	Percent matching with beam width 10
1	Rewrite sequence is only single Commute, uses sequence-to-sequence model	2	1	10	3-19	1-5	S2S	80,000	90.0%	96.2%
2	Rewrite sequence is exactly 2 Commutes, uses sequence-to-sequence model	2	1	10	5-24	3-10	S2S	80,000	80.3%	96.5%
3	Rewrite sequence exactly 2 Commutes	2	1	10	5-24	3-10	G2S	80,000	98.9%	99.8%
4	Rewrite sequence exactly 3 Commutes	2	1	10	7-45	5-15	G2S	80,000	91.4%	99.0%
5	Rewrite sequence 1 to 3 Commutes	2	1	10	3-45	1-15	G2S	180,000	97.1%	99.2%
7	Commute, Noop, Cancel, Distribute Left, Distribute Right	4	5	12	3-45	1-15	G2S	180,000	93.1%	97.4%
8	Scalars, Vectors, and Matrixes	16	5	20	3-30	1-25	G2S	250,000	88.3%	95.6%
9	13 Axioms	16	13	20	3-30	1-25	G2S	400,000	85.5%	95.5%
10	Rewrite sequence or Not_equal	16	13	20	3-30	1-25	G2S	500,000	79.8%	93.8%
11	Test sequence-to-sequence	16	13	20	3-30	1-25	S2S	400,000	59.8%	81.1%
12	Add loop axioms	18	15	20	3-30	1-25	G2S	400,000	83.8%	94.7%

6.6.1 WholeProof Models: Language Complexity and Performance

Table 6.6 shows the result of 12 different experiments and designs for versions of WholeProof models. In particular, we incrementally increase the problem complexity from rows 1 to 10, increasing the number of Operators that can be used in any input program, of Axioms used in the rewrite sequence, of Operands in any input program, of the maximal number of nodes in an input program graph (the Program length, directly influencing the size of the graph network), and the Rewrite rule length, which contains the description of paths from the root node to reach

the position of application of an axiom, this is directly related to the maximal graph height, itself determined by the maximal program size. Details on each row are provided in Supplementary Material.

We specifically compare against a sequence-to-sequence (S2S) approach, to quantify the gains brought by employing graph-to-sequence (G2S). When the space is small enough, S2S still performs well, especially using aggressive beam search. We recall that by design of our system testing the correctness of one sequence is trivial and deterministic, so one can easily use large beam sizes without any correctness impact nor major performance penalty during inference. For example, inference of beam 1 is about 15ms for our most complex networks, but beam 10 only takes 16ms. Checking correctness is $<<$ 1ms.

Contrasting rows 2 and 3 displays the merits of the G2S approach for our problem: on this simple problem, in fact G2S gets near-perfect accuracy already. Progressively increasing the complexity of the search space, till row 9 and 10, displays a slow but steady decrease in quality, while still maintaining excellent scores near or above 95% with beam 10. To reassess the limits of a sequence-to-sequence approach, row 9 and 11 can be constrasted: they operate on the same search space, but S2S peaks at 81% accuracy, while G2S reaches 95%.

Row 10 displays the result when learning using also samples of non-equivalent programs, using the "empty path" symbol Not_equal. We evaluated this system to measure the impact of training on only equivalent programs vs. also sampling pairs of unconnected nodes in the equivalences graph. We recall that by design, if no rewrite rule produced is verified as correct, our system outputs the programs are not equivalent. In other words, whichever the sequence(s) produced by the network, if the two input programs are non-equivalent, the system will *always* output they are not equivalent: no equivalence sequence produced can be verified as correct. So training on only equivalent programs is clearly sensible for such system; furthermore as shown in row 10 vs. 9, even increasing the training set size, training using non-equivalent programs seem to lower the performance slightly.

Human written test expressions from Khan academy exercises Unfortunately there is a dearth of existing large reference datasets for equivalence of linear algebra expressions, which justified our careful dataset creation approach in Sec. 6.4 and their upcoming public release. However numerous math exercises involve exactly this problem, and can provide small but human-written datasets. We solve all of the matrix expression equivalence programs from 2 relevant Khan academy modules designed to test student's knowledge of matrix algebra [114]. Our AxiomStep10 model is able to correctly prove all 15 equivalent pairs from the modules with beam width 1 and wider. With a beam width of 10, the WholeProof10 model proved 12. An example problem solvable by AxiomStep10 but not WholeProof10 is: $c(1A + B) = cB + cA$ which can be proven by applying the rewrite rules NeutralOp, DistributeRight, and Commute to the proper nodes. The WholeProof10 model mostly fails because it was not trained on how to apply repeated transformations at the same point in the AST. This suggests AxiomStep10 has generalized well to these hand-written problems.

6.7 Related Work

Theorem provers The problem of equivalence as we formulated may be solved by other (smart) brute-force approaches, where a problem is solved by pathfinding. This ranges from theoreom proving systems like Coq [31] which supports the formal framework for equivalence we describe in this chapter, to (Approximate Probabilistic) Model Checking [56, 40, 93], where a program equivalence system can also be built, e.g. [206, 55, 224, 168]. Our contribution is not in the formal definition of program equivalence we presented, semantics-preserving rewrite systems have been studied, e.g. [223, 145, 186]. But understanding why this particular formalism was well suited to deep learning graph-to-sequence systems was key. The merits of stochastic search to accelerate such systems has been demonstrated, e.g. [165, 93, 78]. The novelty of our approach is to develop carefully crafted graph-to-sequence neural networks to automatically learn an efficient pathfinding heuristic for this problem. Our approach is potentially applicable in these areas too, however training scalability can become a challenge if increasing the input representation size

excessively. Theorem provers using deep learning have recently started to be investigated, Aygun *et al.* [18] developed a graph neural network system for automatic proof generation. Wu *et al.* [237] explores the ability of theorem provers using GNNs, TreeLSTMs, and BagOfWords architectures to generalize and solve proofs with lengths up to 7 axioms and found that GNNs performed the best of the architectures studied when more complex proofs were required. While our model works in a slightly different problem space, we study the ability of our models to generalize on proofs with lengths up to 10, with 14 different rewrite rules acting on 147 distinct axioms. These frameworks could also be used to prove equivalence between symbolic expressions, as theorem provers.

Static program equivalence Algorithms for static program equivalence have been developed, e.g. [222, 6, 25, 103]. These approaches typically restrict to demonstrating the equivalence of different schedules of the operations, possibly dynamically [24]. In this work we target graph-modifying rewrites (and therefore which alter the operation count). Barthou et al. [25, 6] have developed techniques to recognize algorithm templates in programs. These approaches are restricted to static/affine transformed programs. Karfa *et al.* also designed a method that works for a subset of affine programs using array data dependence graphs (ADDGs) to represent input and transforming behaviors. Operator-level equivalence checking provides the capability to normalize expressions and establish matching relations under algebraic transformations [113]. Mansky and Gunter used the TRANS language [108] to represent transformations. The correctness proof implemented in the verification framework [154] is verified by the Isabelle [175] proof assistant. Other works also include translation validation [127, 169].

Program analysis with machine learning Numerous prior works have employed (deep) machine learning for program analysis, e.g. [9, 12, 216, 128, 185, 28]. code2vec [12] teaches a method for creating a useful embedding vector that summarizes the semantic meaning of a snippet of code. Program repair approaches, e.g. [216, 48] are deployed to automatically repair bugs in a program. Output accuracies of up to 20% on the test set is reported, using sequence-to-sequence models. Wang *et al.* [227] learns to extract the rules for Tomita grammars [214] with recurrent neu-

ral networks. The learned network weights are processed to create a verifiable deterministic finite automata (DFA) representation of the learned grammar. This work demonstrates that deterministic grammars can be learned with RNNs, which we rely on.

Graph Neural Networks Graph neural networks [193, 238] use machine learning to analyze a set of nodes and edges for patterns related to a target problem. Using a graph-to-sequence network with attention has been analyzed for natural language processing [29]. Allamanis *et al.* use graph neural networks to analyze code sequences and add edge types representing LastUse, ComputedFrom, and LastWrite to improve the system's ability to reason about the code [8]. Their work achieves 84% accuracy on correcting variable misuse cases and provides insights to useful edge types. Structure2vec [241] uses a graph neural network to detect binary code similarity. Structure2vec uses a graph neural network to learn an embedding from a annotated control flow graph (ACFG) of a program. This learning process targets the embedding so that equivalent programs will have equivalent embeddings, reporting precision scores of 84% and 85% on various test datasets for correctly predicting program equivalence. It only outputs a probability of equivalence, and not a verifiable proof, which is sufficient in their context.

The G2SKGE model [135] has a similar graph network structure which uses a node embedding (which they refer to as an information fusion mechanism) in order to predict relationships between nodes. This technique of using a neural network to understand and predict node interelationships is common to our approach.

6.8 Conclusion

In this work, we presented `pe-graph2axiom`, the first graph-to-sequence neural network system to generate verifiable axiomatic proofs (via rewrite rules) for equivalence for a class of symbolic programs. Evaluated on a rich language for linear algebra expressions, this system produces correct proofs of up to 10 axioms in length in 93% of the 10,000 equivalent cases evaluated. We believe the performance of our approach comes in part from using graph neural networks for

what they aim to excel at: learning efficient heuristics to quickly find paths in a graph; and the observation that program equivalence can be cast as a path-based solution that is efficiently found by such networks.

6.9 Supplementary Materials

This section provides further details on pe-graph2axiom. We have provided below numerous additional information for completeness. Our supplementary materials are organized as follows:

- Subsection 6.9.1 of this document presents the dataset generation approach we developed.
- Subsection 6.9.2 of this document presents exhaustively the language for complex linear algebra expressions we evalaute on, including the list of all 147 axioms of equivalence we learned.
- Subsection 6.9.3 of this document presents additional details about the neural network architectures we developed.
- Subsection 6.9.4 of this document presents complementary experimental results and additional in-depth details on results presented in the main paper body.

6.9.1 Dataset generation

Generation of Examples

Machine learning benefits from large training sets, so in order to produce this data, we created algorithms that would generate programs meeting a given language grammar along with target programs which could be reached by applying a given axiom set. By creating this process, we could create as large and varied a dataset as our machine learning approach required.

Algorithm 1 provides an overview of the full program generation algorithm. For this generation process, we define a set of operations and operands on scalars, matrices, and vectors. For our process, we presume matrix and vector dimensions are appropriate for the given operation as such dimension checks are simple to implement and are not considered in our procedure. Note the token syntax here is *exactly* the one used by our system, and is *strictly* semantically equivalent to the mathematical notations used to describe these operations, e.g. $1_{\mathbb{N}}$ is `1`.

- Scalar operations: `+s -s *s /s is ns`, where `is` the unary reciprical and `ns` is the unary negation.
- Matrix operations: `+m -m *m im nm tm`, where `im` is matrix inversion, `nm` negates the matrix, and `tm` is matrix transpose.
- Vector operations: `+v -v *s nv`, where `nv` is the unary negation.
- Scalars: `a b c d e 0 1`
- Matrices: `A B C D E O I`, where `O` is the empty matrix and `I` is the identity matrix.
- Vectors: `v w x y z o`, where `o` is the empty vector.
- Summary: 16 operations, 20 terminal symbols

Initially, `GenP1` is called with `GenP1("+s -s *s /s +s -s *s /s +s -s *s /s is ns +m -m *m +m -m *m +m -m *m im nm tm +v -v *v +v -v *v +v -v *v nv", 0.94)`. In this initial call binary operations are repeated so that they are more likely to be created than unary operations, and the initial probability that a child of the created graph node will itself be an operation (as opposed to a terminal symbol) is set to 94%. Since the algorithm subtracts a 19% probability for children at each level of the graph, trees are limited to 7 levels.

Algorithm 1 starts execution by randomly selecting an operation from the set provided as input. When `GenP1` is called recursively, the operation set is limited such that the operation produces the correct type as output (scalar, matrix, or vector). Lines 3 through 15 of the algorithm show an example case where the `*s` operation is processed. This operation requires scalar operands. If the probability of children at this level is met, then `GenP1` is called recursively with only scalar operands available, otherwise a random scalar operand is chosen.

The text for algorithm 1 does not show the process for all operations. Certain operations, such as `*v`, have a variety of operand types that can be chosen. The `*v` operand is a multiplication which produces a vector. As such, Av (matrix times vector), bv (scalar times vector), or vc (vector times scalar) are all valid options and will be chosen randomly.

Algorithm 1: GenP1

```
Result: Prefix notation of computation with parenthesis
Input  : Ops, P
Output: (op L R) or (op L)
1  op = select randomly from Ops
2  // Create subtree for chosen op
3  if op == "*s" then
4      if random < P then
5          L = GenP1("+s -s *s /s +s -s *s /s is ns",P-0.19)
6      else
7          L = select random scalar operand
8      end
9      if random < P then
10         R = GenP1("+s -s *s /s +s -s *s /s is ns",P-0.19)
11     else
12         R = select random scalar operand
13     end
14     return (op L R)
15 end
16 // Other ops may have more complex options for children types.
17 // (For example, "*m" may have a matrix multiplied by a scalar or matrix)
18 ...
```

After generating a program which follows the grammar rules of our language, algorithm 2 will produce a new program along with a set of rewrite rules which transform the source program to the target program.

Algorithm 2 receives as input the source program (or subprogram) along with the `path` to the current root node of the source program. If the source program is a terminal symbol, the algorithm returns with no action taken. Otherwise, the program starts with an operation and the algorithm proceeds to process options for transforming the given operation. For our wholeproof10 and wholeproof5 datasets, algorithm 2 is only called once, simplifying the possible node order and proof complexity. for the axiomstep10 and axiomstep5 datasets, algorithm 2 is called multiple times, allowing for the possibility that after a path is chosen for one axiom any node can be accessed for the next axiom (including the same node).

As shown on line 10 of the algorithm, when the operation and children meet the conditions necessary for a rewrite rule (in this case `NeutralOp`), the rule is applied with some probability (in this case 50%). Note that before processing a node, the left and right operands are further analyzed to determine their operators and operands as well (or $\perp$ if the child is a terminal). Processing the left and right operands allows for complex axioms to be applied, such as distribution or factorization. When a rule is applied, the rewrite rule is added to the rewrite rule sequence and a new target program is generated for any remaining subtrees. When creating the rewrite rules for subtrees, the `path` variable is updated as rewrites are done. In the case of `NeutralOp`, the current node is being updated, so the path is not changed. But in the case of the Commute rule, the return would be generated with `(op GenP2(R,path."left ") GenP2(L,path."right "))` which creates rewrite rules for the prior right and left operands of the `op` and updates the path used to the new node positions. In order to analyze nearly equal programs, illegal rewrites can be optionally enabled; for example, commuting a subtraction operation or mutating one operation into another. In that case, the `GenP2` process continues to create a target program, but `transform_sequence` is set to `Not_equal`.

After these generation algorithms are run, a final data preparation process is done which prunes the data set for the learning algorithm. The pruning used on our final data set ensures that the $(P1, P2)$ program pair total to 100 tokens or fewer (where a token is an operation or terminal), that the graph is such that every node is reachable from the root with a path of length 6 or less, and that there are 10 or fewer rewrite rules applied. But within these restrictions, we assert that our random production rule procedure has a non-zero probability of producing any program allowed by the grammar. Also, the pruning ensures that there are no lexically equivalent programs in the process and removes some of the cases with fewer than 10 rewrite rules generated to bias the dataset to longer rewrite sequences. Table 6.1 details the distribution of rewrite rules created by the full process. Section 6.9.2 details all axioms when variable types and operators are considered.

We produce equivalent program samples by pseudo-randomly applying axioms on one randomly generated program to produce a rewrite sequence and the associated equivalent program.

Algorithm 2: GenP2

```
   Result: Second program and transform_sequence
   Input  : P1, path
   Output: P2
1  if terminal symbol then
2  |  return P1
3  end
4  op = find operator of P1
5  L = find left operand of P1
6  R = find right operand of P1
7  Lop,LL,LR = operator and operands of left child
8  Rop,RL,RR = operator and operands of right child
9  // Randomly apply transform if allowed
10 if random < 0.5 and ((op == "+v" and (L == "o" or R == "o")) or (op == "-v" and R ==
    "o")) then
11 |  append path."NeutralOp " to transform_sequence
12 |  // Eliminate unnecessary operator and 0 vector
13 |  if L == "o" then
14 |  |  return GenP2(R,path)
15 |  else
16 |  |  return GenP2(L,path)
17 |  end
18 end
```

Given a randomly selected node in the program graph, our process checks which axiom(s) can be applied. E.g., the $+_m$ operator can have the Commute axiom applied, or depending on subtrees it may be allowed to have the Factorleft axiom applied, as discussed in Sec. 7.4.4. Generally we choose to apply or not an operator with 50% probability, so that `pe-graph2axiom` is forced to rely on analysis of the two programs to determine whether an operator is applied instead of learning a bias due to the local node features.

Intermediate program generation

The intermediate program generation algorithm is very similar to algorithm 2. For program generation of the target program, algorithm 2 will check that a node can legally apply a given rule, apply the rule with some probability, record the action, and process the remaining program. For intermediate program generation, we begin with a $P1$ and a rewrite rule. We follow the path

provided to identify the node, check that a node can legally accept a rule, apply the rule, and return the adjusted program. If a rule cannot legally be applied, $P1$ is not successfully transformed. If a rule can be legally applied to $P1$, the program is compared lexically to $P2$ and if they match then equivalence has been proven.

Complexity of Proving Equivalence

Table 6.7 shows the complexity of the solution space for our problem for proofs from our AxiomStep10 test dataset up to length 7 (deterministically computing all possible programs requires too many resources for longer proof lengths). The 'All possible nodes and axioms' row includes the total number of proofs of a given length available to our problem space. The entry 5933 for a single axiom represents that for an AST depth of 7 we have 43 axioms which can be applied to all 63 possible operator nodes and 104 axioms which can be applied to the 31 nodes which possibly have child operator nodes themselves: 63*43+31*104=5933. Subsequent columns can select repeatedly from the same set growing as 5933^2 to 5933^7. The 'sample node + axiom group' row is based on our 10,000 sample test dataset and represents the possible selection of any of the 14 axiom groups being applied to any node in the program. The 'sample node + legal axiom' row represents only legal node plus legal axiom group being applied and effectively represents the total number of programs derivable from the start program in the test dataset. The final row 'Sample derivable unique programs' represents the total number of programs derived from legal node and axiom sequences which are lexically unique.

Table 6.7: Counts for equivalence proof possibilities

	Proof length in axioms						
Proof description	1	2	3	4	5	6	7
All Possible nodes and axioms	5933	3.5E+07	2.1E+11	1.2E+15	7.4E+18	4.4E+22	2.6E+26
Sample Node + Any Axiom	226	46900	1.5E+07	8.8E+09	5.0E+12	3.3E+15	2.7E+18
Sample Node + Legal Axiom	11.2	77.8	931	15812	3.4E+05	8.2E+06	1.8E+08
Unique Programs from Sample	9.2	47.4	264	1574	10052	65176	4.6E+05

6.9.2 Language and Axioms for Complex Linear Algebra Expressions

We now provide the complete description of the input language for multi-type linear algebra expressions we use to evaluate our work, and the complete list of all axioms that are used to compute equivalence between programs.

Variable types We model programs made of scalars, vectors and matrices. We limit programs to contain no more than 5 distinct variable names of each type in a program:

- Scalar variables are noted a, b, ..., e.
- Vector variables are noted $\vec{v}$, $\vec{w}$, ..., $\vec{z}$.
- Matrix variables are noted A, B, ..., E.

Note we also explicitly distinguish the neutral and absorbing elements for scalars and matrices, e.g. `1` $= 1_{\mathbb{N}}$. This enables the creation of simplification of expressions as a program equivalence problem, e.g. if $A + B - (B + A) = 0_{\mathbb{K}\times\mathbb{K}}$

Unary operators We model 6 distinct unary operators, all applicable to any variable of the appropriate type:

- `is(a)` $= a^{-1}$ is the unary reciprocal for scalars, `im(A)` $= A^{-1}$ is matrix inverse.
- `ns(a)` $= -a$ is unary negation for scalars, `nv(v)` $= -\vec{v}$ for vectors, `nm(M)` $= -M$ for matrices.
- `tm(M)` $= M^t$ is matrix transposition.

Binary operators We model 10 distinct binary operators that operate on two values. 7 operators require the same type for both operands, while 3 enable multi-type operands (e.g., scaling a matrix by a scalar). Note we do not consider potential vector/matrix size compatibility criterion for these operators, in fact we do not represent vector or matrix sizes at all in our language, for simplicity.

- `+s(a, b)`$= a + b$, the addition on scalars, along with `-s(a,b)` $= a - b$, `*s(a,b)` $= a * b$ and `/s(a,b)` $= a/b$.

