Programming with Neural Surrogates of Programs

by

Alex Renda

B.S., Cornell University, 2018
S.M., Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Alex Renda Department of Electrical Engineering and Computer Science March 8, 2024
Certified by:	Michael Carbin Associate Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Programming with Neural Surrogates of Programs

by

Alex Renda

Submitted to the Department of Electrical Engineering and Computer Science
on March 8, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Surrogate programming, the act of replacing programs with surrogate models of their behavior, is being increasingly leveraged to solve software development challenges. Surrogates are typically machine learning models trained on input-output examples of the program under consideration. With *surrogate compilation*, programmers train a surrogate that replicates the behavior of the original program to deploy to end-users in its place, with the goal of improving performance. With *surrogate adaptation*, programmers first train a surrogate of a program then continue to train the surrogate on a downstream task, with the goal of improving the accuracy of the surrogate on the task. With *surrogate optimization*, programmers train a surrogate of a program then use the surrogate to optimize the program's inputs, with the goal of optimizing inputs more efficiently than with the original program. These emerging design patterns represent an important new frontier of software development. However, we lack a coherent understanding of the applications and methodology underlying surrogate programming.

In this thesis I investigate three hypotheses about surrogate programming: that surrogate programming can be used to achieve state-of-the-art results on large-scale programming tasks; that there is a small set of methodologically distinct design patterns that can be grouped into a single programming methodology, unifying existing uses of surrogates in the literature; and that we can guide surrogate design using facts derived from the modeled program to train surrogates more efficiently and achieve better performance on downstream tasks.

To argue these hypotheses, I present four sets of contributions. I first present DiffTune, a surrogate optimization based approach to tuning the parameters of a large-scale CPU simulator. I next generalize this approach to identify the three design patterns above, and lay out the common methodology underlying all design patterns. I then present Turaco, a program analysis which allows developers to reason about the training data distribution to use to train a surrogate of a given program. I conclude with Renamer, a neural network architecture which mirrors source programs' invariance over variable renaming in their inputs.

Surrogate programming has the potential to change how developers program large-scale computer systems, by abstracting away much of the complexity to machine learning algorithms. Together, the contributions in my thesis lay the groundwork for a principled understanding of the applications and methodology of surrogate programming.

Thesis supervisor: Michael Carbin
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I'd first like to thank my advisor Michael Carbin for his guidance, support, and mentorship over the past six years. Mike has a knack for asking the perfect question to crystallize a problem or agenda, both in research and outside. None of the work in this thesis would have been possible without Mike's cuttingly insightful questions and advice.

I'd also like to thank my collaborators, who have been instrumental in the work in this thesis: Zack Ankner (who lead the Renamer project in Chapter 6), Yi Ding (who coauthored Chapters 3 and 5), Yishen Chen (who coauthored Chapter 2), and Charith Mendis (who coauthored Chapter 2 and was my first grad student mentor at MIT). I've worked with many fantastic people throughout grad school on projects that didn't make it into this thesis: Jonathan Frankle (who has also been a key mentor throughout grad school), Ajay Brahmakshatriya, Aspen Hopkins, Eric Atkinson, Gagandeep Singh, Gintare Karolina Dziugaite, Isha Chaudhary, Jesse Michel, Logan Weber, Ondřej Sýkora, Tian Jin, and Saman Amarasinghe.

The Programming Systems Group has been a fantastic home for the past six years. I'd like to thank the other members of PSG (Ben Sherman, Eric Atkinson, Cambridge Yang, Jesse Michel, Tian Jin, Logan Weber, Charles Yuan, Yi Ding, Ellie Cheng, Zack Ankner, and Jonathan Frankle), many of whom have served as collaborators and mentors, for their support, feedback on ideas and papers, and for making my time at MIT exceptional.

The MIT PL group has been a wonderful place to grow as a researcher, and to be exposed to different ideas and ways of thinking. Though the pandemic kept on throwing wrenches in

the works, I'd like to thank everyone in MIT PL, and in particular Stella Lau for helping to organize countless PL lunches and events.

I'd like to thank my thesis committee, Martin Rinard and Armando Solar-Lezama, for their feedback on drafts of this thesis.

Adrian Sampson plucked me out of CS 4110 at Cornell and gave me my first taste of research, and for that I am forever grateful. Saman Amarasinghe has also been a constant source of good advice and taste in research, especially throughout the early years of grad school.

The work in this thesis was computationally intensive, and I'd like to thank all sources of funding: the work in this thesis was supported by the National Science Foundation (NSF CCF-1918839, CCF-1533753, CCF-2217064, 2030859), the Defense Advanced Research Projects Agency (DARPA Awards #HR001118C0059, #FA8750-17-2-0126, #HR00112190046), a Google Faculty Research Award to my advisor Michael Carbin, cloud computing resources from the MIT Quest for Intelligence, and the MIT-IBM Watson AI Lab.

Finally, I'd like to thank my friends and family – in particular, my brother Nick and sister-in-law Reina, and my parents Greg and Katherine, for their love and support throughout grad school. Alana, thank you for your unrelenting support, and for pushing me to be the best version of myself. This thesis is dedicated to you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programming a modern software system is an increasingly complex task. This is due to several factors. Software systems grow in size and complexity over time, as more features and code are added to the system (Gonzalez-Barahona et al., 2008; Lehman, 1980; Xu et al., 2015). Software systems execute on complex and diverse hardware, from different CPU architectures to architectures like GPUs, TPUs, FPGAs, and other specialized accelerators (Chen et al., 2018; David et al., 2021). Software systems interact with complex and changing environments, including the operating system, standard libraries, and external libraries (Decan et al., 2016; Bommarito and Bommarito, 2019; Bagherzadeh et al., 2018). And, software systems model and interact with the complexities of the real world (Law, 2015). The result is that for any given software system, let alone a complex one built by a team, no single individual understands the entire system. This is further complicated by emergent behavior in the interactions between components of the system, resulting in behavior poorly understood by any developer.

Such systems are already widespread, with significant maintenance costs (Dehaghani and Hajrahimi, 2013; Kelly, 2020; McGeehan, 2020): even as early as 1981, the U.S. government found that two-thirds of software developer time at federal agencies was devoted to maintenance, at a cost of $1.3 billion (United States General Accounting Office, 1981). How does one maintain and evolve these existing systems? Better methodologies for programming com-

plex systems stand to impact both developers and users of programs, reducing the burden to develop complex programs while increasing their performance and reliability.

Surrogate programming. I argue that we should exploit advances in machine learning to abstract away the complexity of modern software systems as a means to address these challenges. In particular, I study *surrogate programming*, a programming methodology that aids the design, analysis, and evolution of complex programs by replacing components in the system with *surrogates* of their behavior (İpek et al., 2006; Esmaeilzadeh et al., 2012; Tercan et al., 2018; She et al., 2019; Tseng et al., 2019; Renda et al., 2020). When the surrogates are easier to manipulate to accomplish programming tasks than the original components are, surrogate programming leads to a more efficient and effective development process.

In the literature, surrogates are constructed using a variety of machine learning models, including neural networks (Goodfellow et al., 2016; Renda et al., 2020), Gaussian processes (Rasmussen and Williams, 2005; Alipourfard et al., 2017), linear models (Gelman and Hill, 2006; Ding et al., 2021), and random forests (Ho, 1995; Nardi et al., 2019). In this thesis I focus on neural network surrogates (Goodfellow et al., 2016; Renda et al., 2020).

Developers construct surrogates from measurements of the behavior of the original program on a dataset of input examples (Santner et al., 2018; Goodfellow et al., 2016; Myers et al., 2009; Gramacy, 2020). Surrogates can model different facets of a program's behavior, including the function computed by a given program (Renda et al., 2020; İpek et al., 2006; Esmaeilzadeh et al., 2012), the control flow trace that a given input induces in the program (She et al., 2019), and the program's wall-clock execution time (Huang et al., 2010; Mendis et al., 2019a).

Programmers use surrogates for a variety of tasks including accelerating computational kernels in numerical programs (Esmaeilzadeh et al., 2012), replacing physical simulators with more accurate versions (Tercan et al., 2018), and tuning parameters of complex simulators (Renda et al., 2020; Tseng et al., 2019). Compared to standard development workflows, programming with surrogates requires lower development costs (Renda et al., 2020; Tseng et al., 2019; She

et al., 2019; Kwon and Carloni, 2020; Kaya and Hajimirza, 2019) and results in programs with lower execution cost (Esmaeilzadeh et al., 2012; Mendis, 2020; Munk et al., 2022; Pestourie et al., 2020) or higher result quality (Tercan et al., 2018; Kustowski et al., 2020; Renda et al., 2020; Tseng et al., 2019). However, the approaches in the literature for both applying and developing surrogates are disparate, with no unifying taxonomy or development methodology.

The central research question that I investigate in this thesis is:

How do we program with surrogates of programs?

1.1 Surrogates of a Physics Simulation

I first motivate surrogate programming by demonstrating how using surrogates of programs helps solve three programming tasks related to developing a physics simulation. Figure 1.1 presents the program under study, a program that simulates the physics of a bouncing ball. The program (with entry point `simulate`) executes the simulation for a specified number of timesteps `i`. On each timestep, three events happen:

1. The ball continues with its current velocity (`step`, line 17)
2. The program checks to see if the ball has collided with the ground, and if so inelastically bounces the ball, negating its velocity (`step`, lines 19-22)
3. The ball accelerates downward due to gravity (`step`, line 24)

Figure 1.2 presents the results of calling `simulate` for a range of timesteps.

Given this program, I show how to solve three programming tasks with surrogates as a key element of each solution. First, with *surrogate compilation*, I accelerate the average execution time of the program over a fixed range of program inputs `i`. Second, with *surrogate adaptation*, I adapt a surrogate of the program to capture the dynamics reflected in ground-truth observations of a ball dropped in an environment that has more complex dynamics than

```python
timestep = 0.5

# Input:
#   i: number of steps to simulate
#   g: gravity. Default g=9.8
# Output:
#   Height after i steps
def simulate(i, g=9.8):
  y = 100  # initial height
  dy = 0   # initial velocity
  for _ in range(i):
    y, dy = step(y, dy, g)
  return y


def step(y, dy, g):
  # 1. move ball position
  y += timestep * dy
  # 2. bounce ball
  if y < 0:
    y = 0
    # elasticity of 0.9
    dy *= -0.9
  # 3. accelerate ball
  dy -= timestep * g
  return y, dy
```

Figure 1.1: Ball bouncing simulation program. The main entry point is `simulate`, which runs the simulation for a given number of steps and returns the final ball height.

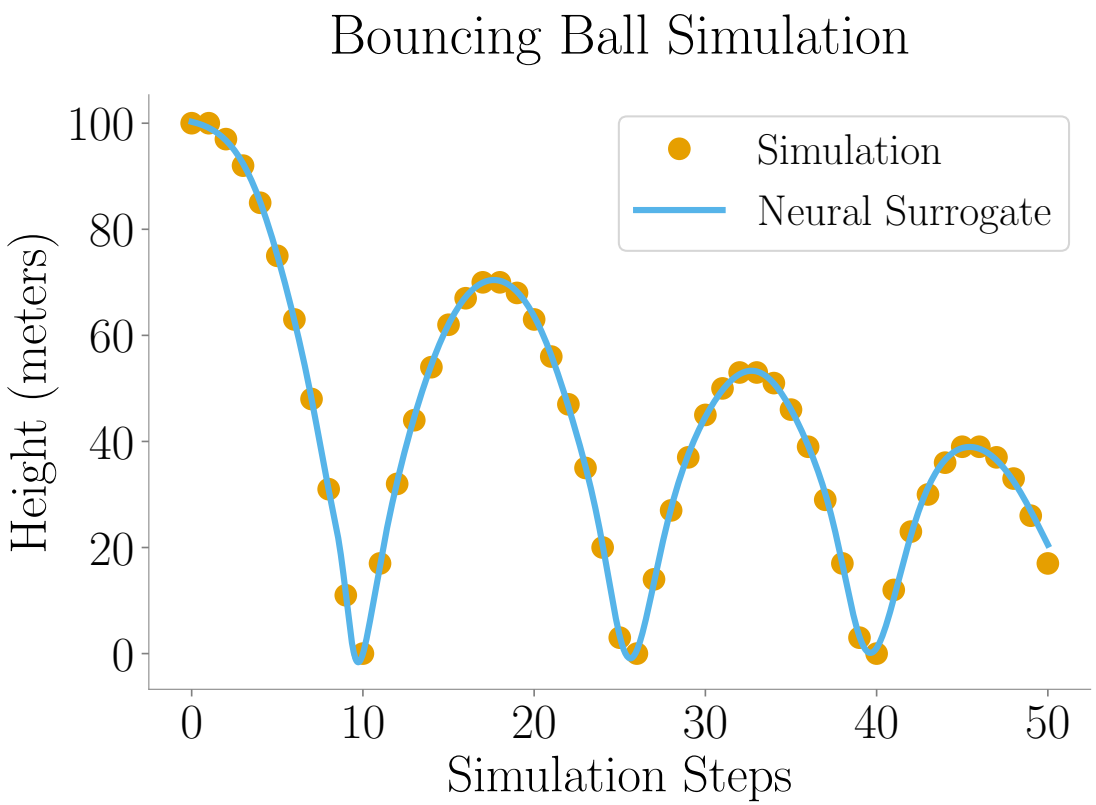


Figure 1.2: Result of the program (`simulate`, orange dots) and surrogate (blue line) when predicting the height of the ball after a specified number of timesteps.

those of the simulation's physical model (including drag). Third, with *surrogate optimization*, I find the value for the gravity parameter `g` that best explains the observed behavior of a bouncing ball in a setting with unknown gravity.

1.1.1 Surrogate Compilation

With surrogate compilation, programmers develop a surrogate that replicates the behavior of the original program to deploy to end-users in place of the original program. Key benefits of this approach include that it is possible to execute the surrogate on different hardware and to bound or accelerate the execution time of the surrogate on inputs within a bounded input domain (Esmaeilzadeh et al., 2012; Mendis, 2020).

For example, when executing the program in Figure 1.1 in a CPython interpreter on an Apple M1 CPU, the mean execution time of the program on input timesteps `i` between 1 and 50 is 6.75×10^{-6} seconds. Approaches in the literature for accelerating this program include memoizing the entire set of results of program execution, using an alternative runtime environment like PyPy (Bolz et al., 2009), manually rewriting the simulation to a closed form solution or approximation thereof, and rewriting the program in another programming language with a compiler or interpreter that results in faster execution.

An alternative approach for accelerating the program is surrogate compilation. With surrogate compilation a programmer develops a surrogate to match the outputs of the original program across all inputs in the domain while accelerating the average execution time of the program. The surrogate is a neural network that takes as input a timestep `i` and predicts the height of the ball after `i` timesteps (i.e., the result of `simulate(i)`). I use a shallow multi-layer perceptron (MLP) as a surrogate for this task.[1]

Figure 1.2 shows the surrogate predictions for the given range of timesteps (the blue line labeled surrogate). Figure 1.3 shows the execution times of the original program and

[1] I use a multilayer perceptron (MLP) with a sigmoid activation function. The MLP has a depth of 4 layers and a width of 64 neurons per layer. The surrogate is trained uniformly on integer-valued numbers of simulation steps `i` between 1 and 50.

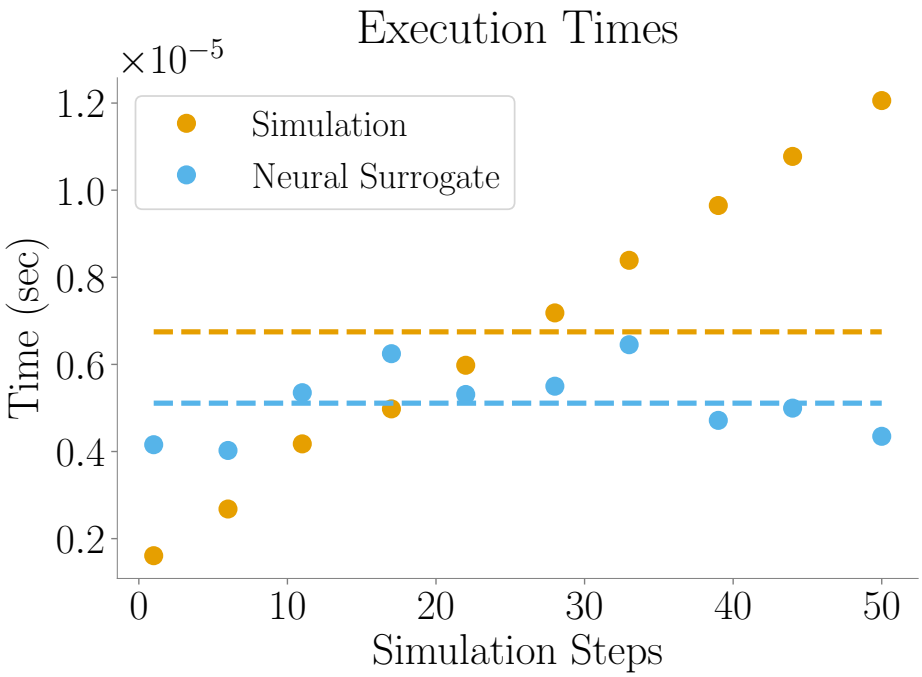


Figure 1.3: Program and surrogate execution times across inputs, with averages as dashed lines.

surrogate across timesteps `i` between 1 and 50, showing the averages as dashed lines. Surrogate compilation accelerates the mean execution time (on the same CPU) of the program across these inputs from taking 6.75×10^{-6} seconds to taking 5.11×10^{-6} seconds, a speedup of $1.32\times$.

1.1.2 Surrogate Adaptation

With surrogate adaptation, programmers first develop a surrogate of a program then continue to train the surrogate on a different task. The key benefit of this approach is that surrogate adaptation makes it possible to alter the semantics of the program to accurately perform a task it otherwise is unable to perform (Tercan et al., 2018; Verma et al., 2019).

For instance consider the behavior of a bouncing ball with the addition of dynamics induced by *drag*, which decreases the magnitude of the velocity with a force proportional to the square of the current velocity. Because the program in Figure 1.1 does not incorporate drag, it has inaccurate predictions compared to ground-truth data with drag, as shown in Figure 1.4.

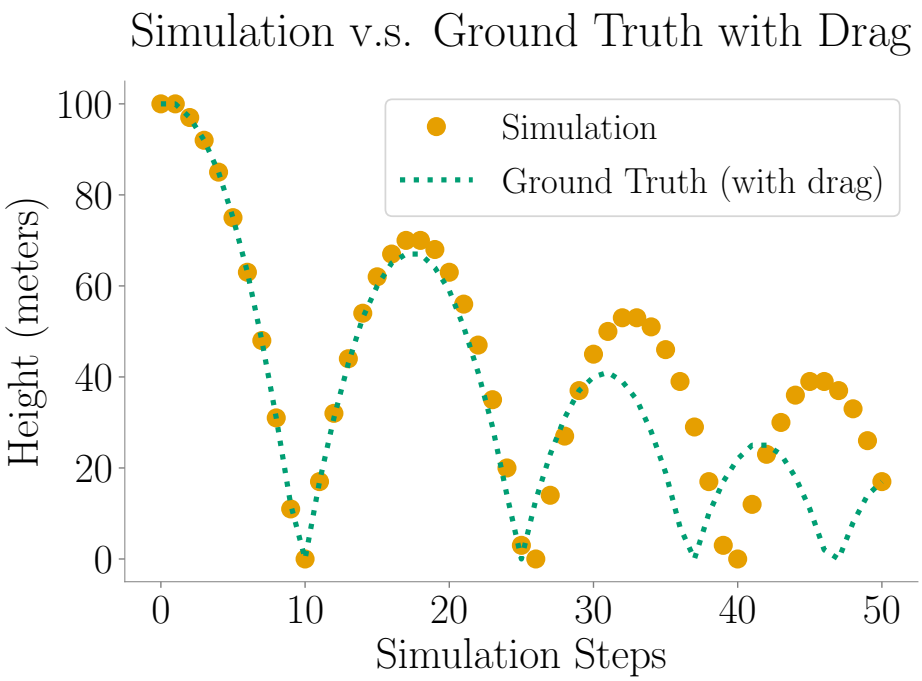


Figure 1.4: The program (orange dots) is not able to accurately model the behavior observed in ground-truth data with drag (green dotted line).

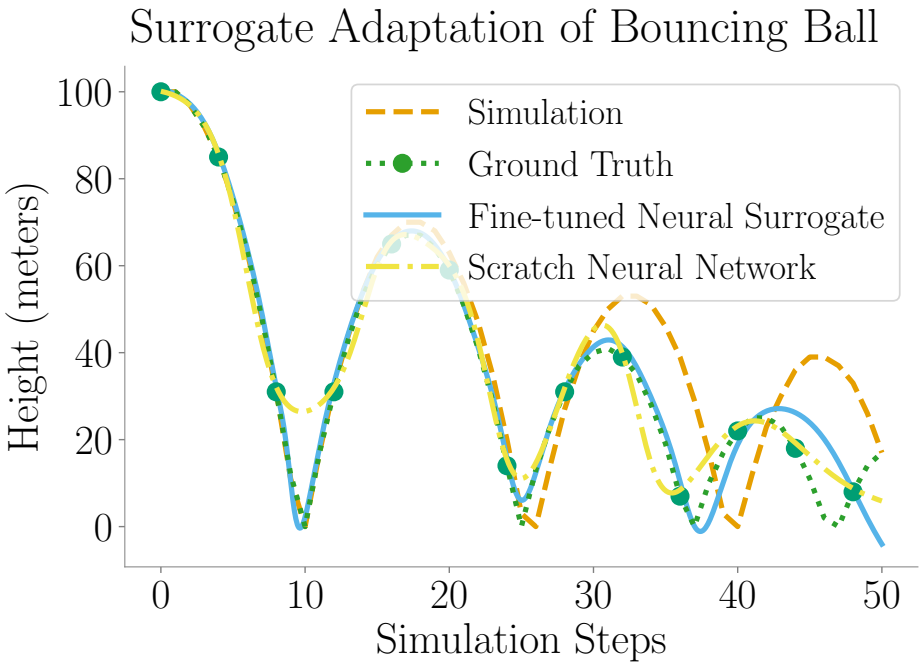


Figure 1.5: Result of using surrogate adaptation (blue) to model ground-truth data with drag (green, with sampled points as dots) compared to both the original program (orange) and a neural network trained from scratch on the ground truth data (yellow).

The simulation has a mean squared error across the 50 timesteps of 542. Approaches for developing a program that does fit the ground-truth data include rewriting the simulation code to include drag, program synthesis (Gulwani et al., 2017), and function approximation approaches like training a neural network from scratch on the ground-truth data.

An alternative approach for improving the accuracy of the program is surrogate adaptation, which adapts the original program's behavior to the new ground-truth behavior with different dynamics with a small number of observations of the ground-truth data. With surrogate adaptation, a programmer designs and trains a surrogate to match the outputs of the original program. The programmer then further trains the surrogate on observed ground-truth data, adapting it to the downstream task.

Figure 1.5 shows the results of several approaches after training on sparse samples of observations from the ground-truth data (every 4 timesteps). After developing the surrogate and continuing to train on these observations from the downstream task, the adapted surrogate is able to closely match the observed ground-truth behavior with drag, including at unobserved ground-truth points, due to its similarity with the original simulation. The adapted surrogate has a mean squared error across the 50 timesteps of 27, 5% that of the original program. In contrast, a neural network trained from scratch on the observed ground-truth data does not match the ground-truth behavior as closely, with a mean squared error across the 50 timesteps of 98, 18% that of the original program.

1.1.3 Surrogate Optimization

With surrogate optimization, programmers develop a surrogate of the original program then optimize inputs of the program against the surrogate on a downstream task. The key benefit of this approach is that surrogate optimization can optimize inputs faster than optimizing directly against the program, due to the potential for faster execution of the surrogate and the potential for the surrogate to be differentiable even when the original program is not (admitting gradient descent) (Tseng et al., 2019; She et al., 2019).

For example, the program in Figure 1.1 assumes a gravity parameter `g` of $9.8m/s^2$, the gravity of Earth. When the program is used to generate predictions for the height of a bouncing ball on another planet with different gravity, the program is inaccurate. The objective of the programming task is to estimate the value for the gravity parameter `g` that best explains the observed behavior of a bouncing ball on a different planet. Approaches in the literature for setting the gravity parameter include rewriting the program in a differentiable programming language (Baydin et al., 2018) and optimizing the gravity parameter with gradient descent, sketch-based program synthesis (Solar-Lezama et al., 2006), and autotuning (Ansel et al., 2014).

An alternative approach for optimizing parameters of the program is surrogate optimization. With surrogate optimization, a programmer develops a surrogate of the program then optimizes input parameters using gradient descent through the surrogate to maximize performance on a downstream task. Gradient descent converges faster to local minima than gradient-free optimization through the original program (Hicken et al., 2020).

In this case, I develop a surrogate to provide a differentiable mapping from an input timestep `i` and gravity parameter `g` to the value of `simulate(i, g)` (i.e., the result if the program were instantiated with the given gravity parameter `g`). Figure 1.6 shows the surrogate's predictions across a range of input timesteps and gravities.

I then optimize the gravity parameter `g`. I initialize the optimization of the gravity parameter `g` at 5, in the middle of the training distribution for gravity in the surrogate. I perform 1000 iterations of gradient descent on the gravity parameter `g` to maximize the similarity between the surrogate's prediction and observed behavior of the bouncing ball on a different planet, which is sufficient to find a local optimum for the gravity parameter.

When optimizing against data with $\texttt{g} = 3.7$ (which is the gravity on Mars), surrogate optimization results in an estimated gravity of $\texttt{g} = 3.57$. Figure 1.7 shows the predictions of the simulation with gravity $\texttt{g} = 3.57$ and the ground-truth data points with gravity $\texttt{g} = 3.7$, showing that surrogate optimization results in an estimated parameter that leads the program to accurate predictions compared to the ground-truth (resulting in a mean squared error of 3.12).

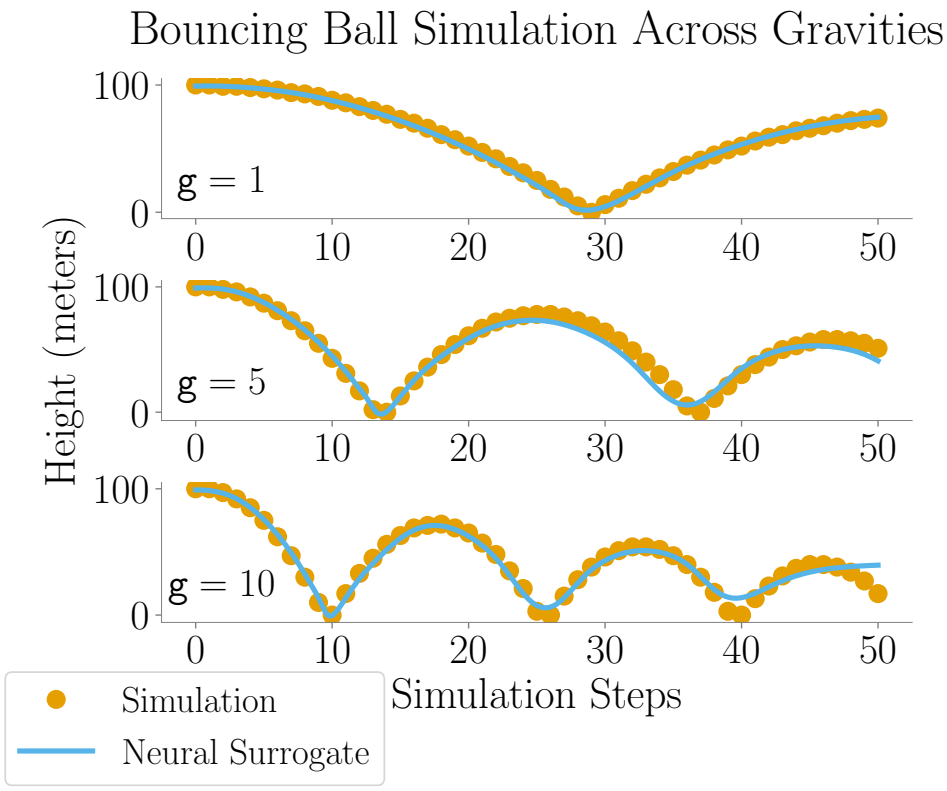


Figure 1.6: Approximating `simulate` (orange dots) across a range of gravity values `g` with a surrogate (blue lines).

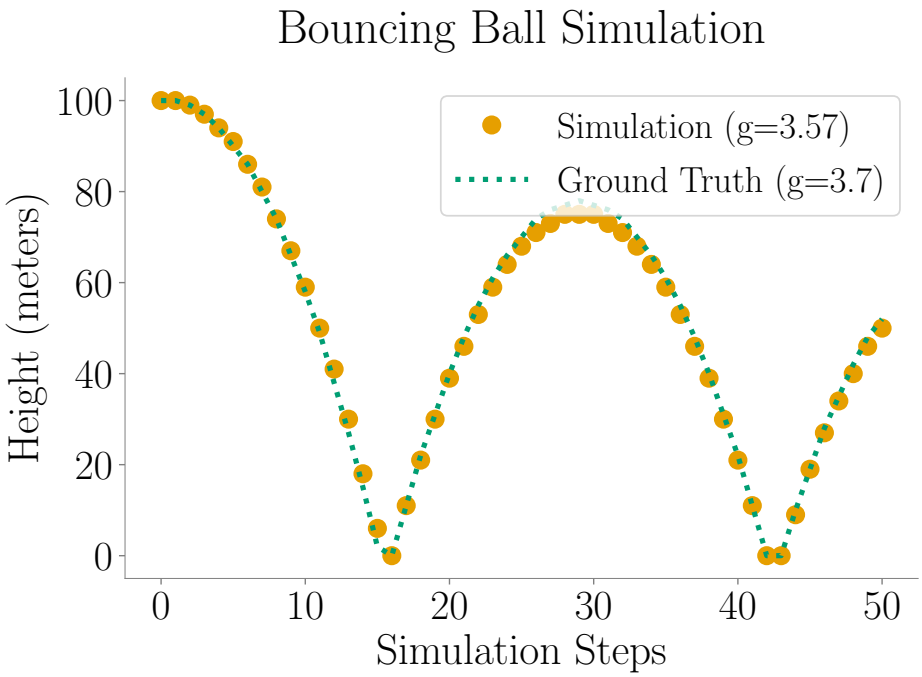


Figure 1.7: Estimating the gravity with surrogate optimization, with the simulation with the estimated gravity shown in orange and the ground-truth values shown in green.

1.2 Thesis

The previous section walked through several examples of using surrogate programming to solve programming tasks, giving a taste of how to program with surrogates of programs. Throughout this thesis I investigate this direction in much more depth. In particular, I investigate the following hypotheses about surrogate programming:

That surrogate programming can be used to achieve state-of-the-art results on large-scale programming tasks involving complex systems. A core driving question for many programming methodologies is their ability to provide utility for large-scale systems. By incorporating surrogate programming into the development of a large-scale application, my work demonstrates that surrogates can achieve state-of-the-art results on large-scale programming tasks. Beyond just demonstrating the utility of surrogate programming,

developing these state-of-the-art results helps us understand both the opportunities and the challenges of applying surrogate programming.

That there is a small set of methodologically distinct design patterns, each unifying existing uses of surrogates in the literature, that can be grouped into a single programming methodology that I call *surrogate programming*. The science and engineering literature contains numerous examples of surrogate programming (İpek et al., 2006; Esmaeilzadeh et al., 2012; Tercan et al., 2018; She et al., 2019; Tseng et al., 2019; Renda et al., 2020; Munk et al., 2022; Pestourie et al., 2020; Kustowski et al., 2020; Kwon and Carloni, 2020). However, these examples are scattered across different domains without a cohesive methodology. I hypothesize that surrogate programming consists of a small set of methodologically distinct design patterns, each unifying existing uses of surrogates in the literature. By surveying the literature of existing applications of surrogate programming, identifying the three surrogate-based design patterns, and describing the programming methodology used to develop neural surrogates, my work provides a taxonomy for reasoning about and developing surrogates.

That we can guide surrogate design using facts derived from the modeled program to train surrogates more efficiently and achieve better performance on downstream tasks. Developers train surrogates to mimic the behavior of a given program, often based on a dataset of input-output examples of the behavior of the program under study. Unlike standard machine learning tasks however, when constructing a surrogate of a program we have access to the precise semantics of the function we are trying to model. I hypothesize that we can exploit this information to guide surrogate design (e.g., training data selection, neural network architecture) using facts derived from the modeled program to train surrogates more efficiently and achieve better performance on downstream tasks.

1.3 Contributions

My thesis investigates these hypotheses through contributions from my and my collaborators' prior work (Renda et al., 2020, 2021; Ankner et al., 2023; Renda et al., 2023).

DiffTune (Renda et al., 2020). As a case study of surrogate programming, I first investigate the hypothesis that surrogate programming can be used to achieve state-of-the-art results on large-scale programming tasks involving complex systems I discuss DiffTune, a detailed case study of using surrogate programming (specifically, surrogate optimization) to find simulation parameters for llvm-mca, a 10,000 line-of-code CPU simulator included in the LLVM compiler infrastructure. These simulation parameters lead llvm-mca to more accurate CPU simulation than expert-set parameters and than parameters found using program tuning techniques that do not use surrogates of the simulator program.

Surrogate programming design patterns (Renda et al., 2021). I then generalize the programming methodology underlying DiffTune. I identify and define three design patterns that use surrogates of programs: *surrogate compilation, surrogate adaptation, and surrogate optimization* (Renda et al., 2021). With *surrogate compilation* programmers develop a surrogate that replicates the behavior of a program to deploy to end-users in place of that program. With *surrogate adaptation* programmers first develop a surrogate of a program then further train that surrogate on data from a different task. With *surrogate optimization* programmers develop a surrogate of a program, then optimize program inputs against that surrogate. I formalize and provide examples of each design pattern and discuss the key benefits and drawbacks of these surrogate programming methodologies. I then classify examples from the literature into this taxonomy of design patterns, demonstrating that these three design patterns generalize across a wide range of surrogate programming applications.

Turaco: Complexity-guided data sampling (Renda et al., 2023). All surrogate-based design patterns share a common first step: training a surrogate of a program under study. Constructing this surrogate necessitates selecting a training dataset and neural network architecture, among other methodological choices. As an example of guiding the surrogate design using facts about the modeled program I present Turaco, an approach to guiding data selection for training surrogates of programs. I first identify a specific challenge in surrogate programming: determining which regions of a program's input space to sample data from to train a surrogate of that program. I present an approach to sampling the space of program inputs to train neural network surrogates of programs that minimizes a theoretical upper bound on the surrogate's error. This approach is based on a novel program analysis that determines the sample complexity of learning a neural network surrogate of a given program. I evaluate this approach on a range of numeric programs, showing quantitatively and qualitatively better performance than baseline sampling approaches.

Renaming-invariant neural surrogates (Ankner et al., 2023). As additional evidence that we can guide surrogate design using facts about the modeled program, I identify another challenge in surrogate programming: developing surrogates invariant to input transformations that the underlying program is invariant to. Mirroring the program's invariance in the surrogate can improve the surrogate's accuracy and reducing its training cost. Mirroring invariance also has particular relevance in surrogate programming, where compared to standard machine learning tasks programs often take structured inputs with more clearly defined and identifiable invariances over the input's structure. I specifically study *renaming invariance*, the property of when a program is invariant to certain renamings of input tokens (e.g., CPU performance prediction is invariant to dependency-preserving register renaming). I introduce and formally characterize the renaming invariance problem and propose the Renamer, a renaming invariant Transformer model architecture. I demonstrate that Renamer results in better performance than vanilla Transformer model on renaming invariant tasks.

1.4 Looking Forward

We are in the early stages of a revolution in the ways we program software systems, where machine learning has caught up and even surpassed human programmers in certain tasks and domains (Chen et al., 2021; Li et al., 2022; OpenAI, 2023; Mankowitz et al., 2023). Several disparate approaches leveraging these new abilities are emerging, ranging from using machine learning to help write programs all the way to using machine learning to replace programs entirely. Surrogate programming is an important emerging element of this new landscape, with the potential to significantly decrease the cost of programming complex systems.

Surrogate programming exploits the unique properties of neural networks (their regularity of computation, parametric form, and differentiability) that allow them to serve as high-quality replacements for programs. I argue that surrogate programming is a coherent set of approaches that can be cast as a single programming methodology. By studying the commonalities between these approaches we can systematize the process and understand the limits of surrogate programming. We can then use tools from the programming languages and machine learning communities to guide improvements to this methodology.

Surrogate programming, though only one approach in this exciting new frontier, serves as a model for how we can use advances in machine learning to improve our programming methodologies. My work constitutes a crucial step towards understanding these new ways of programming in the age of deep learning.

Chapter 2

DiffTune: Optimizing CPU Simulator Parameters with Surrogate Programming

To motivate the opportunities and challenges of surrogate programming, I begin with a detailed case study of using surrogate programming to optimize the parameters of a CPU simulator, demonstrating the hypothesis that surrogate programming can be used to achieve state-of-the-art results on large-scale programming tasks involving complex systems. I also demonstrate the same technique on a different simulator to again achieve state-of-the-art results, showing the generality of the approach. This approach and corresponding results form the foundation for my focus on understanding and improving surrogate programming methodologies throughout the rest of the thesis.

Background: CPU simulators in computer architecture. Simulators are widely used for architecture research to model the interactions of architectural components of a system (Binkert et al., 2011; Di Biagio and Davis, 2018; Intel, 2017; Yourst, 2007; Patel et al., 2011; Sanchez and Kozyrakis, 2013). For example, *CPU simulators*, such as llvm-mca (Di Biagio and Davis, 2018), and llvm_sim (Sykora et al., 2018), model the execution of a processor at various levels of detail, potentially including abstract models of common processor design concepts such as dispatch, execute, and retire stages (Patterson and Hennessy,

1990). CPU simulators can operate at different granularities, from analyzing *basic blocks*, straight-line sequences of assembly code instructions, to analyzing whole programs. Such simulators allow performance engineers to reason about the behavior and bottlenecks of programs run on a given processor.

However, precisely simulating a modern CPU is challenging: not only are modern processors large and complex, but many of their implementation details are proprietary, undocumented, or only loosely specified even given the thousands of pages of vendor-provided documentation that describe any given processor. As a result, CPU simulators are often composed of coarse abstract models of a subset of processor design concepts. Moreover, each constituent model typically relies on a number of approximate design parameters, such as the number of cycles it takes for an instruction to pass through the processor's execute stage. Choosing an appropriate level of model detail for simulation, as well as setting simulation parameters, requires significant expertise. In this chapter, I consider the challenge of setting the parameters of a CPU simulator given a fixed simulation model.

Measurement. One methodology for setting the parameters of such a CPU simulator is to gather fine-grained measurements of each individual parameter's realization in the physical machine (Fog, 1996; Abel and Reineke, 2019) and then set the parameters to their measured values (Colombet, 2014; Topper, 2014). When the semantics of the simulator and the semantics of the measurement methodology coincide, then these measurements can serve as effective parameter values. However, if there is a mismatch between the simulator and the measurement methodology, then measurements may not provide effective parameter settings (Ritter and Hack, 2020, Section 5.2). Moreover, some parameters may not be measurable at all.

Optimizing simulator parameters. An alternative to developing detailed measurement methodologies for individual parameters is to infer the parameters from coarse-grained end-to-end measurements of the performance of the physical machine (Ritter and Hack, 2020). Specifically, given a dataset of benchmarks, each labeled with their true behavior on a given

CPU (e.g., with their execution time or with microarchitectural events, such as cache misses), identify a set of parameters that minimize the error between the simulator's predictions and the machine's true behavior. This is generally a discrete, non-convex optimization problem for which classic strategies, such as random search (Ansel et al., 2014), are intractable because of the size of the parameter space (with $10^{19,336}$ parameter settings in one simulator, llvm-mca).

DiffTune. In this chapter I present DiffTune, an optimization algorithm and implementation for learning the parameters of programs. I use DiffTune to learn the parameters of x86 basic block CPU simulators. DiffTune's algorithm takes as input a program, a description of the program's parameters, and a dataset of input-output examples describing the program's desired output, then produces a setting of the program's parameters that minimizes the discrepancy between the program's actual and desired output. The learned parameters are then plugged back into the original program, leading to higher quality outputs.

The algorithm solves this optimization problem with surrogate optimization against a differentiable surrogate of the original CPU simulator. By requiring the surrogate to be differentiable, it is then possible to compute the surrogate's gradient and apply gradient-based optimization (Robbins and Monro, 1951; Kingma and Ba, 2015) to identify a setting of the program's parameters that minimize the error between the program's output (as predicted by the surrogate) and the desired output.

To apply this to basic block CPU simulators, I instantiate DiffTune's surrogate with a neural network that can mimic the behavior of such a simulator. This neural network takes the original simulator input (e.g., a sequence of assembly instructions) and a set of proposed simulator parameters (e.g., dispatch width or instruction latency) as input, and produces the output that the simulator would produce if it were instantiated with the given simulator's parameters. I derive the neural network architecture of the surrogate from that of Ithemal (Mendis et al., 2019a), a basic block throughput estimation neural network.

Results. I demonstrate that DiffTune can learn the entire set of 11,265 microarchitecture-specific parameters in the respective Intel x86 simulation models of llvm-mca (Di Biagio and Davis, 2018) and llvm_sim (Sykora et al., 2018). Both llvm-mca and llvm_sim are CPU simulators that predict the execution time of basic blocks. The llvm-mca simulator models instruction dispatch, register renaming, out-of-order execution with a reorder buffer, instruction scheduling based on use-def latencies, execution by dispatching to ports, a load/store unit ensuring memory consistency, and a retire control unit.[1] The llvm_sim simulator uses many of the same parameters (from LLVM's backend) as llvm-mca, but uses a different simulation model of the CPU.

I evaluate DiffTune on four different x86 microarchitectures, including both Intel and AMD chips. Using only end-to-end supervision of the execution time measured per-microarchitecture of a large dataset of basic blocks from Chen et al. (2019), DiffTune is able to learn parameters from scratch that lead llvm-mca to an average error of 24.6%, down from an average error of 30.0% with llvm-mca's expert-provided parameters. DiffTune is able to learn parameters that lead llvm_sim to an error of 44.1%, down from an error of 61.3% with expert-provided parameters. In contrast, black-box global optimization with OpenTuner (Ansel et al., 2014) is unable to identify parameters with less than 100% error on either llvm-mca or llvm_sim.

Contributions. In this chapter I present the following contributions:

- I present DiffTune, a surrogate optimization algorithm for learning ordinal parameters of programs from input-output examples.
- I present an implementation of DiffTune for x86 basic block CPU simulators that uses a variant of the Ithemal model as a differentiable surrogate.
- I evaluate DiffTune on llvm-mca and llvm_sim and demonstrate that DiffTune can learn the entire set of microarchitectual parameters in their Intel x86 simulation models.

[1] Note that llvm-mca does not model the memory hierarchy.

- I present case studies of specific parameters learned by DiffTune. This analysis demonstrates cases in which DiffTune learns semantically correct parameters that enable llvm-mca to make more accurate predictions. This analysis also demonstrates cases in which DiffTune learns parameters that lead to higher accuracy but do not correspond to reasonable physical values on the CPU.

These results show that DiffTune offers the promise of a generic, scalable methodology to learn detailed performance models with only end-to-end measurements, reducing performance optimization tasks to simply that of gathering data. This validates the hypothesis that surrogate programming can be used to achieve state-of-the-art results on large-scale programming tasks involving complex systems.

2.1 Background: Simulators

Simulators comprise a large set of tools for modeling the execution behavior of computing systems, at all different levels of abstraction: from cycle-accurate simulators to high-level cost models. These simulators are used for a variety of applications:

- gem5 (Binkert et al., 2011) is a detailed, extensible full system simulator that is frequently used for computer architecture research, to model the interaction of new or modified components with the rest of a CPU and memory system.

- IACA (Intel, 2017) is a static analysis tool released by Intel that models the behavior of modern Intel processors, including undocumented Intel CPU features, predicting code performance. IACA is used by performance engineers to diagnose and fix bottlenecks in hand-engineered code snippets (Laukemann et al., 2018).

- LLVM (Lattner and Adve, 2004) includes internal CPU simulators to predict the performance of generated code (Pohl et al., 2020; Mendis and Amarasinghe, 2018).

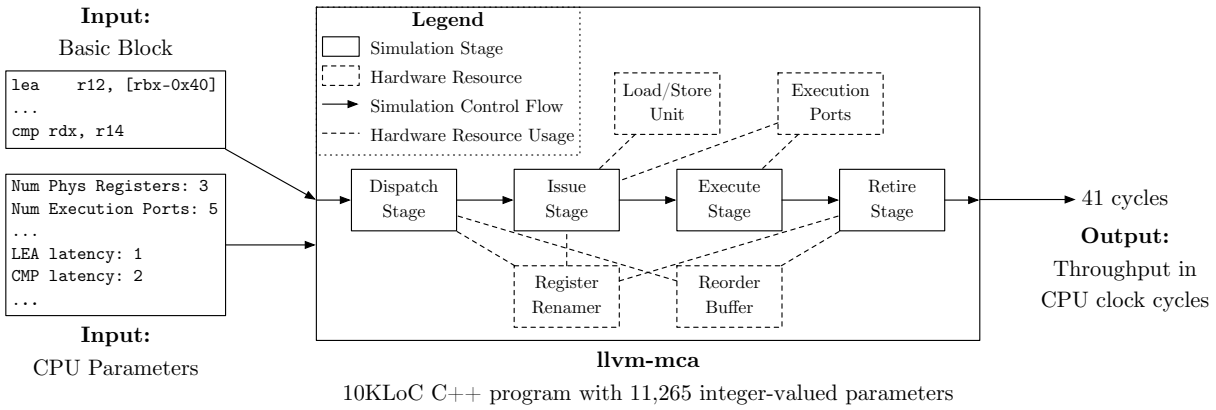


Figure 2.1: Input-output specification and design of llvm-mca.

LLVM uses these CPU simulators to search through the code optimization space, to generate more optimal code.

Though these simulators are all simplifications of the true execution behavior of physical systems, they are still highly complex pieces of software.

2.1.1 Simulation with llvm-mca

For example, consider llvm-mca (Di Biagio and Davis, 2018), a CPU simulator included in the LLVM (Lattner and Adve, 2004) compiler infrastructure. The main design goal of llvm-mca is to expose LLVM's instruction scheduling model for testing.

The llvm-mca simulator is an *out-of-order superscalar* simulator, meaning that it models the behavior of a CPU that can execute instructions out of the order in which they appear in the program and can execute multiple instructions in parallel.

Figure 2.1 presents llvm-mca's input-output specification and design. As input, llvm-mca takes a *basic block*, a sequence of assembly instructions with no jumps or loops, and a set of *CPU parameters*, integers that describe properties of the CPU being modeled. It then outputs a prediction of the *throughput* of the basic block on the CPU, a prediction of the number of CPU clock cycles taken to execute the block when repeated for a fixed number of iterations.

Design. Rather than precisely emulating the behavior of the CPU under study, llvm-mca makes several modeling assumptions about the behavior of the CPU, and simulates basic blocks using an abstract execution model of that CPU. The llvm-mca simulator is structured as a generic, target-independent simulator parameterized on LLVM's internal model of the target hardware. To model basic block performance, llvm-mca makes two core assumptions. First, it assumes that the simulated program is not bottlenecked by the processor frontend; in fact, llvm-mca ignores *instruction decoding*, an often sequential process in which instructions are translated into *micro-ops* that can be executed by the CPU; instead, llvm-mca assumes that the micro-ops are already available. Second, llvm-mca assumes that memory data is always in the L1 cache, and ignores the memory hierarchy.

The simulation model has four main stages: *dispatch*, *issue*, *execute*, and *retire*. Each stage is bottlenecked by the availability of *hardware resources* in the simulation model.

Instructions first enter into the *dispatch* stage. The dispatch stage reserves the hardware resources (e.g., slots in the reorder buffer) needed to track the execution of the instruction in the simulation model, based on the number of micro-ops the instruction is composed of.

Once dispatched, instructions wait in the *issue* stage until they are ready to be executed. The issue stage holds instructions until all of their input operands and all of the hardware resources required to execute the instructions are available.

Instructions then enter the *execute* stage, which reserves the hardware resources required to execute the instruction and holds them for the number of clock cycles specified by the CPU parameters for the instruction.

Finally, once instructions have executed for their duration, they enter the *retire* stage, which frees the resources that were acquired for each instruction in the dispatch phase.

Parameters. Each stage in llvm-mca's model requires parameters. The `NumMicroOps` parameter for each instruction specifies how many micro-ops the instruction is composed of. The `DispatchWidth` parameter specifies how many micro-ops can enter and exit the dispatch

stage in each cycle. The `ReorderBufferSize` parameter specifies how many micro-ops can reside in the issue and execute stages at the same time. The `PortMap` parameters for each instruction specify the number of cycles for which the instruction occupies each execution port. An additional `WriteLatency` parameter for each instruction specifies the number of cycles before destination operands of that instruction can be read from, while `ReadAdvance-Cycles` parameters for each instruction specify the number of cycles by which to accelerate the `WriteLatency` of source operands (representing forwarding paths).

In sum, the 837 instructions in the dataset (Section 2.4.1) lead to 11,265 total parameters with $10^{19,336}$ possible configurations in llvm-mca's Haswell microarchitecture simulation.[2]

2.1.2 Challenges

The default parameter tables are manually written for each microarchitecture, based on processor vendor documentation and manual timing of instructions. Specifically, many of LLVM's `WriteLatency` and `PortMap` parameters are drawn from the Intel optimization manual (int; Topper, 2018), Agner Fog's instruction tables (Fog, 1996; Colombet, 2014), and uops.info (Abel and Reineke, 2019; Topper, 2014), all of which contain latencies and port mappings for instructions across different architectures and microarchitectures.

Measurability. However, these documented and measured values do not directly correspond to parameters in llvm-mca, because llvm-mca's parameters, and abstract simulator parameters more broadly, are not defined such that they have a single measurable value. For instance, llvm-mca defines exactly one `WriteLatency` parameter per instruction. However, Fog (1996) and Abel and Reineke (2019) find that for instructions that produce multiple results in different destinations, the results might be available at different cycles. Further, the latency for results to be available can depend on the actual value of the input operands. Thus, there is no single measurable value that corresponds to llvm-mca's definition of `WriteLatency`.

[2]Based on llvm-mca's default, expert-provided values for these parameters, the 11,265 parameters induce a parameter space of $10^{19,336}$ configurations; the actual values are only bounded by integer representation sizes.

Different choices for how to map from measured latencies to `WriteLatency` values result in different overall errors (as defined in Section 2.4.1). For instance, when instantiating llvm-mca with Abel and Reineke (2019)'s maximum observed latency for each instruction, llvm-mca has an error of 218% when predicting for the Haswell microarchitecture; the median observed latency results in an error of 150%; and the minimum observed latency gives an error of 103%.

2.2 Approach

Tuning llvm-mca's 11,265 parameters among $10^{19,336}$ valid configurations by exhaustive search is impractical. Instead, I present DiffTune, a surrogate optimization algorithm for learning ordinal parameters of arbitrary programs from labeled input and output examples. DiffTune is an example of using surrogate programming (through surrogate optimization) to make this challenging programming task more tractable.

Formal problem statement. Given a program $f : \Theta \to \mathcal{X} \to \mathcal{Y}$ parameterized on parameters $\theta : \Theta$ that maps inputs $x : \mathcal{X}$ to outputs $y : \mathcal{Y}$, and given a function $f^* : \mathcal{X} \to \mathcal{Y}$ that represents ground-truth behavior, find parameters $\theta \in \Theta$ to minimize the expected value of a cost function (called the loss function, representing error) $\mathcal{L} : (\mathcal{Y} \times \mathcal{Y}) \to \mathbb{R}_{\geq 0}$ of the discrepancy between the behavior of the program f and the ground-truth behavior f^* on a dataset of program inputs $\mathcal{I} \subseteq \mathcal{X}$:

$$\arg\min_{\theta} \mathbb{E}_{x \sim \mathcal{I}}[\mathcal{L}(f(\theta, x), f^*(x))] \tag{2.1}$$

Algorithm. Figure 2.2 presents a diagram of using DiffTune to solve Equation (2.1). DiffTune first optimizes a surrogate $\hat{f} : \Theta \to \mathcal{X} \to \mathcal{Y}$ to mimic the original program, where the ideal surrogate has $\forall \theta, x. \hat{f}(\theta, x) \approx f(\theta, x)$. Specifically, DiffTune optimizes the surrogate to

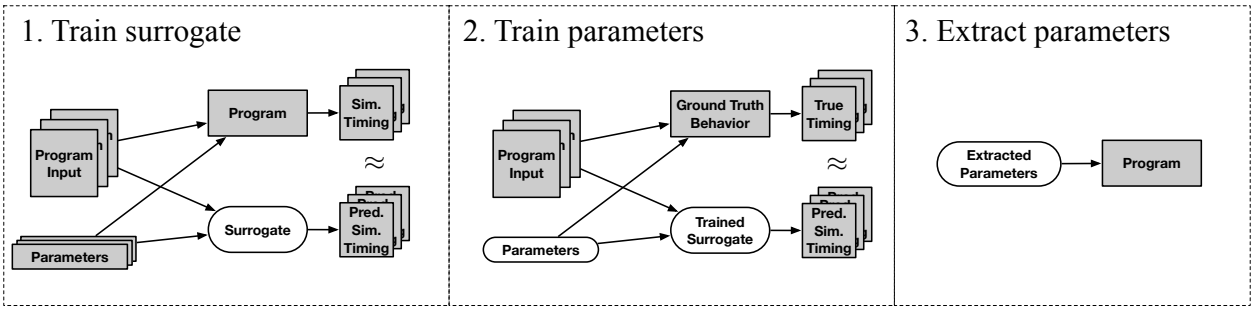


Figure 2.2: DiffTune block diagram.

minimize the expectation of the loss $\mathcal{L}$ over program inputs from $\mathcal{I}$ and samples of θ from a pre-defined parameter sampling distribution D:

$$\arg\min_{\hat{f}} \mathbb{E}_{x\sim\mathcal{I},\theta\sim D}\Big[\mathcal{L}\Big(f(\theta,x),\hat{f}(\theta,x)\Big)\Big] \tag{2.2}$$

DiffTune then optimizes the parameters θ to minimize the discrepancy between predictions of the surrogate $\hat{f}$ and the ground-truth behavior f^*, optimizing the following objective:

$$\arg\min_{\theta} \mathbb{E}_{x\sim\mathcal{I}}\Big[\mathcal{L}\Big(\hat{f}(\theta,x),f^*(x)\Big)\Big] \tag{2.3}$$

Finally, DiffTune extracts the learned parameters θ from the optimization problem and plugs them into the original program f, applying any constraints (e.g., that the parameters must be integers) that were not enforced when optimizing against the surrogate.

Discussion. Note the similarity between Equation (2.1) and Equation (2.3): the two equations only differ by the use of f and $\hat{f}$, respectively. The close correspondence between forms makes clear that $\hat{f}$ stands in as a surrogate for the original program, f. This is a general algorithmic approach (Queipo et al., 2005) that is desirable when it is possible to choose $\hat{f}$ such that it is easier or more efficient to optimize θ using $\hat{f}$ than f.

Optimization. In DiffTune's approach, $\hat{f}$ is a neural network. Neural networks are typically built as compositions of differentiable architectural components, such as *embedding lookup*

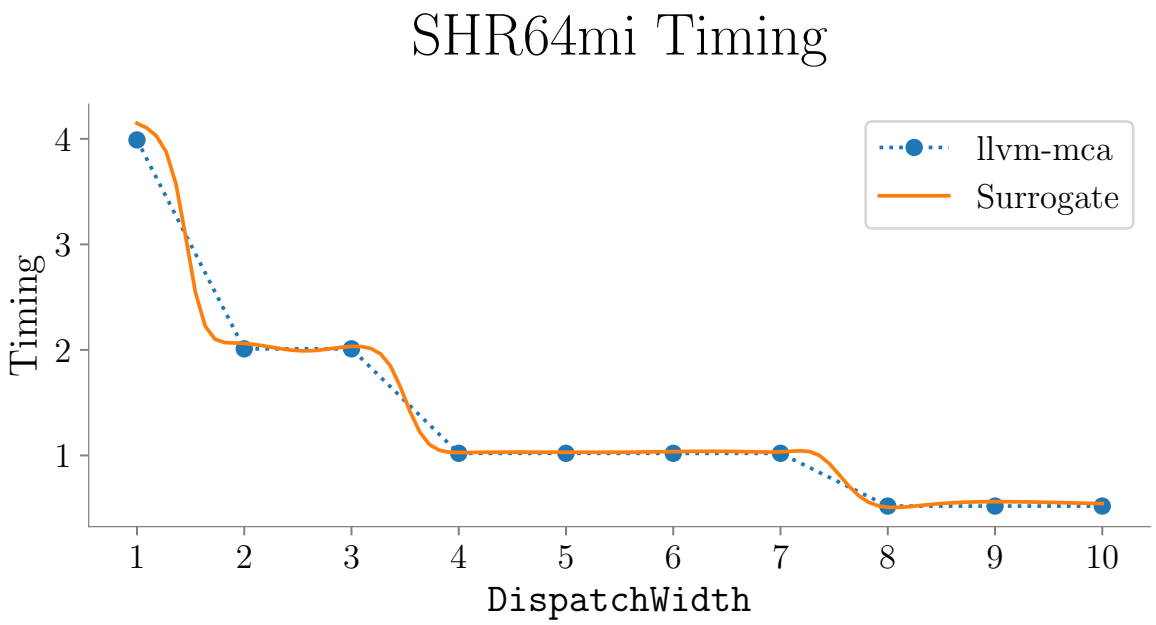


Figure 2.3: Example of timing predicted by llvm-mca (blue) and a surrogate (orange), while varying `DispatchWidth`. By learning the surrogate, DiffTune is able to optimize the parameter value with gradient descent, rather than requiring combinatorial search.

tables, which map discrete input elements to real-valued vectors; *LSTMs* (Hochreiter and Schmidhuber, 1997), which map input sequences of vectors to a single output vector; and *fully connected layers*, which are linear transformations on input vectors. By being composed of differentiable components, neural networks are end-to-end differentiable, so that they are able to be trained using gradient-based optimization. Specifically, neural networks are typically optimized with stochastic first-order optimizations like stochastic gradient descent (SGD) (Robbins and Monro, 1951), which repeatedly calculates the network's error on a small sample of the training dataset and then updates the network's parameters in the opposite of the direction of the gradient to minimize the error.

By selecting a neural network as $\hat{f}$'s representation, DiffTune is able to leverage $\hat{f}$'s differentiable nature not only to train $\hat{f}$ (solving the optimization problem posed in Equation (2.2)) but also to solve the optimization problem posed in Equation (2.3) with gradient-based opti-

mization. This stands in contrast to f which is, generally, non-differentiable and therefore does not permit the computation of its gradients.

Surrogate example. Figure 2.3 presents a visual example of DiffTune, showing an example of the timing predicted by llvm-mca (blue) and a trained surrogate of llvm-mca (orange). The x-axis of Figure 2.3 is the value of the `DispatchWidth` parameter, and the y-axis is the predicted timing of llvm-mca with that `DispatchWidth` for the basic block consisting of the single instruction `shrq $5, 16(%rsp)`. The blue points represent the prediction of llvm-mca when instantiated with different values for `DispatchWidth`. The naive approach of optimizing llvm-mca would be combinatorial search, since without a continuous and smooth surface to optimize, it is impossible to use standard first-order techniques. DiffTune instead first learns a surrogate of llvm-mca, represented by the orange line in Figure 2.3. This surrogate, though not exactly the same as llvm-mca, is smooth and differentiable. Importantly, the surrogate interpolates llvm-mca's predictions even in places where llvm-mca does not have a defined output, such as between the integer-valued parameter settings. Together, these properties mean that it is possible to optimize parameters against the surrogate with first-order techniques like gradient descent.

2.3 Implementation

This section discusses the implementation of DiffTune.

Parameters. In its application to llvm-mca and llvm_sim, DiffTune optimizes two types of parameters: *per-instruction parameters*, which are a uniform length vector of parameters associated with each individual instruction opcode (e.g. for llvm-mca, a vector containing `WriteLatency`, `NumMicroOps`, etc.); and *global parameters*, which are a vector of parameters that are associated with the overall simulator behavior (e.g. for llvm-mca, a vector containing the `DispatchWidth` and `ReorderBufferSize`). DiffTune further supports two types of

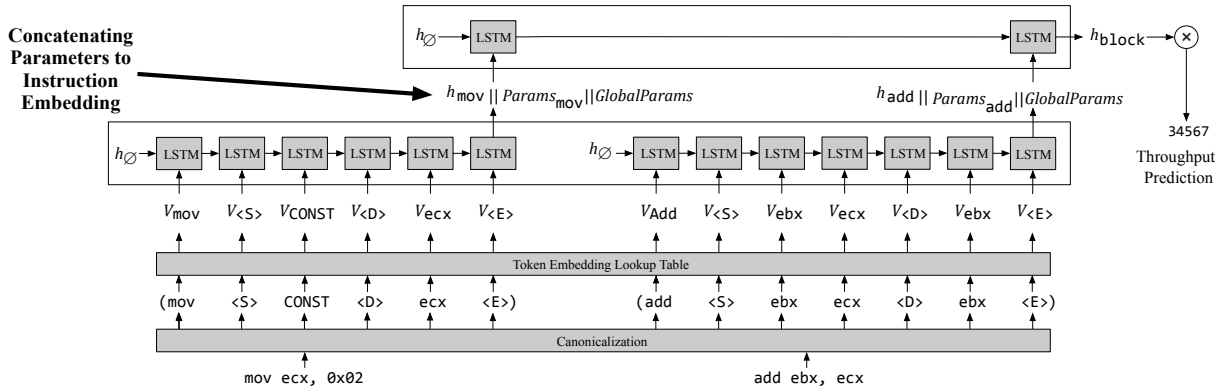


Figure 2.4: Design of the surrogate, from Mendis et al. (2019a) with added parameter inputs. I use || to denote concatenation of parameters to the instruction embedding.

constraints in its implementation: *lower-bounded*, specifying that parameter values cannot be below a certain value (often 0 or 1), and *integer-valued*, specifying that parameter values must be integers. During optimization, all parameters are represented as floating-point.

Surrogate design. Figure 2.4 presents the surrogate design, which is capable of learning parameters for x86 basic block performance models such as llvm-mca.

DiffTune uses a modified version of Ithemal (Mendis et al., 2019a), a neural network based basic block performance model, as the surrogate. In the standard implementation of Ithemal (without DiffTune's modifications), Ithemal first uses an embedding lookup table to map the opcode and operands of each instruction into vectors. Next, Ithemal processes the opcode and operand embeddings for each instruction with an LSTM, producing a vector representing each instruction. Then, Ithemal processes the sequence of instruction vectors with another LSTM, producing a vector representing the basic block. Finally, Ithemal uses a fully connected layer to turn the basic block vector into a single number representing Ithemal's prediction for the timing of that basic block.

DiffTune uses a version of Ithemal with two modifications to act as the surrogate. First, each individual LSTM is replaced with a set of 4 stacked LSTMs, a common technique to increase representative capacity (Hermans and Schrauwen, 2013), to give Ithemal the capacity to represent the dependency of the prediction on the input parameters as well as on the input

basic block.[3] Second, to provide the parameters as input, the per-instruction parameters and the global parameters are concatenated to each instruction vector before processing the instruction vectors with the instruction-level LSTM.

Solving the optimization problems. Training the surrogate requires first defining sampling distributions for each parameter (e.g., a bounded uniform distribution on integers). DiffTune then generates a large simulated dataset by repeatedly sampling a basic block from the ground-truth dataset, sampling a parameter table from the defined sampling distributions, instantiating the simulator with the parameter table, and generating a prediction for the basic block. DiffTune trains the surrogate using SGD against this simulated dataset. During surrogate training, for parameters constrained to be lower-bounded DiffTune subtracts the lower bound before passing them as input to the surrogate.

To train the parameter table, DiffTune first initializes it to a random sample from the parameter sampling distribution. DiffTune generates predictions using the parameter table plugged into the trained surrogate and trains the parameter table by using SGD against the ground-truth dataset. Importantly, when training the parameter table, the weights of the surrogate are not updated. During parameter table training, for parameters constrained to be lower-bounded DiffTune takes the absolute value of the parameters before passing them as input to the surrogate.

Parameter extraction. Once DiffTune has trained the surrogate and the parameter table using the optimization process described in Section 2.2, DiffTune extracts the parameters from the parameter table and uses them in the original simulator. For parameters with lower bounds, DiffTune takes the absolute value of the parameter in the learned parameter table, then adds the lower bound. For integer parameters, DiffTune rounds the learned parameter to the nearest integer. DiffTune does not use any special technique to handle unseen opcodes in the test set, just using the parameters for that opcode from the randomly initialized parameter table.

[3] A stack of 4 LSTMs resulted in the best validation error for the surrogate.

Table 2.1: Parameters learned for llvm-mca.

Parameter	Count	Constraint	Description
`DispatchWidth`	1 global	Integer ≥ 1	How many micro-ops can be dispatched each cycle in the dispatch stage.
`ReorderBufferSize`	1 global	Integer ≥ 1	How many micro-ops can fit in the reorder buffer in total.
`NumMicroOps`	1/instr.	Integer ≥ 1	How many micro-ops per instruction.
`WriteLatency`	1/instr.	Integer ≥ 0	The number of cycles before destination operands of that instruction can be read from. A latency value of 0 means that dependent instructions do not have to wait before being issued, and can be issued in the same cycle.
`ReadAdvanceCycles`	3/instr.	Integer ≥ 0	How much to decrease the effective `WriteLatency` of source operands.
`PortMap`	10/instr.	Integer ≥ 0	The number of cycles the instruction occupies each execution port for. Represented as a 10-element vector per-instruction, where element i is the number of cycles for which the instruction occupies port i.

2.4 Evaluation: llvm-mca

In this section, I report and analyze the results of using DiffTune to learn the parameters of llvm-mca across different x86 microarchitectures. I first describe the methodological details of the evaluation in Section 2.4.1. I then analyze the error of llvm-mca instantiated with the learned parameters, finding the following:

- DiffTune is able to learn parameters that lead to lower error than the default expert-tuned parameters across all four tested microarchitectures. (Section 2.4.2)
- Black-box global optimization with OpenTuner (Ansel et al., 2014) cannot find a full set of parameters for llvm-mca's Intel x86 simulation model that match llvm-mca's default error. (Section 2.4.3)

Table 2.2: Dataset summary statistics.

Statistic		Value
# Blocks		
	Train	230,111
	Validation	28,764
	Test	28,764
	Total	287,639
Block Length		
	Min	1
	Median	3
	Mean	4.93
	Max	256
Median Block Timing		
	Ivy Bridge	132
	Haswell	123
	Skylake	120
	Zen 2	114
# Unique Opcodes		
	Train	814
	Val	610
	Test	580
	Total	837

To show that the implementation of DiffTune is extensible to CPU simulators other than llvm-mca, I evaluate DiffTune on llvm_sim in Section 2.6.

2.4.1 Methodology

Following Chen et al. (2019), the evaluation uses llvm-mca version 8.0.1 (commit hash `19a71f6`). The evaluation specifically focuses on llvm-mca's Intel x86 simulation model: llvm-mca supports behavior beyond that described in Section 2.1 (e.g., optimizing zero idioms, constraining the number of physical registers available, etc.) but this behavior is disabled by default in the Intel microarchitectures evaluated in this chapter. DiffTune does not enable or learn any behavior not present in llvm-mca's default Intel x86 simulation model, including when evaluating on AMD.

Parameters in llvm-mca. For each microarchitecture, DiffTune learns the parameters specified in Table 2.1. Following the default value in llvm-mca for Haswell, DiffTune assumes that there are 10 execution ports available for dispatch for all microarchitectures. While llvm-mca supports simulation of instructions that can be dispatched to multiple different ports in the `PortMap` parameter, the simulation of port group parameters in the `PortMap` does not correspond to standard definitions of port groups (Di Biagio, 2020; Fog, 1996; Ritter and Hack, 2020). For this evaluation I therefore set all the port group parameters in the `PortMap` to zero, removing that component of the simulation.

Dataset. The evaluation uses the BHive dataset from Chen et al. (2019), which contains basic blocks sampled from a diverse set of applications (e.g., OpenBLAS, Redis, LLVM, etc.) along with the measured execution times of these basic blocks unrolled in a loop. These measurements are designed to conform to the same modeling assumptions made by llvm-mca.

The evaluation uses the latest available version of the released timings on GitHub.[4] I evaluate on the datasets released with BHive for the Intel x86 microarchitectures Ivy Bridge, Haswell, and Skylake. I also evaluate on AMD Zen 2, which was not included in the BHive dataset. The Zen 2 measurements were gathered by running a version of BHive modified to time basic blocks using AMD performance counters on an AMD EPYC 7402P, using the same methodology as Chen et al.. Following Chen et al., I remove all basic blocks potentially affected by virtual page aliasing.

I randomly split off 80% of the measurements into a training set, 10% into a validation set for development, and 10% into the test set reported in this chapter. I use the same train, validation, and test set split for all microarchitectures. The training and test sets are block-wise disjoint: there are no identical basic blocks between the training and test set. Summary statistics of the dataset are presented in Table 2.2.

[4] https://github.com/ithemal/bhive/tree/5878a18/benchmark/throughput

Objective. I use the same definition of timing as Chen et al. (2019): the number of cycles it takes to execute 100 iterations of the given basic block, divided by 100. Following Chen et al.'s definition of error, DiffTune optimizes llvm-mca to minimize the mean absolute percentage error (MAPE) against a dataset:

$$\text{Error} \triangleq \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \frac{|f(x) - y|}{y}$$

Note that an error of above 100% is possible when $f(x)$ is much larger than y.

Training methodology. The implementation of DiffTune uses PyTorch-1.2.0 on an NVIDIA Tesla V100 to train the surrogate and parameters. DiffTune trains the surrogate and the parameter table using Adam (Kingma and Ba, 2015), a stochastic first-order optimization technique, with a batch size of 256. DiffTune uses a learning rate of 0.001 to train the surrogate and a learning rate of 0.05 to train the parameter table.

To train the surrogate, DiffTune generates a simulated dataset of 2,301,110 blocks (10× the length of the original training set). For each basic block in the simulated dataset, DiffTune samples a random parameter table, with each `WriteLatency` a uniformly random integer between 0 and 5, each value in the `PortMap` uniform between 0 and 2 cycles to between 0 and 2 randomly selected ports for each instruction, each `ReadAdvanceCycles` between 0 and 5, each `NumMicroOps` between 1 and 10, the `DispatchWidth` uniform between 1 and 10, and the `ReorderBufferSize` uniform between 50 and 250 (all ranges inclusive). A random parameter table sampled from this distribution has error $171.4\% \pm 95.7\%$. See Section 2.8.1 for more discussion of these sampling distributions.

DiffTune loops over this simulated dataset 6 times when training the surrogate, totaling an equivalent of 60 epochs over the original training set. To train the parameter table, DiffTune initializes it to a random sample from the parameter training distribution, then trains it for 1 epoch against the original training set.

Table 2.3: Error of llvm-mca with the default and learned parameters against baselines.

Architecture	Predictor	Error	Kendall's Tau
Ivy Bridge	Default	33.5%	0.788
	DiffTune	25.4% ± 0.5%	0.735 ± 0.012
	Ithemal	9.4%	0.858
	IACA	15.7%	0.810
	OpenTuner	102.0%	0.515
Haswell	Default	25.0%	0.783
	DiffTune	23.7% ± 1.5%	0.745 ± 0.009
	Ithemal	9.2%	0.854
	IACA	17.1%	0.800
	OpenTuner	105.4%	0.522
Skylake	Default	26.7%	0.776
	DiffTune	23.0% ± 1.4%	0.748 ± 0.008
	Ithemal	9.3%	0.859
	IACA	14.3%	0.811
	OpenTuner	113.0%	0.516
Zen 2	Default	34.9%[5]	0.794
	DiffTune	26.1% ± 1.0%	0.689 ± 0.007
	Ithemal	9.4%	0.873
	IACA	N/A	N/A
	OpenTuner	131.3%	0.494

2.4.2 Error of Learned Parameters

Table 2.3 presents the average error and correlation of llvm-mca with the default parameters (labeled default), llvm-mca with the learned parameters (labeled DiffTune). As baselines, Table 2.3 also presents Ithemal's error, as the most accurate predictor evaluated by Chen et al.; IACA's error, as the most accurate analytical model evaluated by Chen et al.; and llvm-mca with parameters learned by OpenTuner (which I discuss further in Section 2.4.3). Because IACA is written by Intel to analyze Intel microarchitectures, it does not generate predictions for Zen 2. I report mean absolute percentage error, as defined in Section 2.4.1, and Kendall's Tau rank correlation coefficient, measuring the fraction of pairs of timing predictions

Table 2.4: Error of llvm-mca with default and learned parameters on Haswell, grouped by BHive applications and categories.

Block Type	**# Blocks**	**Default Error**	**Learned Error**
OpenBLAS	1478	28.8%	29.0%
Redis	839	41.2%	22.5%
SQLite	764	32.8%	21.6%
GZip	182	40.6%	20.6%
TensorFlow	6399	33.5%	22.1%
Clang/LLVM	18,781	22.0%	21.0%
Eigen	387	44.3%	23.8%
Embree	1067	34.1%	21.3%
FFmpeg	1516	30.9%	21.2%
Scalar (Scalar ALU operations)	7952	17.2%	18.9%
Vec (Purely vector instructions)	104	35.3%	39.6%
Scalar/Vec (Scalar and vector arithmetic)	614	53.6%	37.5%
Ld (Mostly loads)	10,850	27.2%	24.4%
St (Mostly stores)	4490	24.7%	8.7%
Ld/St (Mix of loads and stores)	4754	27.9%	30.3%

in the test set that are ordered correctly. For the learned parameters, I report the mean and standard deviation of error and Kendall's Tau across three independent runs of DiffTune.

Across all microarchitectures, the parameter set learned by DiffTune achieves equivalent or better error than the default parameter set. These results demonstrate that DiffTune can learn all of llvm-mca's microarchitecture-specific parameters, from scratch, to equivalent accuracy as the hand-written parameters.

I also analyze the error of llvm-mca on the Haswell microarchitecture using the evaluation metrics from Chen et al. (2019), designed to validate x86 basic block performance models. Chen et al. present three forms of error analysis: overall error, per-application error, and per-category error. Overall error is the error reported in Table 2.3. Per-application error is the average error of basic blocks grouped by the source application of the basic block (e.g., TensorFlow, SQLite, etc.; blocks can have multiple different source applications). Per-

[5]llvm-8.0.1 does not support Zen 2. This default error I report for Zen 2 uses the znver1 target in llvm-8.0.1, targeting Zen 1. The Zen 2 target in llvm-10.0.1 has a higher error of 39.8%.

category error is the average error of basic blocks grouped into clusters based on the hardware resources used by each basic block.

The per-application and per-category errors are presented in Table 2.4. The learned parameters outperform the defaults across most source applications, with the exception of OpenBLAS where the learned parameters result in 0.2% higher error. The learned parameters perform similarly to the default across most categories, with the primary exceptions of the Scalar/Vec category and the St category, in which the learned parameters perform significantly better than the default parameters.

2.4.3 Black-box global optimization with OpenTuner

In this section, I describe the methodology and performance of using black-box global optimization with OpenTuner (Ansel et al., 2014) to find parameters for llvm-mca. I find that OpenTuner is not able to find parameters that lead to equivalent error as DiffTune in llvm-mca's Intel x86 simulation model.

Background. I use OpenTuner as a representative example of a black-box global optimization technique. OpenTuner is primarily a system for tuning parameters of programs to decrease run-time (e.g., tuning compiler flags, etc.), but has also been validated on other optimization problems, such as finding the series of button presses in a video game simulator that makes the most progress in the game.

OpenTuner is an iterative algorithm that uses a multi-armed bandit to pick the most promising search technique among an ensemble of search techniques that span both convex and non-convex optimization: by default, OpenTuner uses "greedy mutation, differential evolution, and two hill climber instances" (Ansel et al., 2014). On each iteration, the bandit evaluates the current set of parameters. Using the results of that evaluation, the bandit then selects a search technique that then proposes a new set of parameters.

Methodology. For computational budget parity with DiffTune, I permit OpenTuner to evaluate the same number of basic blocks as used end-to-end in the learning approach. I initialize OpenTuner with a sample from DiffTune's parameter table sampling distribution. I constrain OpenTuner to search per-instruction (`NumMicroOps`, `WriteLatency`, `ReadAdvanceCycles`, `PortMap`) parameter values between 0 and 5, `DispatchWidth` between 1 and 10, and `ReorderBufferSize` between 50 and 250; these ranges contain the majority of the parameter values observed in the default and learned parameter sets.[6] I evaluate the accuracy of llvm-mca with the resulting set of parameters using the same methodology as in Section 2.4.2.