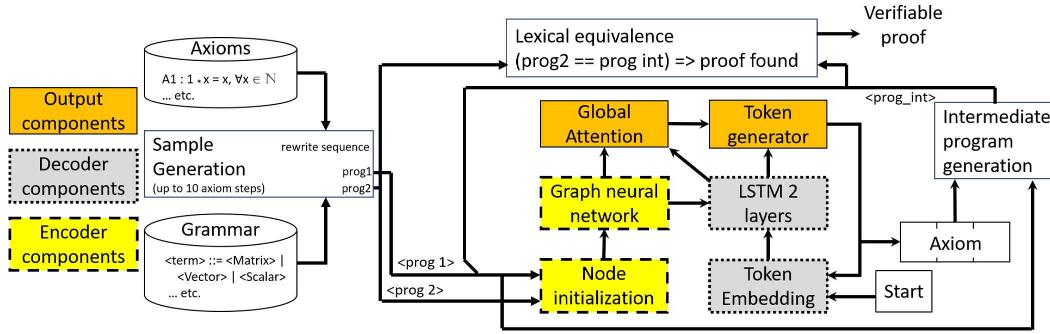


Figure 6.4: `pe-graph2axiom` System Overview

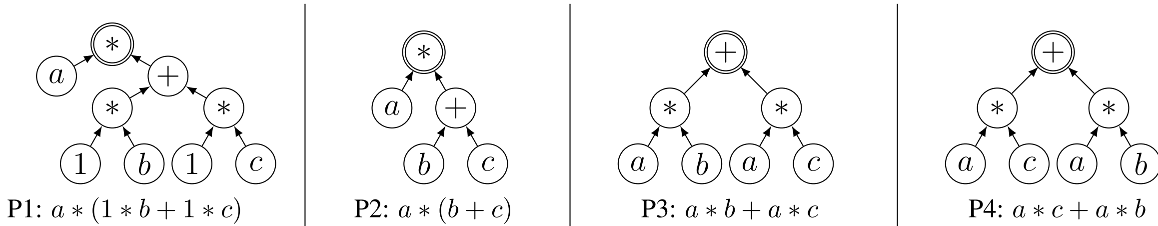


Figure 6.5: Examples of Computations Shown as Symbolic Expressions and Dataflow Graphs

- `+v( v, w)` $= \vec{v} + \vec{w}$, the addition on vectors, along with `-v( v , w)` $= \vec{v} - \vec{w}$, `*v( v, w)` $= \vec{v}.\vec{w}$ the dot product between two vectors, producing a scalar.
- `+m(A, B)` $= A + B$, the addition on matrices, along with `-m(A, B)` $= A - B$, and `*m(A, B)` $= AB$ the product of matrices.
- `*m(a,A)` $= a\dot{A}$ and `*m(A,a)` $= A\dot{a}$ are used to represent scaling a matrix by a scalar.
- `*m(v,A)` $= \vec{v}A$ represents a vector-matrix product.
- `*v(a,v)` $= a\vec{v}$ and `*v(v,a)` $= \vec{v}a$ represent scaling a vector by a scalar.

List of axioms of equivalence Tables 6.8-6.9 show the full 147 axioms supported by our rewrite rules. Many rewrite rules can be applied to all 3 variable types as well as multiple operator types.

6.9.3 Details on neural network model

Figure 6.4 overviews the entire `pe-graph2axiom` architecture including sample generation, the graph-to-sequence network, the intermediate program generation, and lexical equivalence checker. In this section we will discuss the implementation details of these components.

Rewrite Rule	ID	Example(s)
Cancel	1	(a - a) → 0
	2	(b/b) → 1
	3	(A - A) → O
	4	$(A * A^{-1})$ → I
	5	$(A^{-1} * A)$ → I
	6	(v - v) → o
NeutralOp	7	(a + 0) → a
	8	(0 + a) → a
	9	(a - 0) → a
	10	(a * 1) → a
	11	(1 * a) → a
	13	(a / 1) → a
	14	(A + O) → A
	15	(O + A) → A
	16	(A - O) → A
	17	(A * I) → A
	18	(I * A) → A
	19	(v + o) → v
	20	(o + v) → v
	21	(v - o) → v
DoubleOp	22	-(-a)) → a
	23	$(a^{-1})^{-1}$ → a
	24	$-(-A) \rightarrow A$
	25	$(A^{-1})^{-1} \rightarrow A$
	26	$(A^t)^t \rightarrow A$
	27	$-(-v)) \rightarrow v$
DistributeRight	64	a(b + c) → ab + ac
	65	a(b - c) → ab - ac
	66	a(v + w) → av + av
	67	a(v - w) → av - av
	68	A(B + C) → AB + AC
	69	A(B - C) → AB - AC
	70	a(B + C) → aB + aC
	71	a(B - C) → aB - aC

Rewrite Rule	ID	Example(s)
AbsorbOp	28	(a * 0) → 0
	29	(0 * a) → 0
	30	(A * 0) → O
	31	(0 * A) → O
	32	(A * O) → O
	33	(O * A) → O
	34	(A * o) → o
	35	(a * o) → o
	36	(o * a) → o
	37	(0 * v) → o
	38	(v * 0) → o
	39	(O * v) → o
Commute	40	(a + b) → (b + a)
	41	(a * b) → (b * a)
	42	(A + B) → (B + A)
	43	(A * a) → (a * A)
	44	(a * A) → (A * A)
	45	(A * O) → (O * A)
	46	(O * A) → (A * O)
	47	(A * I) → (I * A)
	48	(I * A) → (A * I)
	49	(v + w) → (w + v)
	50	(v * a) → (a * v)
	51	(a * v) → (v * a)
DistributeLeft	52	(a + b)c → ac + bc
	53	(a - b)c → ac - bc
	54	(a + b)/c → a/c + b/c
	55	(a - b)/c → a/c - b/c
	56	(v + w)*a → va + wa
	57	(v - w)*a → va - wa
	58	(A + B)C → AC + BC
	59	(A - B)C → AC - BC
	60	(A + B)v → Av + Bv
	61	(A - B)v → Av - Bv
	62	(A + B)a → Aa + Ba
	63	(A - B)a → Aa - Ba

Table 6.8: Full axiom count with all type options and other supported permutations included (part 1 of 2)

Graph neural network internal representation The sample generation discussed in section 6.4 provides input to the Node Initialization module in Fig. 6.4 to create the initial state of our graph neural network. For each node in the program graph, a node will be initialized in our graph neural network. Each node has a hidden state represented by a vector of 256 floating point values

Rewrite Rule	ID	Example(s)
FactorLeft	72	ab + ac → a(b+c)
	73	ab - ac → a(b-c)
	74	AB + AC → A(B+C)
	75	AB - AC → A(B-C)
	76	Av + Aw → A(v+w)
	77	Av - Aw → A(v-w)
	78	Aa + Ab → A(a+b)
	79	Aa - Ab → A(a-b)
	80	va + vb → v(a+b)
	81	va - vb → v(a-b)
FactorRight	82	ac + bc → (a+b)c
	83	ac - bc → (a-b)c
	84	a/c + b/c → (a+b)/c
	85	a/c - b/c → (a-b)/c
	86	AC + BC → (A+B)C
	87	AC - BC → (A-B)C
	88	Av + Bv → (A+B)v
	89	Av - Bv → (A-B)v
	90	Aa + Ba → (A+B)a
	91	Aa - Ba → (A-B)a
	92	va + wa → (v+w)a
	93	va - wa → (v-w)a
AssociativeLeft	94	a+(b+c) → (a+b)+c
	95	a+(b-c) → (a+b)-c
	96	a(bc) → (ab)c
	97	a(b/c) → (ab)/c
	98	A+(B+C) → (A+B)+C
	99	A+(B-C) → (A+B)-C
	100	A(BC) → (AB)C
	101	A(Ba) → (AB)a
	102	A(aB) → (Aa)B
	103	a(AB) → (aA)B
	104	A(Bv) → (AB)v
	105	A(va) → (Av)a
	106	A(av) → (Aa)v
	107	a(Av) → (aA)v
	108	v+(w+x) → (v+w)+x
	109	v+(w-x) → (v+w)-x
	110	v(ab) → (va)b
	111	a(vb) → (av)b
	112	a(bv) → (ab)v
AssociativeRight	113	(a+b)+c → a+(b+c)
	114	(a+b)-c → a+(b-c)
	115	(ab)c → a(bc)
	116	(A+B)+C → A+(B+C)
	117	(A+B)-C → A+(B-C)
	118	(AB)C → A(BC)
	119	(AB)a → A(Ba)
	120	(Aa)B → A(aB)
	121	(aA)B → a(AB)
	122	(Av)a → A(va)
	123	(Aa)v → A(av)
	124	(aA)v → a(Av)
	125	(va)b → v(ab)
	126	(av)b → a(vb)
	127	(ab)v → a(bv)
	128	(v+w)+x → v+(w+x)
	129	(v+w)-x → v+(w-x)
FlipLeft	130	-(a - b) → b-a
	131	$(a/b)^{-1}$ → b/a
	132	$-(A-B)$ → (B - A)
	133	$-(v-w)$ → (w - v)
FlipRight	134	a/(b/c) → a(c/b)
	135	$a/(b^{-1})$ → ab
	136	a-(b-c) → a+(c-b)
	137	a-(-b) → a+b
	138	A-(B-C) → A+(C-B)
	139	A-(-B) → A+B
	140	v-(w-x) → v+(x-w)
	141	v-(-w) → v+w
Transpose	142	$(AB) \rightarrow (B^tA^t)^t$
	143	$(A+B) \rightarrow (A^t+B^t)^t$
	144	$(A-B) \rightarrow (A^t-B^t)^t$
	145	$(AB)^t \rightarrow B^tA^t$
	146	$(A+B)^t \rightarrow A^t+B^t$
	147	$(A-B)^t \rightarrow A^t-B^t$

Table 6.9: Full axiom count with all type options and other supported permutations included (part 2 of 2)

which are used to create an embedding for the full meaning of the given node. Initially all 256 dimensions of the hidden states of the nodes are set to zero except for 2. Given N tokens in our input program language, one of the dimensions from 1 through N of a node will be set based on the token at the program position that the node represents. For example, if the scalar variable a is assigned to be token 3 in our language, then the a nodes of Fig. 6.5 recalled below would have their 3rd dimension initialized to 1.0. This is a one-hot encoding similar to that used in neural machine translation models which leverage Word2vec [159]. The second non-zero dimension in our node initialization indicates the tree depth, with the root for the program being at depth 1. We set the dimension N+$depth$ to 1.0; hence, the a nodes in Fig 6.5, which vary from level 2 or 3 in the graph, would set dimension $N + 2$ or $N + 3$ to 1. In addition to nodes correlating to all tokens in both input programs, we initialize a root node for program comparison which has edges connecting to the root nodes of both programs. The root node does not represent a token from the language, but it is initialized with a 1.0 in a hidden state dimension reserved for its identification.

For a graph neural network, the edge connections between nodes are a crucial part of the setup. In particular, to match the formulation of our problem, we must ease the ability of the network to walk the input program graphs. We therefore designed a unified graph input, where both program graphs are unified in a single graph using a single connecting root node; and where additional edges are inserted to make the graph fully walkable.

In our full model, we support 9 edge types and their reverse edges. The edge types are: 1) left child of binary op, 2) right child of binary op, 3) child of unary op, 4) root node to program 1, 5) root node to program 2, 6-9) there are 4 edge types for the four node grandchilden (LL, LR, RL, RR). After the node hidden states and edge adjacency matrix are initialized, the network is ready to begin processing. This initial state is indicated in figure 6.6 by the solid circles in the lower left of the diagram.

Beam search A typical approach when using sequence-to-sequence systems is to enable *beam search*, the process of asking for multiple answers to the same question to the network. It is particularly relevant when creating outputs which can be automatically checked [48, 5]. Beam

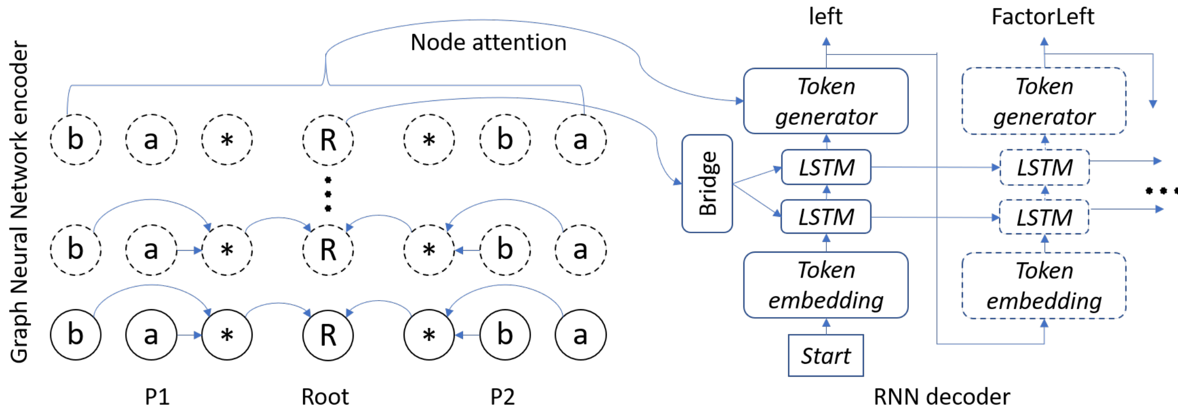


Figure 6.6: Graph-to-sequence neural network data flow details.

search can be viewed as proposing multiple possible axioms to apply. Given the stochastic nature of generation model, a beam width of n can be thought of as creating the n most likely sequences given the training data the model as learned on. Each proposal can be checked for validity, the first valid one is outputted by the system, demonstrating equivalence. Our system builds on the neural network beam search provided by OpenNMT to create a 'system beam search' of variable width. In particular, we set the OpenNMT network beam search to 3, which constrains the token generator to produce 3 possible axiom/node proposals for a given pair of input programs. Using these 3 proposals, when our system beam width is 10, we build up to 10 intermediate programs that are being processed in the search for a proof. To illustrate with a system beam width of 5, after $P1$ and $P2$ are provided to the neural network, 3 possible intermediate programs may be created (so long as all axioms are legal and don't produce duplicates). After those 3 intermediates are processed, 9 possible new intermediates are created, all of which are checked for lexical equivalence with $P2$, but only 5 of which are fed back into the neural network for further axiom generation. This process is continued for up to 12 axioms at which point the system concludes an equivalence proof cannot be found and the programs are likely not equivalent. We evaluate in Sec. 7.4.4 beam sizes ranging from 1 to 10, showing higher success with larger beams.

6.9.4 Details on Experimental Results

Complementary Results and Observations

Table 6.10 describes part of our neural network hyperparameter tuning showing that our golden model has as high a result as other variations explored. Note that the validation token accuracy is not too high (it's not above 90%) despite the ability to predict full correct proofs with over 93% accuracy. This is because the training dataset can have multiple examples of axioms given similar input programs. For example, proving "(a+b)(c+d) = (b+a)(d+c)" requires commuting the left and right subexpressions. The training dataset could have similar programs which are sometimes transformed first with a right Commute and then a left or vice-versa. Given this data, the network would learn to apply one or the other (it would not get trained to use associativity for these program pairs for example), hence the actual output given may or may not match the validation target axiom. We will discuss this further in section 6.9.4.

Table 6.10: Hyperparameter experiments. Summary of best validation token accuracy result after 2 runs for up to 100,000 training iterations. The golden model has 256 graph nodes and decoder dimensions, 2 decoder LSTM layers, starts training with a learning rate of 0.8, and uses 10 steps to stabilize the GGNN encoder.

Parameter	Value	Validation token accuracy
Golden model		83.89
Graph node+decoder LSTM dimension	192	83.89
	320	83.58
Decoder LSTM layers	1	83.53
Initial learning rate	0.75	83.76
	0.85	83.57
GGNN stability steps	12	83.19
	8	83.61

Training convergence Since our model trains on axiomatic proofs which may vary in order (allowing 2 or 3 options to be correct and occur in the training set), we see our training and token accuracies plateau below 90% during training for AxiomStep10 as shown in Figure 6.7. Full

testset proof accuracies for beam width 10 exceed 90%, but also plateau along with the training and validation results. This result differs from our WholeProof10 training, which achieves training and validation accuracies above 96% because the expected axiom sequence is more predictable, but as we have seen less generalized.

As another observation on generalization and overfitting, we note that figure 6.7 shows a slight separation between the training and validation accuracies starting at around iteration 180,000. While the training accuracy rises slowly, validation accuracy plateaus, indicating slight overfitting on the training data. Yet our model continues to slowly increase in quality, with the model snapshot that scores best on both validation and test accuracies occurring at iteration 300,000. This is our golden model, with 93.1% of P1 to P2 proofs accurately found using beam width 10.

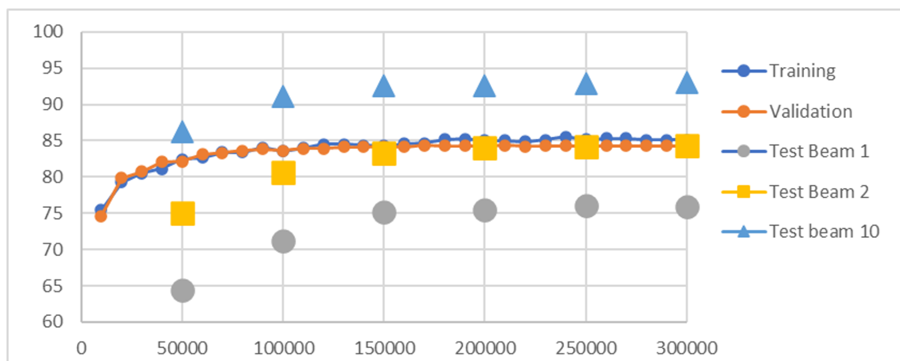


Figure 6.7: Model training percentage accuracy up to 300,000 iterations on AxiomStep10. Training and Validation accuracies are per-token on the target axioms in the samples. Test accuracies are for full correct proofs of P1 to P2.

Testing simpler models In addition to the sequence-to-sequence and graph-to-sequence models, we explored a feed-forward equal/not equal classifier on a simple version of our language. That model uses an autoencoder on the program to find an embedding of the program and then a classifier based on the program embeddings found. It achieves a 73% accuracy on identifying equivalent pairs in the test data, which, as expected, is much lower than the full proof rate of 93% achieved with a graph-to-sequence proof generator on our full language. This simple experiment highlights

the importance of a system which prevents the false positives which a classifier might have by creating a verifiable proof.

We explore initial language generation using a simple language in order to assess feasibility of different approaches. For fine tuning network parameters and architectural features, we add more complexity to the language as shown in table 6.11. Language IDs 1 through 3 are all based on a simple grammar which only allows the "+" or "-" operators on scalar variables labeled a through j. The only axiom is `Commute`, which can be applied on up to 3 nodes in language IDs 2 and 3. Language ID 4 adds the scalar constants 0 and 1, scalar operations * and /, and 4 more axioms. We perform a fair amount of network development on this model in an effort to maintain high accuracy rates. Language ID also 4 expands the operands to 3 types and hence the number of operators also increases. To speed up model evaluation, we reduced the program length for IDs 5, 6, and 7, allowing us to train larger data sets for more epochs. ID 7 is a forward looking-model which makes a minor increment to the language to support the analysis of loop rolling and unrolling, discussed further in section 6.9.4. ID 8 is the WholeProof5 model in relation to these early experiments.

ID	Description	# Operators	# Axioms	# Operands	Program length	Rewrite rules length	Graph2seq (G2S) or seq2seq (S2S)	Training set size	Percent matching with beam width 1	Percent matching with beam width 10
1	Rewrite sequence is only single Commute, uses sequence-to-sequence model	2	1	10	3-19	1-5	S2S	80,000	90.0%	96.2%
2	Rewrite sequence is exactly 2 Commutes, uses sequence-to-sequence model	2	1	10	5-24	3-10	S2S	80,000	80.3%	96.5%
3	Rewrite sequence exactly 2 Commutes	2	1	10	5-24	3-10	G2S	80,000	98.9%	99.8%
4	Rewrite sequence exactly 3 Commutes	2	1	10	7-45	5-15	G2S	80,000	91.4%	99.0%
5	Rewrite sequence 1 to 3 Commutes	2	1	10	3-45	1-15	G2S	180,000	97.1%	99.2%
7	Commute, Noop, Cancel, Distribute Left, Distribute Right	4	5	12	3-45	1-15	G2S	180,000	93.1%	97.4%
8	Scalars, Vectors, and Matrixes	16	5	20	3-30	1-25	G2S	250,000	88.3%	95.6%
9	13 Axioms	16	13	20	3-30	1-25	G2S	400,000	85.5%	95.5%
10	Rewrite sequence or Not_equal	16	13	20	3-30	1-25	G2S	500,000	79.8%	93.8%
11	Test sequence-to-sequence	16	13	20	3-30	1-25	S2S	400,000	59.8%	81.1%
12	Add loop axioms	18	15	20	3-30	1-25	G2S	400,000	83.8%	94.7%

Table 6.11: Results for various language complexities studied, on non-incremental models (WholeProof).

We designed our datasets in section 6.4 with the goal of using the varied models to understand the generalizability of `pe-graph2axiom` and to show that our model is not overfitting on training data. For these next experiments, all results of for beam width 10, which provides for a neural-network directed search of up to 10 axiomatic proofs of equivalence for each program pair. Recall that our most complex dataset is AxiomStep10 which includes $(P1, P2, S)$ samples requiring up to 10 rewrite rules, $P1$ and $P2$ can have up to 50 AST nodes each, and an AST depth of up to 7. AxiomStep5 has samples requiring up to 5 rewrite rules, $P1$ and $P2$ can have up to 25 AST nodes each, and an AST depth of up to 6. Tables 6.12 and 6.13 (repeated from main paper below) demonstrate the ability of a model trained on AxiomStep5 to perform well on the larger distribution of programs from AxiomStep10, implying that the model has generalized well to our program equivalence problem and that `pe-graph2axiom` does not overfit its response to merely the training set distribution.

Table 6.12: Generalizing to longer P1 inputs. Percentage pass rates for equivalence proofs with P1 having increasing program graph nodes. The model trained with the AxiomStep5 dataset had no training examples more than 25 program graph nodes yet it performs relatively well on these more complex problems. The furthest right column shows the `pe-graph2axiom` model results on the most complex dataset.

	Testset Sample Count		Model trained on AxiomStep5		Model trained on AxiomStep10	
P1 nodes	AS5	AS10	AS5	AS10	AS5	AS10
1-5	231	109	100	100	100	**100**
6-10	2147	1050	100	99	99	**99**
11-15	3980	2175	99	96	99	**96**
16-20	2583	2327	98	92	98	**93**
21-25	1059	1989	97	89	98	**92**
26-30	0	1229	N/A	83	N/A	**90**
31-35	0	698	N/A	78	N/A	**88**
36-40	0	304	N/A	74	N/A	**87**
41-45	0	101	N/A	68	N/A	**84**
46-50	0	27	N/A	67	N/A	**85**
All	10000	10000	99	90	99	**93**

Table 6.13 illustrates the ability of a model trained on AxiomStep5 (i.e., limited to proofs of length 5) to perform well when evaluated on the more complex AxiomStep10, which includes

Table 6.13: Performance vs. AST size: counts and percentage pass rates.