Results. Table 2.3 presents the error of llvm-mca when parameterized with OpenTuner's discovered parameters. OpenTuner's parameters perform worse than those of DiffTune, resulting in error above 100% across all microarchitectures. Thus, DiffTune requires substantially fewer examples to optimize llvm-mca than OpenTuner requires.

2.5 Analysis

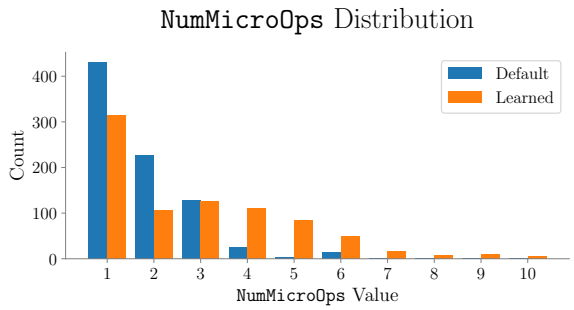
In this section, I analyze the parameters learned by DiffTune on llvm-mca, answering the following research questions:

- How similar are the learned parameters to the default parameters in llvm-mca?
- How optimal are the learned parameters?
- How semantically meaningful are the learned parameters?

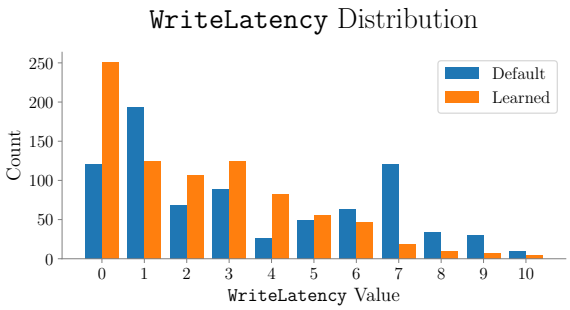
2.5.1 Comparison of Learned Parameters to Defaults

This section compares the default parameters to the learned parameters in Haswell.

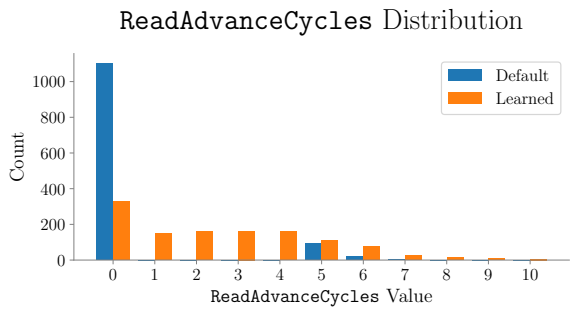
[6] Widening the search space beyond this range resulted in a significantly higher error for OpenTuner.



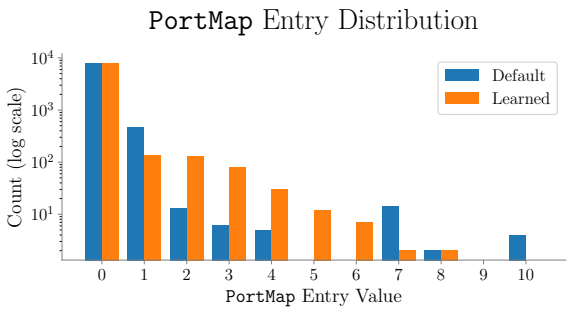
(a) Distribution of default and learned `NumMicroOps` values.



(b) Distribution of default and learned `WriteLatency` values.



(c) Distribution of default and learned `ReadAdvanceCycles` values.



(d) Distribution of default and learned `PortMap` values.

Figure 2.5: Distributions of default and learned parameter values on Haswell.

Table 2.5: Default and learned global parameters.

Architecture	Parameters	DispatchWidth	ReorderBufferSize
Haswell	Default	4	192
	Learned	4	144

Distributional similarities. To determine the distributional similarity of the learned parameters to the default parameters, Figure 2.5 shows histograms of the values of the default and learned per-instruction parameters (`NumMicroOps`, `WriteLatency`, `ReadAdvanceCycles`, `PortMap`). The primary distinctions between the distributions are in `WriteLatency` and `ReadAdvanceCycles`; the learned parameters otherwise follow similar distributions to the defaults.

The distributions of default and learned `WriteLatency` values in Figure 2.5b primarily differ in that only 1 out of the 837 opcodes in the default Haswell parameters has `WriteLatency` 0 (`VZEROUPPER`), whereas 251 out of the 837 opcodes in the learned parameters have `WriteLatency` 0. As discussed in Table 2.1, a `WriteLatency` value of 0 means that dependent instructions need not wait before being issued, and can be issued in the same cycle; instructions may still be bottlenecked elsewhere in the simulation pipeline (e.g., in the execute stage).

The distributions of default and learned `ReadAdvanceCycles` are presented in Figure 2.5c. The default `ReadAdvanceCycles` are mostly 0, with a small population having values 5 and 7; in contrast, the learned `ReadAdvanceCycles` are fairly evenly distributed, with a plurality being 0. As noted in Section 2.1, llvm-mca subtracts `ReadAdvanceCycles` from `WriteLatency` to compute a dependency chain's latency. The result of this subtraction is clipped to be no less than zero; thus, when the `WriteLatency` of a dependent instruction is 0, the `ReadAdvanceCycles` is irrelevant. This may explain why the learned `ReadAdvanceCycles` cycles, which are randomly initialized uniformly between 0 and 5, maintain a near uniform distribution in the learned parameters.

Global parameters. Table 2.5 shows the default and learned global parameters (`DispatchWidth` and `ReorderBufferSize`). The learned `DispatchWidth` parameter is close to the

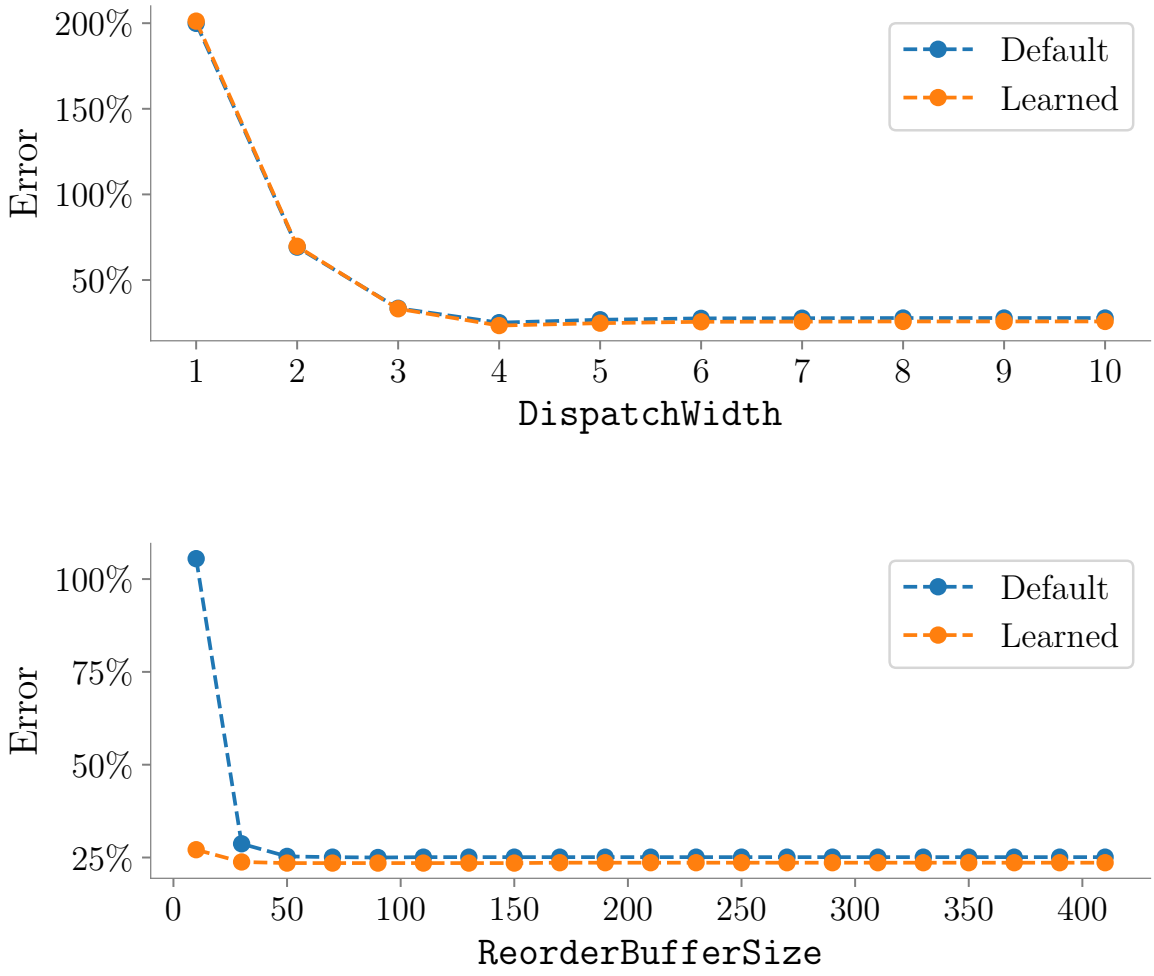


Figure 2.6: The sensitivity to values of `DispatchWidth` (Top) and `ReorderBufferSize` (Bottom) within the default (Blue) and learned (Orange) parameters in llvm-mca.

default `DispatchWidth` parameter, while the learned `ReorderBufferSize` parameter differs significantly from the default. By analyzing llvm-mca's sensitivity to values of `DispatchWidth` and `ReorderBufferSize` within the default and learned parameters in Figure 2.6, I find that although the learned global parameters do not match the default values exactly, they approximately minimize llvm-mca's error because there is a wide range of values that result in approximately the same error.

While llvm-mca is sensitive to small perturbations in the value of the `DispatchWidth` parameter (with the default parameters, a `DispatchWidth` of 3 has error 33.5%, 4 has error 25.0%, and 5 has error 26.8%), it is relatively insensitive to perturbations of the `ReorderBufferSize` (with the default parameters, all `ReorderBufferSize` values above 70 have error 25.0%). This is primarily because one of llvm-mca's core modeling assumptions, that memory accesses always resolve in the L1 cache, means that most instructions spend few cycles in the issue, execute, and retire phases; the `ReorderBufferSize` is therefore rarely a bottleneck in llvm-mca's modeling of the BHive dataset.

2.5.2 Optimality

This section shows that while the parameters learned by DiffTune match the error of the default parameters, the learned values are not optimal: by optimizing just a subset of llvm-mca's parameters, and keeping the rest as their expert-tuned default values, DiffTune is able to find parameters with lower error than when learning the entire set of parameters.

Experiment. I use DiffTune to learn only each instruction's `WriteLatency` in llvm-mca, keeping all other parameters as their default values. The dataset and objective used in this task are otherwise the same as presented in Section 2.4.1.

Methodology. Training hyperparameters are similar to those presented in Section 2.4.1, and are reiterated here with modifications made to learn just `WriteLatency` parameters. DiffTune trains both the surrogate and the parameter table using Adam (Kingma and Ba, 2015)

with a batch size of 256. DiffTune uses a learning rate of 0.001 to train the surrogate and a learning rate of 0.05 to train the parameter table. To train the surrogate, DiffTune generates a simulated dataset of 2,301,110 blocks. For each basic block in the simulated dataset, DiffTune samples a random parameter table, with each `WriteLatency` a uniformly random integer between 0 and 10 (inclusive). DiffTune loops over this simulated dataset 3 times when training the surrogate. To train the parameter table, DiffTune initializes it to a random sample from the parameter training distribution, then trains it for 1 epoch against the original training set.

Results. On Haswell, this application of DiffTune results in an error of 16.2% and a Kendall Tau correlation coefficient of 0.823, compared to an error of 23.7% and a correlation of 0.745 when learning the full set of parameters with DiffTune. These results demonstrate that DiffTune does not find a globally optimal parameter set when learning llvm-mca's full set of parameters. This suboptimality is due in part to the non-convex nature of the problem and the size of the parameter space.

2.5.3 Case Studies

This section presents case studies of basic blocks simulated with the default and with the learned parameters, showing where the learned parameters better reflect the ground truth data, and where the learned parameters reflect degenerate cases of the optimization algorithm. To simplify exposition, the results in this section use just the learned `WriteLatency` values from the experiment in Section 2.5.2.

`PUSH64r`. The default `WriteLatency` with the Haswell parameters for the `PUSH64r` opcode (push a 64-bit register onto the stack, decrementing the stack pointer) is 2 cycles. In contrast, the `WriteLatency` learned by DiffTune is 0 cycles. This leads to significantly more accurate predictions for blocks that contain `PUSH64r` opcodes, such as the following (in which the default and learned latency for `testl` are both 1 cycle):

```
pushq       %rbx
testl       %r8d, %r8d
```

The true timing of this block as measured by Chen et al. (2019) is 1.01 cycles. On this block, llvm-mca with the default Haswell parameters predicts a timing of 2.03 cycles: The `PUSH64r` forms a dependency chain with itself, so the default `WriteLatency` before each `PUSH64r` can be dispatched is 2 cycles. In contrast, llvm-mca with the learned Haswell values predicts that the timing is 1.03 cycles, because the learned `WriteLatency` is 0 meaning that there is no delay before the following `PUSH64r` can be issued, but the `PortMap` for `PUSH64r` still occupies `HWPort4` for a cycle before the instruction is retired; this 1-cycle dependency chain results in a more accurate prediction. In this case, DiffTune learns a `WriteLatency` that leads to better accuracy for the `PUSH64r` opcode.

`XOR32rr`. The default `WriteLatency` in Haswell for the `XOR32rr` opcode (xor two registers with each other) is 1 cycle. The `WriteLatency` learned by DiffTune is again 0 cycles. This is not always correct – however, a common use of `XOR32rr` is as a *zero idiom*, an instruction that sets a register to zero. For example, `xorq %rax, %rax` performs an `xor` of `%rax` with itself, effectively setting `%rax` to zero. Intel processors have a fast path for zero idioms – rather than actually computing the `xor`, they simply set the value to zero. Most of the instances of `XOR32rr` in the dataset (4047 out of 4218) are zero idioms. This leads to more accurate predictions in the general case, as can be seen in the following example:

```
xorl        %r13d, %r13d
```

The true timing of this block is 0.31 cycles. With the default `WriteLatency` value of 1, the Intel x86 simulation model of llvm-mca does not recognize this as a zero idiom and predicts that this block has a timing of 1.03 cycles. With the learned `WriteLatency` value of 0 and the fact that there are no bottlenecks specified by the `PortMap`, llvm-mca executes the `xor`s as quickly as possible, bottlenecked only by the `NumMicroOps` of 1 and the `Dispatch-`

`Width` of 4. With this change, llvm-mca predicts that this block has a timing of 0.27 cycles, again closer to the ground truth.

`ADD32mr`. Unfortunately, it is impossible to distinguish between semantically meaningful values that make the simulator more correct, and degenerate values that improve the accuracy of the simulator without adding interpretability. For instance, consider `ADD32mr`, which adds a register to a value in memory and writes the result back to memory:

```
addl        %eax, 16(%rsp)
```

This block has a true timing of 5.97 cycles because it is essentially a chained load, add, then store—with the L1 cache latency being 4 cycles. However, llvm-mca does not recognize the dependency chain this instruction forms with itself, so even with the default Haswell `WriteLatency` of 7 cycles for `ADD32mr`, llvm-mca predicts that this block has an overall timing of 1.09 cycles. DiffTune is able to learn parameters to lead llvm-mca to predict a higher timing, but is fundamentally unable to change a parameter in llvm-mca to enable llvm-mca to recognize the dependency chain (because no such parameter exists). Instead, the methodology learns a degenerately high `WriteLatency` of 62 for this instruction, allowing llvm-mca to predict an overall timing of 1.64 cycles, closer to the true value. This degenerate value increases the accuracy of llvm-mca without leading to semantically useful `WriteLatency` parameters. This case study shows that the interpretability of the learned parameters is only as good as the simulation fidelity; when the simulation is a poor approximation to the physical behavior of the CPU, the learned parameters do not correspond to semantically meaningful values.

2.6 Evaluation: llvm_sim

To evaluate that the implementation of DiffTune (Section 2.3) is extensible to simulators other than llvm-mca, I evaluate the implementation on llvm_sim (Sykora et al., 2018), learning all parameters that llvm_sim reads from LLVM. This simulation uses many of the same

Table 2.6: Parameters learned for llvm_sim.

Parameter	Count	Constraint	Description
`WriteLatency`	1 per-instruction	Integer, ≥ 0	The number of cycles before destination operands of that instruction can be read from.
`PortMap`	10 per-instruction	Integer, ≥ 0	The number of micro-ops dispatched to each port.

Table 2.7: Learning all parameters: error of llvm_sim with the default and learned parameters.

Architecture	Predictor	Error	Kendall's Tau
Haswell	Default	61.3%	0.7256
	DiffTune	44.1%	0.718
	Ithemal	9.2%	0.854
	IACA	17.1%	0.800
	OpenTuner	115.6%	0.507

parameters (from LLVM's backend) as llvm-mca, but uses a different model of the CPU, modeling the frontend and breaking up instructions into micro-ops and simulating the micro-ops individually rather than simulating instructions as a whole as llvm-mca does.

Behavior. Similar to llvm-mca, llvm_sim (Sykora et al., 2018) is also an out-of-order superscalar simulator exposing LLVM's instruction scheduling model, predicting timings of basic blocks assuming that all data is in the L1 cache. There are three primary ways in which llvm_sim differs from llvm-mca: It models the front-end, it decodes instructions into micro-ops before dispatch and execution, and it is only implemented for the x86 Haswell microarchitecture The llvm_sim simulation has the following pipeline:

- Instructions are fetched, parsed, and decoded into micro-ops (unlike llvm-mca, llvm_sim does model the frontend)
- Registers are renamed, with an unlimited number of physical registers
- Micro-ops are dispatched out-of-order once dependencies are available

- Micro-ops are executed on execution ports
- Instructions are retired once all micro-ops in an instruction have been executed

Parameters. DiffTune learns the parameters specified in Table 2.6. I again assume that there are 10 execution ports available to dispatch for all microarchitectures and do not learn to dispatch to port groups. All other hyperparameters are identical to those in Section 2.4.1.

Results. Table 2.7 presents the average error and correlation of llvm_sim with the default parameters, llvm_sim with the learned parameters, Ithemal trained on the dataset as a lower bound, and the OpenTuner (Ansel et al., 2014) baseline. Learning the parameters that llvm_sim reads from LLVM reduces llvm_sim's error from 61.3% to 44.1%.

2.7 Related Work

Simulators are widely used for architecture research to model the interactions of architectural components of a system (Binkert et al., 2011; Di Biagio and Davis, 2018; Yourst, 2007; Patel et al., 2011; Sanchez and Kozyrakis, 2013). Configuring and validating CPU simulators to accurately model systems of interest is a challenging task (Chen et al., 2019; Gutierrez et al., 2014; Akram and Sawalha, 2019).

Measurement. One methodology for setting the parameters of an abstract model is to gather fine-grained measurements of each individual parameter's realization in the physical machine (Fog, 1996; Abel and Reineke, 2019) and then set the parameters to their measured values (Colombet, 2014; Topper, 2014). When the semantics of the simulator and the semantics of the measurement methodology coincide, then these measurements can serve as effective parameter values. However, if there is a mismatch between the simulator and measurement methodology, then measurements may not provide effective parameter settings.

All fine-grained measurement frameworks rely on accurate hardware performance counters to measure the parameters of interest. Such performance counters do not always exist, such as with per-port measurement performance counters on AMD Zen (Ritter and Hack, 2020). When such performance counters are present, they are not always reliable (Weaver and McKee, 2008).

Optimizing CPU simulators. Another methodology for setting parameters of an abstract model is to infer the parameters from end-to-end measurements of the performance of the physical machine. In the most related effort in this space, Ritter and Hack (2020) present a framework for inferring port usage of instructions based on optimizing against a CPU model that solves a linear program to predict the throughput of a basic block. Their approach is specifically designed to infer port mappings and it is not clear how the approach could be extended to infer different parameters in a more complex simulator, such as extending their simulation model to include data dependencies, dispatch width, or reorder buffer size. DiffTune is the first approach designed to optimize an arbitrary simulator, provided that the simulator and its parameters match DiffTune's scope of applicability (Section 2.8.1).

CPU simulator surrogates. İpek et al. (2006) use neural networks to learn to predict the instructions-per-cycle metric (which is equivalent to llvm-mca's throughput metric) of a cycle-accurate simulator given a set of design space parameters, to enable efficient design space exploration. Lee and Brooks (2007) use regression models to predict the performance and power usage of a CPU simulator, similarly enabling efficient design space exploration. Neither İpek et al. nor Lee and Brooks then use the models to optimize the simulator to be more accurate; both also apply exhaustive or grid search to explore the parameter space, rather than using the gradient of the simulator surrogate.

Differentiating arbitrary programs. Chaudhuri and Solar-Lezama (2010) present a method to approximate numerical programs by executing programs probabilistically, similar to the idea of blurring an image. This approach lets Chaudhuri and Solar-Lezama apply gradient

descent to parameters of arbitrary numerical programs. However, the semantics presented by Chaudhuri and Solar-Lezama only apply to a limited set of program constructs and do not easily extend to the set of program constructs exhibited by large-scale CPU simulators.

2.8 Discussion

CPU simulators are complex software artifacts that require significant measurement and manual tuning to set their parameters. I present DiffTune, a surrogate optimization algorithm for learning parameters within non-differentiable programs, using only end-to-end supervision. My results demonstrate that DiffTune is able to learn the entire set of 11,265 microarchitecture-specific parameters from scratch in llvm-mca.

2.8.1 Limitations and Future Directions

DiffTune is an effective technique to learn simulator parameters, as I demonstrate with llvm-mca (Section 2.4) and llvm_sim (Section 2.6). However, there are several aspects of DiffTune's approach that are designed around the fact that llvm-mca and llvm_sim are basic block simulators that are primarily parameterized by ordinal parameters with few constraints between the values of individual parameters. I believe that DiffTune's overall approach—surrogate optimization—can be extended to whole program and full system simulators that have richer parameter spaces, such as gem5, by extending a subset of DiffTune's individual components.

Whole program and full system simulation. DiffTune requires a differentiable surrogate that can learn the simulator's behavior to high accuracy. Ithemal (Mendis et al., 2019a)—the model I use for the surrogate—operates on basic blocks with the assumption that all data accesses resolve in the L1 cache, which is compatible with the evaluation of llvm-mca and llvm_sim (which make the same assumptions). While Ithemal could potentially model whole

programs and full systems (e.g., branching, caching) with limited modifications, it may require extensions to learn such more complex behavior (Vila et al., 2020; Hashemi et al., 2018).

In addition to the design of the surrogate, training the surrogate would require a new dataset that includes whole programs, along with any other behavior modeled by the simulator being optimized (e.g., memory). Acquiring such a dataset would require extending timing methodologies like BHive (Chen et al., 2019) to the full scope of target behavior.

Non-ordinal parameters. DiffTune only supports ordinal parameters and does not support categorical or boolean parameters. DiffTune requires a relaxation of discrete parameters to continuous values to perform optimization, along with a method to extract the learned relaxation back into the discrete parameter type (e.g., DiffTune relaxes integers to real numbers, and extracts the learned parameters by rounding back to integers). Supporting categorical and boolean parameters would require designing and evaluating a scheme to represent and extract such parameters within DiffTune. One candidate representation is one-hot encoding, but has not been evaluated in DiffTune.

Dependent parameters. All integers in the range $[1, \infty)$ are valid settings for llvm-mca's parameters. However, other simulators, such as gem5, have stricter conditions—expressed as assertions in the simulator—on the relationship among different parameters.[7] DiffTune also does not apply when there is a variable number of parameters: DiffTune is able to learn the port mappings in a fixed-size `PortMap`, but does not learn the number of ports in the `PortMap`, instead fixing it at 10 (the default value for the Haswell microarchitecture). Extending DiffTune to optimize simulators with constrained relationships between parameters motivates new work in efficient techniques to sample such sets of parameters (Dutra et al., 2018).

[7] For an example, see https://github.com/gem5/gem5/blob/v20.0.0.0/src/cpu/o3/decode_impl.hh#L423, which is based on the interaction between different parameters, defined at https://github.com/gem5/gem5/blob/v20.0.0.0/src/cpu/o3/decode_impl.hh#L75.

Sampling distributions. Extending DiffTune to other simulators also requires defining appropriate sampling distributions for each parameter. While the sampling distributions do not have to directly lead to parameter settings that lead the simulator to have low error (e.g., the sampling distributions defined in Section 2.4.1 lead to an average error of llvm-mca on Haswell of $171.4\% \pm 95.7\%$), they do need to contain values that the parameter table estimate may take on during the parameter table optimization phase (because neural networks like DiffTune's modification of Ithemal are not guaranteed to be able to accurately extrapolate outside of their training distribution). Other approaches to surrogate optimization handle this by continuously re-optimizing the surrogate in a region around the current parameter estimate (Shirobokov et al., 2020), a promising direction that could alleviate the need to hand-specify proper sampling distributions.

2.8.2 Conclusion

Looking beyond CPU simulation, DiffTune's approach offers the promise of a generic, scalable methodology to learn the parameters of programs using only input-output examples, potentially reducing many programming tasks to simply that of gathering data. Together, these results validate the hypothesis that surrogate programming can achieve better performance than other techniques on large-scale programming tasks involving complex systems.

Chapter 3

Design Patterns and Methodologies

DiffTune (Chapter 2) demonstrates the opportunity to use surrogate programming to achieve state-of-the-art results on large-scale programming tasks. However, like prior instances of surrogate programming in the literature (İpek et al., 2006; Esmaeilzadeh et al., 2012; Tercan et al., 2018; She et al., 2019; Tseng et al., 2019; Renda et al., 2020) its methodology is presented as ad-hoc – that is, design decisions including the overall optimization process that constitutes DiffTune, along with concrete details like the surrogate design, training, and deployment, are all made without guidance from any overarching methodology.

I argue that we can unify these disparate approaches under a single programming methodology, giving a framework for programmers to reason about and develop surrogates. To demonstrate this, I contribute a taxonomy that classifies the workflows demonstrated in the surrogate programming literature into the three different design patterns first discussed in Section 1.1: *surrogate compilation*, *surrogate adaptation*, and *surrogate optimization*. While Section 1.1 demonstrated them on a toy simulator, I concretize these design patterns by demonstrating how to use each to solve one of three development tasks for llvm-mca (Di Biagio and Davis, 2018), the 10,000 line-of-code CPU simulator studied in Chapter 2. I further discuss the shared programming methodology underlying the three surrogate programming design patterns.

Surrogate compilation. With surrogate compilation, programmers train a surrogate that replicates the behavior of a program to deploy in its place. Key benefits of this approach include the ability to execute the surrogate on different hardware and the ability to bound or to accelerate the execution time of the surrogate (Esmaeilzadeh et al., 2012; Mendis, 2020).

For llvm-mca, I train a neural network to replicate llvm-mca's prediction of the execution time for a given input code snippet. The resulting neural network executes 1.6× faster than llvm-mca on the same hardware, with less than a 10% deviation from llvm-mca's predictions.

Surrogate adaptation. With surrogate adaptation, programmers first train a surrogate of a program then continue to train the surrogate on a downstream task. Key benefits of this approach include that surrogate adaptation makes it possible to perform a different task of interest in a way that is more data-efficient or results in higher accuracy than training a model from scratch for the task (Tercan et al., 2018; Kustowski et al., 2020).

I train a neural network to replicate llvm-mca's predictions then fine-tune that network on measurements of code timing on a physical CPU. This network has as low as 50% of the error of llvm-mca at predicting the ground-truth timings.

Surrogate optimization. With surrogate optimization, programmers train a surrogate of a program then use the surrogate to optimize the program's inputs. The key benefit of this approach is that surrogate optimization can optimize inputs faster than optimizing inputs directly against the program, due to the potential for faster execution speed of the surrogate and the potential for the surrogate to be differentiable even when the original program is not (allowing for optimizing inputs with gradient descent) (Tseng et al., 2019; She et al., 2019).

DiffTune is an example of this design pattern. With DiffTune, I train a neural network to replicate llvm-mca's prediction when llvm-mca is parameterized with different sets of simulation parameters, then optimize against that network to find parameters that lead the network to accurately predict ground-truth timings. I then plug these parameters back into llvm-mca, improving accuracy by 5% relative to expert-selected parameters.

Programming Methodology. The development methodologies common to these surrogate-based design patterns when instantiated with neural networks induce what I term the *neural surrogate programming methodology*, consisting of the *specification* of the task, the *design* of the neural network architecture, the *training* process for the network, and the *deployment* of the system. I present the programming methodology as a set of questions that guide development of the surrogate. A complete set of answers to these questions constitutes a concrete plan for the development and deployment of a neural surrogate.

Surrogates are constructed from input-output examples, meaning that their development methodology is the same as that of any other machine learning technique. I present key insights related to the fact that I study surrogates of programs with known structure and behavior (e.g., how to select a neural network architecture that can represent the original program with high accuracy). I also present insights that arise from the fact that surrogate development is itself a form of programming, constructing a function to meet a correctness specification while trading off among other objectives (e.g., how to minimize execution costs of the surrogate while satisfying an accuracy constraint).

Contributions. In this chapter I present the following contributions:

- I provide case studies of each design pattern of surrogate programming (surrogate compilation, surrogate adaptation, and surrogate optimization) on llvm-mca.
- I formally define each design pattern. I demonstrate that this taxonomy captures examples of surrogate programming from the literature.
- I identify elements of the neural surrogate programming methodology in the form of specifications and design questions that unify these surrogate-based design patterns. I discuss answers to each of these design questions, showing the trade-offs that programmers must consider when developing neural surrogates.

- I lay out directions towards the goal of further systematizing the programming methodology underlying surrogate programming.

Together, these contributions validate the hypothesis that surrogate programming is a coherent set of design patterns with a shared underlying methodology. This serves as a foundation for the remainder of my thesis in which I advance the state of the art in the methodologies of surrogate programming.

3.1 Case Study

To motivate the benefits and challenges of surrogate programming and to demonstrate the opportunity for a unifying surrogate programming methodology, I first demonstrate how developing surrogates of a CPU simulator (llvm-mca) makes it possible to solve three development tasks: (1) increasing the speed of the simulation, (2) simulating the execution behavior of a real-world processor that is not well-modeled by the simulator, and (3) finding simulation parameters that lead the simulator to accurate simulation of the behavior of a real-world processor. I present a surrogate optimization case study from my prior work (Chapter 2, based on Renda et al., 2020) and present two new case studies of surrogate compilation and surrogate adaptation on the same simulator under study.

3.1.1 Program Under Study

The case study focuses on llvm-mca (Di Biagio and Davis, 2018), the CPU simulator included in the LLVM compiler infrastructure (Lattner and Adve, 2004) for which I developed DiffTune in Chapter 2. I describe llvm-mca in detail in Section 2.1.1, but briefly recall key details here.

The llvm-mca system is a C++ program implemented as part of the LLVM compiler infrastructure, comprised of around 10,000 lines of code. The CPU parameters are comprised of 11,265 integer-valued parameters, inducing a configuration space with $10^{19,336}$ possible

configurations. LLVM contains expert-set CPU parameter settings for llvm-mca that target common x86 hardware architectures.

Validation and accuracy. As with DiffTune (Chapter 2), I validate the accuracy of llvm-mca's throughput predictions with BHive, a dataset of x86 basic blocks from a variety of end-user programs (Chen et al., 2019). I again define the mean absolute percentage error (MAPE) of llvm-mca's throughput predictions as the normalized difference between llvm-mca's output y_{pred} and the ground-truth measured throughput y_{true}:

$$\text{err}(y_{\text{pred}}, y_{\text{true}}) \triangleq \frac{|y_{\text{pred}} - y_{\text{true}}|}{y_{\text{true}}}$$

Across basic blocks in the BHive dataset and the CPU platforms that llvm-mca has expert-set parameters for, llvm-mca has a mean absolute percentage error of around 25%.

3.1.2 Surrogate Compilation

As described in Chapter 2, to quickly generate throughput predictions for basic blocks programmers must develop fast CPU simulation models. The standard approach, used by llvm-mca, is to manually implement a fast and sufficiently accurate simulation model, then use compiler optimizations to accelerate the *execution speed* of the simulation code. I define llvm-mca's execution speed as the number of basic blocks per second that llvm-mca is able to generate throughput predictions for.

Other approaches in the literature for accelerating llvm-mca's execution speed include rewriting the simulation software to be faster (Hager and Wellein, 2010) and applying compiler optimizations not included in llvm-mca's default compiler's optimization set, such as superoptimization (Massalin, 1987; Schkufza et al., 2013).

Surrogate compilation. An alternative approach for accelerating llvm-mca is surrogate compilation. With surrogate compilation, programmers develop a surrogate that replicates the behavior of a program to deploy to end-users in place of the original program.

Results. When instantiated with its default set of Haswell CPU parameters, llvm-mca's execution speed is 1742 blocks per second.[1] Surrogate compilation results in a neural surrogate of llvm-mca that has an execution speed of 2820 blocks per second on the same hardware, a speedup of 1.6× over llvm-mca. This surrogate has a mean absolute percentage error (MAPE) of 9.1% compared to llvm-mca's predictions. Against BHive's ground-truth measured data on a real Haswell CPU, the surrogate has an error rate of 27.1%. This error rate is higher (though not quite 9.1% higher) than llvm-mca's error rate of 25.0%.

Programming Methodology

Developing the neural surrogate for surrogate compilation requires thinking about the *specification* of the task, the *design* of the neural network architecture, the *training* process for the neural network, and the *deployment* considerations of the system. I collect these concerns into what I term the neural surrogate programming methodology.

Specification. The primary concern with any programming task is its specification. In the surrogate programming methodology, the specification comes in the form of an optimization problem with an objective and constraints.

The specification for the surrogate in this example is to maximize the execution speed of the surrogate while also constraining the error of the surrogate compared to llvm-mca to be less than 10% as measured by the MAPE. I formalize this using the notation of s as the

[1] All experiments were performed on a Google Cloud Platform `c2-standard-4` instance, using a single core of an Intel Xeon Skylake CPU at 3.1 GHz. I compile llvm-mca in release mode from version 8.0.1, using the version at https://github.com/ithemal/DiffTune/tree/9992f69/llvm-mca-parametric. I invoke llvm-mca a single time and pass it a random sample of 10,000 basic blocks from BHive over stdin. The reported execution speed is the time from invocation to exit of llvm-mca.

surrogate, $\mathcal{D}$ as the dataset of basic blocks x from BHive, p as llvm-mca, and haswell-params as LLVM's default set of Haswell CPU parameters:

$$s^* = \arg\max_s \text{ execution-speed}(s) \text{ such that } \mathop{\mathbb{E}}_{x\sim\mathcal{D}}\left[\frac{|s(x) - p(x, \text{haswell-params})|}{p(x, \text{haswell-params})}\right] \leq 10\%$$

The remainder of this case study walks through the neural surrogate programming methodology, presented as a set of design questions that guide the design, training, and deployment process of the neural surrogate.

Design. When developing a neural surrogate for a given task, the programmer must choose an architecture for the neural network underlying the surrogate, as well as scale the network's capacity appropriately. These choices must be informed by the specification of the surrogate and by the semantics of the program that the surrogate models.

In this example the neural network architecture and capacity must be the network with the highest execution speed that meets the accuracy constraint.

Question 1: *What neural network architecture topology does the surrogate use?*

The neural network architecture topology is the connection pattern of the neurons in the neural network (Goodfellow et al., 2016). The topology determines the types of inputs that the network can process (e.g., fixed-size inputs or arbitrary length sequences) and the *inductive biases* of the network, the assumptions about the task that are baked into the neural network. The topology also determines the computational cost, power, and scalability of the network.

I use a BERT encoder (Devlin et al., 2019), a type of Transformer (Vaswani et al., 2017), as the neural network topology for surrogate compilation of llvm-mca. Though many architectures could provide an acceptable solution to the task, I select and evaluate BERT due to its popularity (Rogers et al., 2020), expressive power (Yun et al., 2020), and relative ease of use (Wolf et al., 2020) for arbitrary sequence modeling tasks (though programmers should in general choose the most appropriate neural network architecture to model the

Table 3.1: The validation error and speedup of BERT models over a range of candidate embedding widths. The MAPE is the best MAPE observed on the validation set over the course of training. The speedup is the speedup relative to the default BERT-Tiny (W=128). An embedding width of 64 results in the fastest BERT model that achieves less than 10% validation MAPE.

Embedding Width	**MAPE**	**Speedup over W=128**
128	8.9%	1×
64	**9.5%**	**1.57×**
32	10.1%	2.01×
16	10.8%	2.22×

program depending on the domain). This BERT architecture processes raw Intel-syntax x86 basic blocks as input and predicts llvm-mca's throughput prediction as output.

Question 2: *How do you scale the surrogate's capacity to represent the original program?*

The *capacity* of the surrogate is the complexity of functions that the surrogate can represent. Higher capacity neural networks better fit the training data (Belkin et al., 2019), but have higher execution cost (Tan and Le, 2019). Scaling the capacity involves adding more layers or increasing the width of each layer.

I search among candidate capacities of the surrogate to find the smallest-capacity BERT architecture that meets the accuracy specification. I base the model on the BERT-Tiny model described by Turc et al. (2019), which has an embedding width of 128, 2 layers, and 2 self-attention heads. From this base architecture I search across alternative embedding widths that are a factor of two between 16 and 128. The objective is to find the fastest-to-execute architecture that has a validation error of less than 10% MAPE. Table 3.1 shows the results of the hyperparameter search, with the bolded row describing the selected model (with an embedding width of 64). Embedding widths of both 128 and 64 achieve less than 10% MAPE; because an embedding width of 64 achieves the fastest execution speed among

this set, it is chosen as the final model. Embedding widths of 32 and 16 provide increasing execution speedups, but do not satisfy the error criteria of a MAPE of less than 10%.

Training. With the architecture in hand, the programmer must train the surrogate model.

Question 3: *What training data does the surrogate use?*

The training data distribution is the distribution of inputs on which the surrogate learns the behavior of the original program. Ideally the training distribution is representative of the distribution of inputs on which the surrogate is expected to perform well.

For surrogate compilation in general, any dataset of inputs can suffice to train the neural surrogate, as long as they constitute a sufficiently large set of representative examples of the distribution of inputs that the programmer wishes to accurately generate predictions for. I use basic blocks from the BHive dataset (Chen et al., 2019) to train the surrogate for consistency with the case studies in Sections 3.1.3 and 3.1.4.

Question 4: *What loss function does the surrogate use?*

The *loss function*, the objective in a neural network's optimization process, is a differentiable, continuous relaxation of the objective and constraints from the specification (which may not be differentiable). Different relaxations of a given loss function may have different properties (Bishop, 2006, pp. 337–338).

Because the objective of maximizing execution speed is handled in the capacity search process, the loss function for training the surrogate for this surrogate compilation example is just the MAPE between the surrogate's prediction and llvm-mca's prediction of throughput.

Question 5: *How long do you train the surrogate?*

The number of training iterations for the neural surrogate determines the trade-off between the training cost of the surrogate and the accuracy of the surrogate. In general, the cost of

training is limited either by an acceptability threshold on the error or by a fixed training budget. Because the training procedure may be run multiple times when designing the surrogate, the threshold or budget should be set to account for the full cost of design and training.

I train the BERT model for 500 passes over the training set (500 epochs), recording the loss over a validation set after each epoch. At the end of training, I select the model with the best validation loss as the final model from training.

Deployment. Once the surrogate has been designed and trained, it must be deployed for its downstream task. This takes different forms depending on the use case of the surrogate: whether the downstream task requires low-latency or high-throughput execution, whether the surrogate is distributed to end-users, what the expected hardware and software platform for the deployment is, or any other considerations related to the downstream use case of the surrogate.

Question 6: *What hardware does the surrogate use?*

For fairness of comparison with llvm-mca the surrogate is deployed on identical hardware to llvm-mca, which in this case is a single Intel Xeon Skylake CPU at 3.1GHz.

Question 7: *What software execution environment does the surrogate use?*

The surrogate uses a network compiled, optimized, and loaded with the ONNX runtime, version 1.7.0 (ONNX Runtime developers, 2021). The surrogate implementation is the Hugging Face Transformers v4.6.1 BertForSequenceClassification with a hidden size of 64, 2 hidden layers, 2 attention heads, an intermediate size of 256, and dropout probability of 0. The surrogate is compiled to ONNX using https://github.com/huggingface/transformers/blob/acc3bd9/src/transformers/convert_graph_to_onnx.py. The surrogate is optimized using the ONNX transformer optimization script with default settings: https://github.com/microsoft/onnxruntime/blob/4fd9fef9ee04c0844d679e81264779402cfa445c/onnxruntime/python/tools/transformers/optimizer.py.

The surrogate is set to use a single thread by setting the OMP, MKL, and ONNX number of threads to 1, and is set to a single CPU affinity. The surrogate uses a batch size of 1. The surrogate is invoked repeatedly by a Python script, and is passed the same 10,000 basic blocks to predict timing values for. The reported execution speed is the time from the invocation of the Python script to its exit.

The BERT-based surrogate does not require any preprocessing of the input assembly.

3.1.3 Surrogate Adaptation

Beyond just being fast, CPU simulators must be accurate. To accurately model behaviors observed in real-world processors, a programmer must develop a model that matches the behavior of that processor. The standard approach, exemplified by llvm-mca, is to manually design, implement, and tune an abstract execution model of the processor. This approach takes significant development effort, and can still result in inaccurate simulation, in part due to modeling assumptions that programmers make that do not accurately reflect real CPUs.

As an alternative to hand-tuning a model, programmers can train a machine learning model from scratch based on observations of the ground-truth behavior of the processor. Though it requires less development effort, this approach requires a significant amount of data to train an accurate model (Lecun et al., 1998; Krizhevsky et al., 2012).

Surrogate adaptation. Another approach for developing an accurate CPU simulation model is surrogate adaptation. With surrogate adaptation, programmers first develop a surrogate of a program then further train that surrogate on data from a different task. Key benefits of this approach include that surrogate adaptation makes it possible to alter the semantics of the program to perform a different task of interest and that it may be more data-efficient or result in higher accuracy than training a model from scratch for the task (Tercan et al., 2018; Kustowski et al., 2020).

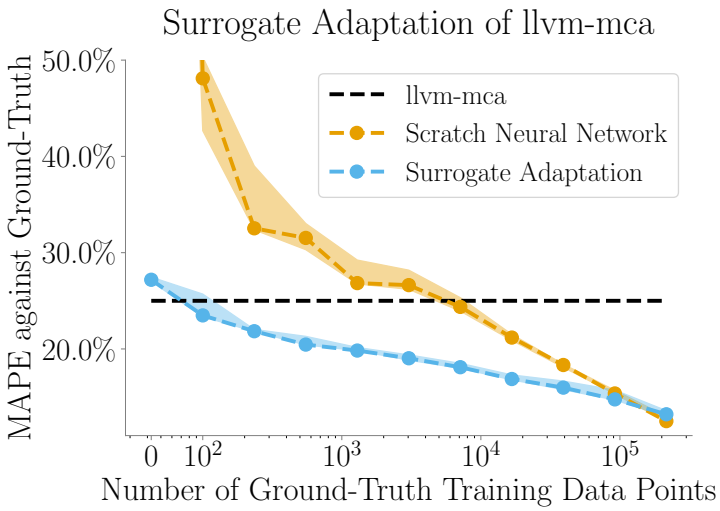


Figure 3.1: Error on ground-truth data of llvm-mca (black), a neural network trained from scratch (orange), and surrogate adaptation of llvm-mca (blue). The rightmost point corresponds to training on the entire BHive dataset.

Results. Figure 3.1 presents the MAPE of several approaches to predicting ground-truth basic block throughputs, as a function of the size of the training dataset of the approach. The black dashed line shows llvm-mca's error rate, which is not a function of the amount of ground-truth training data available, and is constant at 25.0%. The blue dotted line shows surrogate adaptation's error rate, which is upper bounded by llvm-mca's, as surrogate adaptation is first trained to mimic llvm-mca, then decreases with more training data. The orange dots show the error of a neural network trained from scratch, which results in a large error rate when trained with a small number of examples, only matching surrogate adaptation when it is trained on the entire BHive training data set. These results show that surrogate adaptation leads to more accurate simulation than training a neural network from scratch when ground-truth data is not readily available (e.g., in cases where collecting ground-truth data is expensive), but provides no benefit when ground-truth data is plentiful.

Programming Methodology

As with surrogate compilation, developing the surrogate for surrogate adaptation requires a problem specification, a design for the neural network, a training procedure for the network, and a deployment configuration.

Specification. Surrogate adaptation requires two steps, finding the surrogate then adapting to a downstream task. This is represented as two sequential optimization problems. In the first optimization problem for this example, finding a surrogate that mimics llvm-mca, I find a surrogate that minimizes the error against llvm-mca without any other constraints:

$$s_1^* = \arg\min_s \mathop{\mathbb{E}}_{x\sim\mathcal{D}}\left[\frac{|s(x) - p(x, \text{haswell-params})|}{p(x, \text{haswell-params})}\right]$$

where s is the surrogate, $\mathcal{D}$ is the dataset of basic blocks x from BHive, p is llvm-mca, and haswell-params is LLVM's default set of Haswell CPU parameters.

In the second optimization problem, I optimize for accuracy on the ground-truth data:

$$s^* = \arg\min_s \mathop{\mathbb{E}}_{x\sim\mathcal{D}}\left[\frac{|s(x) - m(x)|}{m(x)}\right]$$

where $\mathcal{D}$ is the dataset of basic blocks x from BHive, and m is the ground-truth measured timing of the basic block on a Haswell CPU from BHive.

In surrogate adaptation, the second optimization problem is seeded with the surrogate resulting from the first.

Design. In this example the neural network architecture and capacity must maximize accuracy first against llvm-mca then against the ground-truth measurements. There are no other objectives or constraints on the surrogate design.

Question 1: *What neural network architecture topology does the surrogate use?*

As with the surrogate compilation example, I use a BERT Transformer architecture. In general, surrogate adaptation can use the same architecture as surrogate compilation, though it may not have the same execution time constraints and may require an architecture that is tailored for the downstream objective. In this surrogate adaptation example the downstream objective is similar to the original program's objective, allowing us to use the same architecture.

Question 2: *How do you scale the surrogate's capacity to represent the original program?*

To minimize hyperparameter search cost, I reuse the capacity for the neural surrogate from Section 3.1.2, which has less than 10% error against llvm-mca.

Training.

Question 3: *What training data does the surrogate use?*

As in Section 3.1.2, I use the BHive dataset to train the surrogate. The BHive dataset is the only dataset of basic blocks with timings that correspond to the assumptions made by llvm-mca, making the ground-truth errors pre- and post- surrogate adaptation comparable (though for surrogate adaptation in general the downstream task need not be identical to the task performed by the original program).

In the first optimization problem, the labels for training are llvm-mca's predictions on these basic blocks. In the second optimization problem, the labels are the ground-truth measured timings on a Haswell CPU from BHive.

Question 4: *What loss function does the surrogate use?*

The loss function for training the surrogate in both optimization problems is the MAPE, as specified in the specification. In general, the loss functions for the optimization problems need not be the same if the programmer is adapting the surrogate to a different problem.

Question 5: *How long do you train the surrogate?*

In the first optimization problem of surrogate adaptation, minimizing or constraining training time is not a part of the specification; I therefore reuse the neural surrogate trained in Section 3.1.2, which is the surrogate with minimum validation loss within 500 epochs of training. In the second optimization problem, the surrogate resulting from the first step is used as a warm starting point for optimization. I again use the minimum-validation-loss surrogate within 500 epochs of training, which constrains the surrogate in the second problem to not deviate too much from the original surrogate.

Deployment. Once the programmer has designed and trained the surrogate, the programmer must deploy it for its downstream task. The specification for this surrogate adaptation example does not specify objectives or constraints on the deployment for the surrogate.

Question 6: *What hardware does the surrogate use?*

The neural surrogate is trained on an NVIDIA V100 GPU, which provides sufficient throughput (over 512 training examples per second) to train the surrogate for each optimization problem. Since the specification does not impose constraints on the deployment of the surrogate, I also deploy it on the same GPU for simplicity.

Question 7: *What software execution environment does the surrogate use?*

The neural surrogate is trained in PyTorch (Paszke et al., 2019), which automatically calculates the gradient of the surrogate for both optimization problems. Since the specification does not impose deployment constraints, I also deploy it in PyTorch.

3.1.4 Surrogate Optimization

Surrogate adaptation changes the semantics of the entire simulation to more accurately model ground-truth data, resulting in behavior distinct from that of the original simulation. Such behavior is not always desirable, since it leads to predictions that programmers cannot reason about with the hand-coded simulation model. Programmers may instead want the best version of the simulation that is possible with proper choice of simulation parameters.

In this subsection, I re-present the DiffTune algorithm and results discussed in Chapter 2 in the context of the methodology discussed in this chapter.

To use llvm-mca to accurately model ground-truth data, programmers must find simulation parameters that lead llvm-mca to accurate simulation of the physical CPU. The Haswell parameters in llvm-mca are comprised of 11,265 integer-valued parameters, inducing a configuration space with $10^{19,336}$ possible configurations. Each of these 11,265 parameters must be set for each different CPU that llvm-mca targets.

The standard approach is to have experts manually set the parameters based on documentation, measurement, and intuition. This approach again requires significant developer effort and can still result in high simulation error, due in part to the difficulties of setting llvm-mca's CPU parameters to values that lead llvm-mca to low prediction error.

Alternatively, the parameters may be set by automatic approaches based entirely on measurement. One class of automatic approaches for setting llvm-mca's parameters is to gather measurements of each parameter's realization in the CPU architecture that llvm-mca targets (Fog, 1996; Abel and Reineke, 2019).

Another class of approaches is to gather coarse-grained measurements of entire basic blocks then optimize llvm-mca's parameters to best fit the timings of the basic blocks. Due to the size of the parameter space, this is an optimization problem for which gradient-free optimization techniques (Ansel et al., 2014) are intractable. Gradient descent converges to local minima more quickly than gradient-free optimization with the original program (Hicken et al., 2020).

However, since llvm-mca is not written in a differentiable programming language (Baydin et al., 2018) and operates over discrete values, it is also not possible to calculate its gradient or optimize its parameters with gradient descent.

Surrogate optimization. An alternative approach for optimizing parameters of the program using coarse-grained measurements is to use surrogate optimization as implemented by the DiffTune algorithm in Chapter 2. Here I re-present a case study of using surrogate optimization via DiffTune to optimize llvm-mca's parameters.

With surrogate optimization, programmers develop a surrogate of a program, then optimize program inputs against the surrogate. The key benefit of this approach is that surrogate optimization can optimize inputs faster than optimizing inputs directly against the program, due to the potential for faster execution speed of the surrogate and the potential for the surrogate to be differentiable even when the original program is not (allowing for optimizing inputs with gradient descent) (Tseng et al., 2019; She et al., 2019).

Results. Using surrogate optimization results in parameters that lead llvm-mca to an average error of 23.7% on the Haswell basic blocks in BHive (Chen et al., 2019). In contrast, the expert-tuned default Haswell parameters lead llvm-mca to an average error of 25.0%. OpenTuner (Ansel et al., 2014), a gradient-free optimization technique, is not able to find parameters that lead llvm-mca to lower than 100% error given a computational budget equivalent to that of surrogate optimization.

Programming Methodology

Again, developing the surrogate for surrogate optimization involves a specification, design, training, and deployment.

Specification. Surrogate optimization requires two steps, finding the original surrogate then optimizing inputs to the surrogate. As with surrogate adaptation, this is represented as two sequential optimization problems.

In the first optimization problem for surrogate optimization, the objective is to find a surrogate that minimizes the error against llvm-mca's predicted throughput for any given input basic block and set of CPU parameters:

$$s_1^* = \arg\min_{s} \mathop{\mathbb{E}}_{\substack{x_{\text{block}} \sim \mathcal{D}_{\text{block}} \\ x_{\text{params}} \sim \mathcal{D}_{\text{params}}}} \left[\frac{|s(x_{\text{block}}, x_{\text{params}}) - p(x_{\text{block}}, x_{\text{params}})|}{p(x_{\text{block}}, x_{\text{params}})} \right]$$

where s is the surrogate, $\mathcal{D}_{\text{block}}$ is the dataset of basic blocks x_{block} from BHive, $\mathcal{D}_{\text{params}}$ is a uniform distribution over parameter values x_{params}, and p is llvm-mca.

In the second optimization problem, the objective is to find input parameters that optimize predictive accuracy against the ground-truth data:

$$x_{\text{params}}^* = \arg\min_{x_{\text{params}}} \mathop{\mathbb{E}}_{x_{\text{block}} \sim \mathcal{D}_{\text{block}}} \left[\frac{|s_1^*(x_{\text{block}}, x_{\text{params}}) - m(x_{\text{block}})|}{m(x_{\text{block}})} \right]$$

where $\mathcal{D}_{\text{block}}$ is the dataset of basic blocks x_{block} from BHive, and m is the ground-truth measured timing of the basic block on a Haswell CPU from BHive.

Design. In this surrogate optimization example, the architecture and capacity must maximize accuracy, with no other objectives or constraints on the design.

Question 1: *What neural network architecture topology does the surrogate use?*

Due to including x_{params} as input to the surrogate, the BERT architecture in Sections 3 and 4, which expects just basic blocks as input, is not sufficient for this task. I use the neural network architecture proposed by my prior work (Mendis et al., 2019a).

The architecture consists of a stacked pair of LSTMs (Hochreiter and Schmidhuber, 1997). The bottommost LSTM generates a vector representation of each instruction independently. I then concatenate each of these instruction vector representations with the relevant parameters in x_{params} that affect simulation of the instruction. The topmost LSTM then processes each of these vector representations to generate a final prediction for the basic block. I validate that this model learns to predict the throughput of basic blocks on Intel CPUs with low error (Mendis et al., 2019a), a similar problem to developing a surrogate of llvm-mca.

Question 2: *How do you scale the surrogate's capacity to represent the original program?*

I use a stack of 4 LSTMs in place of each original LSTM, each with a width of 256 neurons. Stacking LSTMs increases their capacity, which is needed due to the complexity induced by adding the CPU parameters as input to the surrogate.

Training.

Question 3: *What training data does the surrogate use?*

In both optimization problems, I use basic blocks from the BHive dataset as input basic blocks x_{block} (Chen et al., 2019). In the first optimization problem, I also use a bounded uniform distribution over parameter values (informed by the range of parameter values for other CPU architectures) as input parameters x_{params}. As with the surrogate adaptation example, in the first optimization problem the throughputs to predict are llvm-mca's predictions on these basic blocks, and in the second they are the measured timings from BHive.

Question 4: *What loss function does the surrogate use?*

The loss function for training the surrogate in both optimization problems of this surrogate optimization example is the MAPE of the surrogate's prediction of llvm-mca's prediction of throughput, as specified in the specification. As with surrogate adaptation, the loss functions for both optimization problems do not have to be the same in general.

Question 5: *How long do you train the surrogate?*

I train the surrogate and the input parameters until convergence on a validation set. This results in training for 60 epochs in the first training phase and 1 epoch in the second phase.

Deployment. Once the surrogate has been designed and trained, it is deployed for its downstream task. Unlike surrogate compilation and surrogate adaptation, in surrogate optimization the surrogate is never directly deployed to end-users, instead being used entirely as an intermediate artifact in the parameter optimization process.

Question 6: *What hardware does the surrogate use?*

The surrogate itself is executed on a GPU, which provides sufficient throughput for optimizing the input parameters. Once found, the input parameters x^*_{params} are plugged back into llvm-mca, which is executed on a CPU.

Question 7: *What software execution environment does the surrogate use?*

The surrogate and input parameters are trained in PyTorch (Paszke et al., 2019), which calculates the gradients for both the surrogate and the input optimization. Once found, the input parameters x^*_{params} are then plugged back into llvm-mca.

3.2 Surrogate-Based Design Patterns

The previous section laid out several case studies of applying surrogate programming to evolve programs. Each case study followed a different methodology, with a different algorithm instantiated by different objectives and constraints. However, these methodologies did share commonalities in their broad algorithms (each first training a surrogate of a program) and in the criteria used in their objectives and constraints.

To generalize these methodologies, I now present a taxonomy of surrogate-based design patterns. I first formalize the definition and specification of a surrogate of a program. I then present the algorithm sketches that define each design pattern, justifying these sketches with concrete examples of each design pattern from the literature. I finally describe and provide examples of the key benefits of each design pattern.

3.2.1 Preliminaries

Let $p \in \mathcal{P}$ denote a program under study. Let $\omega \in \mathcal{P} \to \mathcal{X} \to \mathcal{Y}$ denote an *interpreter*, which takes the program p and an input $x \in \mathcal{X}$ and produces an output $y \in \mathcal{Y}$. Let ω^* denote the *standard interpreter*, corresponding to the standard input-output relationship of the program according to the denotational semantics of the programming language (Winskel, 1993, Chapter 5). Other interpreters may output other aspects of the execution of the program, such as its execution time, memory usage, control flow trace, or any other aspect of its denotational or operational semantics. Finally, let $s \in \mathcal{P}$ denote a surrogate of the program.

The ideal surrogate s of a given interpretation ω_p of a program p is a surrogate such that for all inputs, the standard interpretation ω^* of the surrogate has the same output as the interpretation of the program:

$$\forall x \in \mathcal{X}. \, \omega^*(s)(x) = \omega_p(p)(x) \tag{3.1}$$

3.2.2 Formalization of Design Patterns

I now formalize each of the surrogate-based design patterns. The definitions are in the form of generic optimization problem specifications, showing the set of possible objectives and constraints on the solutions. These generic optimization problem specifications constitute an algorithm sketch for each surrogate-based design pattern.

Let $d : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ measure the error between two outputs. Let $e : (\mathcal{X} \to \mathcal{Y}) \times \mathcal{X} \to \mathbb{R}$ measure the cost of executing a given interpretation of a program on a given input (measured

$$s_1^* = \arg\min_s \; \underset{x \sim \mathcal{D}(\mathcal{X})}{o} \begin{pmatrix} d(\omega^*(s)(x), \omega_p(p)(x)), \\ e(\omega^*(s), x) \end{pmatrix} \text{ subject to } \underset{x \sim \mathcal{D}(\mathcal{X})}{c} \begin{pmatrix} d(\omega^*(s)(x), \omega_p(p)(x)), \\ e(\omega^*(s), x) \end{pmatrix}$$

Figure 3.2: Optimization problem for learning a surrogate s_1^* of the original program p. This optimization problem is the first step of all three surrogate-based design patterns.

in latency, execution cost, energy, etc.). Let $\ell : \mathcal{Y} \times \mathcal{X} \to \mathbb{R}$ measure the error on a downstream task induced by a given prediction of a given input.

Let $\mathcal{D}(\mathcal{X})$ represent a distribution of program inputs that the surrogates are trained on. Let o and c denote generic objective and constraint functions for the optimization problems, which operate as reductions over the distribution of inputs $\mathcal{D}(\mathcal{X})$ (e.g., taking the expectation, supremum, infinimum, or other reduction over the distribution).

All together, the set of free variables for the design patterns include the choice of interpreter ω for the program, the error metric d, the execution cost metric e, the downstream error metric ℓ, the training distribution $\mathcal{D}(\mathcal{X})$, the objective function o, and the constraint function c. The choices for each of these variables select which criteria to consider and how to weigh these criteria when training the surrogate. In the optimization problems presented in the remainder of this section, the choice for any free variable may differ from that of any other repetition of that variable.

Surrogate Construction. The first step of each surrogate-based design pattern is to train a surrogate of the original program. Figure 3.2 presents the generic optimization problem that defines this step. Surrogate construction is defined by an optimization problem that finds a surrogate s_1^* that minimizes a task-dependent objective function o over a distribution of inputs $x \sim \mathcal{D}(\mathcal{X})$ of the error d between the standard interpretation ω^* of the surrogate s on that input x and an interpretation ω_p of the original program p on the input x, and of the execution cost e of the standard interpretation ω^* of the surrogate s on the input x, subject to a constraint function c of the same terms.

$$s^* = \arg\min_{s} \underset{x\sim\mathcal{D}(\mathcal{X})}{o} \begin{pmatrix} \ell(\omega^*(s)(x), x), \\ d(\omega^*(s)(x), \omega^*(s_1^*)(x)), \\ d\big(\omega^*(s)(x), \omega_p(p)(x)\big), \\ e(\omega^*(s), x) \end{pmatrix} \text{ subject to } \underset{x\sim\mathcal{D}(\mathcal{X})}{c} \begin{pmatrix} \ell(\omega^*(s)(x), x), \\ d(\omega^*(s)(x), \omega^*(s_1^*)(x)), \\ d\big(\omega^*(s)(x), \omega_p(p)(x)\big), \\ e(\omega^*(s), x) \end{pmatrix}$$

Figure 3.3: Second optimization problem for surrogate adaptation, which re-trains a surrogate s_1^* to find another surrogate s^* with higher accuracy against a different objective. The surrogate s_1^* is used as a warm start for this problem.

Surrogate Compilation

In surrogate compilation, the programmer simply deploys the surrogate found in the surrogate construction step to the end-user: $s^* = s_1^*$.

Surrogate Adaptation

The first step of surrogate adaptation is the initial surrogate construction step. The second step is to continue to train the surrogate to optimize a different downstream objective.

Figure 3.3 shows the generic optimization problem that defines the second step of surrogate adaptation. This second optimization problem finds a surrogate s^* that minimizes a task-dependent objective function o over a distribution of inputs $x \sim \mathcal{D}(\mathcal{X})$ of the downstream error ℓ of the standard interpretation ω^* of the surrogate s on an input x, the error d between the standard interpretation ω^* of the surrogate s on the input x and the standard interpretation ω^* of the surrogate s_1^* from the first optimization problem on that input x, the error d between the standard interpretation ω^* of the surrogate s on the input x and an interpretation ω_p of the program p on that input x, and the execution cost e of the standard interpretation ω^* of the surrogate s on that input x, subject to a constraint function c of the same terms.

In surrogate adaptation, the surrogate from the first optimization problem is used as a warm starting point for the second optimization problem.

$$x^* = \arg\min_x o\big(\ell(\omega^*(s_1^*)(x), x)\big) \text{ subject to } c\big(\ell(\omega^*(s_1^*)(x), x)\big)$$

Figure 3.4: Second optimization problem for surrogate optimization, which optimizes inputs x of a surrogate s_1^* to minimize a different objective function on the surrogate.

Surrogate Optimization

The first step of surrogate optimization is the surrogate construction step. The second is to optimize inputs to the surrogate against a different objective. Figure 3.4 shows the optimization problem that defines the second step of surrogate optimization.

This second optimization problem finds an input x^* that minimizes a task-dependent objective function o of the downstream error ℓ of the standard interpretation ω^* of the surrogate from the first optimization problem s_1^* on the input x, subject to a constraint function c of the same term.

Specifications in the Literature

Tables 3.2 to 3.4 respectively present surveys of surrogate compilation, surrogate adaptation, and surrogate optimization, showing the terms in the optimization problem solved by each piece of related work. These optimization problem specifications correspond to concrete instantiations of interpreters ω, error functions d, e, and ℓ, and objective functions o and c.

With examples in hand, I now discuss the design considerations and trade-offs that must be considered when specifying the optimization problem for training a surrogate.

Surrogate error. A surrogate must compute a similar function to that computed by its source program. When the surrogate is deployed to end-users as in surrogate compilation and surrogate adaptation, the error metric for the surrogate is that of the domain (Esmaeilzadeh et al., 2012). When the surrogate is used as an intermediate artifact as in surrogate optimization, other error metrics may help to learn a surrogate that allows for successful downstream optimization (Tseng et al., 2019).

Table 3.2: Optimization problem specifications of surrogate compilation from the literature.

Citation and description	Optimization problem specification
	$$s^* = \arg\min_s o\begin{pmatrix} d(s,p), \\ e(s) \end{pmatrix} \text{ subj. to } c\begin{pmatrix} d(s,p), \\ e(s) \end{pmatrix}$$
Esmaeilzadeh et al. (2012): Training neural surrogates of small numerical kernels to decrease their execution latency by executing on a neural network accelerator.	• $o(d(s,p))$: The mean squared error between the outputs of the surrogate and the original kernel is minimized (Esmaeilzadeh et al., 2012, Section 4). • $o(e(s))$: The size of the surrogate (measured by the number of hidden units) is minimized to reduce execution time (Section 4). • $c(d(s,p))$: The end-to-end error of the application that uses the surrogate is constrained to be less than 10% (Section 7.1). • $c(e(s))$: The surrogate is constrained to have lower execution latency than the original kernel (Sections 7, 8).
Mendis (2020, Chapter 4): Training neural surrogates of compiler auto-vectorizers, to replace the original exponential-time vectorizer with a linear time surrogate.	$$s^* = \arg\min_s o(d(s,p)) \text{ subj. to } c(e(s))$$ • $o(d(s,p))$: The cross entropy error between the outputs of the surrogate and the auto-vectorizer is minimized (Mendis, 2020, Chapter 4.4). • $c(e(s))$: The surrogate has predictable (and not data-dependent) linear running time (Chapters 1.3.4, 4.8).
Munk et al. (2022): Training neural surrogates of stochastic simulators to accelerate simulation and inference using the simulator.	$$s^* = \arg\min_s o\begin{pmatrix} d(s,p), \\ e(s) \end{pmatrix} \text{ subj. to } c(e(s))$$ • $o(d(s,p))$: The KL divergence between the outputs of the surrogate and the original stochastic simulator is minimized (Munk et al., 2022, Section 3.1). • $o(e(s))$: The surrogate is as fast as possible to maximize the execution throughput speedup over the original simulator (Section 3.2). • $c(e(s))$: The surrogate is constrained to have higher execution throughput than the original simulator (Section 3.2).
Pestourie et al. (2020): Training neural surrogates of partial differential equation (PDE) solvers to aid designing material composites, using active learning to minimize the training cost of the surrogate.	$$s^* = \arg\min_s o\begin{pmatrix} d(s,p) \\ e(s) \end{pmatrix} \text{ subj. to } c(e(s))$$ • $o(d(s,p))$: The MAPE between the outputs of the surrogate and the original PDE solver is minimized (Pestourie et al., 2020, Figure 5). • $o(e(s))$: The surrogate is as fast as possible to maximize execution latency speedup over the original solver ("Introduction"). • $c(e(s))$: The surrogate must have higher execution throughput than the original PDE solver ("Introduction").

Table 3.3: Optimization problem specifications of surrogate adaptation from the literature.

Citation and description	Optimization problem specification
Tercan et al. (2018): Training neural surrogates of computer simulations of plastic injection molding, then adapting the surrogates on real-world experiments of injection molding to close the gap between simulated and real results.	$s_1^* = \arg\min_s o_1(d(s,p))$ $\quad s^* = \begin{cases} \arg\min_s o_2\begin{pmatrix}\ell(s_1^*,x),\\ e(s)\end{pmatrix} \\ \text{subj. to } c_2(\ell(s_1^*,x)) \end{cases}$ • $o_1(d(s,p))$: The Pearson correlation between the outputs of the surrogate and the original simulation is maximized (Tercan et al., 2018, Section 5.2). • $o_2(\ell(s_1^*,x))$: The Pearson correlation between the surrogate and the real-world experiments is maximized (Section 5.2). • $o_2(e(s))$: The surrogate is cheaper than real experiments (Section 1). • $c_2(\ell(s_1^*,x))$: The L1 loss of the surrogate is less than 0.01 (Section 4).
Kustowski et al. (2020): Training neural surrogates of computer simulations of inertial confinement fusion, then adapting on a small number of results from real-world experiments to close the gap between simulated and real results.	$s_1^* = \arg\min_s o_1(d(s,p))$ $\quad s^* = \arg\min_s o_2\begin{pmatrix}\ell(s_1^*,x),\\ d(s,s_1^*),\\ e(s)\end{pmatrix}$ • $o_1(d(s,p))$: The Pearson correlation between the surrogate and the original simulation is maximized (Kustowski et al., 2020, Section II). • $o_2(\ell(s_1^*,x))$: The Pearson correlation between the trained surrogate and the real-world experiments is maximized (Section II). • $o_2(d(s,s_1^*))$: s^* is biased to be close to s_1^* by freezing most layers in the neural network to be equal to their values in s_1^* (Section III.B). • $o_2(e(s))$: The surrogate is cheaper than real experiments (Section I).
Kwon and Carloni (2020): Training neural surrogates of computer architecture simulations of programs for design space exploration of the architecture, then adapting the surrogates for accurate design space exploration when simulating other programs.	$s_1^* = \arg\min_s o_1(d(s,p))$ $\quad s^* = \arg\min_s o_2\begin{pmatrix}\ell(s_1^*,x),\\ d(s,s_1^*),\\ e(s)\end{pmatrix}$ • $o_1(d(s,p))$: The MSE between the surrogate output and the simulated program running time is minimized (Kwon and Carloni, 2020, Section 1). • $o_2(\ell(s_1^*,x))$: The mean squared error of the surrogate on new programs not in the surrogate's original training set is minimized (Section 2). • $o_2(d(s,s_1^*))$: s^* is biased to be close to s_1^* by using the weights from s_1^* as a warm starting point for the optimization problem (Section 3). • $o_2(e(s))$: The surrogate is cheaper than simulation (Section 1).
Kaya and Hajimirza (2019): Training surrogates of physics simulations of properties of a given material for designing structures with that material, then adapting those surrogates to aid simulation-based design with other materials.	$s_1^* = \arg\min_s o_1(d(s,p))$ $\quad s^* = \begin{cases} \arg\min_s o_2\begin{pmatrix}\ell(s_1^*,x),\\ d(s,s_1^*),\\ e(s)\end{pmatrix} \\ \text{subj. to } c_2(d(s,p)) \end{cases}$ • $o_1(d(s,p))$: The mean squared error between the outputs of the surrogate and simulation on the base material is minimized (Kaya and Hajimirza, 2019, "Results and Discussion – Base Case"). • $o_2(\ell(s_1^*,x))$: The error of the outputs of the trained surrogate on the new material is minimized ("Results and Discussion – Transfer Cases"). • $o_2(d(s,s_1^*))$: s^* is biased to be close to s_1^* by using the weights from s_1^* as a warm starting point for the optimization problem ("Introduction"). • $o_2(e(s))$: The surrogate is cheaper than simulation ("Introduction"). • $c_2(d(s,p))$: If s^* is less accurate than simulation, then the transfer learning results are rejected ("Results and Discussion – Transfer Cases").

Table 3.4: Optimization problem specifications of surrogate optimization from the literature.

Citation and description	Optimization problem specification
Renda et al. (2020) (Chapter 2): Training neural surrogates of CPU simulators that predict execution time of code, then optimizing parameters of the CPU simulator to more closely match ground-truth execution times measured on real hardware.	$s_1^* = \arg\min_s o(d(s,p))$ $\quad x^* = \arg\min_x o(\ell(s_1^*, x))$ • $o(d(s,p))$: The MAPE between the surrogate and the CPU simulator on an input code snippet is minimized (Section 2.3). • $\ell(s_1^*, x)$: The MAPE of the output of the trained surrogate induced by the set of simulation parameters is minimized against the ground-truth data (Section 2.3).
She et al. (2019): Training neural surrogates of the branching behavior of programs to find inputs that trigger branches that cause bugs in the program.	$s_1^* = \arg\min_s o(d(s,p))$ $\quad x^* = \arg\min_x o(\ell(s_1^*, x))$ • $o(d(s,p))$: The binary cross-entropy between the surrogate output and the program's branching behavior is minimized (She et al., 2019, Section IV.B). • $\ell(s_1^*, x)$: Gradient descent tries to find an input that lead to an unseen set of branches taken in the program (Section IV.C).
Tseng et al. (2019): Training neural surrogates of camera pipelines, to find parameters for the pipelines that lead to the cameras producing the most photorealistic images.	$s_1^* = \arg\min_s o(d(s,p))$ $\quad x^* = \arg\min_x o(\ell(s_1^*, x))$ • $o(d(s,p))$: The L2 error between the image from the surrogate and the image from the pipeline is minimized (Tseng et al., 2019, Section 4.2). • $\ell(s_1^*, x)$: Gradient descent tries to find parameters that lead to images being as similar as possible in L2 distance to the ground-truth (Section 4.2).
Shirobokov et al. (2020): Training neural surrogates of physics simulators to find simulation inputs that lead to local optima.	$s_1^* = \arg\min_s o(d(s,p))$ $\quad x^* = \arg\min_x o(\ell(s_1^*, x))$ • $o(d(s,p))$: The error (as measured by a domain-specific loss function per-task) between the outputs of the surrogate and the simulation is minimized (Shirobokov et al., 2020, Section 2.2). • $\ell(s_1^*, x)$: Gradient descent tries to find parameters that lead to local optima in the problem space against the same loss function (Section 2.2).

In the second step of surrogate adaptation, the final surrogate may also be constrained to be close to the original surrogate, another instantiation of surrogate error (treating the original surrogate as a source program) (Kwon and Carloni, 2020; Kaya and Hajimirza, 2019).

Downstream error. For surrogate adaptation and surrogate optimization, the second optimization problems use an error metric beyond that of mimicking the original program. This downstream error metric may be that of the downstream task that the original program targets (Chapter 2; Tercan et al., 2018). The downstream error metric may also be unrelated to the domain of the original program: for instance, Kwon and Carloni (2020) use an error metric for surrogate adaptation that adapts the surrogate to inputs and outputs of a different domain. She et al. (2019) use an error metric for surrogate optimization that measures the extent to which the discovered inputs trigger unseen control flow paths in the program.

Execution Cost. Regardless of the intended use case, a surrogate must be efficient, not exceeding resource budgets to deploy. The execution cost of a surrogate measures the resources required to execute the surrogate in its execution environment. The ideal is a surrogate that is efficient to execute, with low execution latency (Esmaeilzadeh et al., 2012), high throughput (Mendis, 2020), low storage cost (Han et al., 2016b), and minimal energy cost (Esmaeilzadeh et al., 2012).

3.2.3 Key Benefits

I now discuss the key benefits of each different surrogate programming design pattern, detailing examples beyond those of the case studies in Section 3.1.

Surrogate Compilation

Surrogate compilation allows for the ability to execute the surrogate on different hardware and the ability to bound or to accelerate the execution time of the surrogate (Esmaeilzadeh et al., 2012; Mendis, 2020).

Compiling to different hardware. Esmaeilzadeh et al. (2012) develop surrogates of small computational kernels, then deploy the surrogates on a hardware accelerator that reduces the latency and energy cost of executing the surrogate. More generally, surrogates can be deployed on any hardware that supports the surrogate architecture, resulting in different trade-offs compared to the CPU architectures that many conventional programs execute on.

Different algorithmic complexity. Algorithmic complexity can differ between a program and its surrogate: for example, while an algorithm may require an exponential number of operations in the size of the input, a surrogate of that algorithm may only require a linear number of operations to approximate the algorithm to satisfactory accuracy (Mendis et al., 2019b; Karpathy, 2017).

Surrogate Adaptation

Surrogate adaptation makes it possible to alter the semantics of the program to perform a different task of interest. Surrogate adaptation may be more data-efficient or result in higher accuracy than training a model from scratch (Tercan et al., 2018; Kustowski et al., 2020).

Data efficiency. Tercan et al. (2018) develop models that accurately simulate a plastic injection molding process. Tercan et al. train surrogates of computer simulations of injection molding, then adapt the surrogates on real-world experiments of the injection molding process to close the gap between simulation and ground-truth. Tercan et al. show that

the surrogate resulting from surrogate adaptation requires less training data than a neural network trained from scratch.

Accuracy. Kustowski et al. (2020) learn a model of a physical process involved in nuclear fusion, inertial confinement fusion (ICF). Physical simulation is critical for this area of research, but it is not accurate in part due to unknown biases and inaccuracies in the models of ICF. Kustowski et al. use surrogate adaptation to increase the accuracy of simulators by training a surrogate of simulation then adapting the surrogate on data from physical experiments.

Surrogate Optimization

Surrogate optimization optimizes inputs faster than optimizing inputs directly against the program, due to the potential for faster execution speed of the surrogate and the potential for the surrogate to be differentiable even when the original program is not (Tseng et al., 2019; She et al., 2019).

Faster execution time. İpek et al. (2006) perform design space exploration on a simulated computer architecture, finding the physical parameters (e.g., cache size, cache associativity, etc.) that lead to the best performance. İpek et al. use surrogate optimization to optimize these parameters, exploiting the significantly faster execution of the surrogate compared to the execution of the original simulation.