	Testset Sample Count		Model trained on AxiomStep5		Model trained on AxiomStep10	
AST depth	AS5	AS10	AS5	AS10	AS5	AS10
2	5	3	100	100	100	100
3	306	133	100	100	100	100
4	1489	577	100	99	99	99
5	4744	1844	99	94	98	95
6	3456	4308	98	90	98	93
7	0	3135	n/a	86	n/a	92
All	10000	10000	99	90	99	**93**

proofs of unseen length of up to 10. The robustness to the input program complexity is illustrated with the 86% pass rate on AST depth 7, for the model trained on AxiomStep5 which never saw programs of depth 7 during training.

As an indication of the breadth of equivalent programs represented by AxiomStep10 relative to WholeProof10, table 6.14 shows the full detail of models trained on all 4 datasets when tested on test data from all 4 datasets. AxiomStep10, while training on our broadest dataset in which axioms can be applied to nodes repeatedly and in variable order, achieves a 93% average success rate. 72% of the proofs of length 6 from the WholeProof10 testset were solved by the model trained on WholeProof10, but only 5% of such proofs from AxiomStep10 were, suggesting the method of generating AxiomStep pairs covers the problem space more thoroughly.

The complete result for the WholeProof10 model on the WholeProof10 dataset was 8,388 out of 10,000 program pairs had a correct proof found; of those, 8,350 were the exact proof created during $P1, P2$ generation, implying that WholeProof10, while performing well on its own testset distribution, is not learning to generalize to alternative proof paths.

Manual verifications We conducted a series of manual verifications of the system used to produce all the above results. First, we are happy to confirm that most likely $AB \neq BA$ given no verifiable equivalence sequence was produced, but that provably $ab = ba$ indeed. We also verified that $A^{t^t}(B + C - C) = AB$, and that $AB\vec{v} - AB\vec{w} = AB(\vec{v} - \vec{w})$ which would be a much faster

Table 6.14: Generalizing to longer proofs. Percentage pass rates for equivalence proofs of increasing axiom counts when testing each of 4 datasets on models trained using each of 4 datasets.

Axiom Count in	Model trained on WholeProof5 (WP5)				Model trained on WholeProof10 (WP10)				Model trained on AxiomStep5 (AS5)				Model trained on AxiomStep10 (AS10)			
Proof	WP5	WP10	AS5	AS10	WP5	WP10	AS5	AS10	WP5	WP10	AS5	AS10	WP5	WP10	AS5	AS10
1	100	100	100	99	100	100	100	100	100	100	100	100	100	100	100	**100**
2	99	98	66	64	99	99	65	63	100	99	100	99	100	100	100	**100**
3	98	94	34	33	97	95	33	33	100	98	99	98	100	99	99	**99**
4	93	84	16	15	90	88	16	15	98	95	98	97	99	98	98	**98**
5	84	70	8	7	84	82	8	7	96	91	96	95	97	95	96	**96**
6		14		4		72		5		81		88		90		**93**
7		0		1		63		2		67		81		83		**87**
8		0		0		54		1		54		75		73		**82**
9		0		0		47		0		35		64		63		**74**
10		0		0		34		0		24		57		46		**66**
All	95	66	44	27	94	84	44	27	99	87	99	90	99	93	99	**93**

implementation. The system correctly suggests that $AB\vec{v} - BA\vec{w} \neq AB(\vec{v} - \vec{w})$. We ensured that $A^t(AA^t)^{-1}A \neq A^t(AA^{-1})^tA$, from a typo we once made when typing the computation of an orthonormal sub-space. We also verified that indeed $AB + AC + aD - aD = A(B + C)$.

Generalizing variable types We explored the ability of the model to understand variable typing by training a model with the AxiomStep10 distribution but with no samples that included the scalar variable 'e' and scalar multiplication $*_s$. This removed about 50% of the training set, as longer programs were often included both tokens. When tested with the unaltered AxiomStep10 test set and beam width 10, test samples that included a scalar variable not 'e' and $*_s$ were proven equal 90% of the time; test samples that included 'e' and $*_s$ were also proven equal 90% of the time. For beam width 1 the proof success rates were 72% and 70% for without and with 'e', implying that the heavily biased training set did have a small effect on the system generalization. `pe-graph2axiom` was still able to generalize the relation of 'e' to the $*_s$ operator given that 'e' was used in contexts similar to other scalar variables in the training samples that were provided, implying it was forming an internal representation of a 'scalar' type by learning from examples.

Learning that multiple axiom choices are possible

Our AxiomStep10 model is trained on axioms which may be applied in varying order in the training set. For example, $((a + b) * (c + d)) = ((b + a) * (d + c))$ may have the training data to Commute the left node $a + b$ first and then $c + d$ second; in the same dataset, $((a + e) * (b + c)) = ((e + a) * (c + b))$ might occur and the training data has the right node Commuted first. In this way, we expect the model to learn that either commuting the left or right node is a proper first axiom choice. Table 6.15 explores the ability of the model to produce such axiom proposals. Given 5 scalar variables, there are 120 possible expressions where two 2-variable additions are multiplied together such as $((a + b) * (c + d))$. We consider here all 120 program pairs in which the left and right additions are commuted. The table shows which axioms and positions are recommended by the graph-to-sequence neural network model within the `pe-graph2axiom` system as most probably moving the 2 programs closer to equivalence by the beam width 3 on this problem. Note that the 2 correct axioms are always within the top 3 choices and the other 2 axioms (Commute and DistributeLeft on the root), while not necessary for this problem, are at least legal choices for axioms within our expression language.

The results in table 6.15 relate to the value of our approach in relation to reinforcement learning models for proof generation [69, 23]. To make an analogy with reinforcement learning, in our training, the world 'state' is presented as a $P1, P2$ pair and the system must learn to produce an axiom at a location which performs an 'action' on the 'state' of $P1$ in a predictable way. Unlike reinforcement learning, we do not produce a reward function and our system cannot learn from a poor reward produced by an incorrect axiom. However, we have demonstrated that our system, as it is presented with a wide distribution of $(P1, P2, S)$ tuples to train on, learns a probability distribution of possibly correct axioms to produce for a given program pair. There may be value in combining our graph-neural-network within a reinforcement learning framework that used a hindsight mechanism [14] to learn from every attempted axiom, but it is not immediately obvious that our approach of learning only from examples of successful equivalence proofs would be improved.

Table 6.15: Learning multiple output options. When considering scalar expressions that can be proven equivalent by commuting the left and right subexpressions, such as $(a+b)(c+d) = (b+a)(d+c)$, `pe-graph2axiom` learns that either the left or right commute can occur first. The columns show counts for axioms and locations proposed by the token generator with beam width of 3 when given 120 different scalar expression pairs.

	Axiom			
Beam position	Commute left child	Commute right child	Commute root	DistributeLeft root
First	49	35	36	0
Second	58	59	3	0
Third	13	26	45	36
Any of top 3	120	120	84	36

Exploration of alternate designs In order to design the system, we explored parts of the design space quickly and performed several single training run comparisons between 2 options, as shown in Table 6.16.

In cases where 2 options were similar, we chose the model which ran faster, or run the models a second time to get a more precise evaluation, or use our experience based on prior experiments to select an option.

Table 6.16: Example explorations as a single feature or parameter is changed. Each comparison is a distinct experiment, as the entire network and language used was being varied.

Options compared	Match beam 1	Match beam 10
1 layer LSTM vs	198	1380
2 layer LSTM vs	5020	9457
3 layer LSTM	4358	8728
No edges to grandchild nodes vs	9244	9728
Edges to grandchild nodes	9284	9774
Encoder->Decoder only root node vs	8616	9472
Encoder->Decoder avg all nodes	7828	9292

Experiments such as these informed our final network architecture. In `pe-graph2axiom`, for example, we include 4 edges with learnable weight matrices from a node to its grandchildren because such edges were found to improve results on multiple runs. Li *et al.* [135] discusses

the importance of selecting the optimal process for aggregating the graph information hence we explore that issue for our network. Our approach uses the root comparison node to create aggregate the graph information for the decoder as it performs better than a node averaging.

Including Not_equal option Table 6.17 analyzes the challenge related to a model which only predicts Equal or Not_equal for program pairs along with various options which produce rewrite rules which can be checked for correctness. In all 4 output cases shown, 2 programs are provided as input. These programs use an earlier version of our language model with 16 operators, 13 core axioms, and 20 operands generated with a distribution similar to WholeProof5.

Table 6.17: Table showing alternate options for handling not equal programs

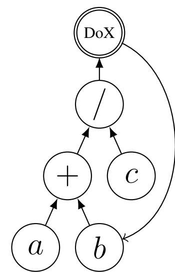
Network output Description	Actual	Predicted NotEq	Predicted Rules or Eq	Correct Rewrite Rules
Eq or NotEq, Beam width 1	Eq	5.4%	94.6%	N/A
	NotEq	90.4%	9.6%	N/A
Rules or NotEq, Beam width 1	Eq	6.6%	93.4%	70.7%
	NotEq	90.9%	9.1%	N/A
Rules only, Beam width 1	Eq	N/A	100%	87.8%
	NotEq	N/A	N/A	N/A
Rules only, Beam width 10	Eq	N/A	100%	96.2%
	NotEq	N/A	N/A	N/A

For the first output case, the output sequence to produce is either `Equal` or `Not_equal`. Given a false positive rate of 9.6%, these results demonstrate the importance of producing a verifiable proof of equivalence when using machine learning for automated equivalence checking. For the second output case, the model can produce either `Not_equal` or a rewrite rule sequence which can be checked for correctness. The source programs for the first and second case are identical: 250,000 equivalent program pairs and 250,000 non-equivalent program pairs. In the second case, the false positive rate from the network is 9.1% (rules predicted for Not_equal programs), but the model only produces correct rewrite rules between actual equivalent programs in 70.7% of the cases.

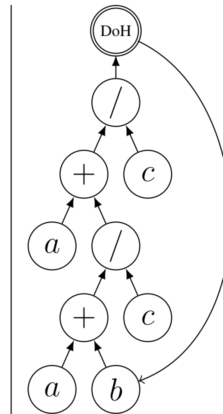
One challenge with a model that produce rules or `Not_equal` is that beam widths beyond 1 are less usable. Consider that with a beam width of 1, if the network predicts `Not_equal` then the checker would conclude the programs are not equal (which is correct for 90.9% of the actually not equal programs). With a beam width of 10, there would be more proposed rewrite rules for equal programs to test with, but if 1 of the 10 proposals is `Not_equal`, should the checker conclude they are not equal? Or should the the checker only consider the most likely prediction (beam width 1) when checking for non-equivalence? The third and fourth network output cases provide an answer. For these 2 cases, the training set is 400,000 equivalent program pairs - none are non-equivalent. 250,000 of these pairs are identical to the equivalent programs in the first 2 cases, and 150,000 are new but were produced using the same random generation process. Note that by requiring the network to focus only on creating rewrite rules, beam width 1 is able to create correct rewrite rules for 87.8% of the equivalent programs. And now, since we've remove the confusion of the `Not_equal` prediction option, beam width 10 can be used to produce 10 possible rewrite rule sequences and in 96.2% of the cases these rules are correct. Hence, we propose the preferred use model for pe-graph2axiom is to always use the model which is trained for rule generation with beam width 10 and rely on our rule checker to prevent false positives. From the 10 rewrite rule proposals, non-equivalent programs will never have a correct rewrite rule sequence produced, hence we guarantee there are no false positives.

An Example of Back-Edge in the Program Graph

Figure 6.8 shows an example of DoX and DoHalf. The new operators result in 2 new edges in our graph representation (along with 2 new back-edges): there is a 'loopbody' edge type from the loop operator node to the start of the subgraph, and there is a 'loopfeedback' edge type from the variable which is written to each loop iteration. These 2 edges types are shown in the figure. The new $Dohalf$ axiom intuitively states that $DoX(g(y)) = DoHalf(g(g(y)))$ (where y is the variable reused each iteration), and Dox states the reverse.



(a) DoX($b = (a + b)/c$)



(b) DoHalf($b = (a + (a + b)/c)/c$)

Figure 6.8: Adding loop constructs creates cycles in the program graph.

Chapter 7

Self-Supervised Learning to Prove Equivalence Between Programs via Semantics-Preserving Rewrite Rules

7.1 Introduction

We now build upon the program equivalence work presented in Chapter 6 by extending programs from complex linear algebra expressions to multiple assignment statements made up of such complex expressions. Additionally, we combine automatic verification concepts from R-HERO which we discuss in Section 4.7 and synthetic equivalence generation techniques from Chapter 6 to create *self-supervised sample selection* as we introduce in Section 3.4.

S4Eq takes as input two programs and generates a sequence of rewrite rules under a well-defined system for equivalence using semantics-preserving rewrite rules [64]. Our work studies programs represented as a list of statements with straight-line control-flow, using multiple variable types and complex mathematical expressions to compute values. S4Eq outputs a verifiable sequence of rewrite rules, meaning that it guarantees no false positives (no programs are stated equivalent if they are not) by design. The system we present in Chapter 6 uses a graph neural network in a graph-to-sequence model; we want to support a larger language (bigger and more complex programs) AND **make the axioms much more complex**: the decision of legally applying them will no longer only need to look at a node and its subtree, but instead may require program-wide analysis (looking at parent nodes), such as for common sub-expression elimination. As we show in Table 7.4, the Transformer model performs significantly better than a graph2sequence model when compared on similar equivalence proof problems, which justified our new proposed design.

The problem domain at hand, generating a provably correct sequence of rewrite rules, requires a specific training procedure. We devise a novel self-supervised learning technique for proving equivalence. We initially train a model in a supervised manner with synthetic data which has a broad distribution on the use of rewrite rules. Then we propose a self-supervised technique based on comparing results between broad and narrow proof searches to incrementally train our model. Rewrite rule sequences demonstrating equivalence found by a quick, narrow search are not considered interesting for further training; while sequences found by a broad search indicate samples for which the model's rewrite rule selections could be improved. We name this procedure *self-supervised sample selection*. We fully implement our learning and inference models in the popular OpenNMT-py framework [117], based on the transformer model.

To sum up, we make the following contributions:

- We present S4Eq, an end-to-end deep learning framework to find equivalence proofs between two complex program blocks. S4Eq produces a sequence of semantics-preserving rewrite rules that can be built to construct one program from the other, via successive rewrites. We consider rewrites which support complex program transformations such as Common Subexpression Elimination and computational strength reduction. S4Eq emits a verifiable sequence of rewrites, leading to no false positive by design.

- We devise an original training technique, tailored to our problem domain, called self-supervised sample selection. This incremental training approach further improves the quality, generalizability and extensibility of the deep learning system.

- We present extensive experimental results to validate our approach, demonstrating our system can successfully prove equivalence on both synthetic programs and programs derived from GitHub with up to 97% success, making the system ready for automated and unsupervised deployment to check equivalence between programs.

Prog A (source code): (a)

```
y_diff = ( particles [ i ] . y_pos - particles [ j ] . y_pos ) ;
r = sqrt ( ( x_diff * x_diff ) + ( y_diff * y_diff ) ) ;
mass = particles [ j ] . mass ;
mass /= ( ( r + EPSILON ) * ( r + EPSILON )
                              * ( r + EPSILON ) ) ;
x_diff *= mass ;
y_diff *= mass ;
sumX += x_diff ;
sumY += y_diff ;
```

Prog A (abstracted): (b)

```
t1 = ( i1 - i2 ) ;
t2 = f1 ( ( i3 * i3 ) + ( t1 * t1 ) ) ;
t3 = i4 ;
t4 = t3 / ( ( t2 + i5 )
     * ( t2 + i5 ) * ( t2 + i5 ) ) ;
t5 = i3 * t4 ;
t6 = t1 * t4 ;
o1 = i6 + t5 ;
o2 = i7 + t6 ;
```

Prog A (prefix encoding of AST): (c)

```
s28 = ( -s s01 s02 ) ;
s27 = ( u1s ( +s ( *s s03 s03 ) ( *s s28 s28 ) ) ) ;
s26 = s04 ;
s25 = ( /s s26 ( *s ( +s s27 s05 )
          ( *s ( +s s27 s05 ) ( +s s27 s05 ) ) ) ) ;
s24 = ( *s s03 s25 ) ;
s23 = ( *s s28 s25 ) ;
s30 === ( +s s06 s24 ) ;
s29 === ( +s s07 s23 ) ;
```

Prog B (source code): (d)

```
distancey = y - src -> center_y ;
rij = sqrt ( distancex * distancex + distancey * distancey ) ;
cst_j = src -> center_mass * 1.0 / ( ( rij + E0 ) * ( rij + E0 )
                                        * ( rij + E0 ) ) ;
cord_x = cst_j * distancex ;
cord_y = cst_j * distancey ;
Fx += cord_x ;
Fy += cord_y ;
```

Prog B (abstracted): (e)

```
t1 = i1 - i2 ;
t2 = f1 ( i3 * i3 + t1 * t1 ) ;
t3 = i4 * 1s / ( ( t2 + i5 )
     * ( t2 + i5 ) * ( t2 + i5 ) ) ;
t4 = t3 * i3 ;
t5 = t3 * t1 ;
o1 = i6 + t4 ;
o2 = i7 + t5 ;
```

Prog B (prefix encoding of AST): (f)

```
s28 = ( -s s01 s02 ) ;
s27 = ( u1s ( +s ( *s s03 s03 ) ( *s s28 s28 ) ) ) ;
s26 = ( *s s04 ( /s 1s ( *s ( +s s27 s05 )
          ( *s ( +s s27 s05 ) ( +s s27 s05 ) ) ) ) ) ;
s25 = ( *s s26 s03 ) ;
s24 = ( *s s26 s28 ) ;
s30 === ( +s s06 s25 ) ;
s29 === ( +s s07 s24 ) ;
```

Equivalence Rewrite Rule Sequence: (g)

```
stm4 MultOne Nr
stm4 Inline s26          stm3 Rename s26
stm3 DeleteStm           stm4 Rename s25
                         stm5 Rename s24
                         stm4 Commute N
                         stm3 NeutralOp Nr
                         stm3 DivOne Nr
                         stm3 FlipRight N
                         stm5 Commute N
```

ProgInt after "stm3 Deletestm": (h)

```
s28 = ( -s s01 s02 ) ;
s27 = ( u1s ( +s ( *s s03 s03 ) ( *s s28 s28 ) ) ) ;
s25 = ( /s s04 ( *s 1s ( *s ( +s s27 s05 )
          ( *s ( +s s27 s05 ) ( +s s27 s05 ) ) ) ) ) ;
s24 = ( *s s03 s25 ) ;
s23 = ( *s s28 s25 ) ;
s30 === ( +s s06 s24 ) ;
s29 === ( +s s07 s23 ) ;
```

Figure 7.1: Equivalence proven between 2 multi-statement programs. The equivalence proof is 11 steps long involving expression and statement rules.

- We provide all our datasets to the community including synthetic generation techniques for the problem of program equivalence via rewrite rules, as well as sequences mined from GitHub [121].

7.2 Problem Statement

We now explain the problem domain of proving program equivalence via rewrite rule sequences.

7.2.1 Scope

In this work we represent programs as a list of statements comprising symbolic expressions made of variables, operators, function calls, and neutral/absorbing elements (*e.g.,* $0, 1$). We support

Table 7.1: The 23 rewrite rules considered by S4Eq. Considering combinations with scalar (a, b, ...) and vector ($\vec{v}$, $\vec{w}$, ...) types, rewrite rules represent multiple operations of linear algebra.

Rewrite Rule	Example or Description	Rewrite Rule	Example or Description
SwapPrev	Swap assign statements	Inline VarID	Replace VarID with expression
UseVar VarID	Replace expr. with VarID	NewTmp NodeID VarID	Assign VarID to NodeID expr.
DeleteStm	Delete assign stm	Rename VarID	Change assignment to VarID
AddZero NodeID	$\vec{v} \to (\vec{0} + \vec{v})$, b→0+b	SubZero NodeID	$\vec{v} \to (\vec{v}$ - $\vec{0})$, b→b-0
MultOne NodeID	$\vec{v} \to (1 \times \vec{v})$, b→1×b	DivOne NodeID	a → a/1
Cancel NodeID	$(\vec{v} - \vec{v}) \to \vec{0}$,(b/b)→1	NeutralOp NodeID	$(\vec{v}$ - $\vec{o}) \to \vec{v}$, 1×a→a
DoubleOp NodeID	$-(-\vec{v}) \to \vec{v}$, 1/1/x→x	AbsorbOp NodeID	$(\vec{v} \times 0) \to \vec{0}$, (b×0)→0
Commute NodeID	(a + b) → (b + a)	DistributeLeft NodeID	(a + b)c → ac + bc
DistributeRight NodeID	a(b + c) → ab + ac	FactorLeft NodeID	ab + ac → a(b+c)
FactorRight NodeID	ac + bc → (a+b)c	AssociativeLeft NodeID	a(bc) → (ab)c
AssociativeRight NodeID	(ab)c→a(bc), (ab)/c→a(b/c)	FlipLeft NodeID	-($\vec{v}$ - $\vec{w}$) → $\vec{w} - \vec{v}$
FlipRight NodeID	a/(b/c) → a(c/b)		

both vector and scalar types, as well as operators and functions that mix these types. We support programs with single or multiple outputs of varying types.

This language is most applicable when comparing complex mathematical computations such as linear algebra expressions, sequences of basic blocks, or straight-line code equivalence [146, 102].

7.2.2 Encoding of Programs and Rewrite Rules

We now discuss how we represent programs and rewrite rules. Figure 7.1 illustrates how programs are represented. The input C programs, shown in boxes (a) and (d), contain multiple assignment statements in sequence. We process this code to create an abstracted representation, shown in boxes (b) and (e). Our lowest level of representation, shown in boxes (c) and (f), uses a prefix representation of the computation tree, with all computations using a one-operand syntax of `(op op1)` or a two-operand syntax of `(op op1 op2)`. This representation is an encoding of the abstract syntax tree (AST) for the abstracted program and it is this precise representation on which our rewrite rules operate.

We first process C programs using variable renaming [126] and global value numbering [188]; that is, to reduce the number of distinct tokens to handle we rename all distinct variables/array accesses to a unique and canonical name for that program. For example, `particles[i].y_pos`

becomes `i1` in the abstracted program. Without any loss of (local) semantics, this abstraction reduces the number of different variable names to handle.