Differentiable output domain of programs. She et al. (2019) construct neural surrogates of programs for *fuzzing*, generating inputs that cause bugs in the program. For a given input, a classical program has an *execution trace*, the set of edges taken in the control flow graph, which can be represented as a bitvector where 1 denotes that a given edge is taken, and 0 denotes that it is not. She et al. construct a neural surrogate that, for a given input, predicts an approximation of the execution trace of the program with each element between 0 and 1 (rather than strictly set to 0 or 1). This allows for a smooth output of the surrogate, which

then allows She et al. to use gradient descent to find inputs that induce a specific execution trace on the original program.

Relaxing the input domain of programs. Grathwohl et al. (2018) use neural surrogates to approximate the gradient of non-differentiable functions, in order to reduce the variance of gradient estimators of random variables. Though the inputs are discrete, Grathwohl et al.'s surrogates take continuous values as input, allowing for optimizing these with gradient descent.

Completeness

This taxonomy of design patterns covers every instance of surrogate programming in the literature that I am aware of at the time of writing. But this does not mean that the taxonomy is complete: there may be other design patterns that I have not yet encountered, or that have not yet been invented. There are also similar neurosymbolic techniques (Sun et al., 2022) that combine neural networks and symbolic reasoning (I provide examples in Section 3.5), but these techniques are not instances of surrogate programming as defined in this thesis.

3.3 Methodologies

I now discuss the methodology used to implement each design pattern, spanning the design, training, and deployment process of the surrogate.

3.3.1 Design

Given a set of optimization problems that constitute a specification for the surrogate, a programmer must then determine how to design, train, and deploy the surrogate to meet the specification. In this and the following sections I detail the design questions driving the neural surrogate programming methodology. I discuss possible answers to each of these design questions, showing the trade-offs that programmers must navigate when developing surrogates.

This section describes the neural network architecture design approaches for neural surrogates used in the literature.

Question 1: *What neural network architecture topology does the surrogate use?*

Domain-agnostic architectures. One design methodology is to use a domain-agnostic architecture for the surrogate, an architecture designed independently of the behavior and domain of application of the program under consideration. A common choice of domain-agnostic architectures for neural surrogates with fixed-size inputs are multilayer perceptrons (MLPs) (İpek et al., 2006; She et al., 2019). Sections 3.1.2 and 3.1.3 use a BERT encoder (Devlin et al., 2019), a type of Transformer (Vaswani et al., 2017), which is a common architecture for sequence processing tasks. While simple to design, such domain-agnostic architectures may have high training costs or low accuracy (Urban et al., 2017; Neyshabur, 2020).

Domain-specific architectures. An alternative is to design the architecture based on the program and domain under study (Section 2.3; Tseng et al., 2019). However designing such architectures requires manual effort and expertise, both in the original program and in its domain. For instance, the surrogate optimization case study uses a derivative of the architecture proposed by my prior work (Mendis et al., 2019a), a model with high accuracy on basic block throughput prediction. This architecture also exploits input sparsity in the simulation: rather than using the entire set of CPU parameters, the architecture only uses parameters that influence simulation of instructions in the basic block.

Question 2: *How do you scale the surrogate's capacity to represent the original program?*

Determining the capacity of the neural surrogate trades off between accuracy and execution cost, core tasks in any approximate programming task (Stanley-Marbell et al., 2020). Possible approaches include manually selecting the architecture based on reasoning about the complex-

ity of the program and automatically searching for the capacity that leads to the optimal trade-offs of the surrogate's objective and constraints in its specification (Esmaeilzadeh et al., 2012).

3.3.2 Training

With the neural network architecture in hand, the programmer must train the neural surrogate.

Question 3: *What training data does the surrogate use?*

The training data of the surrogate defines the distribution of inputs on which the surrogate is expected to perform well. The data must be representative of inputs for the downstream task for which the surrogate is deployed. The data must also be plentiful and diverse enough to train the surrogate model to generalize the observed behavior of the program.

Instrumenting the program. One approach is to instrument the execution of the original program and record observed inputs (Chen et al., 2019; Esmaeilzadeh et al., 2012). This approach is prevalent in surrogate compilation. An underlying challenge is that it may not be possible to guarantee that the training workload is reflective of the workload of the downstream task, especially when the surrogate is deployed directly to end-users.

Manually-defined random sampling. When data reflective of the downstream task is not available, or when the downstream data distribution is not known *a priori*, another common approach is to randomly sample inputs from a hand-defined sampling distribution (Tseng et al., 2019; Tercan et al., 2018).

Neural surrogate and program symmetries. The training data must also reflect the symmetries enforced in the program and the surrogate. For instance, when the original program is invariant to a specific change in the input but the neural surrogate architecture is not (e.g., a program that calculates the area of a shape is invariant to translation of that shape),

the training data should include augmentations on the data that reflect those symmetries, to train the surrogate to be invariant to that symmetry (Shorten and Khoshgoftaar, 2019).

Question 4: *What loss function does the surrogate use?*

The *loss function* is the objective in a neural network's optimization process which measures how bad a neural network's prediction is compared to the ground truth. The loss function should reflect the downstream specification for the surrogate (such that a reduction in the loss results in a better surrogate for the task) while also being a differentiable function that is possible to optimize with gradient descent.

Question 5: *How long do you train the surrogate?*

With training data and loss function in hand, the programmer must then train the surrogate. This results in a trade-off between accuracy and training cost. Because the training procedure may be run multiple times during hyperparameter search, the threshold or budget should be set appropriately to account for the full cost of design and training.

There are two primary approaches in the literature for determining an appropriate training time of the surrogate. One approach is training for a fixed training time, typically determined via experiments on a validation set (Esmaeilzadeh et al., 2012). Another approach is training until an acceptable accuracy is reached, whether via a plateau of the training loss (Tseng et al., 2019) or via reaching a minimum acceptable accuracy (Tercan et al., 2018). Such variable-length approaches are discussed in more depth by Goodfellow et al. (2016, Chapter 7.8).

Determining the training length for surrogate adaptation is especially important due to the challenges imposed by *catastrophic forgetting* (McCloskey and Cohen, 1989; Ratcliff, 1990), when a neural network's performance on a task it was trained on in the past degrades when it is trained on a new task. There are a number of approaches in the literature for addressing catastrophic forgetting (Yosinski et al., 2014; Serrà et al., 2018; Kirkpatrick et al., 2017;

Chronopoulou et al., 2019); in the case study in Section 3.1.3 I simply select the (relatively small) training time that results in the minimum validation error on a held-out test set.

3.3.3 Deployment

Once the programmer has designed and trained the surrogate, the programmer must deploy the surrogate into its execution context. Neural networks can execute with diverse hardware and runtimes, and can require different representations of the input data than the representations used by the original program.

Question 6: *What hardware does the surrogate use?*

The hardware that the surrogate is deployed on impacts the surrogate's execution time properties, efficiency, and available optimization opportunities. When a surrogate is deployed using different hardware than the original program, developers must also consider the costs of data and control transfer between the original program and the surrogate.

GPUs. Modern large-scale deep neural networks can be executed on GPUs (Cireşan et al., 2011), which achieve high throughput (the number of inputs that can be processed per unit time) and low energy consumption per example at the cost of high latency (the end-to-end time to process a single input) and high energy consumption per unit time (Hanhirova et al., 2018; Li et al., 2016; Han, 2017).

CPUs. Other applications use a CPU to deploy the surrogate (İpek et al., 2006). CPUs typically result in lower latency and energy consumption per unit time than GPUs, at the cost of higher energy consumption per example and reduced throughput (Lee et al., 2010; Hazelwood et al., 2018; Han, 2017; Li et al., 2016) (though recent work challenges some of these assumptions (Daghaghi et al., 2021)). CPUs are also more widely available than GPUs, including on edge devices (Wu et al., 2019).

Machine learning accelerators. Esmaeilzadeh et al. (2012) design and deploy a custom neural processing unit (NPU) to accelerate neural surrogates with low latency and energy cost. Other machine learning accelerators offer different trade-offs, such as TPUs increasing throughput even further (Jouppi et al., 2017), or the Efficient Inference Engine decreasing energy costs while approximating the surrogate (Han et al., 2016a).

Question 7: *What software execution environment does the surrogate use?*

Neural networks require specialized software runtime environments. Choosing the runtime environment requires navigating concerns about both the implementation of the program that uses the surrogate and the deployment of the surrogate across varying devices. Software execution environments include custom frameworks and runtimes which provide bespoke trade-offs for specific applications (Esmaeilzadeh et al., 2012).

The choice of software environment can also impact the availability and performance of the surrogate across hardware platforms. Certain software runtimes are only available for certain devices (e.g., CPUs), some devices are supported by specific software runtimes (e.g., TPUs by TensorFlow), and some runtimes are specialized for resource-constrained devices (e.g., TensorFlow Lite for edge devices).

Normalization. *Data normalization*, which involves pre- and post-processing the inputs and outputs to be suitable for neural networks (LeCun et al., 2012), induces complexity into the program that deploys the surrogate, with normalization and denormalization requiring additional code when integrating the surrogate into the original program's execution context. Data processing bugs in such code are difficult to diagnose and lead to reduced accuracy (Sculley et al., 2014). Esmaeilzadeh et al. (2012) address these issues by integrating the normalization and denormalization steps into the custom hardware (the NPU), eliminating the opportunity for software bugs.

Batching. *Batching*, determining the number of inputs to process at a time, induces a trade-off between latency and throughput for the surrogate. Parrot (Esmaeilzadeh et al., 2012), a surrogate compilation approach that deploys the surrogate to end-users, focuses entirely on latency and uses a single data item in each batch, sacrificing throughput for decreased latency. DiffTune (Chapter 2), a surrogate optimization approach, has no explicit latency requirements and focuses entirely on throughput, increasing throughput by batching large numbers of training examples into single invocations of the surrogate.

3.4 Related Work Addressing Similar Tasks

In this section I discuss related work that provides alternative solutions to the surrogate-based design patterns and the neural surrogate programming methodology.

Function approximation. Surrogate construction is an instance of function approximation, which encompasses a broad set of techniques ranging from polynomial approximations like the Taylor series to machine learning approaches like Gaussian processes and neural networks (Trefethen, 2012; Rasmussen and Williams, 2005). The conventional wisdom is that compared to other approaches, neural networks excel at *feature extraction* (Huang and LeCun, 2006), converting function inputs (including discrete and structured inputs) into vectors which can then be processed by machine learning algorithms. Neural networks also excel when given a large amount of training data (Krizhevsky et al., 2012). Other function approximation approaches have different trade-offs relative to neural networks, and may be appropriate in circumstances with limited execution cost or data, or when requiring specific bounds on the behavior of the function approximation.

Program repair. Similar to surrogate adaptation, program repair techniques alter the semantics of a program to meet a downstream objective (Long and Rinard, 2016; Weimer et al., 2009; Perkins et al., 2009). These approaches typically make local changes to a program

in response to a single identified bug. In contrast, surrogate adaptation can change the entire behavior of the program to achieve good performance on a large dataset of examples.

Probabilistic programming. Probabilistic programming is a broad set of techniques for defining probabilistic models, then fitting parameters for these probabilistic models automatically given observations of real-world data (Cusumano-Towner et al., 2019; Goodman et al., 2008). When fitting parameters of a probabilistic program, such techniques require the program to be explicitly specified as a probabilistic program. The parameters are then optimized using inference techniques like Monte Carlo inference (Neal, 1993) and variational inference (Blei et al., 2017). In contrast, when optimizing parameters with surrogate optimization the original program can be specified in any form, while the parameters are optimized with stochastic gradient descent.

Differentiable programming. Differentiable programming is a set of techniques that calculates the derivatives of programs with respect to their input parameters (Baydin et al., 2018). In contrast with estimating the program's gradient with surrogate optimization, differentiable programming calculates the exact derivative without requiring the design and training processes of developing neural surrogates.

While differentiable programming is an appropriate alternative to surrogate optimization in contexts with smooth and continuous original programs, it struggles in cases where the original program is not smooth or is not continuous. For instance, differentiating through control flow constructs like branches and loops results in a discontinuity. Such control flow constructs can also induce a true derivative of 0 almost everywhere, which poses challenges for gradient-based optimization. Differentiable programming also relies on implementing the program in a language amenable to differentiable programming such as PyTorch or TensorFlow (Paszke et al., 2019; Abadi et al., 2016; Bischof et al., 1996).

In contrast, surrogate optimization approximates the program regardless of the provenance of its original implementation. This means that while some points in the original program may

be non-smooth, discontinuous, or have derivative 0, those points may be better behaved in the surrogate model (which approximates the original program) allowing for optimizing inputs with gradient descent despite challenges posed by the original program (Section 2.2, Figure 2.3).

Program smoothing. Chaudhuri and Solar-Lezama (2010) present a method to approximate numerical programs by executing the programs probabilistically. This approach lets Chaudhuri and Solar-Lezama apply gradient descent to optimize parameters of arbitrary numerical programs, similar to surrogate optimization. However, the semantics presented by Chaudhuri and Solar-Lezama only apply to a limited set of program constructs and do not easily extend to the set of program constructs exhibited by large-scale programs. In contrast, surrogate optimization estimates the gradients of arbitrary programs regardless of the constructs used in the program's implementation.

Automating construction of surrogates. Munk et al. (2022) present an approach for automatic construction of neural surrogates of stochastic simulators for surrogate compilation. Munk et al. propose an LSTM architecture that predicts the sequence of samples output by the original stochastic simulator. This approach is applicable to all stochastic simulators, regardless of the number or order of samples output by the original simulator. Munk et al. show that this surrogate executes faster than the original simulator. Though this approach addresses some questions of the neural surrogate programming methodology (specifically, how to design a surrogate for a given program), it does not address questions about how to train and how to deploy the surrogate.

3.5 Related Work Addressing Other Tasks

This section details approaches which, while related in that they use machine learning and programs together, are not examples of surrogates of programs. The intent is to clarify the scope of study of surrogates of programs.

Surrogates of non-programs. Surrogates of black-box processes (beyond just programs) are used across a wide variety of domains from computer systems to physical sciences (Sun and Wang, 2019; Carleo et al., 2019; Mendis et al., 2019a). For example, in my prior work (Mendis et al., 2019a) I train a surrogate of the execution behavior of Intel CPUs to predict the execution time of code. This is not an example of a surrogate of a program because this is performed without precise knowledge of the execution behavior of the CPU. This chapter focuses on constructing surrogates of programs for which there is an intensional representation of the semantics of the program (e.g., program source code) rather than developing surrogates of black-box functions.

Residual models. Another approach is training *residual models* on top of programs, neural networks that add to rather than simply replacing the original program's behaviors (Verma et al., 2019; Watson, 2019; Toderici et al., 2017). Formally, if the original program is a function $f(x)$ then the residual approach learns a neural network $g(x)$ and adds the result to that of the original program, such that the final program computes $f(x) + g(x)$. For example, Verma et al. (2019) train neural networks that augment programmatic reinforcement learning policies (Sutton and Barto, 2018). While learning such residual models is a form of programming, the neural networks are not surrogates of programs, and are thus out of scope for this thesis.

Programs synthesized to mimic neural networks. Several approaches in the literature train neural networks, taking advantage of their relative ease of training for high accuracy on downstream tasks, then synthesize a program that mimics the neural network (Bastani et al., 2018; Verma et al., 2018, 2019). For example, after training a residual model, Verma et al. (2019) synthesize a new program f' that mimics the original program with its residual: $f'(x) \approx f(x) + g(x)$. This class of approaches is also out of the scope of this thesis due to the significant differences in programming methodologies when synthesizing a program that mimics a neural network and developing a surrogate that mimics a program.

3.6 Discussion

In this chapter I have contributed a taxonomy that classifies the workflows demonstrated in the surrogate programming literature into three different design patterns: *surrogate compilation*, *surrogate adaptation*, and *surrogate optimization*. I have further demonstrated the shared methodology underlying each of these design patterns. I now discuss implications and future directions for research in this area.

More mechanization and systematization. I have presented a programming methodology for developing neural surrogates. However, this methodology is not mechanized: programmers still must manually navigate the trade-off space between desiderata. Future work in this domain should mechanize the various aspects of surrogate construction, from automating the surrogate's design based on the semantics of the original program, to automatically training the surrogate based on specifications and objectives over a data distribution, to automatically integrating the surrogate into the original program's execution context. While prior work has addressed some of these concerns (Esmaeilzadeh et al., 2012), fully mechanizing this process is an important direction for future work.

New design patterns. The three design patterns detailed in this chapter cover what I have observed in the literature, but there may be design patterns in use that I have not yet encountered, or design patterns that have not yet been invented. Future work can explore the space of surrogate programming design patterns to identify or invent new design patterns based on new use cases and new technologies like large language models (Chapter 7).

Understanding similar techniques. In Section 3.5 I discuss techniques which are tangentially related to surrogate programming, but fall outside of the scope of this thesis. Future work can explore the relationships between these techniques and surrogate programming, and whether it is possible to cross-pollinate ideas between these different areas of research.

Conclusion. In sum, this chapter validates the hypothesis that there is a small set of methodologically distinct design patterns, each unifying existing uses of surrogates in the literature, that can be grouped into a single programming methodology. This development builds a foundation on which the remainder of my thesis can study the application of surrogates of programs as well as the development of new tools that aid in the development of surrogates of programs.

Chapter 4

Why Focus on Surrogate Accuracy?

All surrogate-based design patterns share a common first step: training a surrogate of a program under study. Constructing this surrogate necessitates selecting a training dataset and neural network architecture, among other methodological choices. Unlike classical machine learning tasks which train on a black-box set of input-output pairs, when constructing a surrogate of a program we have access to the source code and semantics of the program under study. Throughout the remainder of this thesis, I investigate the hypothesis that we can guide surrogate design using facts derived from the modeled program to train surrogates more efficiently and achieve better performance (however measured) on downstream tasks.

There are many ways of measuring the efficiency with which a surrogate is trained and the performance of the resulting surrogate on a downstream task. Surrogate training has costs including the expert time required to design the surrogate's network architecture (see e.g., Section 2.3, which uses a domain-specific modification of the architecture proposed by Mendis et al. (2019a)), the cost of training data acquisition (which I discuss more in Chapter 5), and the resources required to actually train the surrogate on the training data including any hyperparameter or network architecture search that the developer performs (see e.g., Section 3.1.2). The performance of the resulting surrogate on a downstream task is measured by the error of the surrogate on its task, the speedup of the surrogate over

the original program, and (for surrogate optimization) the extent to which the surrogate allows for efficient optimization of the downstream task.[1] Downstream tasks may also have other metrics of interest, such as the resilience of the surrogate to out-of-distribution inputs: for example, Mendis (2020, Section 7.3.4) discusses the promise of surrogates of compiler components that generalize to out-of-distribution inputs.

Throughout the rest of the thesis, I focus on one particular metric: the accuracy of the surrogate in modeling the original program. I focus on this metric for several reasons. First, accuracy can be traded off against other metrics of interest in surrogate programming, meaning that optimizing for accuracy can also improve other metrics of interest. Second, accuracy against the original program is the only metric that is relevant across all surrogate programming design patterns, compared to metrics like speedup which are only relevant for some. Finally, accuracy is the response variable in experiments, and is thus the easiest metric to precisely and reliably measure. The remainder of this chapter walks through each of these reasons in turn, justifying this methodological choice.

4.1 Accuracy as a Proxy for Other Metrics

As with any approximate programming task (Stanley-Marbell et al., 2020; Sidiroglou-Douskos et al., 2011; Hoffmann et al., 2011), surrogate programming requires selecting a surrogate model that balances the tradeoffs between the accuracy of the surrogate and its costs to train and deploy. The machine learning models underlying surrogates allow developers to trade off between these metrics, for example by choosing a smaller neural network architecture to reduce the cost of training and deploying the surrogate at the expense of accuracy.

Training efficiency. In Chapter 6, I demonstrate a technique that improves the accuracy of a surrogate by between 27% and 50% on a set of applications. Without going into the

[1]The specific property for surrogate optimization is that all local minima of the surrogate are also local minima of the original program (and vice versa), and that the ordering of values of local minima is preserved between the surrogate and the original program.

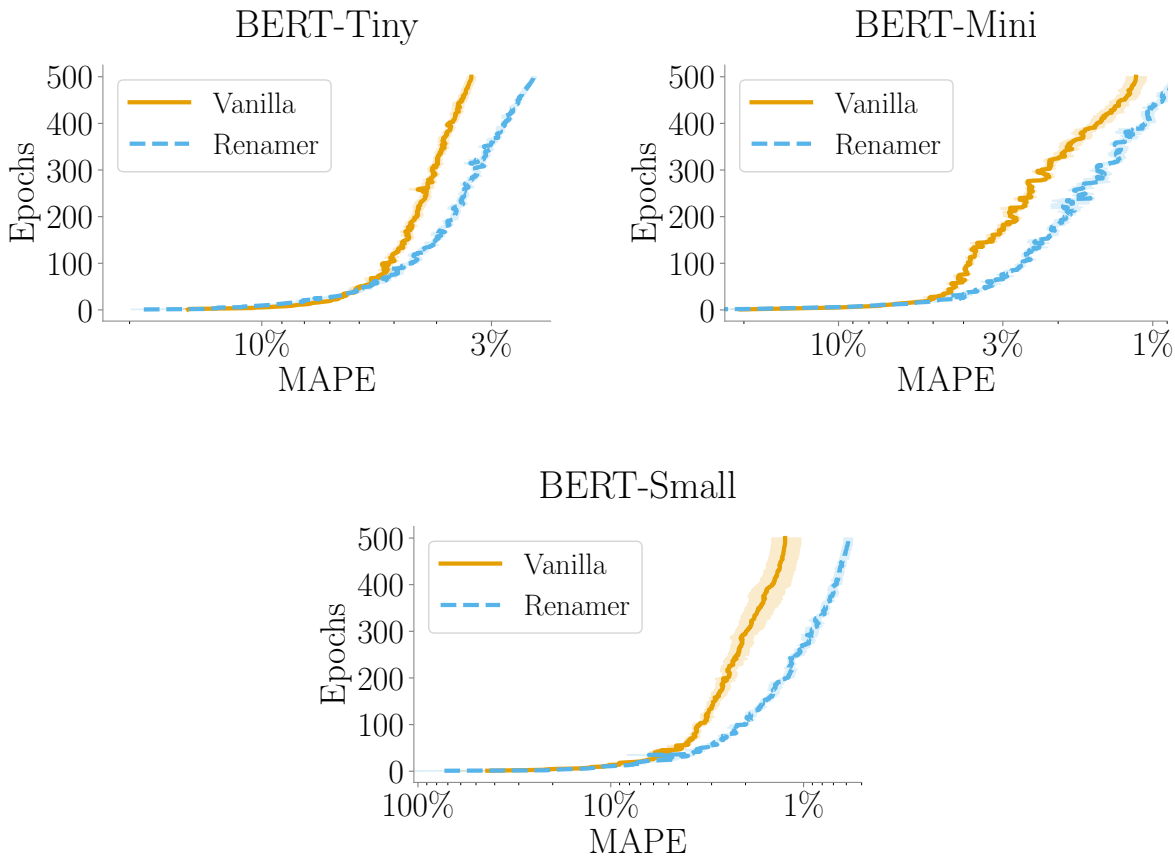


Figure 4.1: Training efficiency of a surrogate of llvm-mca, using a baseline model architecture (Vanilla, in orange) and an improved model architecture (Renamer, in blue). Each plot is of a different model size (BERT-Tiny, BERT-Mini, and BERT-Small). Each plot shows the amount of training required (on the y axis, lower is better) to reach a given test error (on the x axis). Renamer requires fewer training steps to reach a given test error than the baseline Vanilla model.

details of the technique, I demonstrate that this improvement in accuracy can significantly speed up training a surrogate, reducing the time to train a surrogate to a given accuracy target by between 25% and 59%.

Figure 4.1 plots training efficiency curves for a surrogate of llvm-mca (Section 2.1.1) using a baseline model (Vanilla, in orange) and the higher-accuracy variant that I develop (Renamer, in blue). Each plot is of a different model size (BERT-Tiny, BERT-Mini, and BERT-Small). Each plot shows the amount of training required (on the y axis, lower is better) to reach a given test error (on the x axis).

Across all model sizes, Renamer achieves the same performance with fewer training steps than the vanilla model: Renamer reaches the best error of the vanilla model with 213, 123, and 290 fewer epochs for the Tiny, Mini, and Small architectures respectively, which corresponds to a relative decrease in steps to achieve the same performance of 42.77%, 24.75%, and 59.06%. Thus the accuracy improvement of the technique can equivalently be used to instead speed up training the surrogate, reducing the cost of surrogate training.

Deployment efficiency. In Chapter 5, I demonstrate a technique that improves the accuracy of a surrogate by an average of 5% across a range of applications. Such a decrease in error can significantly affect the performance of a system built with surrogate programming.

For example, consider Table 4.1. This table shows the results of a hyperparameter search to choose the fastest-to-execute neural network that meets an error threshold of 10%. This table is a replication of Table 3.1 in the surrogate compilation case study in Section 3.1.2, with an added column of "Error - 5%". With the original error, the methodology dictated choosing the network with size 64, which has a 1.57× speedup; however, a decrease in error of 5% would result instead in choosing the network with size 32, which has a 2.01× speedup, a 28% improvement in application performance.

Section 5.4.2 presents another example of using an accuracy improvement to improve the efficiency of a downstream task. Again, in both of these contexts the accuracy improvement

Table 4.1: The validation error and speedup of BERT models over a range of candidate embedding widths. This table is a replication of Table 3.1 in Section 3.1.2, with an added column of "Error - 5%".

Embedding Width	Error	Error - 5%	Speedup over W=128
128	8.9%	8.5%	1×
64	9.5%	9.0%	1.57×
32	10.1%	9.6%	2.01×
16	10.8%	10.3%	2.22×

of the technique can equivalently be used to instead speed up deployment of the surrogate, reducing the cost of surrogate deployment. Reporting accuracy is thus a useful proxy for other metrics of interest in surrogate programming.

4.2 Accuracy as the Main Measurable Metric of Interest

Beyond being possible to parlay into other metrics, accuracy is also the primary metric of interest across all surrogate programming design patterns.

All design patterns share a common first step: training a surrogate of a program under study. High accuracy is a desirable property of a surrogate in all design patterns (indeed, for a given set of hyperparameters and training cost constraint, developers train the surrogate to minimize a loss function that is a proxy for maximizing accuracy). Other metrics of interest (e.g., speedup) are only relevant for some design patterns.

Similarly, other metrics can be hard to measure, as they are often only relevant in the context of a downstream task. For example in surrogate adaptation, it is not possible to measure the ability of the surrogate to adapt to its downstream tasks; instead, we must adapt and then measure the accuracy of the adapted surrogate on the downstream task. In surrogate optimization, it is not possible to measure the ability of the surrogate to optimize its input parameters; instead, we must perform surrogate optimization and measure the

results. Accuracy is the most generalizable metric of interest, as it is relevant for all design patterns and can be measured in isolation from other metrics of interest.

4.3 Accuracy as the Response Variable

Finally, accuracy is the response variable in experiments: for a given set of hyperparameters and training cost constraint, we train a surrogate and can measure the accuracy of the resulting surrogate. This is in contrast to the various efficiency metrics, which are essentially independent variables. For example, we can control the size of the neural network architecture and measure the resulting accuracy, but we cannot control the accuracy and measure the resulting size of the neural network architecture. As a consequence, optimizing other variables while controlling for accuracy can be quite noisy, as the accuracy of the resulting surrogate is not a deterministic (or even monotonic) function of other hyperparameters. Thus, accuracy is the most natural variable to measure and optimize in experiments.

Together, these three reasons motivate my focus on improving the accuracy of surrogates throughout the remainder of this thesis.

Chapter 5

TURACO: Complexity-Guided Data Sampling for Training Neural Surrogates of Programs

I now focus on techniques to improve the accuracy of surrogates of programs. Among the design decisions in the neural surrogate programming methodology detailed in Section 3.3, one of the most important across all machine learning applications is training dataset selection. Without a fundamental shift in machine learning training paradigms, data quality will remain of paramount importance to the quality of a surrogate model. I now demonstrate that we can leverage facts about the program under study to guide the selection of training data for a surrogate of that program, resulting in a higher quality surrogate.

Dataset Generation. In each surrogate programming design pattern, training a surrogate of a program requires measuring the behavior of the program on a dataset of input examples. There are three common approaches to collecting this dataset. The first is to use data instrumented from running the original program on a workload of interest (Chapter 2; Esmaeilzadeh et al., 2012). In the absence of an available workload, another is to uniformly sample (or another manually defined distribution) from the input space of the program (Tseng

et al., 2019; Kustowski et al., 2020). The third is to use *active learning* (Settles, 2009), a class of online methods that iteratively query labels for the data points that are most useful (however defined) for training the surrogate (İpek et al., 2006; She et al., 2019; Pestourie et al., 2020).

Each of these approaches face challenges on programs with different behaviors in different regions of the input space. For example, in Section 2.5 I identify a scenario in which an instrumented dataset does not exercise a set of control flow paths in the program enough times for the surrogate to learn the program's behavior along those paths, resulting in a surrogate that generates highly inaccurate predictions for inputs in the regions of the input space corresponding to those paths.

Approach. Rather than treating the program as a black box, my approach uses the source code and semantics of the program under study to guide dataset generation for training a surrogate of the program. The core concept is to allocate samples based on both the *complexity* of learning the program's behavior on a given path and the frequency of that path in the input data distribution.

Complexity-guided sampling. The objective is to find how many samples to allocate to each region of the input space to minimize the expected error of the resulting surrogate. To reason about the error of a surrogate, I use neural network sample complexity bounds for learning analytic functions (Arora et al., 2019; Agarwala et al., 2021). These bounds give an upper bound on how many samples are required to learn a surrogate of an analytic function to a given error as a function of a *complexity measure* of that function. The approach calculates a complexity measure for the function induced by each control flow path in the program and combines that with the frequency of each path according to an input data distribution. The output of the approach is the proportion of samples to allocate to each region of the input space, minimizing an upper bound on the surrogate's error.

Utility. The core assumption for this approach's utility is that there is a cost to generating data. Under this assumption, surrogate developers trade off between costs incurred to data generation and costs incurred due to the error of the resulting surrogate. Thus, my approach is applicable under two scenarios. In the *offline* scenario, a developer can sample inputs from any given region of the input space. In the *online* scenario, a developer is presented with a stream of inputs, and must decide whether or not to evaluate the program on that input. In both contexts, my approach results in the lowest upper bound on the surrogate's error for a given number of evaluated samples.[1]

Stratified functions. The core modeling assumption is to represent the program as a *stratified function*, a piecewise[2] function across different regions (*strata*) of the input space. I use *stratified surrogates* to model such functions. A stratified surrogate consists of independent surrogates of each component of the stratified function. During evaluation, a stratified surrogate uses the original program to check which stratum an input is in, then applies the corresponding surrogate.

Complexity analysis. I present a programming language, Turaco, in which programs denote stratified functions with well-defined complexity measures (specifically, stratified analytic functions). I provide a static program analysis for Turaco programs that automatically calculates an upper bound on the complexity of each component of the stratified function that the program denotes.

Evaluation. To demonstrate that complexity-guided sampling using the complexity analysis improves surrogate error on downstream tasks, I evaluate the approach on a range of programs, finding that across this selection of programs complexity-guided sampling improves error relative to baseline distributions by around 5%. I demonstrate that a 5% improvement in

[1] A more fine-grained approach for the online scenario would be to model both the cost of taking an input from the stream and the cost of evaluating that input, but this is beyond the scope of this chapter.

[2] I choose the term *stratified* by analogy with the technique of stratified sampling.

error of a surrogate can result in a 28% improvement in execution speed in an application with a maximum error threshold. I then analyze the classes of problems for which complexity-guided sampling excels, finding potential improvements in error of up to 30%, and the classes of problems for which complexity-guided sampling using Turaco's complexity analysis sampling fails, finding deteriorations in error of up to 500%.

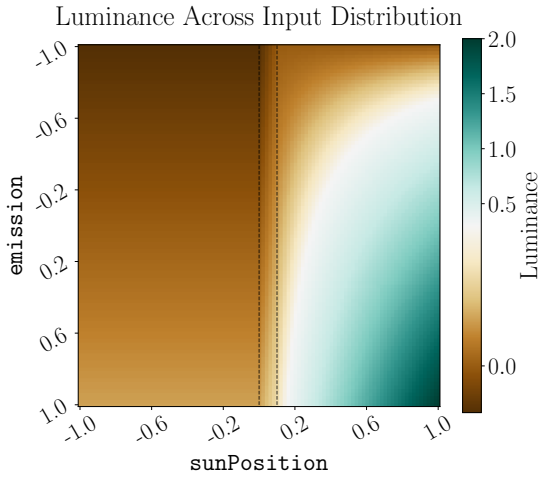
Renderer demonstration. I further present a case study of learning a surrogate of a renderer in a video game engine. I show that the complexity-guided sampling approach results in between 17% and 44% lower error than training using baseline distributions that do not take into account path complexity. These error improvements correspond with perceptual improvements in the generated renders.

Contributions. I present the following contributions:

- An approach to allocating samples among strata to train stratified neural network surrogates of stratified analytic functions that minimizes an upper bound on the surrogate's error.
- A programming language, Turaco, in which all programs are stratified analytic functions, and a program analysis to bound the complexity of learning surrogates of those programs.
- Empirical evaluations on real-world programs demonstrating that complexity-guided sampling using Turaco's complexity analysis results in empirical improvements in error, and that these improvements in error result in improvements in downstream applications.
- Further empirical evaluations of the classes of problems where complexity-guided sampling using Turaco's complexity analysis succeeds and fails.

```
1 fun (sunPosition, emission) {
2   if (sunPosition < 0) {
3     ambient = 0;
4   } else {
5     ambient = sunPosition;
6   }
7   if (sunPosition < 0.1) {
8     emission *= 0.1;
9   } else {
10    emission *= sunPosition;
11  }
12  return ambient + emission;
13 }
```

(a) Graphics program calculating the luminance of each pixel in the scene as a function of ambient light and material properties.



(b) Output of the program on inputs in $[-1, 1]$, with dashes separating the three paths.

```
// assume: sunPosition < 0
fun (sunPosition, emission) {
  ambient = 0;
  emission *= 0.1;
  return ambient + emission;
}
```

(c) Nighttime (`ll`) path.

```
// assume 0<sunPosition<0.1
fun (sunPosition, emission) {
  ambient = sunPosition;
  emission *= 0.1;
  return ambient + emission;
}
```

(d) Twilight (`rl`) path.

```
//assume: sunPosition > 0.1
fun (sunPosition, emission) {
  ambient = sunPosition;
  emission *= sunPosition;
  return ambient + emission;
}
```

(e) Daytime (`rr`) path.

Figure 5.1: Example program, outputs, and traces.

5.1 Example

Figure 5.1a presents an example distilled from the evaluation (Section 5.4.2) that I use to demonstrate how complexity-guided sampling results in a more accurate surrogate than *frequency-based sampling*, sampling according to the frequency of paths alone.

Program under study. Figure 5.1a presents a graphics program that calculates the luminance (i.e., brightness) at a point in a scene as a function of `sunPosition`, the height of the sun in the sky (i.e., the time of day), and `emission`, which describes how reflective the material is at that point.

The program first checks whether it is nighttime (Section 5.1), and sets the ambient lighting variable to zero accordingly. The program next checks whether the sun position is below a threshold indicating direct sunlight (Section 5.1) and sets the emission variable accordingly. The output is then the sum of the ambient light and the light emitted by the material. Figure 5.1b presents the output of this program over the valid input range of `sunPosition` and `emission` (i.e., between -1 and 1 for both variables).

The path conditions (Section 5.1) partition the program into three traces: nighttime, when `sunPosition` is less than 0 (Figure 5.1c); twilight, when `sunPosition` is between 0 and 0.1 (Figure 5.1d); and daytime, when `sunPosition` is greater than 0.1 (Figure 5.1e). These paths are separated by dashed black lines in Figure 5.1b.

Complexity. Training a surrogate of this program poses a particular challenge because these traces have not only different behavior but also different relative complexities: when `sunPosition` is less than 0.1 the function is linear, but when `sunPosition` is above 0.1 the function is quadratic. This notion of complexity is quantified by the *sample complexity* of each trace: traces that are more complex require more samples to learn to a given error than traces that are less complex. Figure 5.2a presents the error as a function of the training dataset size of surrogates of each trace trained in isolation, showing that indeed the quadratic daytime path has the highest error, followed by twilight then nighttime.

Complexity-guided sampling. The objective is to find the number of data points to sample from each path to minimize the expectation of error of a surrogate of the overall program, given a data distribution and a data budget. To accomplish this, the approach leverages the complexity of each path and the frequency of each path in the data distribution, prioritizing sampling paths that are more complex (requiring more samples to learn) and that are more frequent (and thus more important to learn).

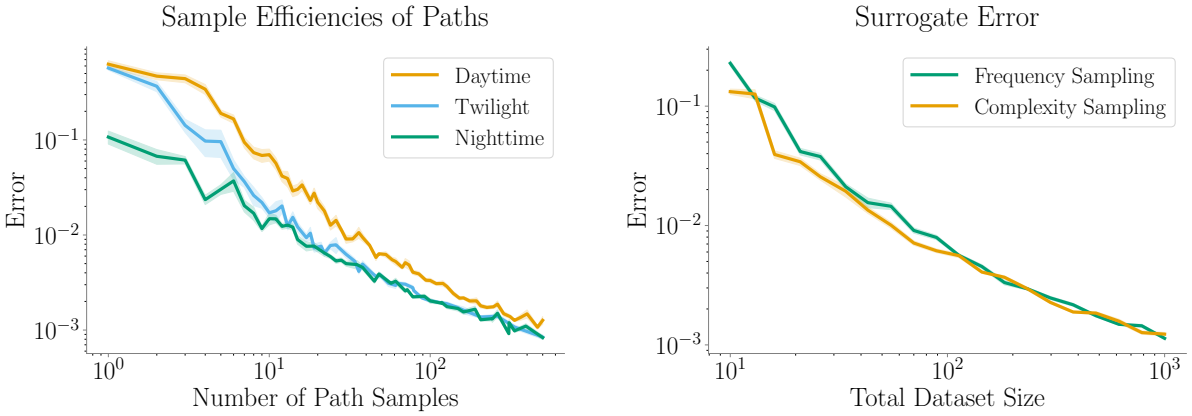
First the approach determines the sample complexity of each trace along each path, the number of samples required to learn a surrogate of the trace (in isolation) to a given error. The

approach extends the sample complexity results of Agarwala et al. (2021), who give an upper bound on the number of samples required to learn a neural network approximation of a given analytic function. Using this bound (as implemented by the TURACO analysis in Section 5.3), the approach determines that the twilight path takes 1.4× as many samples to train a surrogate to a given error as the nighttime path, and the daytime path requires 3.7× as many samples.

Then given a distribution with the frequency of each path, the approach determines the complexity-guided sampling rates for each path. In this example I assume that the data has a uniform distribution over inputs between −1 and 1, resulting in path frequencies for the nighttime path (`sunPosition < 0`) of 50%, the twilight path (`0 < sunPosition < 0.1`) of 10%, and the daytime path (`0.1 < sunPosition`) of 40%. With this, the approach determines that the nighttime path should be sampled at 36.9% of the data budget (undersampling relative to its frequency because it is simple to learn), the twilight path at 14.0%, and the daytime path at 49.1% (oversampling relative to its frequency because it is complex to learn).

Stratified surrogates. The class of surrogate model for which I derive the above approach is that of a *stratified neural surrogate* – a set of disjoint neural networks which are applied based on which path the inputs induce in the program. Concretely, this means that I train one surrogate per path, and pick which to apply for each input at evaluation time. For this example program, picking which surrogate to apply just requires comparing `sunPosition` against constant threshold values.

Results. Figure 5.2b presents the error as a function of the training dataset size of stratified surrogates of the entire program for a baseline of sampling according to path frequency alone and for complexity-guided sampling. Figure 5.2b shows that the complexity-guided sampling approach results in lower error than sampling according to path frequency alone. For datasets of total size below 70 samples, the surrogate trained with complexity-guided sampling has a geometric mean decrease in error of 27.5%. For datasets of total size above 70 samples, the surrogate trained with complexity-guided sampling has a geometric mean decrease in error of



(a) Per-path surrogate errors (log-log plot).

(b) Stratified surrogate errors (log-log plot). The approach decreases the error by 15%.

Figure 5.2: Per-path surrogate errors (left) and combined errors (right) for the example.

5.5%. Across the entire range of dataset sizes evaluated in this plot, the surrogate trained with complexity-guided sampling has a geometric mean decrease in error of 15%. In sum, the approach results in a surrogate that produces a more accurate luminance calculation, and therefore a better final output from the graphics program, than a surrogate trained using frequency-based path sampling.

5.2 Complexity-Guided Sampling

In this section I present the stratified surrogate sample allocation problem and derive my solution, complexity-guided stratified surrogate dataset selection.

5.2.1 The Stratified Surrogate Sample Allocation Problem

The goal is to learn a *stratified surrogate*, $\hat{f}$, of a *stratified function*, f, constrained by a *sample budget*, n, that defines the number of data samples to be used by the learning algorithm.

I approach this problem through the definition of a stratified function as a piecewise function; I term each piece a *stratum*. I then define a stratified surrogate as a stratified function

itself, with each stratum a surrogate of a corresponding stratum of the stratified function. Learning a stratified surrogate therefore requires learning a surrogate for each stratum.

I assume a technique for learning a surrogate of a function, f, given a sample budget. The precise goal is thus to partition the overall sample budget, n, into per-stratum sample budgets for each stratum of the stratified function, with the objective of minimizing the overall error of the stratified surrogate.

Stratified Functions and Surrogates

I define a *stratified function* f as follows:

$$f(x) \triangleq \begin{cases} f_1(x) & \text{if } x \in s_1 \\ \vdots & \\ f_c(x) & \text{if } x \in s_c \end{cases}$$

where f and each f_i is a function from inputs $x : \mathcal{X}$ to outputs $y : \mathcal{Y}$, c is the number of strata, $\{s_i\}_{i=1}^{c}$ are strata, and where $\cup_i s_i = \mathcal{X}$ and $\forall i \neq j.\ s_i \cap s_j = \varnothing$. I define a *stratified surrogate* $\hat{f}$ as a stratified function with components $\hat{f}_i$.

The Stratified Surrogate Sample Allocation Problem

To restate, the goal is to learn a stratified surrogate $\hat{f}$ of a stratified function f.

Formally, I define a *learning algorithm*, a function that learns a surrogate of a given input function, as a random function $tr : (\mathcal{X} \to \mathcal{Y}) \times \mathcal{D} \times \mathbb{N} \times (\mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}) \to (\mathcal{X} \to \mathcal{Y})$ that takes a function $f : \mathcal{X} \to \mathcal{Y}$ from inputs $x : \mathcal{X}$ to outputs $y : \mathcal{Y}$, a distribution $D : \mathcal{D}$ over inputs x, a number of training examples $n : \mathbb{N}$, and a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$ which measures the cost of an incorrect prediction, and returns a function (representing the output surrogate) $\hat{f} : \mathcal{X} \to \mathcal{Y}$.

I also define notation for data distributions D. Let $D(x)$ be the probability that x is sampled from D, and $\int_{x \in s_i} D(x)\mathrm{d}x$ be the probability mass of all data points within s_i over D (reducing to a summation for discrete distributions). Let $D(x|s_i)$, the distribution of x within stratum s_i, be defined as:

$$D(x|s_i) \triangleq \begin{cases} \frac{D(x)}{\int_{x' \in s_i} D(x')\mathrm{d}x'} & x \in s_i \\ 0 & \text{otherwise} \end{cases}$$

I next define a *stratified learning algorithm*. A stratified learning algorithm learns a stratified surrogate of a stratified function by learning each component surrogate independently (given their respective dataset budgets). I use the following notation to denote the operation of a stratified learning algorithm, where $\vec{n}$ is a vector of sample budgets for each stratum:

$$\hat{f} \sim tr(f, D, \vec{n}, \ell) \triangleq \left\{ \hat{f}_i \sim tr(f_i, D(x|s_i), \vec{n}_i, \ell) \right\}$$

I now formalize the stratified surrogate sample allocation problem:

$$\arg\min_{\vec{n}} \mathbb{E}_{\hat{f} \sim tr(f, D, \vec{n}, \ell)} \left[\mathop{\mathbb{E}}_{x \sim D} \left[\ell\Big(\hat{f}(x), f(x) \Big) \right] \right] \text{ such that } \sum_i \vec{n}_i \leq n \tag{5.1}$$

The objective of this problem is to find a vector of per-stratum sample budgets $\vec{n}$ that in the expectation over the outcomes of the stratified surrogate learning algorithm (the outer expectation) minimize the expected loss over the data distribution (the inner expectation), subject to a constraint that the total number of samples used is no more than n.

5.2.2 Complexity-Guided Stratified Surrogate Dataset Selection

In this section, the goal is to solve Equation (5.1). To solve this optimization problem, we need to model the relationship between the sample budget afforded to the learning algorithm for each stratum and the error of the resulting surrogate. I leverage the PAC learning framework

for neural networks to derive a conservative probabilistic upper bound on the error of the surrogate. I then solve this optimization problem with the derived upper bound in place of the original objective.

PAC Learning

To reason about the error of a surrogate, I use the *probably approximately correct* (PAC) learning framework (Valiant, 1984). The PAC learning framework bounds the number of examples needed to learn a surrogate as a function of the allowable error threshold for the surrogate.

Equation (5.2) defines a given function f as *probably approximately correctly learnable* (Valiant, 1984) (abbreviated as learnable) for a given learning algorithm tr and loss function ℓ if for all distributions D, with probability $1-\delta$ the learning algorithm returns a surrogate $\hat{f}$ that approximately matches the original function f over the distribution D (i.e., the expectation of the error is bounded by ϵ):

$$\forall D, \epsilon \in (0,1), \delta \in (0,1). \exists n. \underset{\hat{f} \sim tr(f,D,n,\ell)}{\mathrm{P}} \left(\mathop{\mathbb{E}}_{x \sim D} \left[\ell\left(\hat{f}(x), f(x) \right) \right] \leq \epsilon \right) \geq 1 - \delta \tag{5.2}$$

Neural Network Sample Complexity Measures

It is an open problem to determine the exact relationship between the number of samples n and the target error threshold ϵ in the PAC bound (Equation (5.2)) for neural networks on arbitrary target functions f. Rather than use the exact relationship, I use an upper bound on ϵ as a function of n, and minimize the induced upper bound.

Arora et al. (2019) and Agarwala et al. (2021) present such an upper bound for learning analytic functions f with neural networks. Agarwala et al. define a *sample complexity measure* $\zeta(f) \in \mathbb{R}_{\geq 0}$, where higher values denote functions that require more samples n to learn f to a given error ϵ. With this sample complexity measure $\zeta(f)$, Equation (5.2) holds

for all analytic f, n, D, ϵ, and δ with:

$$\exists K.\, \epsilon \leq K\sqrt{\frac{\zeta(f) + \log(\delta^{-1})}{n}} \tag{5.3}$$

where K is an unknown constant.

Agarwala et al. define $\zeta(f)$ using the *tilde* $\tilde{f}$ of f, defined as follows for univariate functions:

$$f(x) = \sum_{k=0}^{\infty} a_k x^k \qquad \tilde{f}(x) \triangleq \sum_{k=0}^{\infty} |a_k| x^k \tag{5.4}$$

The tilde measures the magnitude of each coefficient of f's analytic representation; this measures the influence of hard-to-model higher-order terms. I work with the following generalization of the tilde to multivariate analytic functions, where $\vec{x}\|1$ denotes concatenating a 1 to $\vec{x}$:

$$f(\vec{x}) = \sum_{k=0}^{\infty} \sum_{v \in V_k} a_v \prod_{i=1}^{k} (\beta_{v,i} \cdot \vec{x}\|1) \qquad \tilde{f}(x) = \sum_{k=0}^{\infty} \left(\sum_{v \in V_k} |a_v| \prod_{i=1}^{k} \|\beta_{v,i}\|_2 \right) x^k \tag{5.5}$$

Agarwala et al. present the multivariate generalization; I contribute the novel generalization to $\vec{x}\|1$, which allows us to handle functions that are not analytic around 0 such as log.

With the definition of the tilde, I now present Agarwala et al.'s core theorem, which says that the tilde induces a sample complexity measure for analytic functions:

Theorem 1. *For a sufficiently wide (see Arora et al. (2019, Theorem 5.1)) 2-layer neural network trained with gradient descent for sufficient steps (ibid.), if f is analytic, $\vec{x}$ is on the d-dimensional unit sphere, and ℓ is 1-Lipschitz, then $f(\vec{x})$ is learnable in the sense of Equations (5.2) and (5.3) with:*

$$\zeta(f) = \tilde{f}'(1)^2$$

I present the proof of this theorem in Section 5.2.3. The proof is a novel extension to inputs $\vec{x}\|1$ of Agarwala et al. (2021)'s proof.

Complexity-Guided Stratified Surrogate Dataset Selection

Coming back to the stratified surrogate sample allocation problem (Equation (5.1)), the goal is to find per-stratum sample budgets $\vec{n}$ that minimize the expectation of error of the stratified surrogate. To help solve this optimization problem, we can refactor Equation (5.1) to separate out each stratum as follows:

$$\arg\min_{\vec{n}} \mathop{\mathbb{E}}_{s_i \sim \int_{x \in s_i} D(x)\mathrm{d}x} \left[\mathop{\mathbb{E}}_{\hat{f}_i \sim tr(f_i, D(x|s_i), \vec{n}_i, \ell)} \left[\mathop{\mathbb{E}}_{x \sim D(x|s_i)} \left[\ell\Big(\hat{f}_i(x), f_i(x)\Big) \right] \right] \right] \quad \text{such that} \quad \sum_i \vec{n}_i \leq n \tag{5.6}$$

This refactoring exploits that a stratified learning algorithm learns each surrogate independently:[3] I decompose the expected loss of the learning algorithm into the expectation over strata (the outermost expectation in Equation (5.6)) of the expectation over the outcomes of the surrogate learning algorithm on that stratum (the middle expectation) of the expectation of the expectation of the loss over the data distribution on that stratum (the inner expectation).

Predicted Error of a Surrogate

Instead of optimizing Equation (5.6) directly, my approach is to optimize the conservative probabilistic upper bound ϵ_i given by the PAC framework for each surrogate.

I define the *predicted error* $\hat{\epsilon}_{f_i, n_i, \delta_i}$ of a stratified surrogate component to be the upper bound (with probability $1 - \delta_i$) of the error of the surrogate $\hat{f}_i$ against the function f_i. Concretely, the predicted error is the error for a given n_i and δ_i assuming that Equation (5.3) is tight with $K = 1$ (the value of K cancels out in the analysis, so this choice is just for notational convenience):

$$\hat{\epsilon}_{f_i, n_i, \delta_i} \triangleq \sqrt{\frac{\zeta(f_i) + \log\big(\delta_i^{-1}\big)}{n_i}} \tag{5.7}$$

[3]Specifically, I decompose the innermost expectation in Equation (5.1) over strata using the law of total expectation, move the expectation over strata to the outside using that expectation is linear, then rewrite the expectation over the stratified learning algorithm to be the expectation over the single stratum under consideration, using that the stratified learning algorithm learns the surrogate for each stratum independently.

I then replace the expectation of error in Equation (5.6) in each stratum with the predicted error of that stratum, resulting in the objective that the approach optimizes:

$$\arg\min_{\vec{n}} \mathop{\mathbb{E}}_{s_i \sim \int_{x \in s_i} D(x)\mathrm{d}x} [\hat{\epsilon}_{f_i, \vec{n}_i, \delta_i}] \quad \text{such that} \quad \sum_i \vec{n}_i \leq n \tag{5.8}$$

The objective of this problem is to find a vector of per-stratum sample budgets $\vec{n}$ that in the expectation over strata (the outer expectation) minimize the predicted error of the surrogate for that stratum, subject to a constraint that the total number of samples used is no more than n.

Finally we can solve this optimization problem. For a given stratified function f, sample budget n, and per-stratum failure probabilities δ_i:

Theorem 2. *Equation (5.8) is minimized at:*

$$\vec{n}_i = n \frac{\left(\left(\int_{x \in s_i} D(x)\mathrm{d}x\right)\sqrt{\zeta(f_i) + \log\left(\delta_i^{-1}\right)}\right)^{\frac{2}{3}}}{\sum_{j=1}^{c}\left(\left(\int_{x \in s_j} D(x)\mathrm{d}x\right)\sqrt{\zeta(f_j) + \log\left(\delta_j^{-1}\right)}\right)^{\frac{2}{3}}} \tag{5.9}$$

Theorem 2 defines how much data the complexity-guided sampling approach samples from each stratum. Specifically, data is sampled from each stratum proportionally to:

$$\left(\left(\int_{x \in s_i} D(x)\mathrm{d}x\right)\sqrt{\zeta(f_i) + \log\left(\delta_i^{-1}\right)}\right)^{\frac{2}{3}}$$

This term incorporates the frequency of that stratum ($\int_{x \in s_i} D(x)\mathrm{d}x$), the complexity of that stratum ($\zeta(f_i)$), and a term from the failure probability δ_i. I present the proof in Section 5.2.3.

For convenience, throughout the rest of this chapter I assume that all δ_i are set to be equal ($\forall i, j.\, \delta_i = \delta_j$). Because each surrogate training is independent, this induces an overall PAC failure probability $\delta = 1 - \prod_i (1 - \delta_i)$.

Tightness of the Predicted Error Optimization

Note that the optimal solution to Equation (5.8) is not necessarily the optimal solution to Equation (5.1). First, optimizing the predicted error is not the same as optimizing the expectation of error: specifically, there is a gap between the optimal solution to Equation (5.8) and the optimal solution to Equation (5.1). Assuming that the per-example loss is bounded by some value L, the expectation of error found by the optimal $\vec{n}$ for Equation (5.8) is bounded by:

$$\mathop{\mathbb{E}}_{s_i \sim \int_{x \in s_i} D(x)\mathrm{d}x} \left[(1 - \delta_i) \sqrt{\frac{\zeta(f_i) + \log\left(\delta_i^{-1}\right)}{\vec{n}_i}} + \delta_i L \right] \tag{5.10}$$

Second, the bound on the predicted error itself may be loose. Note that while the predicted error itself may be a loose bound on the error, the approach does not require exact values from these bounds, but instead compares the predicted error of each different component of the stratified function to minimize the overall predicted error.

5.2.3 Proofs

This section presents proofs of Theorems 1 and 2, supporting lemmas, and other results.

Complexity Calculus

Theorem 1. *For a sufficiently wide (see Arora et al. (2019, Theorem 5.1)) 2-layer neural network trained with gradient descent for sufficient steps (ibid.), if f is analytic, $\vec{x}$ is on the d-dimensional unit sphere, and ℓ is 1-Lipschitz, then $f(\vec{x})$ is learnable in the sense of Equations (5.2) and (5.3) with:*

$$\zeta(f) = \tilde{f}'(1)^2$$

Proof. The proof is similar to that of Theorem 8 in Agarwala et al. (2021), with two deviations.

First, note that Equation 15 in Agarwala et al. (2021) has a typo, which I correct below:

$$\sqrt{M_g} = \sum_k k|a_k|\|\beta_k\|_2^k = \|\beta_k\|_2 \sum_{k=1}^{\infty} k|a_k|\|\beta_k\|_2^{k-1}$$

Thus, the $|a_0|$ term is not necessary, meaning that the $\tilde{g}(0)$ term in Equation 14 in Agarwala et al. (2021) is not necessary. Second, I note that the proof of Corollary 3 in Agarwala et al. (2021) involves appending a 1 to the neural network input; thus, the complexity of learning any function $f(\vec{x})$ is the same as learning the complexity of a function $f(\vec{x}\|1)$. Otherwise the proof is identical to Theorem 8 in Agarwala et al. (2021). □

Complexity-Guided Sampling

Theorem 2. *Equation (5.8) is minimized at:*

$$\vec{n}_i = n \frac{\left(\left(\int_{x\in s_i} D(x)\mathrm{d}x\right)\sqrt{\zeta(f_i) + \log\left(\delta_i^{-1}\right)}\right)^{\frac{2}{3}}}{\sum_{j=1}^{c}\left(\left(\int_{x\in s_j} D(x)\mathrm{d}x\right)\sqrt{\zeta(f_j) + \log\left(\delta_j^{-1}\right)}\right)^{\frac{2}{3}}} \tag{5.9}$$

Proof. The task is to find n_i for each surrogate $\hat{f}_i$ that find a minimal error ϵ using (using the upper bound in Equation (5.3)), while also meeting the total sample size constraint: $\sum_i n_i \leq n$.

For convenience, define:

$$p_i \triangleq \left(\int_{x\in s_i} D(x)\mathrm{d}x\right)\sqrt{\left[\zeta(f_i) + \log\left(\delta_i^{-1}\right)\right]}$$

We must show the following KKT conditions:

$$\begin{aligned}
\mathcal{L}(n_i, \mu_n, \{\mu_i\}) &= \mathop{\mathbb{E}}_{s_i\sim\left\{\left(\int_{x\in s_i} D(x)\mathrm{d}x\right)\right\}}\left[\sqrt{\frac{1}{n}\left[\zeta(f_i) + \log\left(\delta_i^{-1}\right)\right]}\right] + \mu_n\left(\sum_i n_i - n\right) + \sum_i \mu_i n_i \\
&= \sum_i p_i n_i^{-\frac{1}{2}} + \mu_n\left(\sum_i n_i - n\right) + \sum_i \mu_i n_i
\end{aligned}$$

Stationarity: $\forall i.\, 0 = -\frac{1}{2}p_i n_i^{-\frac{3}{2}} + \mu_n + \mu_i$

Primal feasibility: $\sum n_i \leq n$ and $\forall i.\, 0 \leq n_i$

Dual feasibility: $0 \leq \mu_n$ and $\forall i.\, 0 \leq \mu_i$

Complementary slackness : $0 = \mu_n \left(n - \sum_i n_i \right)$ and $\forall i.\, \mu_i n_i = 0$

I now show that the following is a solution to the optimization problem:

$$\begin{aligned} n_i &= n \frac{p_i^{\frac{2}{3}}}{\sum_j p_j^{\frac{2}{3}}} \\ \mu_n &= \left(\frac{1}{n} \sum_i \left(\frac{1}{2} p_i \right)^{\frac{2}{3}} \right)^{\frac{3}{2}} \\ \mu_i &= 0 \end{aligned}$$

Stationarity.

$$\begin{aligned} -\frac{1}{2} p_i n_i^{-\frac{3}{2}} + \mu_n + \mu_i &= -\frac{1}{2} p_i \left(n \frac{p_i^{\frac{2}{3}}}{\sum_j p_j^{\frac{2}{3}}} \right)^{-\frac{3}{2}} + \left(\frac{1}{n} \sum_j \left(\frac{1}{2} p_j \right)^{\frac{2}{3}} \right)^{\frac{3}{2}} + 0 \\ &= -\frac{1}{2} p_i p_i^{-1} n^{-\frac{3}{2}} \left(\sum_j p_j^{\frac{2}{3}} \right)^{\frac{3}{2}} + \frac{1}{2} n^{-\frac{3}{2}} \left(\sum_j p_j^{\frac{2}{3}} \right)^{\frac{3}{2}} \\ &= 0 \end{aligned}$$

Primal feasibility.

$$\begin{aligned} n_i = n \frac{p_i^{\frac{2}{3}}}{\sum_j p_j^{\frac{2}{3}}} &\geq 0 \quad (\text{if } n \geq 0) \\ \sum_i n_i = n \frac{1}{\sum_j p_j^{\frac{2}{3}}} \sum_i p_i^{\frac{2}{3}} &= n \leq n \end{aligned}$$

Dual feasibility.

$$\mu_n = \left(\frac{1}{n}\sum_i \left(\frac{1}{2}p_i\right)^{\frac{2}{3}}\right)^{\frac{3}{2}} \geq 0$$

$$\mu_i = 0 \geq 0$$

Complementary slackness.

$$\mu_n\left(n - \sum_i n_i\right) = \mu_n\left(n - \sum_i n\frac{p_i^{\frac{2}{3}}}{\sum_j p_j^{\frac{2}{3}}}\right) = \mu_n(n-n) = 0$$

$$\mu_i n_i = 0$$

Note that the objective is convex; therefore this is the globally optimal solution.

Expanding p_i, we have:

$$n_i = n\frac{\left(\left(\int_{x\in s_i} D(x)\mathrm{d}x\right)\sqrt{\zeta(f_i) + \log\left(\delta_i^{-1}\right)}\right)^{\frac{2}{3}}}{\sum_j\left(\left(\int_{x\in s_j} D(x)\mathrm{d}x\right)\sqrt{\zeta(f_j) + \log\left(\delta_j^{-1}\right)}\right)^{\frac{2}{3}}}$$

□

Note 5.2.1. *Assuming* $\exists Z.\, \forall i.\, \zeta(f_i) \leq Z$*, then with* n_i *as above,* $\lim_{c\to\infty} n_i = \dfrac{\left(\int_{x\in s_i} D(x)\mathrm{d}x\right)^{\frac{2}{3}}}{\sum_j\left(\int_{x\in s_j} D(x)\mathrm{d}x\right)^{\frac{2}{3}}}$.

Note 5.2.2. *For a given* $\delta = 1 - \prod_i(1-\delta_i)$*, all choices of* δ_i *result in most strata being dominated by the* $\log \delta_i^{-1}$ *term in the limit of infinite paths:*

$$\forall k.\, \lim_{c\to\infty}\left[\min_{\{\delta_i | i\in[c]\}} \mathbb{E}\left[\mathbb{1}\left[\log\delta_i^{-1} > k\right]\right] \text{ such that } 0 < \delta_i \leq 1 \text{ and } 1 - \prod_i(1-\delta_i) \geq \delta\right] = 1$$

Proof. The δ_i constraint is equivalent to:

$$\log(1-\delta) \geq \sum_i \log(1-\delta_i)$$

$$
\begin{aligned}
p &::= \texttt{fun (}x,\ldots,x\texttt{) \{}s\texttt{ ; return }x\texttt{\}} \\
s &::= \texttt{skip} \mid s\texttt{ ; }s \mid x\texttt{ = }e \\
&\quad \mid \texttt{if (}e\texttt{ > 0) \{}s\texttt{\} else \{}s\texttt{\}} \\
e &::= x \mid v \mid e + e \mid e * e \mid -e \mid \texttt{sin(}e\texttt{)} \\
&\quad \mid \texttt{exp(}e\texttt{)} \mid \texttt{log\{}v\texttt{\}(}e\texttt{)} \\
x &::= \text{set of variable names} \\
v &::= \text{set of floating-point values}
\end{aligned}
$$

Figure 5.3: Syntax of Turaco.

To minimize $\mathbb{E}\left[\mathbb{1}\left[\log \delta_i^{-1} > k\right]\right]$ subject to $1 - \prod(1 - \delta_i) \geq \delta$, we must set as many δ_i as small as possible without exceeding $\log \delta_i^{-1} > k$. To do this, I set $\delta_i = e^{-k}$ for as many as possible. However, because of the constraint, we can do this for no more than $\frac{\log(1-\delta)}{\log\left(1-e^{-k}\right)}$ strata; the rest must exceed $\log \delta_i^{-1} > k$. Thus, $\forall k.\ \lim_{c\to\infty} \min_{\{\delta_i\}} \mathbb{E}\left[\mathbb{1}\left[\log \delta_i^{-1} > k\right]\right] = 1$. □

5.3 Turaco: Programs as Stratified Functions

In this section I present Turaco, a programming language in which programs denote learnable stratified functions. I provide an analysis for Turaco programs that calculates an upper bound on the complexity of each component of the stratified function that the program denotes.

5.3.1 Syntax and Standard Interpretation

Figure 5.3 presents the syntax of Turaco, a loop-free language similar to IMP (Winskel, 1993). A Turaco program p takes a list of inputs x, executes a top-level statement s, and returns a single variable x. Statements s are skips, sequences, assignments, or if statements. Expressions e are variables x, floating-point values v, binary operations, or unary operations.

Turaco supports analytic functions (e.g., sin, exp), including those which are analytic only on a subset of the reals (e.g., log). I restrict the supported operations to those required to implement the evaluation in Section 5.4.

$$\frac{}{\langle \sigma, v \rangle \Downarrow v} \qquad \frac{}{\langle \sigma, x \rangle \Downarrow \sigma(x)} \qquad \frac{\langle \sigma, e_1 \rangle \Downarrow v_1 \qquad \langle \sigma, e_2 \rangle \Downarrow v_2}{\langle \sigma, e_1 \texttt{+} e_2 \rangle \Downarrow v_1 + v_2} \qquad \frac{\langle \sigma, e_1 \rangle \Downarrow v_1 \qquad \langle \sigma, e_2 \rangle \Downarrow v_2}{\langle \sigma, e_1 \texttt{*} e_2 \rangle \Downarrow v_1 \cdot v_2}$$

$$\frac{\langle \sigma, e \rangle \Downarrow v}{\langle \sigma, -e \rangle \Downarrow -v} \qquad \frac{\langle \sigma, e \rangle \Downarrow v}{\langle \sigma, \texttt{sin(}e\texttt{)} \rangle \Downarrow \sin(v)} \qquad \frac{\langle \sigma, e \rangle \Downarrow v}{\langle \sigma, \texttt{exp(}e\texttt{)} \rangle \Downarrow \exp(v)} \qquad \frac{\langle \sigma, e \rangle \Downarrow v \qquad |b - v| < b}{\langle \sigma, \texttt{log\{}b\texttt{\}(}e\texttt{)} \rangle \Downarrow \log(v)}$$

Figure 5.4: Big-step evaluation relation for expressions in Turaco.

$$\frac{}{\langle \sigma, \texttt{skip} \rangle \Downarrow \sigma} \qquad \frac{\langle \sigma, s_1 \rangle \Downarrow \sigma' \qquad \langle \sigma', s_2 \rangle \Downarrow \sigma''}{\langle \sigma, s_1 \texttt{ ; } s_2 \rangle \Downarrow \sigma''} \qquad \frac{\langle \sigma, e \rangle \Downarrow v}{\langle \sigma, x \texttt{ = } e \rangle \Downarrow \sigma[x \mapsto v]}$$

$$\frac{\langle \sigma, e \rangle \Downarrow v \qquad v > 0 \qquad \langle \sigma, s_1 \rangle \Downarrow \sigma_l}{\langle \sigma, \texttt{if (}e \texttt{ > 0) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \rangle \Downarrow \sigma_l} \qquad \frac{\langle \sigma, e \rangle \Downarrow v \qquad v \leq 0 \qquad \langle \sigma, s_2 \rangle \Downarrow \sigma_r}{\langle \sigma, \texttt{if (}e \texttt{ > 0) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \rangle \Downarrow \sigma_r}$$

Figure 5.5: Big-step evaluation relation for statements.

$$\frac{\langle \sigma, s \rangle \Downarrow \sigma'}{\langle \sigma, \texttt{fun (}x_0, x_1 \ldots, x_n\texttt{) \{}s \texttt{ ; return } x\texttt{\}} \rangle \Downarrow \sigma'(x)}$$

Figure 5.6: Big-step evaluation relation for Turaco.

Standard execution semantics. Figure 5.4 presents the big-step evaluation relation for expressions in Turaco. The expression relation $\langle \sigma, e \rangle \Downarrow v$ says that under variable store σ (assigning values to all variables in e), the expression e evaluates to value v. These semantics are standard to IMP-like languages with the exception of that for `log{`b`}(`e`)`: note that the expression `log{`b`}(`e`)` takes an additional parameter b and requires $|b - v| < b$. I discuss this requirement in Section 5.3.6.

Figure 5.5 presents the big-step evaluation relation for statements in Turaco. The statement relation $\langle \sigma, s \rangle \Downarrow \sigma'$ says that under variable store σ, the statement s evaluates to a new variable store σ'.

Figure 5.6 presents the big-step evaluation relation for Turaco programs. The program relation $\langle \sigma, \texttt{fun (}x_0, x_1 \ldots, x_n\texttt{) \{}s\texttt{ ; return }x\texttt{\}} \rangle \Downarrow v$ says that under variable store σ (representing the inputs to the program), the program evaluates to value v.

5.3.2 Complexity Analysis

I now present a program analysis that gives an upper bound on the complexity of *traces* of Turaco programs, sequences of statements without if statements. The analysis uses two core concepts: a *complexity interpretation* of expressions to calculate an upper bound on the tilde of expressions (Section 5.2.2), and a standard *dual-number execution* (Wengert, 1964; Griewank and Walther, 2008) of the complexity interpretation to calculate the derivative of the upper bound on the tilde, which as I show below is also an upper bound on the derivative of the tilde. The result of the dual-number execution allows us to upper bound the complexity of a trace of a Turaco program.

Program Analysis

First I walk through the rules of the analysis, presented as a big-step evaluation relation.

$$\frac{}{\langle \tilde{\sigma}, v\rangle \;\tilde{\Downarrow}\; (|v|, 0)} \qquad \frac{}{\langle \tilde{\sigma}, x\rangle \;\tilde{\Downarrow}\; \tilde{\sigma}(x)}$$

$$\frac{\langle \tilde{\sigma}, e_1\rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}') \qquad \langle \tilde{\sigma}, e_2\rangle \;\tilde{\Downarrow}\; (\tilde{v}_2, \tilde{v}_2')}{\langle \tilde{\sigma}, e_1\texttt{+}e_2\rangle \;\tilde{\Downarrow}\; (\tilde{v} + \tilde{v}_2, \tilde{v}' + \tilde{v}_2')} \qquad \frac{\langle \tilde{\sigma}, e_1\rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}') \qquad \langle \tilde{\sigma}, e_2\rangle \;\tilde{\Downarrow}\; (\tilde{v}_2, \tilde{v}_2')}{\langle \tilde{\sigma}, e_1\texttt{*}e_2\rangle \;\tilde{\Downarrow}\; (\tilde{v} \cdot \tilde{v}_2, \tilde{v} \cdot \tilde{v}_2' + \tilde{v}_2 \cdot \tilde{v}')}$$

$$\frac{\langle \tilde{\sigma}, e\rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}')}{\langle \tilde{\sigma}, \texttt{-}e\rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}')} \qquad \frac{\langle \tilde{\sigma}, e\rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}')}{\langle \tilde{\sigma}, \texttt{sin(}e\texttt{)}\rangle \;\tilde{\Downarrow}\; (\sinh(\tilde{v}), \tilde{v}' \cosh(\tilde{v}))} \qquad \frac{\langle \tilde{\sigma}, e\rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}')}{\langle \tilde{\sigma}, \texttt{exp(}e\texttt{)}\rangle \;\tilde{\Downarrow}\; (\exp(\tilde{v}), \tilde{v}' \exp(\tilde{v}'))}$$

$$\frac{\langle \tilde{\sigma}, e\rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}') \qquad b > \tilde{v}\sqrt{b^2+1}}{\langle \tilde{\sigma}, \texttt{log\{}b\texttt{\}(}e\texttt{)}\rangle \;\tilde{\Downarrow}\; \left(|\log(b)| + \log(b) - \log\left(b - \tilde{v}\sqrt{b^2+1}\right), \frac{\tilde{v}'}{b - \tilde{v}\sqrt{b^2+1}}\right)}$$

Figure 5.7: Tilde relation for expressions in TURACO.

Figure 5.7 presents the relation used to calculate the tilde for expressions in TURACO. The relation $\langle \tilde{\sigma}, e\rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}')$ says that under the variable complexity mapping $\tilde{\sigma}$ (mapping variables to tuples with their respective tilde and tilde derivative), the expression e has $\tilde{e} \leq \tilde{v}$ and $\tilde{e}' \leq \tilde{v}'$.

Broadly, I define the rules in Figure 5.7 using the definition of the tilde, the fact that the tilde is compositional (as I prove in Section 5.3.2) and the definition of a dual-number execution. For instance, the tilde of a constant v is the absolute value $|v|$ of that constant with a derivative of 0, and the tilde of e_1`+`e_2 is the sum of the tilde of each expression with a derivative of the sum of their derivatives.

A slightly more complex rule is that of `sin(`e`)`, which computes $\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$. Thus, $\widetilde{\sin}(x) = \sum_{n=0}^{\infty} \left|\frac{(-1)^n}{(2n+1)!}\right| x^{2n+1} = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!} = \sinh(x)$. The derivative is then $\sinh'(x) = x' \cosh(x)$. Because $\langle \tilde{\sigma}, e\rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}')$ and because the tilde is compositional (Lemma 5.3.1), we can plug in $\tilde{v}$ and $\tilde{v}'$ to get $(\sinh(\tilde{v}), \tilde{v}' \cosh(\tilde{v}))$.

The most complex rule is the rule for `log{`b`}(`e`)`. To handle that $\log(x)$ is not analytic around 0, `log{`b`}(`e`)` expands $\log(x)$ around $x = b$. The value of b is a nuisance parameter that must be set to allow the expansion around $x = b$ to converge for all inputs (inducing the $|b - v| < b$ requirement in Figure 5.4) while minimizing the overall program complex-

$$\frac{}{\langle \tilde{\sigma}, \texttt{skip} \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}} \qquad \frac{\langle \tilde{\sigma}, t_1 \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}' \qquad \langle \tilde{\sigma}', t_2 \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}''}{\langle \tilde{\sigma}, t_1 \texttt{ ; } t_2 \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}''} \qquad \frac{\langle \tilde{\sigma}, e \rangle \,\tilde{\Downarrow}\, (\tilde{v}, \tilde{v}')}{\langle \tilde{\sigma}, x \texttt{ = } e \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}[x \mapsto (\tilde{v}, \tilde{v}')]}$$

Figure 5.8: Tilde relation for traces in TURACO.

$$\frac{\langle \{x_i \mapsto (1,1)\}, t \rangle \,\tilde{\Downarrow}\, \tilde{\sigma} \qquad \tilde{\sigma}(x) = (\tilde{v}, \tilde{v}')}{\zeta_{\,\tilde{\Downarrow}}\,(t, x) \leq \tilde{v}'^2}$$

Figure 5.9: Complexity relation for traces in TURACO.

ity. Note that the condition $b > \tilde{v}\sqrt{b^2+1}$ can always be satisfied by applying the identity $\log(x) = \log\left(\frac{x}{c}\right) + \log(c)$.

Figure 5.8 presents the relation for calculating the tilde of all variables computed by traces (branch-free statements) in TURACO. The relation $\langle \tilde{\sigma}, t \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}'$ says that under the variable complexity mapping $\tilde{\sigma}$, executing the trace t computes variables with tildes and tilde derivatives upper-bounded by those of $\tilde{\sigma}'$.

Figure 5.9 presents the complexity relation for traces in TURACO. The relation $\zeta_{\,\tilde{\Downarrow}}\,(t, x) \leq z$ says that the trace t has complexity upper bounded by z for computing variable x (under the assumptions in Agarwala et al. (2021)).

Figure 5.10 presents the trace collection relation for TURACO statements. The relation $\langle \tau, s \rangle \rightsquigarrow \tau'$ says that under the trace mapping τ (mapping paths that reach this statement to the trace of statements executed thus far), executing the statement s can result in possible paths and corresponding traces τ'.

$$\frac{}{\langle \tau, \texttt{skip} \rangle \rightsquigarrow \tau} \qquad \frac{}{\langle \tau, x \texttt{ = } e \rangle \rightsquigarrow \{p \mapsto \tau(p); x = e \,|\, p \in \tau\}} \qquad \frac{\langle \tau, s_1 \rangle \rightsquigarrow \tau' \qquad \langle \tau', s_2 \rangle \rightsquigarrow \tau''}{\langle \tau, s_1 \; s_2 \rangle \rightsquigarrow \tau''}$$

$$\frac{\langle \tau, s_1 \rangle \rightsquigarrow \tau_1 \qquad \langle \tau, s_2 \rangle \rightsquigarrow \tau_2}{\langle \tau, \texttt{if (}e \texttt{ > 0) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \rangle \rightsquigarrow \{\texttt{l}.p \mapsto \tau_1(p) \,|\, p \in \tau_1\} \cup \{\texttt{r}.p \mapsto \tau_2(p) \,|\, p \in \tau_2\}}$$

Figure 5.10: Trace collection relation for statements, where . denotes string concatenation.

$$\frac{\langle \{\cdot \mapsto \texttt{skip}\}, s\rangle \rightsquigarrow \tau}{\langle \texttt{fun } (x_0, x_1 \ldots, x_n) \ \{s \ ; \texttt{return } x\}\rangle \rightsquigarrow \tau}$$

Figure 5.11: Trace collection relation for TURACO programs, using $\cdot$ to mean the empty string.