We identify all inputs to a program block as those variables which are not assigned within the block; and outputs of the block are variables which are not read after assignment within the block. Our abstracted programs use `i1,i2,...` for input variables, `t1,t2,...` for temporary variables, and `o1,o2,...` for output variables.

Next, a prefix encoding of the AST is designed to provide for precise specification of rewrite rules with consideration for deep learning language representations, as discussed further in Section 7.3. The parenthesis positioning in this encoding allows for direct recognition of subtrees, and nodes of this tree can be used when rewrite rules are specified. For example, with reference to Figure 7.1, ProgA in subfigure (c) is transformed into ProgB in subfigure (f) with the 11 step rewrite rule sequence shown in subfigure (g).

Our rewrite rule syntax is:

```
stm# RuleName [NodeID] [VarID]
```

where `stm#` is the assignment statement number in ProgA which should have `RuleName` applied. `NodeID` optionally identifies the node within the right hand side of the statement, and `VarID` is the optional name of the variable to use for applying the rule.

For example, the very simple `MultOne` rewrite rule represents the semantically correct rewrite $A1 : x = 1_{\mathbb{N}} * x, \ \forall x \in \mathbb{N}$ when considering natural arithmetic. The first rewrite in Figure 7.1 is performed on statement 4, and applies the `MultOne` rule at NodeID `Nr`, which is our syntax to model the right child (r) of the root expression node (N). For example, the left child of node Nr would be denoted Nrl. Hence here, `s25 = (/s s26 ( *s ... ) )` is transformed into `s25 = (/s s26 (*s 1s (*s ... ) ) )`. The `Inline` rewrite rule replaces all instances of a given variable in the indicated statement with the RHS of its most recent assigment, and the `DeleteStm` rewrite rule will remove an assignment statement (this rule is only legal when the variable assigned is not an output of the program and is not used after being assigned). Hence,

the second and third rules in box (g) inline the value of `s26` into the current statement 4 and then delete the unnecessary assignment of `s26` resulting in the intermediate program shown in box (h).

We include in our rewrite rules 2 main groups: linear algebra axioms sufficient to allow our set of vectors to be considered an Abelian group mathematically and our set of scalars to be considered a field; then statement interactions which allow standard inter-statement compilation actions to be applied (such as replacing a variable by its assigned expression, which we refer to as (variable) inlining, or defining a new variable which can be shared by multiple statements). A description of all 23 of our rewrite rules is provided in Table 7.1.

Ultimately, the goal is produce a *verifiable* sequence of rewrite rules to transform ProgA into ProgB. Indeed, a sequence of rewrite rules forms a transformation recipe for a program: taking ProgA, we apply the first rewrite rule to obtain ProgA', then we apply the second rule in the sequence on ProgA' to obtain ProgA", etc. till the end of the sequence to obtain ProgA^{trans}, by successive rewrite steps. Then, to determine whether ProgA and ProgB are equivalent under this rewrite sequence, we check whether ProgA^{trans} is *syntatically* identical to ProgB. We also need to ensure the validity of the rewrite sequence: before each application of one rewrite rule, we first verify it is legal to apply by ensuring the conditions of applications are met (e.g., it is applied on an appropriate node/subtree structure). If the sequence can be legally applied, meaning each step does preserve semantics, and ProgA^{trans} is syntactically identical to ProgB, then we have computed a verified proof of equivalence between ProgA and ProgB. Verifying the validity of a sequence is trivial as it amounts to a simple (sub)tree shape and token values matching in our rewrite system.

7.2.3 Pathfinding Rewrite Rule Sequences

Intuitively, we can view the program equivalence solution space as a very large graph, where every possible syntactically different program in the language is represented by its own vertex v. Then, two vertices v_i and v_j are connected by a labeled edge iff applying one particular rewrite rule on a particular node of v_i is valid, and leads to producing the program v_j. The edge is labeled by the rewrite rule applied (as defined above). This graph is a multigraph, as multiple different

rewrites may connect the same two programs. It also contains cycles, as a sequence of rewrites can "undo" changes. Therefore, any two programs connected by a path in this graph are semantically equivalent: the rewrite sequence is the set of labels along the edges forming the path.

Building the rewrite rule sequence S for $ProgB \equiv S(ProgA)$ amounts to exposing one path (out of possibly many) from $ProgA$ to $ProgB$ in this graph when it exists, the path forming the proof of equivalence. We build a deep learning system to learn a stochastic approximation of an iterative algorithm to construct such feasible path when possible. Our approach avoids entirely the need to craft smart exploration heuristics for such large equivalence graph to make this path-finding problem practical (akin to building tactics in theorem provers): instead, the traversal heuristic is learned automatically, without any user input, by deep learning.

7.3 S4Eq: Deep Learning to Find Rewrite Rule Sequences

We propose to use a deep learning model to find rewrite rule sequences which transform one program into a semantically equivalent target program. The idea is to learn from correct rewrite sequences, and then to solve previously unseen program equivalence problems.

7.3.1 Overview of S4Eq

Prior work has shown that source code has patterns that are similar to human language [95], and thus techniques used in natural language processing can work on source code as well, including deep learning [48]. Deep learning has the ability to learn syntactic structure and expected outputs related to programs. For S4Eq we aim to create a deep learning model which, given 2 programs, will predict a sequence of rewrite rules which can formally prove equivalence between the 2 programs.

For S4Eq we use the state of the art sequence-to-sequence deep learning model known as the transformer model [208]. Because sequence-to-sequence models are stochastic, they can be used to produce multiple answers for the same query; this is called *beam search*. By using beam search

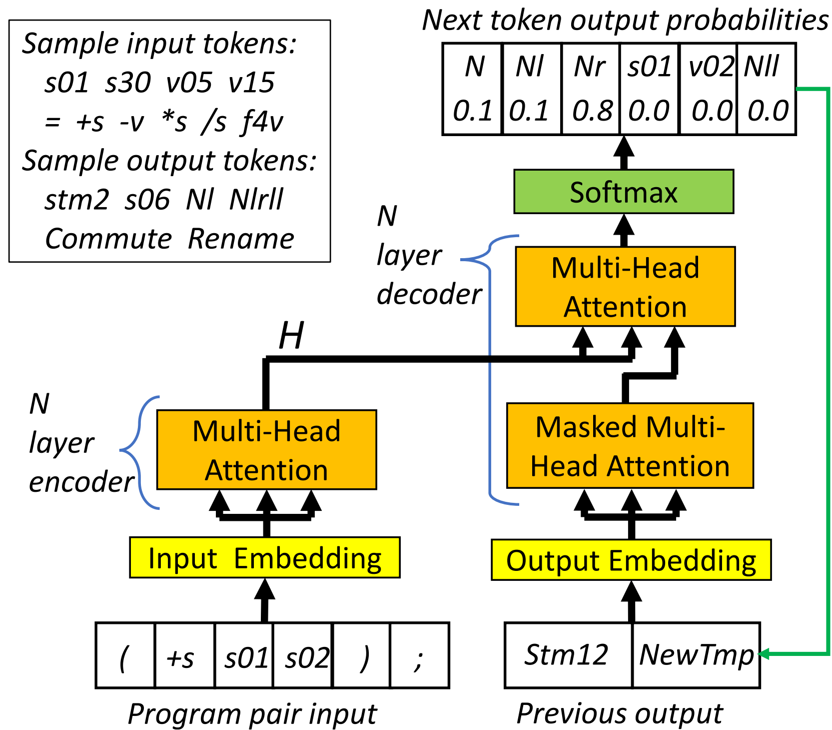


Figure 7.2: Transformer model used to predict rewrite rules given 2 input programs.

we can order rewrite rules proposed by the model. By composing beam search recursively, we can construct proofs which use heuristics learned by the model to guide proof search.

In S4Eq, we devise an original training process specifically for the task of proving equivalence. The S4Eq training process combines supervised and self-supervised learning and does not require human labeling.

7.3.2 Transformer Model

For S4Eq, we use a transformer model for sequence to sequence modeling based on the system we cover in Section 2.1.6. We introduce the input and output languages for X and Z in Section 7.2. Subfigures 7.1(c) and 7.1(f) are examples of such inputs. The output language for Z is illustrated in subfigure 7.1(g). For input to the transformer, we add a special token Y to separate the 2 programs, with ProgA being input before ProgB. The input and output languages are fully detailed in our GitHub repository [121].

In S4Eq, the transformer model we use is shown in Figure 7.2. The yellow boxes represent the model's learned interpretation for the tokens in the input and output. Tokens such as '*s' and '=' in

the input language or ’stm3’ and ’Commute’ in the output language have learned embeddings used by the transformer model. The model accomplishes context-dependent interpretation with multiple attention layers which learn a complex representation for a program node by learning which other nodes should be “attended to” while creating the higher level representation.

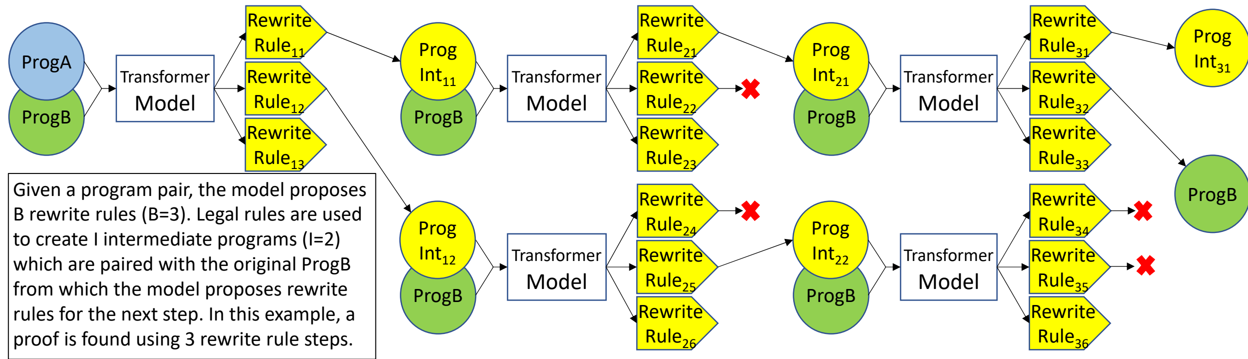


Figure 7.3: Proof search: Transformer model produces multiple rewrite rules at each step. Unusable rules are marked with **X**.

In the transformer model, the encoder and decoder use layers which provide an attention mechanism which relates different positions of a single sequence in order to compute a representation of the entire sequence. For example, the attention layers provide a mechanism for information related to the assignment of a variable to affect the representation of other parts of the program where the variable is used.

Multi-head attention allows for the different heads to learn different aspects of the data. For our problem of program equivalence, the model is trained to produce correct rewrite rules and hence all of the learnable functions are learning representations useful for this task. To illustrate, one of the heads may tend to build a representation for addition and subtraction of vectors, while another head might build a representation for multiplication and division of scalars. What the heads learn is not constrained by the architecture, so the true meaning of the heads at each layer is not easily decipherable, but multi-head attention has been shown by others and by our own ablation studies to be valuable.

Figure 7.2 identifies ’N layers’ for both the encoder on the left (which encodes the input programs to an intermediate representation) and the decoder on the right (which decodes the represen-

tation to produce the rewrite rule output). The 'N layer encoder' is using self-attention in which the information processed by a given layer is provided by the layer below. A similar situation holds for the 'N layer decoder', however the H connection from the encoder layers to the decoder layers allows the decoder to process information from the decoder and encoder. The transformer model we employ also includes residual and feed-forward sublayers and a full description of the interactions within the model can be read in the work by Vaswani, *et al.* [220]. We used the Adam optimizer [116] to adjust the weights in the network based on the loss function between the target token expected in the training sample and the result produced by the current network.

The intermediate representation H encodes the complex representations for each token of the input, in our case a pair of programs. The decoder model will generate a first output token based on H and then generate subsequent tokens based on H and the previously output tokens. The Softmax function normalizes the output of the final decoder layer to a probability distribution:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \texttt{for } \mathbf{z} = (z_1, ..., z_K) \in \mathbb{R}^K$$

The effect of the Softmax layer is to create an output that can be interpreted as representing the probability that a given token is correct, given the training the model has been exposed to. As the output is generated, when the tokens with the highest Softmax values are selected we create a rewrite rule which represents the most likely next edge in our path through the program space from ProgA to ProgB.

7.3.3 Beam Search to Explore Multiple Possible Proofs

A typical approach when using sequence-to-sequence models is called "beam search", it is the process of asking the deep learning model to produce multiple answers for the same question. As the network produces likely token selections, the Softmax layer shown in Figure 7.2 is effectively producing probabilities for each token. Using these probabilities we can produce multi-token outputs sorted in order based on the total probability the model assigns to the sequences.

In S4Eq, we use this beam search to enumerate the possible next rewrite rule to apply. Each proposal is then checked for legality (whether the proposed rewrite rule can indeed be applied at the given program location) and novelty (the program resulting from the rule application does not match a previously seen program for this search).

In addition to the rewrite rule beam, S4Eq uses a second type of beam during the proof search. The idea is to feed the network again based on the result of the application of the previously suggested rewrite rules. We denote the number of enumerated rewrite rules B. As the search advances, the B outputs from the neural network all lead to potential intermediate programs from which a search can continue. After having checked for legality, we limit this potential exponential growth in the search to at most configurable I intermediate programs which may be explored at a given proof step.

Consider a search where we set B to 3 and I to 2, as diagrammed in Figure 7.3. When a transformation search between 2 programs is attempted, at first there is only 1 sample (the original 2 programs to prove equivalent) fed into the transformer model which will propose 3 rewrite rules. Perhaps the first 2 are legal rewrite rules; both of these are checked for equivalence to the ProgB goal and assuming there is no match both will be fed into the transformer model on the next step. This will produce 6 proposed rewrite rules. If a rewrite rule would create an intermediate program that is already being searched (for example commuting the same node twice in a row) then the search process will not create the duplicate intermediate program. In the figure, rewrite rules which are illegal or create duplicate search programs are marked with a red X. The search routine will select the most likely proposed rule for each ProgInt/ProgB pair if it legally creates a novel intermediate program.

As diagrammed, the ProgInt_{11}/ProgB pair produces a legal novel program which is used for the next step, but the ProgInt_{12}/ProgB pair's 1st proposal is not usable. Since the 2nd proposal from the ProgInt_{11}/ProgB pair is also not usable, the legal 2nd proposal from the ProgInt_{12}/ProgB pair is used to complete the 2 entry I beam for the continuing search. Our search will 1st try to use the 1st proposed rule from each ProgInt/ProgB pair, then the 2nd, and so on until the next I intermediate

programs are created. We will limit the intermediate programs that feed into the transformer to I as the search is continued up to the rewrite rule sequence step limit N_s (such as 25 steps). In rare cases, none of the proposed rewrite rules for any of the intermediate programs will produce a legal novel intermediate program and the search will terminate before the step limit. In our example, the 2nd rewrite rule proposed from the model when given ProgInt_{21}/ProgB to the transformer rewrites ProgInt_{21} into the lexical equivalent of ProgB, and hence, a 3 step proof has been found. ProgA is transformed into ProgB by applying Rewrite Rule_{11}, Rewrite Rule_{21}, and then Rewrite Rule_{32}.

7.3.4 Training Process

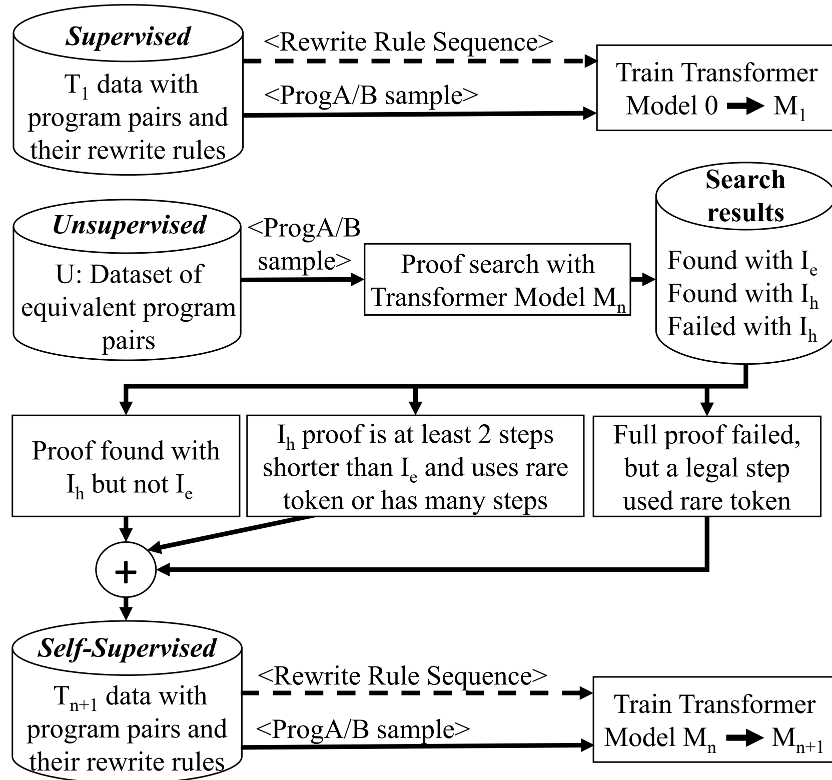


Figure 7.4: Self-Supervised Sample Selection: proof attempts with $I = I_e$ and I_h intermediate programs are used to incrementally train the model.

A key novelty of S4Eq lies is the way we train the neural network. We devise an original training process which is dedicated to the challenges of synthesizing equivalence proofs. This training process involves three kinds of training which we now present.

Initial Supervised Training

The typical technique for training sequence-to-sequence models involves providing supervised training samples which provide the target output paired with sample input. S4Eq performs initial training using program pairs for which a rewrite rule sequence is known between the two programs ProgA and ProgB. The details of how to obtain such a supervised dataset are discussed in Section 7.4.1. We refer to this initial supervised dataset of program pairs and their rewrite rules as T_1, and the we refer to the initial model trained with it as M_1.

Incremental training with challenging proofs

After we have an initially trained model, we can use it to attempt new proofs. Because a proposed rewrite sequence between 2 programs can be checked automatically, we can automatically verify the generated outputs and use them to further optimize the model. At this point, to further optimize the model, we don't need to generate program pair inputs with rewrite rule outputs in a supervised manner, but only program pair inputs. In other words, this can be considered as self-supervision since the labeling is automated.

To be effective, the core challenge becomes to effectively select new samples. Hence, we term this technique *self-supervised sample selection*, and since our framework uses this technique on the problem of program equivalence, we name our framework S4Eq.

First, we need a dataset (or a data sample generator) of known equivalent programs which does not include the rewrite rule steps. We call this dataset U, for Unsupervised, as the pairs do not have target rewrite rules associated with them. Relative to the samples in T_1, the looser restrictions on the U dataset allows S4Eq to generalize to program distributions and proof distributions which may be different than those found in T_1.

Figure 7.4 gives a complete overview of our training process for the transformer models. As we show in Section 7.3.4, we start with supervised training and produce model M_1 using program pairs and their expected rules. Now we can use model M_1 to attempt proofs with known equivalent programs in dataset U.

Our key idea is to focus on challenging problems for the current model. Given an intermediate program limit, we can say that the model can 'easily' prove equivalence if a small number of intermediate programs are available at each search step (such as 1 or 2). In such a case, the most likely rewrite rules proposed by the model are checked for equivalence up to a given proof step limit, and if the proof is found, then the model is already well trained for the given problem. Our variable representing the number of intermediate programs searched at each proof step is I and the number of steps to search is N_s steps. To check for 'easy' proofs, we search for a proof with $I = I_e$ where I_e is a small number. To check for 'challenging' proofs, we set $I = I_h$ where I_h is a larger number (10 or more), allowing the proof search to explore more possible rewrite rule paths which are considered less likely to be correct by the current model. We attempt to prove each known-equivalent pair with $I = I_e$ and also $I = I_h$. We define challenging proofs as those found with $I = I_h$ but not $I = I_e$. When proofs are attempted with model M_n, the next training dataset (T_{n+1}) includes samples with proof steps found with $I = I_h$ but not $I = I_e$; thus, the model is more likely to propose similar steps in the future.

In addition to including challenging proofs, if $I = I_h$ found a proof at least 2 steps shorter than the $I = I_e$ proof we include that proof given certain conditions. We set the probability of including such proofs based on length in order to bias the self-supervised samples to solve complex proofs. Also, based on distributions discussed in Section 16, we include the $I = I_h$ proofs when they include rare output tokens such as referring to higher statement numbers, or deeper expressions nodes, or rare rewrite rule names.

After creating our initial training samples in T_1, we train on various model hyperparameter options and use validation test sets to determine the model best suited for continued training. During incremental training, the model size parameters (number of layers, *etc.*) are constant but we train with variations on initial learning rate and decay rate. We then select the best model to continue training using the same validation sets. These validation sets help prevent catastrophic forgetting [80]. But additionally, if a model becomes weaker at solving certain problems then

problems similar to those will get selected in the next iteration of the training, again reducing catastrophic forgetting.

Boosting methods in which models weak in one part of the problem distribution are boosted by other models have been shown to reduce risk of overfitting [65] and we anticipate that our methodology for incremental training based on challenging proofs will similarly resist overfitting.

Incremental training with rare tokens

If some input or output tokens are not yet well understood by a given model M_n, it is because the training datasets so far do not have sufficient samples demonstrating the use of those tokens. To overcome this problem, we propose another kind of incremental training based on rare tokens. The core idea is to oversample those proofs and rewrite rules that involve rare tokens. Even when a full proof search fails, when a rare output token is used legally by a single rewrite rule step in the proof, we keep it in the training dataset. This type of training improves the model representation for the rare token and the situations in which it should be applied and is based on the hindsight experience replay concept [14].