Figure 5.11 presents the trace collection relation for TURACO programs. The trace collection relation $\langle \texttt{fun } (x_0, x_1 \ldots, x_n) \ \{s \ ; \texttt{return } x\}\rangle \rightsquigarrow \tau$ says that executing the program can result in possible paths and corresponding traces τ.

Tilde Calculus

This section presents the core lemma stating that the upper bound on the tilde is compositional, and that the derivative of the upper bound is an upper bound on the derivative. The bounds on the tilde are from Agarwala et al. (2021); I extend these bounds to also bound the derivative of the tilde.

Lemma 5.3.1. *The tilde and its derivative both have upper bounds that are compositional with respect to the function f:*

$$f(\vec{x}) = g(\vec{x}) + h(\vec{x}) \Rightarrow \forall x \geq 0.\ \tilde{f}(x) \leq \tilde{g}(x) + \tilde{h}(x) \wedge \tilde{f}'(x) \leq \tilde{g}'(x) + \tilde{h}'(x)$$

$$f(\vec{x}) = g(\vec{x}) \cdot h(\vec{x}) \Rightarrow \forall x \geq 0.\ \tilde{f}(x) \leq \tilde{g}(x) \cdot \tilde{h}(x) \wedge \tilde{f}'(x) \leq \tilde{g}'(x)\tilde{h}(x) + \tilde{g}(x)\tilde{h}'(x)$$

$$f(\vec{x}) = g(h(\vec{x})) \Rightarrow \forall x \geq 0.\ \tilde{f}(x) \leq \tilde{g}\Big(\tilde{h}(x)\Big) \wedge \tilde{f}'(x) \leq \tilde{g}'\Big(\tilde{h}(x)\Big) \cdot \tilde{h}'(x)$$

(when $\tilde{h}(x)$ is in the radius of convergence of g)

The proof of this lemma is presented in Section 5.3.7.

5.3.3 Soundness

This section proves that the TURACO complexity analysis is sound: that it computes an upper bound on the true complexity of learning a trace. I prove this by induction on ex-

pressions and traces. The approach is based on the observation that at a given program point the value of each variable was computed by some function f_x applied to the input. I use the notation $\{f_x\}$ as shorthand for $\{f_x \,|\, x \in \sigma\}$, a set of functions indexed by $x \in \sigma$. The inductive hypothesis requires that $\tilde{\sigma}$ contain the tilde and tilde derivative of each of these functions (evaluated at 1, as in Theorem 1). I use the notation $\tilde{\sigma} \vdash \{f_x\}$ to denote the predicate that each f_x have tilde and tilde derivative bounded by $\tilde{\sigma}$:

$$\tilde{\sigma} \vdash \{f_x\} \Leftrightarrow \forall x \in \tilde{\sigma}. \left(\tilde{\sigma}(x) = (\tilde{v}, \tilde{v}') \Rightarrow \left(0 \leq \tilde{f}_x(1) \leq \tilde{v} \wedge 0 \leq \tilde{f}_x{}'(1) \leq \tilde{v}'\right)\right)$$

I also note that the standard execution semantics big-step relation $\Downarrow$ both for expressions and for traces is a function. I use $[\![\cdot]\!]$ as notation to refer to that function for expressions and $[\![\cdot]\!]_x$ to refer to that function for traces followed by taking the value of the variable x:

$$[\![e]\!](\sigma) = v \Leftrightarrow \langle \sigma, e \rangle \Downarrow v$$
$$[\![t]\!]_x(\sigma) = v \Leftrightarrow \langle \sigma, t \rangle \Downarrow \sigma' \wedge \sigma'(x) = v$$

I use the notation $\circ$ to denote function composition. The functions denoted by expressions and traces have multiple inputs; in this context, composition with a set of functions $\{f_x\}$ is defined as follows:

$$([\![\cdot]\!] \circ \{f_x\})(\sigma) \triangleq [\![\cdot]\!](\{x \mapsto f_x(\sigma)\})$$

Now I state the core theorems of correctness for the TURACO analysis:

Lemma 5.3.2. *The tilde big-step expression relation upper bounds the tilde and its derivative:*

$$\left(\langle \tilde{\sigma}, e \rangle \;\tilde{\Downarrow}\; (\tilde{v}, \tilde{v}') \wedge \tilde{\sigma} \vdash \{f_x\}\right) \Rightarrow \left(\widetilde{[\![e]\!] \circ \{f_x\}}(1) \leq \tilde{v} \wedge \widetilde{[\![e]\!] \circ \{f_x\}}'(1) \leq \tilde{v}'\right)$$

Lemma 5.3.3. *The tilde big-step trace relation upper bounds on the tilde and tilde derivative:*

$$\left(\langle\tilde{\sigma}, t\rangle \;\tilde{\Downarrow}\; \tilde{\sigma}' \wedge \tilde{\sigma} \vdash \{f_x\}\right) \Rightarrow \tilde{\sigma}' \vdash \left\{[\![t]\!]_y \circ \{f_x\}\right\}$$

Theorem 3. *The complexity relation computes an upper bound on the true complexity:*

$$\zeta_{\;\tilde{\Downarrow}}\,(t, x) \leq z \Rightarrow \zeta([\![t]\!]_x) \leq z$$

The proofs of Lemmas 5.3.2 and 5.3.3 and Theorem 3 are presented in Section 5.3.7.

5.3.4 Precision

Note that the analysis is a sound but imprecise approximation of complexity, in that the upper bound it computes is not tight. For example, consider the expression x+(-x): under the TURACO analysis, $\langle\{x \mapsto (1,1)\}, x\texttt{+(-}x\texttt{)}\rangle \;\tilde{\Downarrow}\; (2,2)$ even though $[\![x\texttt{+(-}x\texttt{)}]\!](\sigma) = 0$.

5.3.5 Extensions

My implementation extends TURACO to support vector-valued variables, applying all operations elementwise. Following Agarwala et al. (2021), I define the complexity of learning a vector-valued function to be the sum of the complexity of learning each output component. My implementation also supports other syntactic sugar including a minus operation and division by constants.

Loops. My implementation of TURACO also supports fixed- and bounded-length loops, though they are not required for any case study in the chapter (thus I do not present them in this chapter). However, unbounded loops pose a challenge because the approach trains a distinct surrogate per path, which is not possible with unbounded loops. This restriction to statically bounded length loops is a common feature of analyses that reason about numerical

approximation, including reliability analyses (Carbin et al., 2013; Misailovic et al., 2014; Boston et al., 2015) and floating-point error analyses (Darulova and Kuncak, 2014; Magron et al., 2017; Solovyev et al., 2018). Reasoning about loops with dynamic, input-dependent bounds requires separate techniques (e.g., Boston et al. (2015)).

5.3.6 Log Rule

The `log{`b`}(`e`)` rule requires the parameter b to expand around since log is not analytic around 0. The value set for this parameter must satisfy the $|b - v| < b$ condition for all inputs in the standard interpretation (to ensure that all values are in the radius of convergence) and the $b > \tilde{v}\sqrt{b^2 + 1}$ condition in the tilde interpretation, but is otherwise free to be set to a value that minimizes the upper bound on the complexity.

As an example of a program that uses value of b value other than 1, consider the following:

```
fun(x) {
    x = log{3.88}(0.75 * x);
    x = x * x;
    return x;
}
```

This program cannot use $b = 1$ (since it fails the condition in the tilde interpretation: $\tilde{v} = 0.75$ so $\tilde{v}\sqrt{b^2 + 1} > b$). Instead, this program requires $b > \frac{3}{\sqrt{7}}$, and is minimized around $b = 3.88$. Automatically inferring a value for this parameter that satisfies the constraints and optimizes the complexity bound could alleviate this burden, but this would require as-of-yet undeveloped techniques.

5.3.7 Proofs

This section presents the proofs of Lemmas 5.3.1 to 5.3.3 and Theorem 3.

Complexity Algebra

I first prove each component of Lemma 5.3.1:

Lemma 4.4. *The tilde and its derivative both have upper bounds that are compositional with respect to the function f:*

$$f(\vec{x}) = g(\vec{x}) + h(\vec{x}) \Rightarrow \forall x \geq 0.\, \tilde{f}(x) \leq \tilde{g}(x) + \tilde{h}(x) \wedge \tilde{f}'(x) \leq \tilde{g}'(x) + \tilde{h}'(x)$$

$$f(\vec{x}) = g(\vec{x}) \cdot h(\vec{x}) \Rightarrow \forall x \geq 0.\, \tilde{f}(x) \leq \tilde{g}(x) \cdot \tilde{h}(x) \wedge \tilde{f}'(x) \leq \tilde{g}'(x)\tilde{h}(x) + \tilde{g}(x)\tilde{h}'(x)$$

$$f(\vec{x}) = g(h(\vec{x})) \Rightarrow \forall x \geq 0.\, \tilde{f}(x) \leq \tilde{g}\Big(\tilde{h}(x)\Big) \wedge \tilde{f}'(x) \leq \tilde{g}'\Big(\tilde{h}(x)\Big) \cdot \tilde{h}'(x)$$

(when $\tilde{h}(x)$ is in the radius of convergence of g)

Aspects of these are shown without proof in Agarwala et al. (2021). Here I flesh out the proof and further prove bounds on the tilde derivatives.

Lemma 5.3.1. *If $f(\vec{x}) = g(\vec{x}) + h(\vec{x})$, then for $x \geq 0$, $\tilde{f}(x) \leq \tilde{g}(x) + \tilde{h}(x)$ and $\tilde{f}'(x) \leq \tilde{g}'(x) + \tilde{h}'(x)$.*

Proof. Given:

$$g(\vec{x}) = \sum_k \sum_{v \in V_{g,k}} a_{g,v} \prod_{i=1}^{k} \beta_{g,v,i} \cdot \vec{x}$$

$$h(\vec{x}) = \sum_k \sum_{v \in V_{h,k}} a_{h,v} \prod_{i=1}^{k} \beta_{h,v,i} \cdot \vec{x}$$

Define:

$$V_{f,k} = V_{g,k} \sqcup V_{h,k} \qquad \text{(where } \sqcup \text{ is the disjoint union)}$$

$$a_{k,f,v} = \begin{cases} a_{g,v} & v \in V_{g,k} \\ a_{h,v} & \text{otherwise} \end{cases}$$

$$\beta_{f,v,i} = \begin{cases} \beta_{g,v,i} & v \in V_{g,k} \\ \beta_{h,v,i} & \text{otherwise} \end{cases}$$

Then:

$$\begin{aligned}
f(\vec{x}) &= g(\vec{x}) + h(\vec{x}) \\
&= \sum_k \sum_{v \in V_{g,k}} a_{g,v} \prod_{i=1}^{k} \beta_{g,v,i} \cdot \vec{x} + \sum_k \sum_{v \in V_{h,k}} a_{h,v} \prod_{i=1}^{k} \beta_{h,v,i} \cdot \vec{x} \\
&= \sum_k \sum_{v \in V_{f,k}} a_{f,v} \prod_{i=1}^{k} \beta_{f,v,i} \cdot \vec{x} \\
\tilde{f}(x) &= \sum_k \left(\sum_{v \in V_{f,k}} |a_{f,v}| \prod_{i=1}^{k} \|\beta_{f,v,i}\|_2 \right) x^k \\
&= \sum_k \left(\sum_{v \in V_{g,k}} |a_{g,v}| \prod_{i=1}^{k} \|\beta_{g,v,i}\|_2 + \sum_{v \in V_{h,k}} |a_{h,v}| \prod_{i=1}^{k} \|\beta_{h,v,i}\|_2 \right) x^k \\
&= \sum_k \left(\sum_{v \in V_{g,k}} |a_{g,v}| \prod_{i=1}^{k} \|\beta_{g,v,i}\|_2 \right) x^k + \sum_k \left(\sum_{v \in V_{h,k}} |a_{h,v}| \prod_{i=1}^{k} \|\beta_{h,v,i}\|_2 \right) x^k \\
&= \tilde{g}(x) + \tilde{h}(x) \\
\tilde{f}'(x) &= \sum_k \left(\sum_{v \in V_{f,k}} |a_{f,v}| \prod_{i=1}^{k} \|\beta_{f,v,i}\|_2 \right) k x^{k-1} \\
&= \sum_k \left(\sum_{v \in V_{g,k}} |a_{g,v}| \prod_{i=1}^{k} \|\beta_{g,v,i}\|_2 + \sum_{v \in V_{h,k}} |a_{h,v}| \prod_{i=1}^{k} \|\beta_{h,v,i}\|_2 \right) k x^{k-1}
\end{aligned}$$

$$= \sum_k \left(\sum_{v \in V_{g,k}} |a_{g,v}| \prod_{i=1}^{k} \|\beta_{g,v,i}\|_2 \right) kx^{k-1} + \sum_k \left(\sum_{v \in V_{h,k}} |a_{h,v}| \prod_{i=1}^{k} \|\beta_{h,v,i}\|_2 \right) kx^{k-1}$$
$$= \tilde{g}'(x) + \tilde{h}'(x)$$

Note that other choices of β_f are possible and may result in a smaller $\tilde{g}$ (hence the imprecision noted in Section 5.3.4). □

Lemma 5.3.2. *If $f(\vec{x}) = g(\vec{x}) \cdot h(\vec{x})$, then for $x \geq 0$, $\tilde{f}(x) \leq \tilde{g}(x) \cdot \tilde{h}(x)$ and $\tilde{f}'(x) \leq \tilde{g}'(x)\tilde{h}(x) + \tilde{g}(x)\tilde{h}'(x)$.*

Proof. Given:

$$g(\vec{x}) = \sum_k \sum_{v \in V_{g,k}} a_{g,v} \prod_{i=1}^{k} \beta_{g,v,i} \cdot \vec{x}$$
$$h(\vec{x}) = \sum_k \sum_{v \in V_{h,k}} a_{h,v} \prod_{i=1}^{k} \beta_{h,v,i} \cdot \vec{x}$$

Define:

$$V_{f,l} = \{(v_g, v_h) \mid j + k = l \wedge v_g \in V_{g,j} \wedge v_h \in V_{h,k}\}$$
$$a_{f,(v_g,v_h)} = a_{g,v_g} \cdot a_{h,v_h}$$
$$\beta_{f,(v_g,v_h),i} = \begin{cases} \beta_{g,v_g,i} & i \leq j \\ \beta_{h,v_h,i-j} & i > j \end{cases}$$

Then:

$$f(\vec{x}) = g(\vec{x}) \cdot h(\vec{x})$$
$$= \left(\sum_k \sum_{v \in V_{g,k}} a_{g,v} \prod_{i=1}^{k} \beta_{g,v,i} \cdot \vec{x} \right) \cdot \left(\sum_k \sum_{v \in V_{h,k}} a_{h,v} \prod_{i=1}^{k} \beta_{h,v,i} \cdot \vec{x} \right)$$

$$= \sum_j \sum_k \sum_{v_g \in V_{g,j}} \sum_{v_h \in V_{h,k}} a_{g,v_g} a_{h,v_h} \prod_{i=1}^{j} (\beta_{g,v_g,i} \cdot \vec{x}) \prod_{i=1}^{k} (\beta_{h,v_h,i} \cdot \vec{x})$$

$$= \sum_l \sum_{j+k=l} \sum_{v_g \in V_{g,j}} \sum_{v_h \in V_{h,k}} a_{g,v_g} a_{h,v_h} \prod_{i=1}^{j} (\beta_{g,v_g,i} \cdot \vec{x}) \prod_{i=1}^{k} (\beta_{h,v_h,i} \cdot \vec{x})$$

$$= \sum_l \sum_{v \in V_{f,l}} a_{f,v} \prod_{i=1}^{l} \beta_{f,v,i} \cdot \vec{x}$$

$$\tilde{f}(x) = \sum_l x^l \sum_{v \in V_{f,l}} |a_{f,v}| \prod_{i=1}^{l} \|\beta_{f,v,i}\|_2$$

$$= \sum_l \sum_{j+k=l} x^{j+k} \sum_{v_g \in V_{g,j}} \sum_{v_h \in V_{h,k}} |a_{g,v_g} a_{h,v_h}| \prod_{i=1}^{j} \|\beta_{g,v_g,i}\|_2 \prod_{i=1}^{k} \|\beta_{h,v_h,i}\|_2$$

$$= \sum_j \sum_k x^{j+k} \sum_{v_g \in V_{g,j}} \sum_{v_h \in V_{h,k}} |a_{g,v_g}||a_{h,v_h}| \prod_{i=1}^{j} \|\beta_{g,v_g,i}\|_2 \prod_{i=1}^{k} \|\beta_{h,v_h,i}\|_2$$

$$= \left(\sum_j x^j \sum_{v_g \in V_{g,j}} |a_{g,v_g}| \prod_{i=1}^{j} \|\beta_{g,v_g,i}\|_2 \right) \cdot \left(\sum_k x^k \sum_{v_h \in V_{h,k}} |a_{h,v_h}| \prod_{i=1}^{k} \|\beta_{h,v_h,i}\|_2 \right)$$

$$= \tilde{g}(x) \cdot \tilde{h}(x)$$

$$\tilde{f}'(x) = \sum_l l x^{l-1} \sum_{v \in V_{f,l}} |a_{f,v}| \prod_{i=1}^{l} \|\beta_{f,v,i}\|_2$$

$$= \sum_l \sum_{j+k=l} (j+k) x^{j+k-1} \sum_{v_g \in V_{g,j}} \sum_{v_h \in V_{h,k}} |a_{g,v_g} a_{h,v_h}| \prod_{i=1}^{j} \|\beta_{g,v_g,i}\|_2 \prod_{i=1}^{k} \|\beta_{h,v_h,i}\|_2$$

$$= \left(\sum_j j x^{j-1} x^k \sum_{v_g \in V_{g,j}} \sum_{v_h \in V_{h,k}} |a_{g,v_g} a_{h,v_h}| \prod_{i=1}^{j} \|\beta_{g,v_g,i}\|_2 \prod_{i=1}^{k} \|\beta_{h,v_h,i}\|_2 \right)$$
$$+ \left(\sum_k k x^{k-1} x^j \sum_{v_g \in V_{g,j}} \sum_{v_h \in V_{h,k}} |a_{g,v_g} a_{h,v_h}| \prod_{i=1}^{j} \|\beta_{g,v_g,i}\|_2 \prod_{i=1}^{k} \|\beta_{h,v_h,i}\|_2 \right)$$

$$= \left(\left(\sum_j j x^{j-1} \sum_{v_g \in V_{g,j}} |a_{g,v_g}| \prod_{i=1}^{j} \|\beta_{g,v_g,i}\|_2 \right) \cdot \left(\sum_k x^k \sum_{v_h \in V_{h,k}} |a_{h,v_h}| \prod_{i=1}^{k} \|\beta_{h,v_h,i}\|_2 \right) \right)$$
$$+ \left(\left(\sum_k k x^{k-1} \sum_{v_h \in V_{h,k}} |a_{h,v_h}| \prod_{i=1}^{k} \|\beta_{h,v_h,i}\|_2 \right) \cdot \left(\sum_j x^j \sum_{v_g \in V_{g,j}} |a_{g,v_g}| \prod_{i=1}^{j} \|\beta_{g,v_g,i}\|_2 \right) \right)$$

$$= \tilde{g}'(x) \cdot \tilde{h}(x) + \tilde{g}(x) \cdot \tilde{h}'(x)$$

□

Lemma 5.3.3. *If $f(\vec{x}) = g(h(\vec{x}))$, then $\tilde{h}(x) \leq \tilde{g}(\tilde{f}(x))$ and $\tilde{h}'(x) \leq \tilde{g}'(\tilde{f}(x)) \cdot \tilde{f}'(x)$.*

Proof. The proof is similar to that of Corollary 2 in Agarwala et al. (2021), following fairly directly from Fact 1 in Agarwala et al. (2021):

$$
\begin{aligned}
f(x) &= \sum_k a_{g,k} h(\vec{x})^k \\
\tilde{f}(x) &\leq \sum_k \widetilde{a_{g,k} h(\vec{x})^k} && (f = g + h \Rightarrow \tilde{f} \leq \tilde{g} + \tilde{h}) \\
&\leq \sum_k |a_{g,k}| \widetilde{h(\vec{x})^k} && (f(x) = g(cx) \Rightarrow \tilde{f}(x) = |c| \tilde{g}(x) \\
&\leq \sum_k |a_{g,k}| \Big(\tilde{h}(x)\Big)^k && (f = g \cdot h \Rightarrow \tilde{f} \leq \tilde{g} \cdot \tilde{h}) \\
&\leq \tilde{g}\Big(\tilde{h}(x)\Big) \\
\tilde{f}'(x) &= \widetilde{\frac{d}{dx} \sum_k a_{g,k} h(\vec{x})^k} \\
&\leq \sum_k \frac{d}{dx} \widetilde{a_{g,k} h(\vec{x})^k} && (f = g + h \Rightarrow \tilde{f}' \leq \tilde{g}' + \tilde{h}') \\
&= \sum_k |a_{g,k}| \frac{d}{dx} \widetilde{h(\vec{x})^k} \\
&\leq \sum_k |a_{g,k}| k \tilde{h}(x)^{k-1} \tilde{h}'(x) && (f = g \cdot h \Rightarrow \tilde{f}' \leq \tilde{g}' \cdot \tilde{h} + \tilde{g} \cdot \tilde{h}') \\
&= \tilde{g}'\Big(\tilde{h}(x)\Big) \cdot \tilde{h}'(x)
\end{aligned}
$$

□

Lemma 5.3.4. *The tilde and its derivative are monotonic function for $x \geq 0$:*

$$\forall x \geq x' \geq 0. \tilde{f}(x) \geq \tilde{f}(x) \wedge \tilde{f}'(x) \geq \tilde{f}'(x)$$

Proof.

$$\tilde{f}(x) = \sum_{k=0}^{\infty} |a_k| x^k$$

The derivative of a power series with all nonnegative coefficients is also a power series with all nonnegative coefficients:

$$\tilde{f}'(x) = \sum_{k=0}^{\infty} k|a_k|x^{k-1}$$

For $x > 0$, and any power series with all nonnegative coefficients g, $g(x) \geq 0$. Thus, the derivative of $\tilde{f}$ and $\tilde{f}'$ are both nonnegative everywhere, thus they are both monotonic. $\square$

TURACO Analysis Proof of Soundness

This section proves that the TURACO complexity analysis is sound: that it computes an upper bound on the true complexity of learning the program.

I first note that the standard execution semantics big-step relation $\Downarrow$ both for expressions and for traces is a function. I use $[\![\cdot]\!]$ as notation to refer to that function for expressions and $[\![\cdot]\!]_x$ to refer to that function for traces followed by taking the value of the variable x:

$$[\![e]\!](\sigma) = v \Leftrightarrow \langle \sigma, e \rangle \Downarrow v$$
$$[\![t]\!]_x(\sigma) = v \Leftrightarrow \langle \sigma, t \rangle \Downarrow \sigma' \wedge \sigma'(x) = v$$

I use the notation $\{f_x\}$ as shorthand for $\{f_x \,|\, x \in \sigma\}$, a set of functions indexed by $x \in \sigma$. I use the notation $\tilde{\sigma} \vdash \{f_x\}$ to denote the predicate that f_x have tildes and tilde derivatives that are bounded by $\tilde{\sigma}$:

$$\tilde{\sigma} \vdash \{f_x\} \Leftrightarrow \forall x \in \tilde{\sigma}. \left(\tilde{\sigma}(x) = (\tilde{v}, \tilde{v}') \Rightarrow \left(0 \leq \tilde{f}_x(1) \leq \tilde{v} \wedge 0 \leq \tilde{f}_x{}'(1) \leq \tilde{v}'\right)\right)$$

I use the notation $\circ$ to denote function composition:

$$([\![\cdot]\!] \circ \{f_x\})(\sigma) \triangleq [\![\cdot]\!](\{x \mapsto f_x(\sigma)\})$$

Lemma 4.2. *The tilde big-step expression relation upper bounds the tilde and its derivative:*

$$\Big(\langle\tilde{\sigma}, e\rangle \tilde{\Downarrow} (\tilde{v}, \tilde{v}') \wedge \tilde{\sigma} \vdash \{f_x\}\Big) \Rightarrow \Big(\widetilde{[\![e]\!] \circ \{f_x\}}(1) \leq \tilde{v} \wedge \widetilde{[\![e]\!] \circ \{f_x\}}'(1) \leq \tilde{v}'\Big)$$

Proof. I prove this by induction.

Case: $\dfrac{}{\langle\tilde{\sigma}, v\rangle \tilde{\Downarrow} (|v|, 0)}$

$[\![v]\!](\sigma) = v$, so $([\![v]\!] \circ \{f_x\})(\sigma) = v$. Thus, $\widetilde{[\![v]\!] \circ \{f_x\}}(1) = |v|$ and $\widetilde{[\![v]\!] \circ \{f_x\}}'(1) = 0$.

Case: $\dfrac{}{\langle\tilde{\sigma}, x\rangle \tilde{\Downarrow} \tilde{\sigma}(x)}$

$[\![x]\!](\sigma) = \sigma(x)$, so $([\![x]\!] \circ \{f_x\})(\sigma) = f_x(\sigma)$. For $\tilde{\sigma}(x) = (\tilde{v}, \tilde{v}')$, $\tilde{f}_x(1) \leq \tilde{v}$ and $\tilde{f}_x{}'(1) \leq \tilde{v}'$. Thus, $\widetilde{[\![x]\!] \circ \{f_x\}}(1) = \tilde{f}_x(1) \leq \tilde{v}$ and $\widetilde{[\![x]\!] \circ \{f_x\}}'(1) = \tilde{f}_x{}'(1) \leq \tilde{v}'$.

Case: $\dfrac{\langle\tilde{\sigma}, e\rangle \tilde{\Downarrow} (\tilde{v}, \tilde{v}')}{\langle\tilde{\sigma}, \texttt{-}e\rangle \tilde{\Downarrow} (\tilde{v}, \tilde{v}')}$

$[\![-e]\!](\sigma) = -[\![e]\!](\sigma)$, so $([\![-e]\!] \circ \{f_x\})(\sigma) = -([\![e]\!] \circ \{f_x\})(\sigma)$. By the inductive hypothesis, $\widetilde{[\![e]\!] \circ \{f_x\}}(1) \leq \tilde{v}$ and $\widetilde{[\![e]\!] \circ \{f_x\}}'(1) \leq \tilde{v}'$. Thus, $\widetilde{[\![-e]\!] \circ \{f_x\}}(1) = \widetilde{[\![e]\!] \circ \{f_x\}}(1) \leq \tilde{v}$ and $\widetilde{[\![-e]\!] \circ \{f_x\}}'(1) = \widetilde{[\![e]\!] \circ \{f_x\}}'(1) \leq \tilde{v}'$, proving this case.

Case: $\dfrac{\langle\tilde{\sigma}, e\rangle \tilde{\Downarrow} (\tilde{v}, \tilde{v}')}{\langle\tilde{\sigma}, \texttt{sin(}e\texttt{)}\rangle \tilde{\Downarrow} (\sinh(\tilde{v}), \tilde{v}' \cosh(\tilde{v}))}$

$[\![\texttt{sin(}e\texttt{)}]\!](\sigma) = \sin([\![e]\!](\sigma))$, so $([\![\texttt{sin(}e\texttt{)}]\!] \circ \{f_x\})(\sigma) = \sin([\![e]\!] \circ \{f_x\}(\sigma))$. By the inductive hypothesis, $\widetilde{[\![e]\!] \circ \{f_x\}}(1) \leq \tilde{v}$ and $\widetilde{[\![e]\!] \circ \{f_x\}}'(1) \leq \tilde{v}'$. Note that $\widetilde{\sin}(x) = \sinh(x)$. By Lemma 5.3.3, $\widetilde{[\![\texttt{sin(}e\texttt{)}]\!] \circ \{f_x\}}(1) \leq \sinh\Big(\widetilde{[\![e]\!] \circ \{f_x\}}(1)\Big) \leq \sinh(\tilde{v})$ (Lemma 5.3.4) and $\widetilde{[\![\texttt{sin(}e\texttt{)}]\!] \circ \{f_x\}}'(1) \leq \widetilde{[\![e]\!] \circ \{f_x\}}'(1) \cosh\Big(\widetilde{[\![e]\!] \circ \{f_x\}}(1)\Big) \leq \tilde{v}' \cosh(\tilde{v})$ (Lemma 5.3.4).

Case: $\dfrac{\langle\tilde{\sigma}, e\rangle \tilde{\Downarrow} (\tilde{v}, \tilde{v}')}{\langle\tilde{\sigma}, \texttt{exp(}e\texttt{)}\rangle \tilde{\Downarrow} (\exp(\tilde{v}), \tilde{v}' \exp(\tilde{v}))}$

$[\![\texttt{exp(}e\texttt{)}]\!](\sigma) = \exp([\![e]\!](\sigma))$, so $([\![\texttt{exp(}e\texttt{)}]\!] \circ \{f_x\})(\sigma) = \exp([\![e]\!] \circ \{f_x\}(\sigma))$. By the inductive hypothesis, $\widetilde{[\![e]\!] \circ \{f_x\}}(1) \leq \tilde{v}$ and $\widetilde{[\![e]\!] \circ \{f_x\}}'(1) \leq \tilde{v}'$. Note that $\widetilde{\exp}(x) = \exp(x)$. By Lemma 5.3.3, $\widetilde{[\![\texttt{exp(}e\texttt{)}]\!] \circ \{f_x\}}(1) \leq \exp\Big(\widetilde{[\![e]\!] \circ \{f_x\}}(1)\Big) \leq \exp(\tilde{v})$ and $\widetilde{[\![\texttt{exp(}e\texttt{)}]\!] \circ \{f_x\}}'(1) \leq \widetilde{[\![e]\!] \circ \{f_x\}}'(1) \cdot \widetilde{\exp}\Big(\widetilde{[\![e]\!] \circ \{f_x\}}(1)\Big) \leq \tilde{v}' \exp(\tilde{v})$ (Lemma 5.3.4).

Case: $\dfrac{\langle\tilde{\sigma}, e\rangle \tilde{\Downarrow} (\tilde{v}, \tilde{v}') \qquad b > \tilde{v}\sqrt{b^2+1}}{\langle\tilde{\sigma}, \texttt{log\{}b\texttt{\}(}e\texttt{)}\rangle \tilde{\Downarrow} \left(|\log(b)| + \log(b) - \log\left(b - \tilde{v}\sqrt{b^2+1}\right), \dfrac{\tilde{v}'}{b - \tilde{v}\sqrt{b^2+1}}\right)}$

$[\![\texttt{log\{}b\texttt{\}(}e\texttt{)}]\!](\sigma) = \log([\![e]\!])$, so $([\![\texttt{log\{}b\texttt{\}(}e\texttt{)}]\!] \circ \{f_x\})(\sigma) = \log(([\![e \circ \{f_x\}]\!])(\sigma))$. By the inductive hypothesis $[\![e \circ \{f_x\}]\!](\sigma) \leq \tilde{v}$ and $[\![e \circ \{f_x\}]\!]'(\sigma) \leq \tilde{v}'$. Expanding around $x = b$, $\log(x) = \log(b) + \sum_{k=1}^{\infty}(-1)^{k+1}\frac{(x-b)^k}{kb^k}$. Note that this is a multivariate analytic power series in the sense of Equation (5.5) (with the 1 appended) with $V_k = \{v_k\}$, $a_{v_0} = \log(b)$, $a_{v_{k>0}} = \frac{(-1)^{k+1}}{kb^k}$, and $\beta_{v_k,i} = \begin{bmatrix} 1 \\ -b \end{bmatrix}$. Thus, $\widetilde{\log}(x) = |\log(b)| + \sum_{k=1}^{\infty}\frac{\left(1+b^2\right)^{\frac{k}{2}}}{kb^k}x^k = |\log(b)| + \log(b) - \log\left(b - x\sqrt{b^2+1}\right)$, which converges for $b > x\sqrt{b^2+1}$. By Lemma 5.3.3, $[\![\widetilde{\texttt{log\{}b\texttt{\}(}e\texttt{)}}]\!](\sigma) \leq |\log(b)| + \log(b) - \log\left(b - \tilde{v}\sqrt{b^2+1}\right)$ and $[\![\widetilde{\texttt{log\{}b\texttt{\}(}e\texttt{)}}]\!]'(\sigma) \leq \frac{\tilde{v}'}{b-\tilde{v}\sqrt{b^2+1}}$ (Lemma 5.3.4).

Case: $\dfrac{\langle\tilde{\sigma}, e_1\rangle \tilde{\Downarrow} (\tilde{v}, \tilde{v}') \qquad \langle\tilde{\sigma}, e_2\rangle \tilde{\Downarrow} (\tilde{v}_2, \tilde{v}_2')}{\langle\tilde{\sigma}, e_1\texttt{+}e_2\rangle \tilde{\Downarrow} (\tilde{v} + \tilde{v}_2, \tilde{v}' + \tilde{v}_2')}$

$[\![e_1 \texttt{+} e_2]\!](\sigma) = [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma)$, so $([\![e_1 + e_2]\!] \circ \{f_x\})(\sigma) = ([\![e_1]\!] \circ \{f_x\})(\sigma) + ([\![e_2]\!] \circ \{f_x\})(\sigma)$. By the inductive hypothesis, $[\![e_1]\!] \circ \{f_x\}(\sigma) \leq \tilde{v}$, $[\![e_1]\!] \circ \{f_x\}'(\sigma) \leq \tilde{v}'$, $[\![e_2]\!] \circ \{f_x\}(\sigma) \leq \tilde{v}_2$, and $[\![e_2]\!] \circ \{f_x\}'(\sigma) \leq \tilde{v}_2'$ Thus by Lemma 5.3.1, $[\![e_1 + e_2]\!] \circ \{f_x\}(\sigma) \leq \tilde{v} + \tilde{v}_2$ and $[\![e_1 + e_2]\!] \circ \{f_x\}'(\sigma) \leq \tilde{v}' + \tilde{v}_2'$.

Case: $\dfrac{\langle\tilde{\sigma}, e_1\rangle \tilde{\Downarrow} (\tilde{v}, \tilde{v}') \qquad \langle\tilde{\sigma}, e_2\rangle \tilde{\Downarrow} (\tilde{v}_2, \tilde{v}_2')}{\langle\tilde{\sigma}, e_1\texttt{*}e_2\rangle \tilde{\Downarrow} (\tilde{v} \cdot \tilde{v}_2, \tilde{v}' \cdot \tilde{v}_2 + \tilde{v} \cdot \tilde{v}_2')}$

$[\![e_1 * e_2]\!](\sigma) = [\![e_1]\!](\sigma) \cdot [\![e_2]\!](\sigma)$, so $([\![e_1 * e_2]\!] \circ \{f_x\})(\sigma) = ([\![e_1]\!] \circ \{f_x\})(\sigma) \cdot ([\![e_2]\!] \circ \{f_x\})(\sigma)$. By the inductive hypothesis $[\![e_1]\!] \circ \{f_x\}(\sigma) \leq \tilde{v}$, $[\![e_1]\!] \circ \{f_x\}'(\sigma) \leq \tilde{v}'$, $[\![e_2]\!] \circ \{f_x\}(\sigma) \leq \tilde{v}_2$, and $[\![e_2]\!] \circ \{f_x\}'(\sigma) \leq \tilde{v}_2'$ Thus by Lemma 5.3.2, $[\![e_1 * e_2]\!] \circ \{f_x\}(\sigma) \leq \tilde{v} \cdot \tilde{v}_2$ and $[\![e_1 * e_2]\!] \circ \{f_x\}'(\sigma) \leq \tilde{v}' \cdot \tilde{v}_2 + \tilde{v} \cdot \tilde{v}_2'$.

□

Lemma 4.3. *The tilde big-step trace relation upper bounds on the tilde and tilde derivative:*

$$\left(\langle\tilde{\sigma}, t\rangle \tilde{\Downarrow} \tilde{\sigma}' \wedge \tilde{\sigma} \vdash \{f_x\}\right) \Rightarrow \tilde{\sigma}' \vdash \left\{[\![t]\!]_y \circ \{f_x\}\right\}$$

Proof. The proof proceeds by induction on t.

Case: $\dfrac{}{\langle \tilde{\sigma}, \texttt{skip} \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}}$

$[\![\texttt{skip}]\!]_x(\sigma) = \sigma(x)$. Thus $[\![\texttt{skip}]\!]_y \circ \{f_x\} = f_y$. $\tilde{\sigma} \vdash \{f_x\}$, that $\tilde{\sigma}' = \tilde{\sigma}$, and that $\left\{[\![\texttt{skip}]\!]_y \circ \{f_x\}\right\} = \{f_x\}$, so $\tilde{\sigma}' \vdash \left\{[\![t]\!]_y \circ \{f_x\}\right\}$.

Case: $\dfrac{\langle \tilde{\sigma}, t_1 \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}' \qquad \langle \tilde{\sigma}', t_2 \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}''}{\langle \tilde{\sigma}, t_1 \,;\, t_2 \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}''}$

$[\![t_1 \,;\, t_2]\!]_x(\sigma) = [\![t_2]\!]_x\left(\left\{y \mapsto [\![t_1]\!]_y(\sigma)\right\}\right)$. By the IH, $\tilde{\sigma}' \vdash \left\{[\![t_1]\!]_y \circ \{f_x\}\right\}$, and $\tilde{\sigma}'' \vdash \left\{[\![t_2]\!]_z \circ \left\{[\![t_1]\!]_y \circ \{f_x\}\right\}\right\}$. Since composition is associative, this is equivalent to $\tilde{\sigma}'' \vdash \left\{\left([\![t_2]\!]_z \circ \left\{[\![t_1]\!]_y\right\}\right) \circ \{f_x\}\right\}$. Because $[\![t_2]\!]_z \circ \left\{[\![t_1]\!]_y\right\} = [\![t_1 \,;\, t_2]\!]_z$, this proves this case.

Case: $\dfrac{\langle \tilde{\sigma}, e \rangle \,\tilde{\Downarrow}\, (\tilde{v}, \tilde{v}')}{\langle \tilde{\sigma}, x = e \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}[x \mapsto (\tilde{v}, \tilde{v}')]}$

$[\![x = e]\!]_y(\sigma)$ is equal to $[\![e]\!](\sigma)$ for $y = x$ and $\sigma(y)$ for $y \neq x$. By Lemma 5.3.2, $\widetilde{[\![e]\!] \circ \{f_x\}}(1) \leq \tilde{v}$ and $\widetilde{[\![e]\!] \circ \{f_x\}}'(1) \leq \tilde{v}'$. Thus $\tilde{\sigma}[x \mapsto (\tilde{v}, \tilde{v}')] \vdash \{[\![e]\!] \circ \{f_x\}\}$.