Consider again Figure 7.3 and a case where a required rewrite rule to prove ProgA equal to ProgB was, for example, `stm2 Commute Nlrll`. The node `Nlrll` is an example of a node ID which specifies the 5th level of the AST for statement 2. If there haven't been sufficient training samples with `Nlrll` then the model may not have a good internal representation for when the node should be produced by the final Softmax layer of the transformer model and the proof might fail. However, if, for example, $RewriteRule_{25}$ was `stm2 AssociativeLeft Nlrll` and this was a legal application of `AssociativeLeft` then the pair with $ProgInt_{12}$ and $ProgInt_{22}$ can be proven equal by applying $RewriteRule_{25}$. If `Nlrll` is a rare token, this sample can be included in the next training dataset and this will improve the model's representation for this rare token in order to improve use of this token after incremental training.

7.4 Experimentation

We now describe our experiments to assess `S4Eq`. We start by carefully devising two datasets for training and evaluating our system.

7.4.1 Dataset Generation

We devise two separate datasets with different properties. First, we wish to evaluate our algorithm broadly on actual programs from open source C functions found on GitHub, we call this dataset K. Second, we develop a process to create synthetic program pairs based on applying rewrite rules, we call this dataset R.

Equivalent program pairs from GitHub

We want to have a dataset representative of developer code with straight-line programs matching our grammar. For this, we use an existing dataset of C programs mined from GitHub suitable for machine learning [48].

We process these C functions to find sequences of assign statements that correspond to our straight-line program grammar. We search for C snippets of mathematical computations, with at least 2 assignments, at least one multiply or divide statement, and require at least 1 temporary variable is used in an output variable. To create our abstracted programs (a process similar to function outlining [247]), we collapse complex data structure accesses into an intermediate variable. For example, C code of the form `delta = ca->tcp_cwnd - cwnd ; max_cnt = cwnd / delta ;` will be abstracted to `t1 = i1 - i2 ; o1 = i2 / t1 ;`. Two complete examples of abstracted programs are shown in subfigures 7.1(b) and 7.1(e).

Algorithm 3 provides an overview of the process we use. After finding source GitHub programs and abstracting them, we perform 3 high-level compilation steps on the abstracted C code: common subexpression elimination, strength reduction, and variable reuse. For training sample generation, Encode will transform abstracted C code into the prefix encoding of the AST. Encode will randomly reassign scalar variable IDs to temporary variables with each iteration of the `foreach` loop; so `t1` may be assigned to `s03` for one program and `s25` for another program.

Algorithm 3: GenerateKnownEqual

Input : Tokenized C Functions from GitHub: F
Output: Prefix encoded known equivalent pairs K

```
1  K ← ∅  {Compiler equivalence program pairs}
2  S ←FindSourcePrograms(F)  {Possible programs}
3  foreach s in S do
4  |  s̃ ←Abstraction(s) {Abstracted source code}
5  |  s̃_cse ←CommonSubexpressionElimination(s̃)
6  |  s̃_str ←StrengthReduction(s̃_cse)
7  |  s̃_reuse ←VariableReuse(s̃_str)
8  |  p ←Encode(s̃) {Encode into prefix format}
9  |  p_cse ←Encode(s̃_cse)
10 |  p_str ←Encode(s̃_str)
11 |  p_reuse ←Encode(s̃_reuse)
12 |  p_rules ←Rules(p_reuse) {Probabilistic application of rewrite rules}
13 |  if CheckLimits(p, p_cse, p_str, p_reuse, p_rules) then
14 |  |  K ← K+MixPairs(p, p_cse, p_str, p_reuse, p_rules)
15 |  end
16 end
```

The goal of the random assignment by Encode is to help with generalization. The high-level compilation steps utilize many of the 23 rewrite rules shown in Table 7.1, but in order to ensure all rewrite rules are represented in K, we call the Rules function on p_{reuse} (the encoded program after all compilation steps) which may apply one or more rewrite rules to create p_{rules}. The Rules function is used heavily in Section 16 and is discussed further there. The CheckLimits function ensures that our samples meet the model limits[9].

Using these rules, the dataset K derived from C functions from Github eventually contains 49,664 unique known equivalent program pairs for our experimentation.

Synthetic equivalent program pairs

As we discuss in Section 7.3.4, we need to create an initial training set with broad distribution on the input and output tokens necessary for our problem of proving programs equivalent. We create legal input programs by probabilistically applying production rules from a grammar which

[9]We limit each program to 20 statements, at most 100 AST nodes, at most 30 scalar variables, an expression depth of 5 levels of parenthesis (expression AST depth of 6), and at most 2 outputs.

defines our target program space. This approach allows us to create arbitrarily large amounts of training data.

Algorithm 4: GenerateRewrites

Input : Probabilistic grammar: G, Number of samples desired: n
Output: Prefix encoded equivalent program pairs R

1 $R \leftarrow \emptyset$ {Rewrite rule equivalence program pairs}
2 **while** *samples in R < n* **do**
3 $\quad p_A \leftarrow$GenerateProgA(G) {Probabilistic production rule generation}
4 $\quad p_B \leftarrow$Rules(Rules(Rules(p_A))) {3 passes over p_A with probabilistic application of rewrite rules}
5 $\quad$ **if** *CheckLimits*(p_A, p_B) **then**
6 $\quad\quad R \leftarrow R + (p_A, p_B)$
7 $\quad$ **end**
8 **end**

Prog A (prefix encoding of AST): (a)	**Inputs:** v27, s01
v26 = (-v v27 (h5v v27 v27)) ;	**Outputs:** v08, s26
v27 = (+v (-v 0v 0v) (-v 0v v26)) ;	
s21 = (u4s (is s01)) ;	
s01 = (ns (ns s21)) ;	
v08 === (*v (-s s21 s01) (-v v26 v26)) ;	
s26 === (h4s v26 (nv v27)) ;	

Prog B (prefix encoding of AST): (b)
v26 = (-v v27 (h5v v27 v27)) ;
v08 === 0v ;
v27 = (-v 0v v26) ;
s26 === (h4s v26 (nv v27)) ;

Rewrite Rule Sequence: (c)
stm2 AssociativeLeft N
stm5 Cancel Nr
stm5 AbsorbOp N
stm4 DeleteStm
stm2 NeutralOp Nll
stm3 SwapPrev
stm4 SwapPrev
stm2 DeleteStm
stm3 NeutralOp Nl

Figure 7.5: Equivalence proven between 2 multi-statement programs generated synthetically. The equivalence proof is 9 steps long involving expression and statement rules.

Algorithm 4 shows the synthesis algorithm. Our program grammar defines a program as made up of a series of assign statements which assign scalar and vector variables to complex mathematical expressions. Our program generation process starts by creating assignment statements for the output variable(s). Then a subset of the variables used in the expression may have earlier assign

statements added. This process continues adding assign statements randomly to the beginning of the program.

Variables which are used but never assigned are considered program inputs. For example, in the program "`c = b + a; d = c * a + b;`"; a and b are inputs, d is an output, and c is a temporary variable. Subfigure 7.5(a) shows an example ProgA generated using our algorithm. It includes 6 statements and produces one vector output v08 and one scalar output s26 identified with === tokens.

Rules In order to create training samples with known paths between equivalent programs, after creating a program we randomly apply legal rewrite rules to the start program. For example, Figure 7.5(c) shows the rewrite rules randomly selected which transform ProgA in subfigure (a) into ProgB in subfigure (b).

Synthetic distribution Figure 7.6 diagrams the distribution of samples generated by Algorithm 4 with a plot of the number of AST nodes in ProgA and the number of rewrite rules used to generate ProgB in the sample. When GenerateProgA generates a program with more AST nodes, more rewrite rules are found by invoking of Rules function. We limit the number of AST nodes to 100, as shown in the distribution, but the number of rewrite rule steps is not strictly limited and we have some cases with over 40 steps between ProgA and ProgB (not shown in figure).

Certain rewrite rules, such as `Commute`, tend to be applicable to many nodes in a program AST, while others, such as `FactorLeft`, require rarer patterns in order to be legally applied. We adjust the likelihood of rewrite rules being applied to help balance the likelihood a proof will use any given rewrite rule. All 23 rules shown in Table 7.1 will occur in our synthetic dataset. Because the `Commute` rule can be applied to a large number of operators in our AST, we limit the likelihood it will be applied to about 9% per location with the result that 60.2% of the proofs use it for generating ProgB. Conversely, given our random program generation process, `FactorLeft` and `FactorRight` are not so likely to be applicable, so we bias the application so that about 65% of the AST nodes which would allow these rules have them applied which results in 7.1% and 7.2% of proofs using these rules.

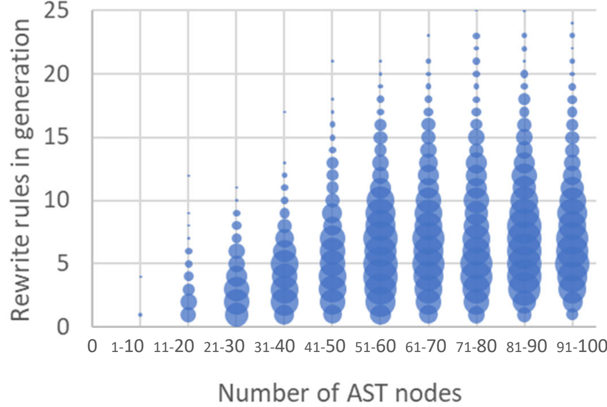


Figure 7.6: Distribution of proof length for synthetic program equivalence dataset R.

For algorithm 4 the probabilistic grammar expansion used in GenerateProgA is tuned so as to create a range of program sizes. We skew the generation to prefer creation of programs with large AST node counts which have either many statements or deep expressions and rely on CheckLimits to ensure programs outside our transformer model ranges are pruned out. For the T_1 initial training dataset, we create 150,000 equivalent program pairs; given that the average pair takes multiple rewrite rules to transform, we create 640,000 rewrite rule steps from these pairs for training the M_1 model.

7.4.2 Experimental setup of incremental training

Recall from Section 7.3.4 that the model initially trained on synthetic programs with target rewrite rules is labeled M_1. We train our initial model M_1 for 100,000 steps, which is 5 epochs of our 640,000 sample T_1 dataset (after dividing by our effective batch size of 32). As per Figure 7.4, M_7 has gone through 6 iterations of incremental training beyond the initial M_1 model. In order to validate partially trained models during training, we create R_v and K_v, which are each 1,000 equivalent program pairs randomly sampled from the R and K datasets. The success of the intermediate models on proving equivalences in R_v and K_v is used to select the model to use in the next step of incremental training.

For self-supervised sample selection, after training M_1 with T_1 we want to create training data which ensures we continue improving performance on the synthetic dataset R but also learn

to solve equivalent programs in K. Referring to Figure 7.4, we create the unsupervised set of program pairs U at each iterative learning step by selecting 60,000 equivalent pairs from R and 40,000 equivalent pairs from K.

In order to attempt to prove equal the program pairs in U, we chose $I_e = 2$ for the "easy" beam width as a beam width of only 1 can fail with a single misstep and we wanted to allow recovery from that case. Due to machine time constraints, we chose $I_h = 20$ for the "hard" beam width and $N_s = 25$ for the maximum number of proof steps to search. From this data, we create T_n for use in training model M_n.

During incremental training, we train 4 model versions (we vary the learning rates) for 50,000 steps with the new T_n training dataset. Similar to early stopping techniques [178], we use our validation datasets to select the best performing model during incremental training. We save the model every 10,000 training steps and select M_n as the model with the highest total proofs found in the 2,000 test samples when R_v and K_v are combined. Our final iteration is selected when both R_v and K_v have improved performance by 1% or less when training the next model. For our experiments, this is M_7.

7.4.3 Research Questions

In this section, we describe our research questions for S4Eq and the protocol methodologies for evaluating them.

Our research questions are:

- RQ1: How effective is S4Eq on the synthetic test dataset?
- RQ2: How useful is incremental learning with self-supervised sample selection in improving S4Eq's model?
- RQ3: To what extent does our model generalize outside the training data?

For RQ1 we create a 10,000 sample test dataset drawn from the synthetic programs R which we call R_t. For RQ2 we create a 10,000 sample test dataset drawn from K which we call K_t.

The samples in the test datasets do not overlap with any samples used for training nor with the validation datasets R_v and K_v. To more broadly understand the system behavior, in both RQ1 and RQ2 we analyze subsets of R_t and K_t based on characteristics of the rewrite sequence which transforms the first program in the sample into the second. Of the 23 rewrite rules shown in Table 7.1, `Rename` is the one which occurs the least in our initial T_1 training data as it is least used when synthetically creating program pairs in R (552 out of 10,000 samples in R_t use it), so we report on the subset of proofs which use `Rename` to observe the system behavior on this group. `Newtmp` is the rule which improves most between M_1 and M_7 in the K_t test set - improving from 545 proofs found by M_1 to 2,277 proofs found by M_7 so we also report based on this rule. The `DistributeLeft` rule is the one most improved on between M_1 and M_7 when attempting proofs in the R_t test set - improving from 796 proofs found by M_1 and 1,771 proofs found by M_7. Proofs that use these 3 rewrite rules will tend to be longer proofs (longer proofs are more likely to use any given rewrite rule), so we also include a rule category which is subtractive. We report on proofs which do not include any statement rules (*i.e.,* they don't use `SwapPrev`, `Usevar`, `Inline`, `Newtmp`, `DeleteStm`, or `Rename`), which also allows for comparisons with our prior work [123]. Because our hindsight methodology focuses on NodeIDs at depth 5 of the AST (and these are individually our least common tokens), we report on proofs which use such an identifier. Our last 2 proof subsets are based on the length of the rewrite rule sequence used to transform ProgA to ProgB in the sample - we report on proofs of 1-10 steps as a group as well as proofs of 11 steps or more.

We perform all our experiments on systems with 12 Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz CPU cores and NVIDIA GeForce GTX 1060 6GB for GPU compute. Our model is based on the transformer model implemented in OpenNMT-py. Proof searches for incremental model training may use up to 30 systems in parallel.

Methodology for RQ1

To evaluate the effectiveness of `S4Eq` on finding rewrite rule sequences for the program pairs selected from the synthetic program dataset R, we analyze the distribution of programs and rewrite

rules included in R and present data on the success rate for proving various subsets of the test dataset R_t. Our goal is to understand if our system is unable to perform well on any of our selected subsets of R_t (*i.e.,* solve at less than a 90% success rate).

Methodology for RQ2

Our incremental training approach is able to learn to solve proofs for which the supervised proof sequence is not provided during sample generation. To determine how well self-supervised sample selection improves the quality of the S4Eq model, we study the 10,000 sample K_t dataset drawn from K alongside our rewrite rule test dataset R_t.

In addition to our golden model M_7, which is trained using self-supervised sample selection, we create a model for comparison called Q which is trained in a more traditional way. Q is trained for the same number of training steps as M_7 but it continues training on the T_1 dataset from the M_1 model in a supervised manner. To align with our M_n training protocol, we train multiple models from M_1 with varying learning rates and select the strongest model using the R_v and K_v datasets. We will evaluate the ability of our models to prove the known equivalent program pairs in K_v in order to determine if self-supervised sample selection adds value to our model.

Methodology for RQ3

We explore the ability of S4Eq to generalize using 2 different methods. The first method focuses on using the sample generation algorithm 4 for R with different hyperparameters to create programs outside of the training distribution. The second method relates to actual use cases for program equivalence and focuses on finding equivalent programs from within different GitHub C functions. This search could be viewed as a demonstration of the system to find semantic equivalence for a variety of uses, such as identifying opportunities for library calls [163], or grouping student submissions into semantically equal groups for grading [57], *etc.*

Regarding the first method, we use algorithm 4 to create programs with exactly 3 outputs and 101 to 120 AST nodes. Recall that in training we limit R and K to include only 1 or 2 output

programs with up to 100 AST nodes. We test our golden model on this dataset to determine if it has overfit the training data or if can solve problems from outside the training distribution.

Regarding the second method, the FindSourcePrograms routine in algorithm 3 finds 13,215 unique multi-statement program blocks from GitHub. As we are interested in comparing the more complex programs in this set, we select only programs with at least 30 AST nodes resulting in a set of 4,600 programs. We then group the programs and test all pairs of programs which have the same number of inputs, outputs, and functions. This results in 152,874 unique pairs of programs to check for equivalence. Since we search in both directions for each pair, our full GitHub test set G has 305,748 program pairs to attempt equivalence proofs on.

7.4.4 Experimental Results

RQ1: Effectiveness on Synthetic Test Dataset

In this research question, we aim to show the breadth of program pairs and rewrite rule sequences in our synthetic dataset and to demonstrate the effectiveness of our M_7 golden model. Table 7.2 details the performance of a variety of subsets of our 10,000 pair synthetic test dataset R_t. Each cell in the table gives the passing rate and sample counts for each subset. Here the passing rate means the percentage of program pairs proven equivalent with an appropriate sequence of rewrite rules (recall that all program pairs in R are equivalent by construction). The first data row gives the effectiveness over the whole dataset. The other rows in the table provide data on subsets of the sample based on the rewrite rules used to generate the ProgB in the sample given the synthetically generated ProgB. A full description of the rows is given in Section 7.4.3. Orthogonal to the rewrite rules used to generate ProgB, the columns explore subsets of interest for the original ProgA in the sample. The first column provides data for all samples which conform to the rewrite rule subset given by the rows. The 2nd column shows results for the 6,390 samples that used at least 3 functions in ProgA. The 3rd column shows results for the 3,340 samples where ProgA starts with an expression of depth 4-6 (note that a NodeID at depth 5 is used in 533 samples given any ProgA but only 506 samples when ProgA started with a deep expression; this is due to some rewrite

Table 7.2: Success rate for subsets of test dataset R_t. Each entry includes tuned passing percentage and sample count within R_t.

Rewrite Rules	ALL	Functions 3 or more	Maximum Expression Depth 4-6	Nodes 30-100
Whole dataset	98%(10000)	98%(6390)	97%(3340)	98%(9470)
Rename	97%(552)	97%(354)	95%(74)	97%(546)
Newtmp	94%(844)	94%(591)	93%(477)	94%(828)
DistributeLeft	96%(2273)	95%(1650)	95%(1524)	96%(2214)
No statement rules	99%(4923)	98%(3208)	98%(2312)	99%(4524)
NodeID at depth 5	92%(533)	92%(429)	92%(506)	92%(522)
Rewrite steps 1-10	99%(8253)	99%(5066)	99%(2269)	99%(7725)
Rewrite steps 11+	94%(1747)	93%(1324)	92%(1071)	94%(1745)

rules, such as `MultOne`, adding depth to the original ProgA). The final column provides data on the larger programs from the dataset (and the size corresponds to the program sizes considered in RQ3).

In the upper left data cell in Table 7.2, we find that of the 10,000 samples in R_t, M_7 was able to find a rewrite rule sequence from ProgA to ProgB for 98% of them, which is arguably very high. We see that some subsets of R_t performed better than this overall result, such as cases where ProgA to ProgB proofs contain 1-10 rewrite rule steps (99% effectiveness). Other subsets performed worse, such as proving samples where a depth 5 NodeID was use to create ProgB. In general we see that program pairs generated with shorter rewrite rule sequences are more easily proven equal than longer ones, corresponding to the intuition that shorter proofs are easier to find. In the table, there are 5 subsets tied for the poorest result of 92% success: all 4 subsets with NodeID at depth 5, and cases where ProgA has a deep expression and generating ProgB used over 10 rewrite rule steps. These results show that `S4Eq` is challenged by deeply nested expressions and long proofs, however it still achieves over 90% success.

Table 7.2 also shows that `S4Eq` well handles rare tokens. The output token least represented in our 640,000 samples in T_1 is `Nrlrr`, used in only 278 samples. `Nrlrr` is one of 16 tokens used to indicate that a rule should be applied to a depth 5 node; all 15 of the other such tokens make up the 15 other least commonly used tokens in the T_1 output group. During self-supervised training, we compensate for this rarity to increase the number of such samples in T_{2-7}. As we show with

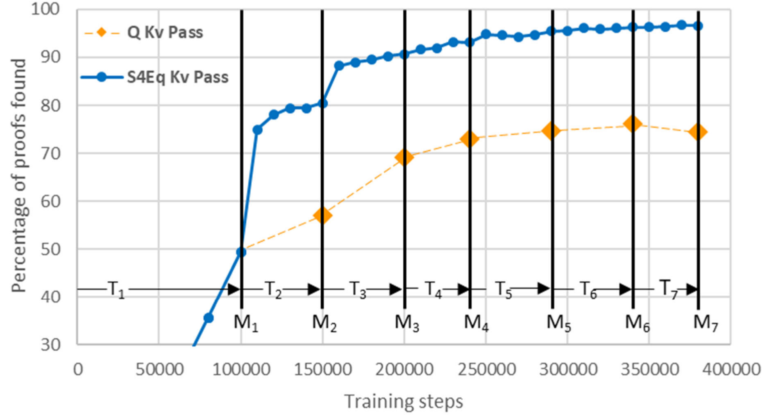


Figure 7.7: Performance metrics on K_v through training process. M_7 is the final golden model used for experiments, Q is purely supervised training on the T_1 training data.

the "NodeID at depth 5" row in Table 7.2, our fully trained model has learned to use these tokens effectively.

Our synthetic ProgA algorithm given in algorithm 4 is designed to ensure a relatively balanced use of input tokens in T_1 for training. This works well, because the least used input token is one of the 5 tokens representing functions which receive 2 scalars and produce a vector output (`f4v`) with 43,458 of the 640,000 samples using it. Table 7.2 shows that those programs using at least 3 functions are proven well by S4Eq.

> Answer to RQ1: S4Eq is effective on the R dataset, achieving over 90% successful proofs on all subsets analyzed. S4Eq handles well both rare input tokens used in the programs to be analyzed (such as function names) and rare output tokens (such as depth 5 NodeIDs).

RQ2: Incremental Training Benefit

In order to show how the an initial model improves with self-supervised sample selection, we compare S4Eq training up to model M_7 against a model trained only on the supervised training set T_1. For this comparison, we study the K_v and K_t datasets derived from GitHub program compilation steps since these include program pairs which can be proven equal using our rewrite rules. In Figure 7.7, the orange squares indicate the percentage of proofs found on the K_v dataset as the model trains only in a supervised manner on T_1 (the initial supervised training set of synthetic programs with known rewrite rules). The blue circles represent training with self-supervised sample

selection. With traditional training, we see the best performance of the Q model on K_v occurs at 340,000 total training steps (S4Eq had trained to M_6 after that many steps). This result is significantly below the performance S4Eq achieves. The significant improvement of S4Eq on the K_v dataset demonstrates the benefits on our incremental training procedure.

After training from 340,000 steps to 380,000 steps, Q decreased performance on the GitHub compiled test dataset. This indicates that Q was starting to overfit on the distribution for R and the latest learnings were not as applicable to the K problems. On the contrary, since self-supervised sample selection generates new training samples, S4Eq is able to avoid overfitting and continues to improve the overall model's ability to find proofs on the K_v dataset.

Also, we note that Q was able to slightly outperform M_7 on the R_v dataset (98.96% pass versus 98.44%, not shown in the figure) which shows that the T_1 dataset had sufficient samples to train for the problem distribution in R_v.

Table 7.3: Performance of M_1, Q, and M_7 models on the 10,000 K_t test pairs based on GitHub code. The self-selected samples in T_{2-7} are biased to areas where M_1 is weak, ultimately allowing M_7 to outperform Q in all categories.

Sample or Proof Sequence Used	T_1 Samples	T_{2-7} Samples	M_1 Proved	Q Proved	M_7 Proved
Any rewrite rule	640,000	714,332	4,866	7,531	9,688
Rename	5,267	54,925	2,591	4,791	6,315
Newtmp	7,706	33,053	545	810	2,277
DistributeLeft	30,029	24,243	526	649	774
No statement rules	506,217	479,637	968	1,109	1,111
NodeID at depth 5	5,969	35,842	18	91	323
Rewrite steps 1-10	367,976	336,344	4,690	7,329	8,997
Rewrite steps 11+	272,024	377,988	176	202	691

Table 7.3 shows the benefit of training with samples that are sampled from challenging proofs. The first row of the table summarizes the total number of samples available in T_1 used to train M_1 and Q, the total number of samples in T_{2-7} used to train incrementally up to M_7, and the total proofs in the 10,000 sample GitHub test dataset K_t found by the models M_1, Q and M_7. For example, in the upper right corner we show that there were 9,688 (out of 10,000) GitHub program pairs for which proofs were found by the best model M_7. Different subsets of these 9,688 proofs are

shown in later rows regarding the rewrite rules needed to prove the GitHub program compilation cases. The 2nd-4th rows introduce the mechanism through which self-supervised sample selection most benefits the model. For the `Rename` rewrite rule, we see that, of the 9,688 samples proven equal by M_7, 6,315 of them used `Rename` for at least one step, but M_1 only used `Rename` in 2,591 of its proofs (less than half of the M_7 usage). We can see that from the T_1 and T_{2-7} columns that self-supervised selection recognized that M_1 was weak in this area (implying searches with I_e rarely found the proof) but was able to augment the training data with more samples using the `Rename` rule (implying searches with I_h were able to provide example successes). We can see that model Q, which continued training only with T_1 samples, did not learn this rule as well as M_7. We see a similar effect for the `Newtmp` rule; 2,277 proofs used this rule for M_7, but only 545 used it for M_1. We see that T_{2-7} included more of these samples to help improve the model. As a counterexample, we see that `DistributeLeft` is used in 774 successful proofs of the high-performing M_7 model, and in 526 of the proofs for the initial M_1 model. Given that M_7 was able to solve 97% of the problems in K_t, M_1 had already solved over two thirds of these cases which implies T_{2-7} only needed to contain certain cases of `DistributeLeft` where the model was still challenged in order to improve the model. In this way, self-supervised sample selection provides new training samples to improve the areas in which the model is weak.

Table 7.3 also includes 'NodeID at depth 5', which indicates a rewrite rule was used on an AST node at depth 5 of an expression. Since we use NodeIDs to identify rule application positions, there are $2^{5-1} = 16$ different IDs needed to correctly locate a given instance. Fewer than 1% of T_1 samples (5969) include a depth 5 node, which correlates with poor base model performance (18 cases proven). However, part of our incremental training data are the hindsight steps which use a depth 5 node. Consequently, the performance increases to 323 found proofs, meaning that the self-supervised training greatly improves the results.

Self-supervised sample selection also improves performance on long proofs by increasing the number of samples from such proofs. The last 2 rows of Table 7.3 report on the number of rewrite rule steps required by the 3 different models to prove pairs equivalent. The last column shows

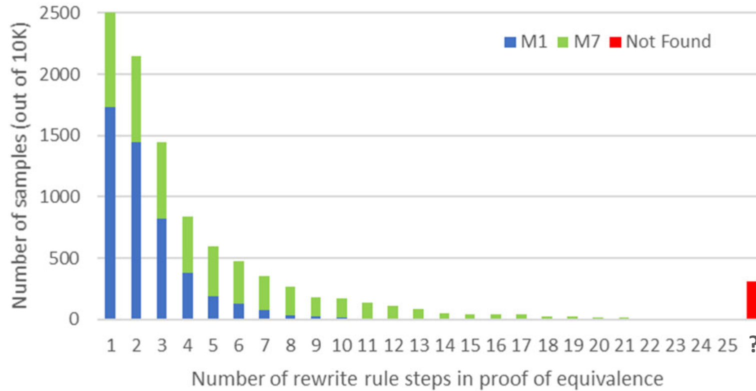


Figure 7.8: Proofs found on test set K_t. The length charted is the proof length found with the tuned model M_7, with the unproven cases shown as unknown step count.

us that M_7 successfully used long proofs for 691 samples, while M_1 only found 176 long proofs. Indeed, we also see that T_{2-7} contained more samples from long proofs than T_1 to help the model improve this category, explaining this difference.

The details on long proofs shown with Figure 7.8 demonstrate another benefit of self-supervised sample selection. Recall that M_1 can only prove 176 samples equivalent with a proof over 10 steps. Of the 691 samples that M_7 solves with over 10 steps, M_1 only solves 23 of them. The other 153 samples for which M_1 found a long proof are still proven by M_7, but in 10 steps or fewer. Figure 7.8 shows the benefit of self-supervised sample selection on the performance of proofs of increasing length in the K_t dataset. Here we see that proofs which M_7 proved with only 1-3 steps were solved with over 50% success by M_1; yet proofs that M_7 found with over 10 steps were rarely solved by M_1.

We now assess the ability of self-supervised sample selection to avoid catastrophic forgetting by analyzing the samples which M_1, Q, and M_7 are able to prove. Of the 4,866 samples that M_1 is able to prove equivalent, *ALL* of them are included in the 9,688 samples that M_7 proves after training based on self-supervised sample selection. However, only 4,774 of them are included in the 7,531 which Q proves - Q 'forgot' how to prove 92 samples that the M_1 model it trained from had proven. *This shows clearly the benefit of self-supervised sample selection for avoiding catastrophic forgetting.*

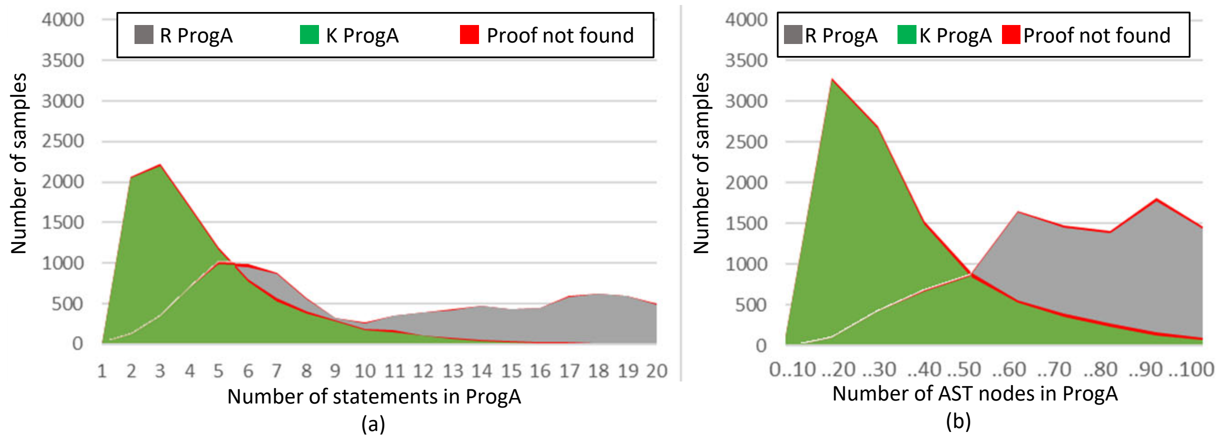


Figure 7.9: Proofs found on test sets R_t and K_t plotted for varying statement and AST node counts. In all cases the performance of S4Eq closely tracks the dataset distribution.

> Answer to RQ2: Compared to supervised training, using self-supervised sample selection improved performance on our target dataset from 75% success (supervised training) to 97% success (S4Eq's novel self-supervised training). Self-supervised sample selection does focus on areas where the model needed the most improvement by selecting the most interesting new training samples.

RQ3: Generalization Ability

A concern with machine learning is that it may learn to perform well within the samples on which it is trained but not generalize well to unseen problems that humans would consider related. We asses this risk by presenting data on how well S4Eq has generalized beyond its training distribution.

Let us first discuss the differences in the distribution of programs and rewrite rules between R and K. Figure 7.9 shows the histograms of the number of assign staments and tokens in ProgA for R and K. We see the human-written code in K shown as the green distribution tends to have many samples with fewer than 5 statements and fewer than 50 tokens, while the synthetic code shown as the grey distribution was designed to create more complex programs and hence has ProgAs with more statements and more AST nodes. This clearly shows a different distribution. The red areas show the proofs which are not found. The thinness of the red area showing that there is no obvious

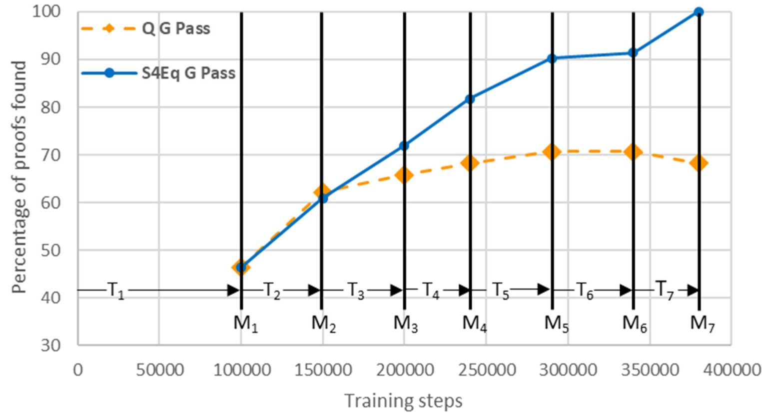


Figure 7.10: M_7 searched 305,748 program pairs based on GitHub functions and found 82 equivalent pairs. This chart shows the percentage of these 82 cases found as S4Eq trains and as Q trains (Q trains only on T_1 data).

weakness on any area of the ProgA distribution. The rewrite rules needed to prove equivalence also varies between R and K: 49% of the proofs for samples in R_t are solved without rewrite rules related to statements while less than 12% of proofs on the K_t dataset could be solved without these rules (compare the "No statement rules" rows from tables 7.2 and 7.3). Taken together, these data show that S4Eq has generalized well over 2 different datasets.

In order to show S4Eq can generalize outside its training domain, we used algorithm 4 with CheckLimits and GenerateProgA adjusted such that we created test samples with 101-120 AST nodes and 3 outputs, which is outside the initial training distribution. Recall that on R (with up to 100 AST nodes and 1 or 2 outputs), S4Eq achieves 98% success on the R_t test set. Now, on programs with 101-120 AST nodes and 3 outputs it is still able to prove 60% of the program pairs equivalent, showing that S4Eq can generalize to larger examples.

For our final evidence of generalization, we present equivalences found between human-written programs on GitHub. Of the 305,748 program pairs in G, we found 82 provable cases of equivalence. Figure 7.1 shows an example of proven equivalence of GitHub samples. Figure 7.10 illustrates the generalization of S4Eq to this problem as training progresses by showing the percentage of the 82 proofs found by M_1 through M_7. The figure also shows the performance of Q (which only trains on T_1 sampled from R) on the same 82 proofs. We see that after 150,000 training steps M_2 and Q perform about equally well but ultimately Q plateaus with just over 70% of

Table 7.4: S4Eq ablation study. With the exception of the 'Golden model', results are shown with only initial base training of the model. The synthetic datasets for the last 2 rows were regenerated based on the modified language grammar.

Model description	Percent proofs found R_v	K_v
M_7 (Golden model)	98%	97%
M_1 (Golden model before incremental training)	66%	50%
Faster learning rate (0.0002 vs 0.0001 in M_1)	2%	1%
Slower learning rate (0.00005 vs 0.0001 in M_1)	49%	35%
Fewer transformer layers (6 vs 8 in M_1)	56%	23%
Limit language to scalars (vs scalars+vectors in M_1)	63%*	40%
Linear algebra expressions only (and 50 AST node limit vs 100 in M_1)	99%*	9%

the proofs found at 340,000 steps. Like Figure 7.7 on K_t data, Q seems to start overfitting on the T_1 data after 340,000 steps and does not generalize well to different problem areas. However, we see that S4Eq continued to improve on the G test set from M_6 to M_7 showing it has generalized to the problem of finding equivalence between human-written programs well.

> Answer to RQ3: S4Eq is able to generalize. S4Eq is able to solve 60% of synthetic programs that are outside of any training data it was presented with. Furthermore, it is able to progressively generalize to find up to 82 proofs of equivalence in the golden human-written samples from GitHub.

7.4.5 Ablation Study

Table 7.4 illustrates some of the model variations we explored. Except for the golden model, all of the results are reported only after initial training using the appropriate synthetic dataset. All of the models are also evaluated on the GitHub dataset.

The faster learning rate shown in Table 7.4 has poor results presumably due to known risks of divergence with large learning rates. We studied this result further by attempting to do an iteration with self-supervised sample selection using the poor proof success. As discussed in Section 7.4.4, self-supervised sample selection requires that sufficient examples of the input and output tokens are provided for successful incremental training. For example, using the M_1 that led to our golden model, the T_2 dataset had 126,630 samples for use during training, 11,737 of which used the

Rename rule. For the high learning rate model, since even the I=20 search had poor success, its T_2 only had 1,482 samples in it and NONE included a successful use of Rename. Hence, unlike our golden model, incremental training on the high learning rate model did not significantly improve performance on R_v nor K_v. We see also that a slower learning rate produced a slightly worse result than our M_1 result, hence we selected M_1 for our S4Eq training base.

We tested our transformer model with different numbers of attention layers, as well as different numbers of attention heads and hidden feature sizes. We show a typical result of these searches with only 6 attention layers. This parameter had a small loss for the R_v success, but did not generalize as well to the K_v dataset and hence it was not pursued further.

The last 2 rows of Table 7.4 explore training models on alternate language grammars. For 2 these cases only the synthetic proofs found column indicates the number of proofs found for the synthetic dataset which aligns with the model description. Perhaps surprisingly we see that a language with only scalar variables and operators performs worse than M_1 (which has both scalars and vectors) on both its own synthetic validation set as well as on K_v. One possible explanation for the weakness on both the synthetic validation set with only scalars and the compiled GitHub programs in K_v, which only use scalars, is that the existence of vectors in the training set helps the attention layers in the model generalize better to complex uses of the rewrite rules. The final row of the ablation study shows results for a language which has only a single statement with up to 50 AST nodes, but matrixes, vectors, and scalars are supported. We see strong success on a synthetic dataset with the same features, but this model does poorly on the GitHub compiled equivalence checks. When compared with prior work on a similar dataset [123] this row demonstrates that *for our problem of program equivalence the transformer model outperforms a well tuned graph-to-sequence model which itself was found to outperform a bidirectional RNN sequence-to-sequence model.*

This ablation study does not include the full breadth of models and language representations we explored. For our transformer interface, we tested input variations (such as using infix instead of prefix or including statement numbers in the input) and output variations (such as different

Table 7.5: Execution time statistics on S4Eq tasks. Machine hours are approximate as the systems are pre-emptable by students.

Task Description	Approx Machine Hours
Train 16 M_1 candidates with varying hyperparameters	375
Search for easy and hard proofs on total of 600,000 equivalent pairs with models M_1 through M_6	540
Train 4 candidates each for models M_2 through M_7 with varying learning rates and learning rate decays	270
Total to create M_7 with self-supervised sample selection	1185
Total to create Q with only supervised samples	645
Search 305,748 mostly unequal abstractions of human-written code in G using M_7	775

rewrite rule syntax including left/right path listing to identify expression nodes and also outputting the full rewrite sequence as a single long output). We also tested sequence-to-sequence RNN and graph-to-sequence models on early versions of our language [123], and we explored transformer model parameters guided by OpenAI's work on neural language model scaling [111]. In the end the parameters for our golden model performed best overall in these studies.

7.4.6 Execution time

Table 7.5 shows the machine hours needed for the key steps related to M_7 training and usage. For some steps we use up to 30 machines in parallel as indicated in Section 7.4.3. This table shows that while our proof search for self-supervised sample selection takes time, it does not double the model creation time for M_7 relative to a model trained with traditional hyperparameter searches such as Q. Also of note is that almost all of the pairs in G are not equal, and the average search time per pair with a 50-step limit takes about 9 seconds. Meanwhile, the 600,000 equivalent pairs used for self-supervised sample selection are each attempted twice in about 3 seconds per pair because once the proof is found the search terminates.

7.5 Related Work

7.5.1 Static Program Equivalence

Algorithms for proving program equivalence restricted to specific classes of programs have been developed [222, 6, 25, 103]. These approaches are typically restricted to proving the equivalence of different schedules of operations, possibly via abstract interpretation [195, 53] or even dynamically [24]. Popular techniques also involve symbolic execution to compare program behavior [163, 20]. The problem of program equivalence we target may be solved by other brute-force (or heuristical) approaches, where a problem is solved by pathfinding. This includes theorem provers [31, 175], which handle inference of axiomatic proofs. Program rewrite systems have been heavily investigated, [64, 206, 55, 224, 168, 108, 154]. While semantics-preserving rewrite systems for program equivalence have been studied [223, 145, 186, 235], our contribution recognizes this compositional formalism is well suited to deep learning sequence generator systems. The merits of stochastic search to accelerate such systems has been demonstrated [165, 93, 78]. The novelty of our work is to develop carefully crafted sequence generator neural networks to automatically learn an efficient pathfinding heuristic for this problem.

EqBench [19] proposes a test suite of 147 pairs of equivalent C/Java programs. As they include if-conditionals, it is not immediately usable with S4Eq. In contrast, we mine and build tens of thousands of equivalent program pairs, using a richer set of rewrite rules for expressions and functions. Our complete dataset, including the samples extracted from GitHub, is publicly available as a program equivalence test suite [121], providing a rich suite complementing EqBench's.

7.5.2 Incremental and Transfer Learning

For incremental learning in S4Eq, we use an "instance incremental scenario" in that for our problem we keep the output vocabulary constant while creating new data for incremental learning model updates [147]. Ye *et al.* discuss using an output verification process (in their case compilation and test for program repair) to adjust the loss function in later training iterations [244]; our

approach is related in that we test outputs but instead of adjusting the training loss we create new training samples which helps to generalize the model to a different problem domain.

To the best of our knowledge, there are only a few works that use transfer learning in the software engineering domain, and none of them use it for generating equivalence proofs. Recently, Ding has done a comprehensive study on applying transfer learning on different software engineering problems [66], such as code documentation generation and source code summarization. He found that transfer learning improves performance on all problems, especially when the dataset is tiny and could be easily overfitted. In our work, we deploy transfer learning on program equivalence proofs and show that it also improves generalization.

Huang, Zhou, and Chin used transfer learning to avoid the problem of having a small dataset for the error type classification task [99]. They trained a Transformer model on the small dataset and achieved 7.1% accuracy. When training first on a bigger source dataset and tuning afterward on the small dataset, they reached 69.1% accuracy. However, they do not develop self-supervised sample selection, and implement a limited analysis of transfer learning. In our work, we develop a form of transfer learning for program equivalence, and carefully analyze its merits and limitations, reaching 97% accuracy on our dataset.

7.5.3 Symbolic Mathematics Using Machine Learning

Hussein *et al.* [69] develop a system which learns to apply a set of inequality axioms and derived lemmas using a reinforcement learning model with a feed-forward neural network. A challenge of using reinforcement learning is the determination of a feasible reward function from the environment and the resulting training time. HOList [23, 171] is a system that can interact with a large rule set to score tactics in the search for a proof but faces compute time limitations when training a reinforcement learning network for theorem proving. Unlike their model, we output both the tactic (rewrite rule) as well as the location to apply it (eliminating the need to score various premises individually). Similar to recent work on using transformers to propose actions [45], our network has learned the rewrite rules (actions on a program) which are most likely to transform the

program into the target program. Our work aims at laying the foundation for program equivalence proofs by studying a language subset that includes multiple computation statements and maintains a high accuracy. Our approach to synthetic data modeling is similar to Lample, *et al.* [130], who randomly create representative symbolic equations to develop a deep learning sequence-to-sequence transformer model which can perform symbolic integration. They note that their system requires an external framework to guarantee validity. Similarly, work by Mali, *et al.* [153] reasons on mathematical problems but produces outputs which are not guaranteed correct. Our system outputs a verifiable reasoning that is straightforward to check for correctness, guaranteeing no false positive and the correct handling of all true negatives.

A Deep Reinforcement Learning Approach to First-Order Logic Theorem Proving [61] provides the theorem prover the allowed axioms as input to the model. In contrast, we produce the rewrite rules as a sequence. As it is a reinforcement learning model, they create training iterations as proof rewards improve with better models, while our approach creates incremental samples specifically chosen through beam search to improve output by providing a correct output for a proof the model currently is challenged by.

7.5.4 Program Analysis using Machine Learning

Numerous prior works have employed (deep) machine learning for program analysis [9, 12, 216, 128, 185, 28]. PHOG [34] presents a probabilistic grammar to predict node types in an AST. Program repair approaches, [216, 48] are deployed to automatically repair bugs in a program. Wang *et al.* [227] learn to extract the rules for Tomita grammars [214] with recurrent neural networks. The learned network weights are processed to create a verifiable deterministic finite automata representation of the learned grammar. This work demonstrates that deterministic grammars can be learned with neural networks, which we rely on. Recent work by Rabin *et al.* [183] shows that neural networks (in their case GGNNs) learn more general representations of semantically equivalent programs than code2vec [12], which creates code representations using AST paths. Bui, *et al.* [38] show that using semantics preserving transformation can improve machine

learning on code, and continue the work with a study on using self-supervised learning to create similar embeddings for semantically equivalent code [39]. We use beam search to identify model weaknesses and target learning to generate the transformations that prove semantic equivalence.

7.6 Conclusion

In this work, we presented S4Eq, the first transformer neural network system to generate verifiable proofs of equivalence for a class of symbolic programs. Evaluated on a rich language for statements with scalars and vectors, this system produces correct proofs of up to 49 rewrite rules in length in our synthetic test dataset. We contribute a dataset created from GitHub samples usable for equivalence evaluation. We develop self-supervised sample selection to incrementally improve our model performance on the samples. We produce correct proofs for 97% of our cases generated by applying compiler steps to these GitHub samples. We also analyze unique GitHub programs and find 82 proofs of equivalence between them. We believe the performance of our approach comes in part from using transformers for what they aim to excel at: learning efficient representations to analyze patterns in syntactic input; and the observation that program equivalence can be cast as a path-based solution that is efficiently found by such networks.

Chapter 8

Conclusion

Deep learning has been expanding into diverse fields as hardware advances and novel algorithms allow approaches to succeed in new areas [9, 161, 211, 140]. For the problems of program repair and program equivalence, in this thesis we demonstrate how machine learning techniques that have roots in natural language processing can be successfully applied to computer languages. Chapter 4 shows how historic techniques for searching for program repairs using transformation schemas can be mimicked and improved on by training a network to "translate" buggy code to fixed code, resulting in similar transformations being learned. As machine learning uses probabilistic sequence generation, it is capable of mimicking probabilistic models of language grammars.

And yet, generative models using machine learning continue to be imprecise [111, 133]. Methods to automatically evaluate machine learning outputs for use in settings with a low error tolerance expand the range of problems on which machine learning can be applied. We demonstrate in Section 4.8 how to automate test set execution to verify program patches generated with machine learning. We show in Chapters 6 and 7 how formally verifying generated proof sequences for proving programs equivalent can yield systems with a guarantee of no false positives. This thesis yields valuable results which allow machine learning to be applied to new problem domains in the field of computer aided programming.

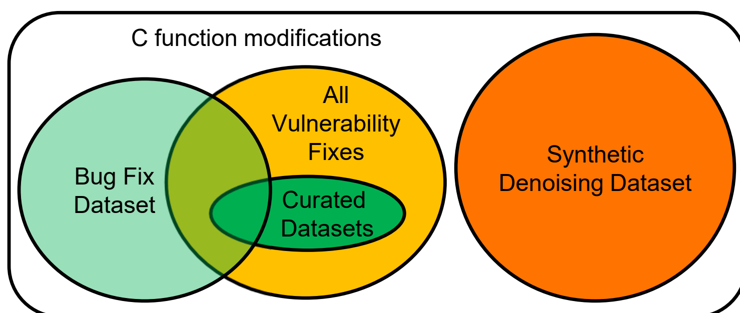


Figure 8.1: For the vulnerability repair task addressed by VRepair, pretraining with a denoising task has some benefit to teaching the model about C function modifications, but using transfer learning from a bug fix dataset trains the model on a more similar task.

8.1 Contributions

Data representation In this thesis we have contributed dataset organization and presentation techniques which allow computer programming problems to be mapped into input and output formats on which we show how to apply machine learning. In Chapter 4 we present the SEQUENCER system [48] which combines useful program information with the *abstract buggy context* which can be used by a *token copy mechanism* when attempting program repairs for buggy Java programs. In Chapter 5 we present the VREPAIR system [47] which can represent multi-line fixes by using *token context diff* to identify where patches should be applied to repair security vulnerabilities in C functions. In Chapter 6 we present the PE-GRAPH2AXIOM system [123] showing both how to map a linear algebra expression equivalence problem into an AST for use with a graph neural network as well as how to represent transformations on the expression using rewrite rules which can be learned by a neural network. And in Chapter 7 we present S4Eq [124] showing how full straight-line programs can be represented to a transformer model and how such a model can generate rewrite rules on program statements.

Transfer learning benefit Figure 8.1 is a Venn diagram showing the grouping of datasets used in our vulnerability repair work in Chapter 5. In that work, we created VREPAIR which is capable of creating some of the fixes in the "All Vulnerability Fixes" set. Because the size of curated vulnerability data on which to train a neural network was small, we tested pretraining this model with a synthetic denoising dataset as well as a bug fix dataset. As the diagram indicates, the synthetic dataset can provide useful examples of modifying C functions but does not overlap with our target problem space. Our technique for *transfer learning from bug fix to security vulnerability fix* was a more successful way to train the model.

Self-supervised sample selection Figure 8.2 is a Venn diagram showing the grouping of datasets used in our program equivalence work in Chapter 7. In that work, we created S4Eq which is capable of proving some equivalent program pairs equal. We studied 3 subsets of equivalent programs: 1) Synthetically generated programs can be used to provide a relatively smooth distribution on

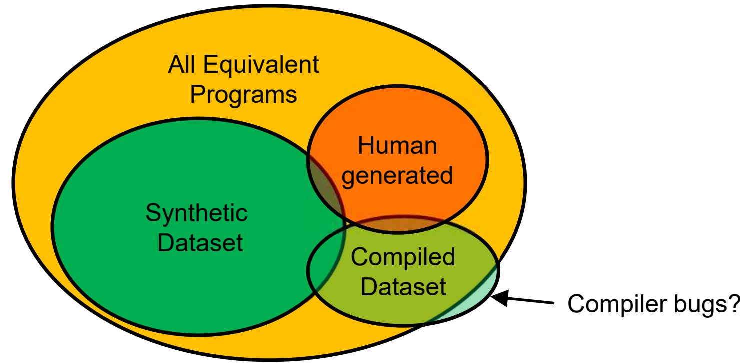


Figure 8.2: For the program equivalence proof task addressed by S4Eq, synthetic and compiled training cases improve the ability of the model to find human-generated cases of equivalence. Some cases of proof failure can indicate true non-equivalence, for example, if there is a bug in the compiler.

tokens and rewrite rules for model training 2) Compiler steps can be used to create equivalent programs that are related to common optimizations 3) Human written code can provide examples of alternate human implementations for equivalent programs. Because we can automatically create examples for the compiled and synthetic datasets, and because we can verify proof correctness, we developed *self-supervised sample selection* to improve model coverage and learning on these datasets. In addition to applications for program equivalence itself, our iterative learning technique provides an opportunity to confirm that the training data is indeed equivalent. We found during development that errors in the compiler optimizations could not be proven equivalent and hence, an occasional review of the failures which occurred with a well tuned model may show that some input programs which could not be proven equal are indeed not equal, as indicated in Figure 8.2. Ultimately, learning with self-supervised sample selection created a model which was applicable to proving human-generated programs equal.

8.2 Avenues for Future Research

This work opens up several promising avenues for continued research and we now summarize them. Most of these areas relate to the field of computer aided programming; but there are also general avenues of research in machine learning opened up by the contributions of this dissertation.

Machine learning for automated program repair The SEQUENCER work presented in Chapter 4 can immediately be extended based on other work in this thesis. While SEQUENCER studied a sequence-to-sequence model, we have seen that a transformer model will likely work better for such code transformations in Chapter 5. Also in Chapter 5 we developed and tested *token context diff* which could be used in any system that makes program modifications, including ones attempting general program repair. Finally, we have shown some benefit of using a denoising task to pretrain a program modification model. All of these techniques would likely improve results for program repair.

Machine learning for automated vulnerability repair We have seen that training on a denoising task is not as valuable as training first on a bug fixing task for developing a vulnerability fix model. However, it may be useful to do both - first pretrain with denoising, then train with bug fixing, and finally tune the model with a small number of actual vulnerability fixes. We also note that our bug fix and vulnerability datasets could be useful for training a vulnerability localization system.

Machine learning for program equivalence The success rate of 97% for our current model presented in Chapter 7 is very high for the field of machine learning. With more examples and more compute time we would expect that a richer language could be used and programs could still be proven equivalent. The ability to prove complex loop transformations equivalent could have performance benefits to compiler development.

Machine learning for NP-hard problems using self-supervised sample selection The general technique for *self-supervised sample selection* presented in Chapter 7 could be applied to any problem in which random problems can be generated and a correct answer can be easily verified. A variety of NP-hard problems fit in this category and this training technique could be used to improve machine learning results on such problems.

Transformer models for interfacing symbolic reasoning with neural networks As we have shown in Chapters 5 and 7, transformer models can successfully provide an interface between symbolic data (such as programs) and a stochastic neural network. Problems of symbolic reasoning on the world are gaining visibility in the artificial intelligence community and the ability of the multi-layer attention network used by Transformers to find connections between varied data will certainly continue to be explored.

Bibliography

[1] Scott Aaronson. "BQP and the Polynomial Hierarchy". In: *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*. STOC '10. Cambridge, Massachusetts, USA: Association for Computing Machinery, 2010, pp. 141–150. ISBN: 9781450300506.

[2] Oliver Adams et al. "Cross-lingual word embeddings for low-resource language modeling". In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*. 2017, pp. 937–947.

[3] Wasi Ahmad et al. "A Transformer-based Approach for Source Code Summarization". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, 2020, pp. 4998–5007.

[4] Wasi Uddin Ahmad et al. "Unified Pre-training for Program Understanding and Generation". In: *arXiv preprint arXiv:2103.06333* (2021).

[5] Umair Z Ahmed et al. "Compilation error repair: for the student programs, from the student programs". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ACM. 2018, pp. 78–87.

[6] Christophe Alias and Denis Barthou. "On the recognition of algorithm templates". In: *Electronic Notes in Theoretical Computer Science* 82.2 (2004), pp. 395–409.

[7] Miltiadis Allamanis. "The adverse effects of code duplication in machine learning models of code". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2019, pp. 143–153.

[8] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. 2018.

[9] Miltiadis Allamanis et al. "A survey of machine learning for big code and naturalness". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), p. 81.

[10] Miltos Allamanis et al. "Bimodal Modelling of Source Code and Natural Language". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, pp. 2123–2132.

[11] Uri Alon et al. "code2seq: Generating sequences from structured representations of code". In: *arXiv preprint arXiv:1808.01400* (2018).

[12] Uri Alon et al. "Code2Vec: Learning Distributed Representations of Code". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 40:1–40:29. ISSN: 2475-1421.

[13] Dario Amodei et al. "Concrete Problems in AI Safety". In: *arXiv e-prints*, arXiv: 1606.06565 (2016). arXiv: `1606.06565 [cs.AI]`.

[14] Marcin Andrychowicz et al. "Hindsight Experience Replay". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 5048–5058.

[15] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. 2nd. New York, NY, USA: Cambridge University Press, 2003. ISBN: 052182060X.

[16] Andrea Arcuri and Lionel Briand. "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 1–10. ISBN: 978-1-4503-0445-0.

[17] Thanassis Avgerinos et al. "The mayhem cyber reasoning system". In: *IEEE Security & Privacy* 16.2 (2018), pp. 52–60.

[18] Eser Aygün et al. "Learning to Prove from Synthetic Theorems". In: *arXiv e-prints*, arXiv: 2006.11259 (2020). arXiv: `2006.11259 [cs.LO]`.

[19] Sahar Badihi, Yi Li, and Julia Rubin. "EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 2021, pp. 610–614.

[20] Sahar Badihi et al. "ARDiff: Scaling Program Equivalence Checking via Iterative Abstraction and Refinement of Common Code". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 13–24. ISBN: 9781450370431.

[21] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[22] Jacques M. Bahi et al. "Neural networks and chaos: Construction, evaluation of chaotic networks, and prediction of chaos with multilayer feedforward networks". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 22.1 (2012), p. 013122. eprint: https://doi.org/10.1063/1.3685524.

[23] Kshitij Bansal et al. "HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, 2019, pp. 454–463.

[24] Wenlei Bao et al. "Polycheck: Dynamic verification of iteration space transformations on affine programs". In: *ACM SIGPLAN Notices*. Vol. 51. 1. ACM. 2016, pp. 539–554.

[25] Denis Barthou, Paul Feautrier, and Xavier Redon. "On the equivalence of two systems of affine recurrence equations". In: *Euro-Par 2002 Parallel Processing*. 2002.

[26] Paul Bassett. "Computer Aided Programming: Techniques for Software Manufacturing". In: ACM '83. New York, NY, USA: Association for Computing Machinery, 1983, pp. 18–19. ISBN: 0897911202.

[27] Benoit Baudry et al. “A Software-Repair Robot Based on Continual Learning”. In: *IEEE Software* 38.4 (2021), pp. 28–35.

[28] Rohan Bavishi, Michael Pradel, and Koushik Sen. *Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts*. 2017.

[29] Daniel Beck, Gholamreza Haffari, and Trevor Cohn. “Graph-to-Sequence Learning using Gated Graph Neural Networks”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, 2018, pp. 273–283.

[30] Yoshua Bengio. “Practical recommendations for gradient-based training of deep architectures”. In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478.

[31] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq’Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[32] Guru Bhandari, Amara Naseer, and Leon Moonen. “CVEfixes: automated collection of vulnerabilities and their fixes from open-source software”. In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2021, pp. 30–39.

[33] S. Bhatia, P. Kohli, and R. Singh. “Neuro-Symbolic Program Corrector for Introductory Programming Assignments”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering*. 2018, pp. 60–70.

[34] Pavol Bielik, Veselin Raychev, and Martin Vechev. “PHOG: Probabilistic Model for Code”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, pp. 2933–2942.

[35] Ondrej Bojar et al. "Findings of the 2014 Workshop on Statistical Machine Translation". In: *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Baltimore, Maryland, USA: Association for Computational Linguistics, 2014, pp. 12–58.

[36] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[37] Joseph Tobin Buck and Edward A Lee. "Scheduling dynamic dataflow graphs with bounded memory using the token flow model". In: *1993 IEEE international conference on acoustics, speech, and signal processing*. Vol. 1. IEEE. 1993, pp. 429–432.

[38] Nghi D. Q. Bui. "Efficient Framework for Learning Code Representations through Semantic-Preserving Program Transformations". In: *arXiv e-prints*, arXiv:2009.02731 (2020). arXiv: 2009.02731 [cs.SE].

[39] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. "Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations". In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 511–521. ISBN: 9781450380379.

[40] Jerry R Burch et al. "Symbolic model checking: 1020 states and beyond". In: *Information and computation* 98.2 (1992), pp. 142–170.

[41] José Campos et al. "GZoltar: An Eclipse Plug-in for Testing and Debugging". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, 2012, pp. 378–381. ISBN: 978-1-4503-1204-2.

[42] Sicong Cao et al. "BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection". In: *Information and Software Technology* (2021), p. 106576.

[43] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. "Tree2Tree Neural Translation Model for Learning Source Code Changes". In: *arXiv* abs/1810.00314 (2018).

[44] Saikat Chakraborty et al. "Codit: Code editing with tree-based neural models". In: *IEEE Transactions on Software Engineering* (2020).

[45] Lili Chen et al. "Decision Transformer: Reinforcement Learning via Sequence Modeling". In: *CoRR* abs/2106.01345 (2021). arXiv: 2106.01345.

[46] Z. Chen and M. Monperrus. "The CodRep Machine Learning on Source Code Competition". In: *ArXiv e-prints* (2018). arXiv: 1807.03200 [cs.SE].

[47] Zimin Chen, Steve Kommrusch, and Martin Monperrus. "Neural Transfer Learning for Repairing Security Vulnerabilities in C Code". In: *CoRR* abs/2104.08308 (2021). arXiv: 2104.08308.

[48] Zimin Chen et al. "SEQUENCER : Sequence-to-Sequence Learning for End-to-End Program Repair". In: *IEEE Transactions on Software Engineering* (2019). QC 20191106.

[49] Jianlei Chi et al. "SeqTrans: Automatic Vulnerability Fix via Sequence to Sequence Learning". In: *arXiv preprint arXiv:2010.10805* (2020).

[50] KyungHyun Cho et al. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches". In: *CoRR* abs/1409.1259 (2014). arXiv: 1409.1259.

[51] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1724–1734.

[52] Kyunghyun Cho et al. "On the properties of neural machine translation: Encoder-decoder approaches". In: *arXiv preprint arXiv:1409.1259* (2014).

[53] Berkeley Churchill et al. "Semantic program alignment for equivalence checking". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 1027–1040.

[54] *clang: a C language family frontend for LLVM*. http://clang.llvm.org/ ; accessed 3-Apr-2021.

[55] Edmund Clarke, Daniel Kroening, and Karen Yorav. “Behavioral consistency of C and Verilog programs using bounded model checking”. In: *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*. IEEE. 2003, pp. 368–371.

[56] Edmund M Clarke, Orna Grumberg, and David E Long. “Model checking and abstraction”. In: *ACM transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994), pp. 1512–1542.

[57] Joshua Clune et al. “Program Equivalence for Assisted Grading of Functional Programs”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020).

[58] John Cocke. “Global common subexpression elimination”. In: *Proceedings of a symposium on Compiler optimization*. 1970, pp. 20–24.

[59] Stephen Cook. “The P versus NP problem”. In: *Clay Mathematical Institute; The Millennium Prize Problem*. 2000.

[60] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158.

[61] Maxwell Crouse et al. “A Deep Reinforcement Learning Approach to First-Order Logic Theorem Proving”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.7 (2021), pp. 6279–6287.

[62] Jia Deng et al. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255.

[63] David Denison. “Parts of speech: Solid citizens or slippery customers?” In: *Journal of the British Academy*. The British Academy, 2013, pp. 151–185.

[64] Nachum Dershowitz. “Computing with rewrite systems”. In: *Information and Control* 65.2-3 (1985), pp. 122–157.

[65] Thomas G. Dietterich. “Ensemble Methods in Machine Learning”. In: *Proceedings of the First International Workshop on Multiple Classifier Systems*. MCS ’00. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 1–15. ISBN: 3540677046.

[66] Wei Ding. “Exploring the Possibilities of Applying Transfer Learning Methods for Natural Language Processing in Software Development”. MA thesis. Technische Universität München, 2021.

[67] Facebook. *Infer*. 2021. URL: https://fbinfer.com/ (visited on 03/15/2021).

[68] Jiahao Fan et al. “A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 508–512.

[69] Alhussein Fawzi et al. “Learning dynamic polynomial proofs”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 4179–4188.

[70] Zhangyin Feng et al. “Codebert: A pre-trained model for programming and natural languages”. In: *arXiv preprint arXiv:2002.08155* (2020).

[71] Qing Gao et al. “Safe memory-leak fixing for c programs”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 459–470.

[72] Xiang Gao et al. “Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction”. In: (2020).

[73] *GH Archive*. 2015. URL: https://www.gharchive.org (visited on 03/15/2021).

[74] Philip Ginsbach, Bruce Collie, and Michael FP O’Boyle. “Automatically harnessing sparse acceleration”. In: *Proceedings of the 29th International Conference on Compiler Construction*. 2020, pp. 179–190.

[75] *GitHub Octoverse 2020 Security Report*. GitHub, 2021.

[76] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. “VulinOSS: a dataset of security vulnerabilities in open-source systems”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 18–21.

[77] Benny Godlin and Ofer Strichman. “Inference rules for proving the equivalence of recursive procedures”. In: *Acta Informatica* 45.6 (2008), pp. 403–439.

[78] Vibhav Gogate and Pedro Domingos. “Probabilistic theorem proving”. In: *arXiv preprint arXiv:1202.3724* (2012).

[79] Robert Goldblatt and Marcel Jackson. “Well-structured program equivalence is highly undecidable”. In: *ACM Transactions on Computational Logic (TOCL)* 13.3 (2012), p. 26.

[80] Ian J Goodfellow et al. “An empirical investigation of catastrophic forgetting in gradient-based neural networks”. In: *arXiv preprint arXiv:1312.6211* (2013).

[81] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. Cambridge, MA, USA: MIT Press, 2016.

[82] Jiatao Gu et al. “Incorporating copying mechanism in sequence-to-sequence learning”. In: *arXiv preprint arXiv:1603.06393* (2016).

[83] Rahul Gupta, Aditya Kanade, and Shirish Shevade. “Deep Reinforcement Learning for Syntactic Error Repair in Student Programs”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (2019), pp. 930–937.

[84] Rahul Gupta et al. “DeepFix: Fixing Common C Language Errors by Deep Learning.” In: *AAAI*. 2017, pp. 1345–1351.

[85] Eitan M. Gurari and Oscar H. Ibarra. “Some simplified undecidable and NP-hard problems for simple programs”. In: *Theoretical Computer Science* 17.1 (1982), pp. 55–73. ISSN: 0304-3975.

[86] Jacob Harer et al. “Learning to repair software vulnerabilities with generative adversarial networks”. In: *arXiv preprint arXiv:1805.07475* (2018).

[87] Hideaki Hata, Emad Shihab, and Graham Neubig. "Learning to Generate Corrective Patches using Neural Machine Translation". In: *arXiv preprint 1812.07170* (2018).

[88] Simon S. Haykin. *Neural networks and learning machines*. Third. Pearson Education, 2009, pp. 383–398.

[89] Kaiming He, Ross Girshick, and Piotr Dollár. "Rethinking imagenet pre-training". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 4918–4927.

[90] Vincent J Hellendoorn and Premkumar Devanbu. "Are deep neural networks the best choice for modeling source code?" In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 763–773.

[91] Vincent J Hellendoorn and Premkumar Devanbu. "Are deep neural networks the best choice for modeling source code?" In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 763–773.

[92] Jordan Henkel et al. "Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces". In: *Proceedings of ESEC/FSE*. 2018.

[93] Thomas Hérault et al. "Approximate probabilistic model checking". In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2004, pp. 73–84.

[94] Karl Moritz Hermann et al. "Teaching machines to read and comprehend". In: *Advances in neural information processing systems*. 2015, pp. 1693–1701.

[95] Abram Hindle et al. "On the naturalness of software". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 837–847.

[96] G. E. Hinton and R. R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786 (2006), pp. 504–507. ISSN: 0036-8075.

[97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[98] Hossein Homaei and Hamid Reza Shahriari. “Seven years of software vulnerabilities: The ebb and flow”. In: *IEEE Security & Privacy* 15.1 (2017), pp. 58–65.

[99] Shan Huang, Xiao Zhou, and Sang Chin. “Application of Seq2Seq Models on Code Correction”. In: *Frontiers in artificial intelligence* 4 (2021), p. 590215. ISSN: 2624-8212.

[100] Zhen Huang et al. “Using safety properties to generate vulnerability patches”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 539–554.

[101] Iu I. Ianov. “On the Equivalence and Transformation of Program Schemes”. In: *Commun. ACM* 1.10 (1958), pp. 8–12. ISSN: 0001-0782.

[102] Oscar Ibarra and Shlomo Moran. “Probabilistic Algorithms for Deciding Equivalence of Straight-Line Programs”. In: *J. ACM* 30 (1983), pp. 217–228.

[103] Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. “On program equivalence with reductions”. In: *International Static Analysis Symposium*. Springer. 2014, pp. 168–183.

[104] Tiantian Ji et al. “The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques”. In: *2018 IEEE third international conference on data science in cyberspace (DSC)*. IEEE. 2018, pp. 53–60.

[105] Jiajun Jiang et al. “Shaping Program Repair Space with Existing Patches and Similar Code”. In: (2018).

[106] Marcin Junczys-Dowmunt and Roman Grundkiewicz. “An Exploration of Neural Sequence-to-Sequence Architectures for Automatic Post-Editing”. In: *IJCNLP 2017*. 2017.

[107] René Just, Darioush Jalali, and Michael D Ernst. “Defects4J: A database of existing faults to enable controlled testing studies for Java programs”. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM. 2014, pp. 437–440.

[108] Sara Kalvala, Richard Warburton, and David Lacey. “Program transformations using temporal logic side conditions”. In: *ACM Trans. on Programming Languages and Systems (TOPLAS)* 31.4 (2009), p. 14.

[109] Aditya Kanade et al. “Learning and evaluating contextual embedding of source code”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5110–5121.

[110] Donald M Kaplan. “Regular expressions and the equivalence of programs”. In: *Journal of Computer and System Sciences* 3.4 (1969), pp. 361–386.

[111] J. Kaplan et al. “Scaling Laws for Neural Language Models”. In: *ArXiv* abs/2001.08361 (2020).

[112] Rafael-Michael Karampatsis and Charles Sutton. “Maybe Deep Neural Networks are the Best Choice for Modeling Source Code”. In: *CoRR* abs/1903.05734 (2019). arXiv: `1903.05734`.

[113] Chandan Karfa et al. “Verification of loop and arithmetic transformations of array-intensive behaviors”. In: *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 32.11 (2013), pp. 1787–1800.

[114] Sal Khan. “Properties of matrix multiplication”. In: *Khan Academy (accessed May 20, 2020)* (2020).

[115] Jack Kiefer, Jacob Wolfowitz, et al. “Stochastic estimation of the maximum of a regression function”. In: *The Annals of Mathematical Statistics* 23.3 (1952), pp. 462–466.

[116] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015.

[117] Guillaume Klein et al. “OpenNMT: Open-Source Toolkit for Neural Machine Translation”. In: *Proc. ACL*. 2017.

[118] Andrew J Ko et al. “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks”. In: *IEEE Transactions on software engineering* 12 (2006), pp. 971–987.

[119] Philipp Koehn and Rebecca Knowles. "Six Challenges for Neural Machine Translation". In: *CoRR* abs/1706.03872 (2017). arXiv: 1706.03872.

[120] S. Kommrusch and L. Pouchet. "Synthetic Lung Nodule 3D Image Generation Using Autoencoders". In: *2019 International Joint Conference on Neural Networks (IJCNN)*. 2019, pp. 1–9.

[121] Steve Kommrusch. *S4Eq Software*. 2021. URL: github.com/SteveKommrusch/PrgEq.

[122] Steve Kommrusch. "Self-Supervised Learning for Multi-Goal Grid World: Comparing Leela and Deep Q Network". In: *Proceedings of the First International Workshop on Self-Supervised Learning*. Ed. by Henry Minsky et al. Vol. 131. Proceedings of Machine Learning Research. PMLR, 2020, pp. 72–88.

[123] Steve Kommrusch, Théo Barollet, and Louis-Noël Pouchet. "Proving Equivalence Between Complex Expressions Using Graph-to-Sequence Neural Models". In: *CoRR* abs/2106.02452 (2021). arXiv: 2106.02452.

[124] Steve Kommrusch, Martin Monperrus, and Louis-Noël Pouchet. *Self-Supervised Learning to Prove Equivalence Between Programs via Semantics-Preserving Rewrite Rules*. 2021. arXiv: 2109.10476 [cs.LG].

[125] Steve Kommrusch et al. "Optimizing Coherence Traffic in Manycore Processors Using Closed-Form Caching/Home Agent Mappings". In: *IEEE Access* 9 (2021), pp. 28930–28945.

[126] D. J. Kuck et al. "Dependence Graphs and Compiler Optimizations". In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '81. Williamsburg, Virginia: Association for Computing Machinery, 1981, pp. 207–218. ISBN: 089791029X.

[127] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. "Proving optimizations correct using parameterized program equivalence". In: *ACM SIGPLAN Notices* 44.6 (2009), pp. 327–337.

[128] Jeremy Lacomis et al. "DIRE: A Neural Approach to Decompiled Identifier Naming". In: *International Conference on Automated Software Engineering*. ASE '19. 2019.

[129] Alex M Lamb et al. "Professor Forcing: A New Algorithm for Training Recurrent Networks". In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., 2016, pp. 4601–4609.

[130] Guillaume Lample and François Charton. "Deep Learning For Symbolic Mathematics". In: *International Conference on Learning Representations*. 2020.

[131] Claire Le Goues et al. "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 3–13. ISBN: 978-1-4673-1067-3.

[132] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. "Memfix: static analysis-based repair of memory deallocation errors for c". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 95–106.

[133] Mike Lewis et al. "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension". In: *arXiv preprint arXiv:1910.13461* (2019).

[134] Der-Chiang Li et al. "Using mega-trend-diffusion and artificial samples in small data set learning for early flexible manufacturing system scheduling knowledge". In: *Computers & Operations Research* 34.4 (2007), pp. 966–982.

[135] W. Li et al. "Graph2Seq: Fusion Embedding Learning for Knowledge Graph Completion". In: *IEEE Access* 7 (2019), pp. 157960–157971.

[136] Yi Li, Shaohua Wang, and Tien N Nguyen. "DLfix: Context-based code transformation learning for automated program repair". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 602–614.

[137] Yujia Li et al. "Gated Graph Sequence Neural Networks". In: *Proceedings of ICLR'16*. 2016.

[138] Zhen Li et al. "Vuldeepecker: A deep learning-based system for vulnerability detection". In: *arXiv preprint arXiv:1801.01681* (2018).

[139] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv e-prints*, arXiv:1509.02971 (2015). arXiv: `1509.02971 [cs.LG]`.

[140] Guanjun Lin et al. "Software vulnerability detection using deep neural networks: a survey". In: *Proceedings of the IEEE* 108.10 (2020), pp. 1825–1848.

[141] Long-Ji Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine Learning* 8.3 (1992), pp. 293–321. ISSN: 1573-0565.

[142] Kui Liu et al. "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems". In: *arXiv preprint arXiv:1812.07283* (2018).

[143] Fan Long and Martin Rinard. "Staged Program Repair with Condition Synthesis". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 166–178. ISBN: 978-1-4503-3675-8.

[144] Net Losses. "Estimating the global cost of cybercrime". In: *McAfee, Centre for Strategic & International Studies* (2014).

[145] Dorel Lucanu and Vlad Rusu. "Program equivalence by circular reasoning". In: *Formal Aspects of Computing* 27.4 (2015), pp. 701–726.

[146] Lannan Luo et al. "Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection". In: *IEEE Transactions on Software Engineering* 43.12 (2017), pp. 1157–1177.

[147] Yong Luo et al. "An Appraisal of Incremental Learning Methods". In: *Entropy* 22.11 (2020). ISSN: 1099-4300.

[148] Minh-Thang Luong et al. "Addressing the rare word problem in neural machine translation". In: *arXiv preprint arXiv:1410.8206* (2014).

[149] Thang Luong, Hieu Pham, and Christopher Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, 2015, pp. 1412–1421.

[150] Thibaud Lutellier et al. "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 101–114.

[151] Thibaud Lutellier et al. "Encore: Ensemble learning using convolution neural machine translation for automatic program repair". In: *arXiv preprint arXiv:1906.08691* (2019).

[152] Siqi Ma et al. "Vurle: Automatic vulnerability detection and repair by learning from examples". In: *European Symposium on Research in Computer Security*. Springer. 2017, pp. 229–246.

[153] Ankur Arjun Mali et al. "Recognizing and Verifying Mathematical Equations using Multiplicative Differential Neural Units". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 5006–5015.

[154] William Mansky and Elsa Gunter. "A framework for formal verification of compiler optimizations". In: *Interactive Theorem Proving*. Springer, 2010.

[155] Matias Martinez et al. "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset". In: *Empirical Software Engineering* 22.4 (2017), pp. 1936–1964.

[156] John McCarthy et al. “A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, August 31, 1955”. In: *AI Magazine* 27.4 (2006), p. 12.

[157] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. “Angelix: Scalable multiline program patch synthesis via symbolic analysis”. In: *Proceedings of the 38th international conference on software engineering*. ACM. 2016, pp. 691–701.

[158] Ali Mesbah et al. “DeepDelta: learning to repair compilation errors”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 925–936.

[159] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *arXiv e-prints*, arXiv:1301.3781 (2013). arXiv: `1301.3781 [cs.CL]`.

[160] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533. ISSN: 00280836.

[161] Martin Monperrus. “Automatic Software Repair: a Bibliography”. In: *ACM Computing Surveys* 51 (2017), pp. 1–24.

[162] Gordon E. Moore. “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35.

[163] Federico Mora et al. “Client-Specific Equivalence Checking”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: Association for Computing Machinery, 2018, pp. 441–451. ISBN: 9781450359375.

[164] Paul Muntean et al. “Intrepair: Informed repairing of integer overflows”. In: *IEEE Transactions on Software Engineering* (2019).

[165] Andrzej S Murawski and Joël Ouaknine. “On probabilistic program equivalence and refinement”. In: *International Conference on Concurrency Theory*. Springer. 2005, pp. 156–170.

[166] David J Musliner et al. "Fuzzbomb: Autonomous cyber vulnerability detection and repair". In: *Fourth International Conference on Communications, Computation, Networks and Technologies (INNOV 2015)*. 2015.

[167] Ramesh Nallapati et al. "Abstractive text summarization using sequence-to-sequence rnns and beyond". In: *arXiv preprint arXiv:1602.06023* (2016).

[168] Kedar S Namjoshi and Robert P Kurshan. "Syntactic program transformations for automatic abstraction". In: *International Conference on Computer Aided Verification*. Springer. 2000, pp. 435–449.

[169] George C Necula. "Translation validation for an optimizing compiler". In: *ACM SIGPLAN Notices* 35.5 (2000), pp. 83–94.

[170] Vadim Okun, Aurelien Delaitre, and Paul E Black. "Report on the static analysis tool exposition (sate) iv". In: *NIST Special Publication* 500 (2013), p. 297.

[171] Aditya Paliwal et al. "Graph Representations for Higher-Order Logic and Theorem Proving". In: *arXiv e-prints*, arXiv:1905.10006 (2019). arXiv: `1905.10006 [cs.LG]`.

[172] Kishore Papineni et al. "BLEU: A Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318.

[173] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[174] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035.

[175] Lawrence C. Paulson. *Isabelle Page*. `https://www.cl.cam.ac.uk/research/hvg/Isabelle`.

[176] Jeff H Perkins et al. "Automatically patching errors in deployed software". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 87–102.

[177] Serena Elisa Ponta et al. "A manually-curated dataset of fixes to vulnerabilities of open-source software". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 383–387.

[178] Lutz Prechelt. "Early stopping-but when?" In: *Neural Networks: Tricks of the trade*. Springer, 1998, pp. 55–69.

[179] Yewen Pu et al. "sk_p: a neural program corrector for MOOCs". In: *CoRR* abs/1607.02902 (2016). arXiv: 1607.02902.

[180] Yuhua Qi et al. "The Strength of Random Search on Automated Program Repair". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 254–265. ISBN: 978-1-4503-2756-5.

[181] Zichao Qi et al. "An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 24–36. ISBN: 978-1-4503-3620-8.

[182] Zichao Qi et al. "An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 24–36. ISBN: 978-1-4503-3620-8.

[183] Md Rafiqul Islam Rabin et al. "On the generalizability of Neural Program Models with respect to semantic-preserving program transformations". In: *Information and Software Technology* 135 (2021), p. 106552. ISSN: 0950-5849.

[184] Danijel Radjenović et al. "Software fault prediction metrics: A systematic literature review". In: *Information and software technology* 55.8 (2013), pp. 1397–1418.

[185] Veselin Raychev, Martin Vechev, and Andreas Krause. "Predicting Program Properties from "Big Code"". In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Sympo-*

sium on Principles of Programming Languages. POPL '15. Mumbai, India: ACM, 2015, pp. 111–124. ISBN: 978-1-4503-3300-9.

[186] Uday S Reddy. "Rewriting techniques for program synthesis". In: *International Conference on Rewriting Techniques and Applications*. Springer. 1989, pp. 388–403.

[187] Sofia Reis and Rui Abreu. "A ground-truth dataset of real security patches". In: 2021.

[188] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. "Global value numbers and redundant computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 12–27.

[189] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[190] Rebecca Russell et al. "Automated vulnerability detection in source code using deep representation learning". In: *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE. 2018, pp. 757–762.

[191] Ripon K Saha et al. "Elixir: Effective object-oriented program repair". In: *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE. 2017, pp. 648–659.

[192] Eddie Antonio Santos et al. "Syntax and sensibility: Using language models to detect and correct syntax errors". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 311–322.

[193] Franco Scarselli et al. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20 (2009), pp. 61–80.

[194] Jonathan Schmidt et al. "Predicting the thermodynamic stability of solids combining density functional theory and machine learning". In: *Chemistry of Materials* 29.12 (2017), pp. 5090–5103.

[195] Markus Schordan et al. “Verification of Polyhedral Optimizations with Constant Loop Bounds in Finite State Space Computations”. In: *Proc. of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2014.

[196] Mike Schuster and Kuldip K Paliwal. “Bidirectional recurrent neural networks”. In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[197] Abigail See, Peter J. Liu, and Christopher D. Manning. “Get To The Point: Summarization with Pointer-Generator Networks”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, 2017, pp. 1073–1083.

[198] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural machine translation of rare words with subword units”. In: *arXiv preprint arXiv:1508.07909* (2015).

[199] Tushar Sharma et al. “Code smell detection by deep direct-learning and transfer-learning”. In: *Journal of Systems and Software* (2021), p. 110936.

[200] Hoo-Chang Shin et al. “Deep convolutional neural networks for computer-aided detection: CNN architectures, dataset characteristics and transfer learning”. In: *IEEE transactions on medical imaging* 35.5 (2016), pp. 1285–1298.

[201] Richard Shin, Illia Polosukhin, and Dawn Song. “Towards Specification-Directed Program Repair”. In: *ICLR Workshop*. 2018.

[202] Simeon Kostadinov. *Understanding Encoder-Decoder Sequence to Sequence Model*. https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346. [Online; accessed 10-November-2021]. 2019.

[203] Edward K Smith et al. “Is the cure worse than the disease? overfitting in automated program repair”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 532–543.

[204] Robert Solé and Dominique Valbelle. *The Rosetta Stone: the story of the decoding of hieroglyphics*. Profile, 2001.

[205] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[206] Bernhard Steffen. "Data flow analysis as model checking". In: *International Symposium on Theoretical Aspects of Computer Software*. Springer. 1991, pp. 346–364.

[207] Chen Sun et al. "Revisiting unreasonable effectiveness of data in deep learning era". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 843–852.

[208] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[209] C. Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9.

[210] Christian Szegedy et al. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[211] Chuanqi Tan et al. "A survey on deep transfer learning". In: *International conference on artificial neural networks*. Springer. 2018, pp. 270–279.

[212] Daniel Tarlow et al. "Learning to fix build errors with graph2diff neural networks". In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 2020, pp. 19–20.

[213] *The 2020 State of the Octoverse*. 2021. URL: `https://octoverse.github.com/` (visited on 03/15/2021).

[214] M. Tomita. "Dynamic Construction of Finite Automata from examples using Hill-climbing". In: *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*. Ann Arbor, Michigan, 1982, pp. 105–108.

[215] Lisa Torrey and Jude Shavlik. “Transfer learning”. In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 2010, pp. 242–264.

[216] Michele Tufano et al. “An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation”. In: *ACM Transactions on Software Engineering and Methodology* 28.4 (2019), 19:1–19:29. ISSN: 1049-331X.

[217] Michele Tufano et al. “On Learning Meaningful Code Changes via Neural Machine Translation”. In: *Proceedings 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019)*. 2019.

[218] A. M. Turing. “Computing Machinery and Intelligence”. In: *Mind* 59.236 (1950), pp. 433–460. ISSN: 00264423, 14602113.

[219] Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265.

[220] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[221] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. “Equivalence checking of static affine programs using widening to handle recurrences”. In: *Computer aided verification*. Springer. 2009, pp. 599–613.

[222] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. “Equivalence checking of static affine programs using widening to handle recurrences”. In: *ACM Trans. on Programming Languages and Systems (TOPLAS)* 34.3 (2012), p. 11.

[223] Eelco Visser. “Program transformation with Stratego/XT”. In: *Domain-specific program generation*. Springer, 2004, pp. 216–238.

[224] Willem Visser et al. “Model checking programs”. In: *Automated software engineering* 10.2 (2003), pp. 203–232.

[225] R.T. Vought. “Guidance for Regulation of Artificial Intelligence Applications.” In: *U.S. White House Announcement, Washington, D.C.* (2020). https://www.whitehouse.gov/wp-content/uploads/2020/11/M-21-06.pdf.

[226] Ke Wang, Rishabh Singh, and Zhendong Su. “Dynamic Neural Program Embedding for Program Repair”. In: *arXiv preprint arXiv:1711.07163* (2017).

[227] Qinglong Wang et al. “An Empirical Evaluation of Rule Extraction from Recurrent Neural Networks”. In: *Neural Comput.* 30.9 (2018), pp. 2568–2591. ISSN: 0899-7667.

[228] Tielei Wang, Chengyu Song, and Wenke Lee. “Diagnosis and emergency patch generation for integer overflow exploits”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2014, pp. 255–275.

[229] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. “Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results”. In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE’13. Silicon Valley, CA, USA: IEEE Press, 2013, pp. 356–366. ISBN: 978-1-4799-0215-6.

[230] Ming Wen et al. “Context-Aware Patch Generation for Better Automated Program Repair”. In: ICSE. 2018.

[231] Martin White et al. “Deep Learning Code Fragments for Code Clone Detection”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 87–98. ISBN: 978-1-4503-3845-5.

[232] Martin White et al. “Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities”. In: *Proceedings of SANER*. 2019.

[233] Wikipedia contributors. *Abstract syntax tree — Wikipedia, The Free Encyclopedia*. [Online; accessed 3-March-2019]. 2019.

[234] Wikipedia contributors. *Computational complexity theory — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Computational_complexity_theory&oldid=1039096789. [Online; accessed 9-November-2021]. 2021.

[235] Max Willsey et al. "Egg: Fast and Extensible Equality Saturation". In: *Proc. ACM Program. Lang.* 5.POPL (2021).

[236] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[237] Yuhuai Wu et al. "INT: An Inequality Benchmark for Evaluating Generalization in Theorem Proving". In: *arXiv e-prints*, arXiv:2007.02924 (2020). arXiv: `2007.02924`.

[238] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *CoRR* abs/1901.00596 (2019). arXiv: `1901.00596`.

[239] Qi Xin and Steven P Reiss. "Leveraging syntax-related code for automated program repair". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, pp. 660–670.

[240] Yingfei Xiong et al. "Precise condition synthesis for program repair". In: *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press. 2017, pp. 416–426.

[241] Xiaojun Xu et al. "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: ACM, 2017, pp. 363–376. ISBN: 978-1-4503-4946-8.

[242] Michihiro Yasunaga and Percy Liang. "Graph-based, self-supervised program repair from diagnostic feedback". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 10799–10808.

[243] He Ye, Matias Martinez, and Martin Monperrus. “Automated Patch Assessment for Program Repair at Scale”. In: *Empirical Software Engineering* (2020). eprint: `arXiv:1909.13694`.

[244] He Ye, Matias Martinez, and Martin Monperrus. “Neural Program Repair with Execution-based Backpropagation”. In: *CoRR* abs/2105.04123 (2021). arXiv: `2105.04123`.

[245] He Ye et al. “Automated Classification of Overfitting Patches with Statically Extracted Code Features”. In: *IEEE Transactions on Software Engineering* under press (2021).

[246] Pengcheng Yin et al. “Learning to represent edits”. In: *arXiv preprint arXiv:1810.13337* (2018).

[247] Peng Zhao and José Nelson Amaral. “Ablego: A Function Outlining and Partial Inlining Framework: Research Articles”. In: *Softw. Pract. Exper.* 37.5 (2007), pp. 465–491. ISSN: 0038-0644.

[248] Qihao Zhu et al. “A Syntax-Guided Edit Decoder for Neural Program Repair”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece, 2021, pp. 341–353. ISBN: 9781450385626.

[249] Daming Zou et al. “An empirical study of fault localization families and their combinations”. In: *IEEE Transactions on Software Engineering* (2019).