□

Theorem 3. *The complexity relation computes an upper bound on the true complexity:*

$$\zeta_{\,\tilde{\Downarrow}}(t, x) \leq z \Rightarrow \zeta([\![t]\!]_x) \leq z$$

Proof. $\langle \{x_i \mapsto (1,1)\}, t \rangle \,\tilde{\Downarrow}\, \tilde{\sigma}$. Note that $\{x_i \mapsto (1,1)\} \vdash \{f_{x_i}(x) = x\}$. Thus by Lemma 5.3.3, $\tilde{\sigma} \vdash \{[\![t]\!]_x\}$, so for $\tilde{\sigma}(x) = (\tilde{v}, \tilde{v}')$ and $z = \tilde{v}'^2$ thus $\widetilde{[\![t]\!]_x}' \leq \tilde{v}'$, so $\left(\widetilde{[\![t]\!]_x}'\right)^2 \leq z$. □

5.4 Evaluation

In this section I evaluate the complexity-guided sampling approach using Turaco's complexity analysis to determine sampling rates for a range of benchmark programs. I demonstrate that complexity-guided sampling consistently results in more accurate surrogates than those trained using baseline distributions (the frequency distribution of paths and the uniform dis-

tribution of paths). I also demonstrate that such an improvement in surrogate error can result in an improvement in execution speed in an application with a maximum error threshold.

In Section 5.4.1 I first evaluate across both a set of real-world programs, showing expected error improvements in practice, and also a set of synthetic programs, showing cases where the complexity-guided sampling approach shines and cases where it fails. Then in Section 5.4.2 I dive into a case study on a specific large-scale program, a demonstration 3D renderer (Lettier, 2019), such as forms the core of a graphics rendering pipeline for a movie or 3D game engine (Christensen et al., 2018; Tatarchuk, 2009).

Utility. The core assumption for this approach's utility is that there is a cost to generating data. Under this assumption, surrogate developers trade off between costs incurred to data generation and costs incurred due to the error of the resulting surrogate. These costs are application-specific: different applications have different objectives and constraints that determine the relative importance of these costs.

In this section, the primary metric that I evaluate is the improvement of the error of the surrogate for a given cost of data generation. In Section 5.4.2, I also evaluate the improvement in the number of samples required to achieve a given error threshold, as well as the improvement in execution speed that results from the improvement in error at a given number of samples.

5.4.1 Evaluation Across Programs

In this section I evaluate the complexity-guided sampling approach using Turaco's complexity analysis to determine sampling rates for a range of benchmark programs. I evaluate both a set of real-world programs, showing expected error improvements in practice, and also a set of synthetic programs, showing cases where the complexity-guided sampling approach shines and cases where it fails.

Methodology. Following the input scale assumptions from Agarwala et al. (2021), I sample each input variable uniformly between $[-1, 1]$ or $[0, 1]$ as appropriate for the program.

Table 5.1: Average change in error across all budgets from using complexity-guided sampling compared to baselines on each benchmark (higher values means complexity-guided sampling has lower error).

Benchmark			Baseline			
Program	**LoC**	**Paths**	**Frequency (Predicted)**	**Frequency (Empirical)**	**Uniform (Predicted)**	**Uniform (Empirical)**
Luminance	14	3	2.58%	15.01%	6.97%	15.17%
Huber	13	3	0.49%	8.15%	1.93%	9.54%
BlackScholes	15	2	4.43%	3.61%	1.30%	4.00%
Camera	69	3	2.83%	0.56%	0.22%	1.36%
EQuake	34	2	7.45%	2.25%	7.45%	2.25%
Jmeint	176	18	2.34%	0.01%	8.44%	1.02%
Geomean			3.33%	4.81%	4.33%	5.43%

I insert scale factors as appropriate given the expected data distribution of the original program. I then uniformly sample inputs from these ranges. This induces both a data distribution over inputs and a path frequency distribution.

For all benchmarks other than the Jmeint benchmark, I evaluate using a training data budget using 10 points logarithmically spaced between 10 and 1000. For the Jmeint benchmark, which is more data intensive, I evaluate using a training data budget using 10 points logarithmically spaced between 1000 and 10,000. When computing the complexity-guided sampling distribution, I use $\delta = 0.1$.

For each path in each benchmark, I train a 1-hidden-layer MLP with 1024 hidden units with a ReLU activation, using 10,000 steps of Adam with learning rate 0.0005 and batch size 128. I run the training for 5 trials.

I report both the predicted error (Equation (5.7)) improvement and the empirical improvement, the geometric mean improvement in error across trials. As in Section 5.4.2, improvement is defined as the mean percentage error between the predicted error for complexity-guided sampling and the baseline sampling method.

Table 5.2: Benchmark statistics.

Benchmark	Path	Complexity	Frequency Distribution	Uniform Distribution	Complexity Distribution
Luminance	`ll`	0.01	50.00%	33.33%	36.94%
	`rl`	1.21	10.00%	33.33%	13.98%
	`rr`	9.00	40.00%	33.33%	49.07%
Huber	`ll`	9.00	50.00%	33.33%	44.25%
	`lr`	9.00	25.00%	33.33%	27.88%
	`r`	9.00	25.00%	33.33%	27.88%
BlackScholes	`l`	165.72	75.00%	50.00%	59.34%
	`r`	485.23	25.00%	50.00%	40.66%
Camera	`ll`	0.86	44.54%	33.33%	36.96%
	`lrl`	0.81	35.48%	33.33%	31.63%
	`rrr`	9.53	19.98%	33.33%	31.41%
EQuake	`l`	56.29	50.00%	50.00%	26.99%
	`r`	1169.50	50.00%	50.00%	73.01%
Jmeint	`lllrrrll`	7236100.00	18.74%	5.56%	13.25%
	`lllrrrlrl`	7236100.00	5.31%	5.56%	5.72%
	`lllrrrlrrl`	7236100.00	5.31%	5.56%	5.71%
	`lllrrrrll`	7236100.00	5.30%	5.56%	5.71%
	`lllrrrrlrl`	7236100.00	2.52%	5.56%	3.48%
	`lllrrrrlrrl`	7236100.00	2.51%	5.56%	3.47%
	`lllrrrrrll`	7236100.00	5.30%	5.56%	5.71%
	`lllrrrrrlrl`	7236100.00	2.52%	5.56%	3.48%
	`lllrrrrrlrrl`	7236100.00	2.52%	5.56%	3.48%
	`rrrrrrll`	7236100.00	18.67%	5.56%	13.22%
	`rrrrrrlrl`	7236100.00	5.29%	5.56%	5.71%
	`rrrrrrlrrl`	7236100.00	5.29%	5.56%	5.70%
	`rrrrrrrll`	7236100.00	5.34%	5.56%	5.74%
	`rrrrrrrlrl`	7236100.00	2.52%	5.56%	3.48%
	`rrrrrrrlrrl`	7236100.00	2.51%	5.56%	3.47%
	`rrrrrrrrll`	7236100.00	5.29%	5.56%	5.70%
	`rrrrrrrrlrl`	7236100.00	2.53%	5.56%	3.49%
	`rrrrrrrrlrrl`	7236100.00	2.52%	5.56%	3.48%

```
fun(x, delta) {

  if (x > -delta) {
    if (x < delta) {
      res = x*x / 2  + delta
        ↪  * delta / 2;
    } else {
      res = x * delta;
    }
  } else {
      res = -x * delta;
  }

  return delta - res;
}
```

(a) Huber benchmark, which calculates a variant of the Huber loss for $\texttt{x} \in [-1, 1]$ and $\delta \in [0, 1]$.

```
fun (rate,time,sptprice,strike
   ↪ ,otype,NofXd1,NofXd2) {
 FutValueX = strike
    ↪  * exp(-rate*time);
 if (otype > 0) {
  OptPrice = sptprice*NofXd1
     ↪  - FutValueX*NofXd2;
 } else {
  NegNofXd1 = (1.0 - NofXd1);
  NegNofXd2 = (1.0 - NofXd2);
  OptPrice
     ↪  = FutValueX*NegNofXd2
     ↪  - sptprice*NegNofXd1;
 }
 return OptPrice;
}
```

(b) BlackScholes benchmark, which performs a part of the Black Scholes option pricing model (with `otype` positive for puts and negative for calls.)

Figure 5.12: Huber and BlackScholes benchmarks.

Results

Table 5.1 presents the results of the evaluation across 6 benchmark programs: Luminance, Huber, BlackScholes, Camera, EQuake, and Jmeint. Table 5.2 presents path and distribution statistics for each benchmark program.

I find that across this selection of programs, from predicted error improvements of 3.33% against the frequency sampling baseline, complexity-guided sampling results in an empirical improvement of 4.81%; from predicted error improvements of 4.33% against the uniform sampling baseline, complexity-guided sampling results in an empirical improvement of 5.43%. Such a decrease in error can significantly affect a system end-to-end, as shown in Chapter 4.

Luminance. The luminance benchmark is that of Section 5.1, and is presented in Figure 5.1a. This benchmark has 3 paths: when `sunPosition` < 0 (path `ll` with com-

plexity 0.01), when $0 <$ `sunPosition` < 0.1 (path `rl` with complexity 1.2), and when `sunPosition` > 0.1 (path `rr` with complexity 9).

Against the frequency baseline, compared to a predicted error improvement of 2.58%, complexity-guided sampling results in an improvement of 15.01%. Against the uniform baseline, compared to a predicted error improvement of 6.97%, complexity-guided sampling results in an improvement of 15.17%.

Huber. Figure 5.12a presents the Huber benchmark, which calculates the Huber loss for `x` $\in [-1, 1]$ and `delta` $\in [0, 1]$. This benchmark has 3 paths: when $-$`delta` $<$ `x` $<$ `delta` (path `ll` with complexity 9), when `x` $<$ $-$`delta` (path `lr` with complexity 9), and when `x` $>$ `delta` (path `r` with complexity 9).

Against the frequency baseline, compared to a predicted error improvement of 0.49%, complexity-guided sampling results in an improvement of 8.15%. Against the uniform baseline, compared to a predicted error improvement of 1.93%, complexity-guided sampling results in an improvement of 9.54%.

BlackScholes. Figure 5.12b presents the BlackScholes benchmark, which performs a part of the Black Scholes option pricing model (with `otype` positive for puts and negative for calls), for inputs uniform in $[0, 1]$ (other than `otype`, which is uniform in $[-1, 1]$). This benchmark is a fragment of the Black-Scholes benchmark in the AxBench benchmark suite (Yazdanbakhsh et al., 2017). This benchmark has 2 paths: when `otype` > 0 (path `l` with complexity 165.72; for puts) and when `otype` < 0 (path `r` with complexity 485.23; for calls).

Against the frequency baseline, compared to a predicted error improvement of 4.43%, complexity-guided sampling results in an improvement of 3.61%. Against the uniform baseline, compared to a predicted error improvement of 1.30%, complexity-guided sampling results in an improvement of 4.00%.

Camera. Figure A.1 in Appendix A.1 presents the Camera benchmark, which performs a part of the conversion from blackbody radiator color temperature to the CIE 1931 x,y chromaticity approximation function, for inputs $\texttt{x} \in [-1, 1]$, $\texttt{y} \in [-1, 1]$, $\texttt{invKiloK} \in [0, 1]$, and $\texttt{T} \in [0.1, 0.5]$ (note that `T` is used exclusively to determine the path). This benchmark is included in the Frankencamera platform (Adams et al., 2010), and is based off of an implementation by Kang et al. (2002). This benchmark has three paths: when $\texttt{T} < 0.2222$ (path `ll` with complexity 0.86), when $0.2222 < \texttt{T} < 0.4$ (path `lrl` with complexity 0.81), and when $0.4 < \texttt{T}$ (path `rrr` with complexity 9.53).

Against the frequency baseline, compared to a predicted error improvement of 2.83%, complexity-guided sampling results in an improvement of 0.56%. Against the uniform baseline, compared to a predicted error improvement of 0.22%, complexity-guided sampling results in an improvement of 1.36%.

EQuake. Figure A.2 in Appendix A.1 presents the EQuake benchmark, which computes the displacement of an object after one timestep in an earthquake simulation. This benchmark is a fragment of the 183.equake benchmark in the SPECfp2000 benchmark suite (Henning, 2000). This benchmark has 2 paths: when $\texttt{t} > 0.5$ (path `l` with complexity 56.29) and when $\texttt{t} < 0.5$ (path `r` with complexity 1169.50).

Against both the frequency and uniform baselines, compared to a predicted error improvement of 7.45%, complexity-guided sampling results in an improvement of 2.25%.

Jmeint. Figure A.3 in Appendix A.1 presents the Jmeint benchmark, which calculates whether two 3D triangles intersect, and several auxiliary variables related to their intersection. All inputs are sampled from $[-1, 1]$. This benchmark is a fragment of the Jmeint benchmark in the AxBench benchmark suite (Yazdanbakhsh et al., 2017). This benchmark has 18 paths; each path has the same complexity of 72,361,000, but with different frequencies.

Against the frequency baseline, compared to a predicted error improvement of 2.34%, complexity-guided sampling results in an improvement of 0.01%, a negligible change in er-

ror. Against the uniform baseline, compared to a predicted error improvement of 8.44%, complexity-guided sampling results in an improvement of 1.02%.

Note that this benchmark has the highest complexity of any evaluated program (requiring more samples and still resulting in higher overall errors), and also that empirically some paths do appear to be significantly easier to learn despite the matching complexities.

Analysis: Complexity-Guided Sampling Successes

In this section, I demonstrate examples of where the complexity-guided sampling technique results in significantly better error than baselines.

Complex paths. The first case is when some paths are significantly more complex than others: neither the frequency nor the uniform baseline take into account path complexity, so both baselines should undersample the complex path.

Figure 5.13a presents an example of such a case. In this example, the complexity of the $\texttt{x} < 0.5$ path (`l`) is 137677, while the complexity of the $\texttt{x} < 0.5$ path (`r`) is 57. The frequency of both the `l` and the `r` paths are 50%. The complexity-guided sampling approach samples the `l` path with probability 93% and the `r` path with probability 7%.

Against both the frequency and uniform baselines, compared to a predicted error improvement of 22.72%, complexity-guided sampling results in an improvement of 10.9%.

Skewed frequency distribution. The second case is when some paths are significantly more frequent than others. This confers advantages over the uniform baseline, which does not take into account path frequency, and also over the frequency baseline, which does not take into account the functional form of the learning bound in Equation (5.3) (i.e., that error decreases proportionally to the square root of the number of samples).

Figure 5.13b presents an example of such a case. In this example, all paths have a complexity of 14, while the frequencies are either 10% (for paths `l`, `rl`, and `rrl`) or 70% (for

path `rrr`). The complexity-guided sampling approach samples the 10%-frequency paths with probability 15%, and the 70%-frequency path with probability 55%.

Against the frequency baseline, compared to a predicted error improvement of 3.75%, complexity-guided sampling results in an improvement of 28.38%. Against the uniform baseline, compared to a predicted error improvement of 14.08%, complexity-guided sampling results in an improvement of 20.76%.

Note that this type of path distribution (with rare paths below 1% of the input data distribution) matches the distribution of paths in the renderer evaluation in Section 5.4.2, and results in a similar significant overperformance of the predicted error improvement.

Analysis: Complexity-Guided Sampling Failures

In this section, I demonstrate core examples of where the complexity-guided sampling technique results in significantly worse error than baselines.

Complexity imprecision. The first case is when the complexity results in too loose of an upper bound on the resulting error of a surrogate of that function. In this case, the complexity-guided sampling approach can oversample from the corresponding path.

Figure 5.13c presents an example of such a case. In this example, the complexity of the `l` path is 18,638 and the complexity of the `r` path is 16. Though I am not aware of any tighter bounds on the complexity of learning $\sin(4x)$ for $x \in [0.5, 1]$, in practice neural networks are able to learn this function to low error with relatively few samples.

The frequency of each path is 50%. The complexity-guided sampling approach samples the `l` path with probability 90.9% and the `r` path with probability 9.1%. Against both the frequency and uniform baselines, compared to a predicted error improvement of 20.88%, complexity-guided sampling results in a change of -92.59%, a significant increase in error.

```
fun(x, y) {
 if (x < 0.5) {
  res = sin(5 * y);
 } else {
  res = sin(2 * y);
 }
 return res;
}
```

(a) Success: synthetic example with skewed path complexities (137,678 v.s. 57) where complexity-guided sampling significantly improves error.

```
fun(x, y) {
 y = y + 1;
 if(x <
    ↪ 0.1) {r = sin(y);} else{
 if(x <
    ↪ 0.2) {r = sin(y);} else{
 if(x <
    ↪ 0.3) {r = sin(y);} else{
                  r = sin(y);}}}
 return r;
}
```

(b) Success: synthetic example with skewed path frequencies (10% v.s. 70%) where complexity-guided sampling significantly improves error.

```
fun(x, y) {

 if (x > 0.5) {
  y = sin(4*x);
 } else {
  y = y * 4;
}

 return y;
}
```

(c) Failure: synthetic example with a function on which the complexity bound is imprecise.

```
fun(x, y) {
 if (x > 0.5) {
  y = sin(10*
     ↪ y) / 1000;
 } else {
  y = sin(y);
}

 return y;
}
```

(d) Failure: synthetic example with a function that has different effective complexities across scales.

```
fun(x, y) {
 if (x < 0.5) {
  y = y + (y * 100);
  y = y-(y/101)*100;
  y = y * 2;
 } else {
  y = sin(2*y);
 }
 return y;
}
```

(e) Failure: synthetic example with a function on which the TURACO analysis is imprecise.

Figure 5.13: Examples of complexity-guided sampling successes and failures.

Nonuniform complexity. The second case is when the complexity of a learning function varies significantly across different scales. In this case, the complexity-guided sampling approach can oversample from the corresponding path.

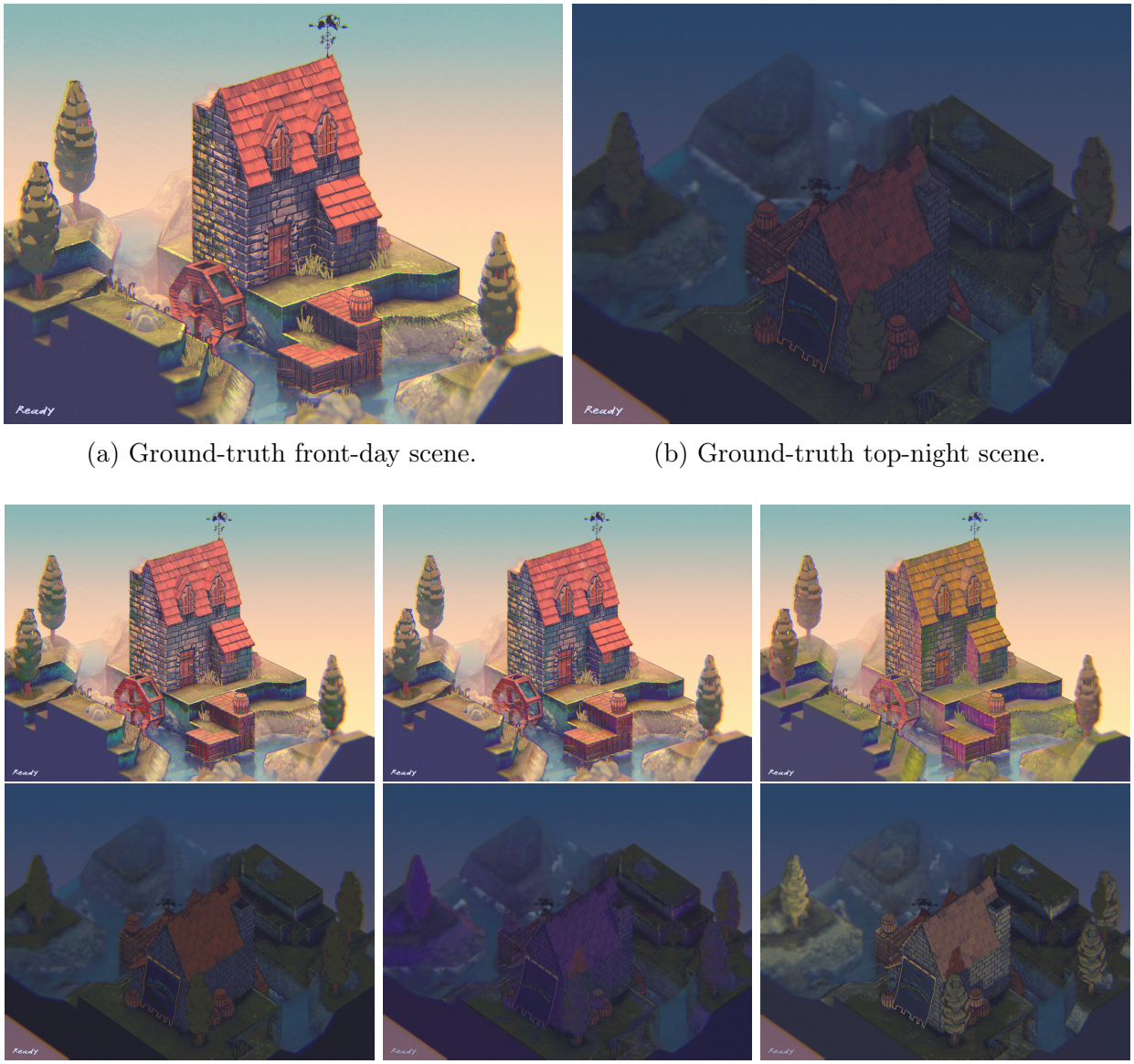
Figure 5.13d presents an example of such a case. In this example, the complexity of the `l` path is 12129 and the complexity of the `r` path is 2.38. This causes an issue with the complexity-guided sampling because with a large target error (e.g., $\epsilon > 0.01$), the `l` path is essentially zero (and therefore should have low complexity). However, with a small target error (e.g., $\epsilon < 0.00001$), the `l` path is very complex. Because the sample complexity bounds themselves are scale-independent upper bounds, they do not by default incorporate this knowledge.

The frequency of each path in this example is 50%. The complexity-guided sampling approach samples the `l` path with probability 92.9% and the `r` path with probability 7.1%. Against both the frequency and the uniform baselines, compared to a predicted error improvement of 22.7%, complexity-guided sampling results in a change of −351%, a 3.5× increase in error.

Analysis imprecision. The third case is when TURACO's analysis of the complexity is imprecise: Theorem 3 proves that TURACO's complexity analysis computes an upper bound on the tilde, but this upper bound may also be loose (as discussed in Section 5.3.4). In this case, the complexity-guided sampling approach can oversample from the corresponding path.

Figure 5.13e presents an example of such a case. In this example, the calculated complexity of the `l` path is 161604 and the calculated complexity of the `r` path is 56.6. With algebraic simplification, which TURACO does not perform, the complexity of the `l` path would be 4.

The frequency of each path is 50%. With TURACO's computed complexities, the complexity-guided sampling approach samples the `l` path with probability 93.3% and the `r` path with probability 7.7%. Against both the frequency and the uniform baselines, compared to a predicted error improvement of 23.03%, complexity-guided sampling results in a change of −491.22%, a 5× increase in error.



(a) Ground-truth front-day scene.

(b) Ground-truth top-night scene.

(c) Complexity-guided surrogate.

(d) Frequency-based surrogate.

(e) Uniform surrogate.

Figure 5.14: Ground-truth (top) and surrogate renderings (bottom) of scenes generated by the renderer.

```
fun (rimLight[4], isCelShadingEnabled[2],
    ↪ sunPositionBase[2], gamma[2], worldNormal[3], ssao[3], diffuseColor[4],
    ↪ diffuse[4], specular[4], emissionBase[3], isWater[2], isParticle[2]) {
 sunPosition = sin(sunPositionBase[0] * 0.1745329252);

 sunMixFactor = 0.5 - sunPosition / 2;
 ambientCoolBase = exp([-1.19732826, -0.79628794, -0.75289718] * gamma[0]);
 ambientWarmBase = exp([-0.26787945, -0.55686956, -0.91629073] * gamma[0]);

 if (0.5 > sunMixFactor) {
  ambientCool = ambientCoolBase / 2;
  ambientWarm = ambientWarmBase / 2;
 } else {
  ambientCool = ambientCoolBase * sunMixFactor;
  ambientWarm = ambientWarmBase * sunMixFactor;
 }

 if (0 > sunMixFactor) {
  skyLight = ambientCool;
  groundLight = ambientWarm;
 } else {
  if (sunMixFactor > 1) {
   skyLight = ambientWarm;
   groundLight = ambientCool;
  } else {
   skyLight = ambientCool * (1 - sunMixFactor) + ambientWarm * sunMixFactor;
   groundLight = ambientWarm * (1 - sunMixFactor) + ambientCool * sunMixFactor;
  }
 }

 worldNormalMixFactor = (1.0 + worldNormal[2]) / 2;
 ambientLight
    ↪ = groundLight * (1 - worldNormalMixFactor) + skyLight * worldNormalMixFactor;
 ambient = ambientLight * ssao * [diffuseColor[0], diffuseColor[1], diffuseColor[2]];

 if (0.01745240643728351 > sunPosition) {
  emission = emissionBase * 0.1;
 } else {
  sunPositionPow = exp(log{1}(sunPosition) * 0.4);
  emission = emissionBase * sunPositionPow;
 }

 out0rgb = [ambient[0], ambient[1], ambient[2]] + [diffuse[0],
    ↪ diffuse[1], diffuse[2]] + [rimLight[0], rimLight[1], rimLight[2]] + emission;


 if (isWater[0] > 0) {
  out0a = 0;
 } else {
  out0a = diffuseColor[3];
 }

 out1a = diffuseColor[3];

 if (isParticle[0] > 0) {
  out1rgb = [0,0,0];
 } else {
  out1rgb = [specular[0], specular[1], specular[2]];
 }

 out = [out0rgb[0],out0rgb[1],out0rgb[2],
    ↪ out0a, out1rgb[0],out1rgb[1],out1rgb[2], out1a];
 return out;
}
```

Figure 5.15: Full code for the renderer case study.

5.4.2 Renderer Demonstration

In this section I present a case study of the complexity-guided sampling results and complexity analysis. The program under study is a demonstration 3D renderer (Lettier, 2019), such as forms the core of a graphics rendering pipeline for a movie or 3D game engine (Christensen et al., 2018; Tatarchuk, 2009). Figures 5.14a and 5.14b show scenes that the renderer generates. I demonstrate that the sampling and analysis techniques in Sections 5.2 and 5.3 consistently result in more accurate surrogates than those trained using baseline distributions (the frequency distribution of paths and the uniform distribution).

Compared to training surrogates on the frequency distribution of paths, complexity-guided sampling decreases error by 17%. Compared to training on the uniform distribution of paths, complexity-guided sampling decreases error by 44%. These improvements in error correspond to perceptual improvements in the generated images, as shown in Figures 5.14c to 5.14e.

Program Under Study

The full renderer program is a 2750 lines-of-code C++ program, which invokes 38 different GLSL shader programs totaling 2446 lines of code. I learn a surrogate of a section of one core shader, totaling 60 lines of code.[4] Figure 5.15 presents the code for the renderer case study.[5]

This program is a good candidate to train a surrogate of for several reasons. First, it is an approximable program: as long as the outputs of a surrogate of the program are sufficiently close to the ground-truth outputs, the generated image will be perceptually indistinguishable. Second, its paths are all determined by *uniform* input variables, variables in GLSL that are constant across each invocation of the shader. This means that relative to the cost of executing the program, it is cheap to determine which path a given input induces in the program (and

[4]Lines 278 through 337 of https://github.com/lettier/3d-game-shaders-for-beginners/blob/29700/demonstration/shaders/fragment/base.frag.

[5]The original program is written in GLSL. I present a semantically equivalent translation (preserving all paths) of the program to Turaco for simplicity of presentation.

Table 5.3: Top: the identifier, lines of code, complexity, and description of each path present in the dataset. Bottom: the distribution (abbreviated distr.) of each path across each dataset: the frequency (Freq.) of each observed path, and the complexity-guided sampling rate (Com.) of that path.

Path		**lrrllr**	**lrrlrl**	**lrrlrr**	**lrrrlr**	**lrrrrl**	**lrrrrr**	**rrrllr**	**rrrlrl**	**rrrlrr**
Lines of Code		17	17	17	18	18	18	17	17	17
Complexity		6115	5806	6272	6401	6084	6562	8804	8433	8993
Description		Twilight Water	Twilight Smoke	Twilight Solids	Nighttime Water	Nighttime Smoke	Nighttime Solids	Daytime Water	Daytime Smoke	Daytime Solids
Dataset	**Distr.**	**lrrllr**	**lrrlrl**	**lrrlrr**	**lrrrlr**	**lrrrrl**	**lrrrrr**	**rrrllr**	**rrrlrl**	**rrrlrr**
Front Day	Freq.							5.0%	7.9%	87.1%
	Com.							11.0%	14.7%	74.3%
Front Night	Freq.				5.0%	7.9%	51.4%			35.6%
	Com.				8.9%	11.9%	42.4%			36.9%
Top Day	Freq.							6.7%	13.1%	80.1%
	Com.							12.9%	19.8%	67.4%
Top Night	Freq.	0.16%	0.06%	1.2%	0.3%	0.1%	2.4%	6.3%	12.9%	76.5%
	Com.	0.87%	0.45%	3.3%	1.4%	0.7%	5.3%	11.1%	17.7%	59.2%
Front	Freq.				2.5%	4.0%	25.7%	2.5%	4.0%	61.4%
	Com.				5.2%	7.0%	24.9%	5.8%	7.8%	49.4%
Top	Freq.	0.08%	0.03%	0.6%	0.2%	0.1%	1.2%	6.5%	13.0%	78.3%
	Com.	0.56%	0.29%	2.1%	0.9%	0.5%	3.5%	11.7%	18.4%	62.1%
Day	Freq.							5.9%	10.5%	83.6%
	Com.							11.9%	17.4%	70.7%
Night	Freq.	0.08%	0.03%	0.6%	2.7%	4.0%	26.9%	3.1%	6.5%	56.1%
	Com.	0.50%	0.26%	1.9%	5.2%	6.7%	24.4%	6.4%	10.3%	44.3%
All	Freq.	0.04%	0.02%	0.3%	1.3%	2.0%	13.5%	4.5%	8.5%	69.8%
	Com.	0.33%	0.17%	1.2%	3.4%	4.4%	16.0%	8.5%	12.8%	53.2%

Table 5.4: Average decrease in error across all budgets from using complexity-guided sampling compared to baselines on each dataset (higher values means complexity-guided sampling has lower error).

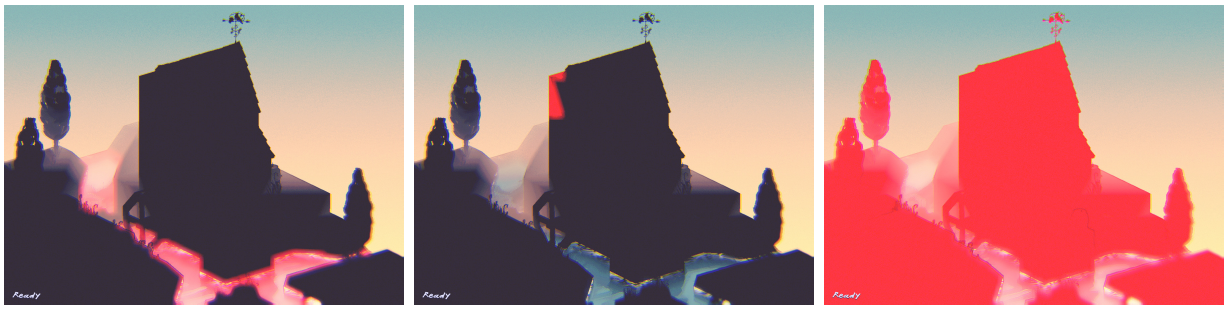
Baseline	**Front Day**	**Front Night**	**Top Day**	**Top Night**	**Front**	**Top**	**Day**	**Night**	**All**	**Mean**
Frequency	5%	-3%	-1%	48%	3%	31%	2%	21%	27%	17%
Uniform	39%	31%	36%	40%	42%	61%	34%	52%	52%	44%

thus which surrogate to apply). Third, its execution environment is well suited to be replaced with a neural network, since the original program itself performs batch processing on a GPU.

Input-output specification. This program is a shader which assigns colors to pixels in the image based on the scene geometry, materials, lights, and other properties. The program is called for each pixel that is rendered in the image. Each invocation of the program takes as input a set of 11 fixed-size vectors, totaling 35 inputs. The program returns as output a set of 4 fixed-size vectors, totaling 8 outputs. These outputs represent two RGBA colors, the first representing the base color of the pixel, and the second representing the color and intensity of a specular map at that pixel.

Scenes and datasets. I evaluate the renderer on four different scenes, which I combine into nine different datasets. Figures 5.14a and 5.14b present two of the four different scenes under consideration; the four scenes are all combinations of views from the front and top, during the day and night. I combine these scenes into nine datasets: a dataset with each scene, a dataset combining each scene from each angle (front day and front night, top day and top night), a dataset combining each scene from each time of day, and a dataset combining all scenes.

Paths. The program is a conjunction of 48 different paths, 9 of which are exercised by the renderer. The top part of Table 5.3 presents statistics about the paths under study, showing the identifier (a trace of `l` and `r` characters denoting which branch of each if statement the path takes), the lines of code in the corresponding trace, and the complexity of the corresponding trace according to the analysis in Section 5.3.2. The paths are broken up into a path for rendering smoke particles from the chimney, water particles in the river, and the solids of the ground and house. Each set of paths is duplicated for twilight, nighttime, and daytime. Within each time of day, the smoke paths are the least complex, followed by water then solids. Across different times, twilight paths are the least complex, followed by nighttime then daytime.



(a) Water path. (b) Smoke path. (c) Solids path.

Figure 5.16: Daytime scene with each different path highlighted red, and all others black.

Figure 5.16 shows side-by-side comparisons of the three classes of paths: water, smoke, and solids. In each of these images, one path returns red for all pixels while the other paths return black for all pixels. The base scene is the front daytime scene in Figure 5.14a.

Table 5.3 also presents the observed distribution and the complexity-guided distribution of paths for each dataset. In general, the twilight paths are rarer than the nighttime paths, which are rarer than the daytime paths: this is because data collection for the nighttime scenes extends through twilight and into the morning. For all datasets, the smoke paths are rarer than the water paths, which are in turn rarer than the solids paths; this is purely due to the scene geometry.

Surrogate Training and Deployment Methodology

To create and deploy a surrogate of the renderer, I train a surrogate of each path, then create a stratified surrogate which branches on the set of path conditions and applies the corresponding surrogate.

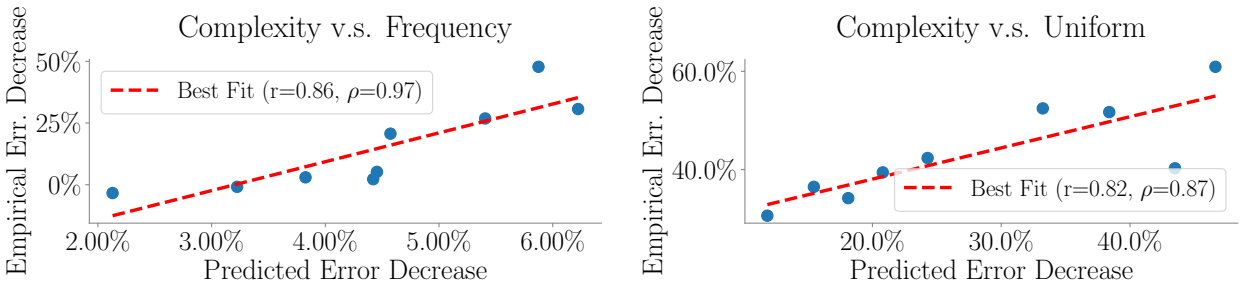
The goal is to compare the errors achieved by training on the complexity-guided distribution of paths against those of baseline distributions of paths. I compare the approaches across different training datasets, different total numbers of training data points, and evaluating across different evaluation sets, all with multiple trials.

For the design of each surrogate, I use a simple MLP architecture with a single hidden layer of 512 units and a ReLU activations. This architecture matches that of Agarwala et al. (2021), except using 512 rather than 1000 hidden units (I found that the accuracy of each path surrogate plateaued by 512 hidden units). I train the surrogate using the Adam optimizer with a learning rate of 0.0001 and a batch size of 256 for 50,000 steps. I run 5 trials of all experiments, and report the error as an arithmetic mean when reported in isolation for a given surrogate (e.g., as in Figure 5.18) and a geometric mean error when comparing relative error rates across different settings (e.g., as in the headline error improvement numbers in Table 5.4).

Surrogate Errors

Table 5.4 presents the geometric mean decrease in error of using complexity-guided path sampling compared to each baseline, on each dataset. Across most datasets, complexity-guided path sampling results in lower error than both frequency-based path sampling and uniform path sampling. On datasets with few paths (Front Day) and in which all paths are well represented (minimum 5% frequency), the gap is minimal, and frequency-based path sampling matches or outperforms complexity-guided path sampling. On datasets with more and rarer paths (e.g., Top Night), the gap widens and complexity-guided path sampling outperforms frequency-based path sampling; I discuss this phenomenon in Section 5.4.1. On all datasets, complexity-guided path sampling outperforms uniform path sampling.

Figure 5.17 presents the correlation between the predicted error (Equation (5.7)) and the observed empirical error for each dataset, showing a strong correlation. The left plot shows this correlation for surrogates trained with frequency-based path sampling, and the right plot shows this correlation for surrogates trained with uniform path sampling. The x axis is the decrease in predicted error (specifically, the mean percentage error between the predicted error for complexity-guided sampling and the sampling method in the plot), and the y axis is the decrease in empirical error (the mean percentage error between the error observed from complexity-guided sampling and the sampling method in the plot). Each



(a) Correlation between predicted and empirical surrogate error decrease between surrogates trained with complexity-guided path sampling compared to frequency-based path sampling.

(b) Correlation between predicted and empirical surrogate error decrease between surrogates trained with complexity-guided path sampling compared to uniform path sampling.

Figure 5.17: Correlation between predicted and empirical surrogate error decreases for the renderer case study.

point represents a different dataset (e.g., front-day, top-night, etc.). The red dotted line shows the line of best fit. For the frequency-based surrogates, the Pearson correlation is $r = 0.86$ and the Spearman correlation is $\rho = 0.97$. For the uniform surrogates, the Pearson correlation is $r = 0.82$ and the Spearman correlation is $\rho = 0.87$.

Figure 5.18 presents the error of surrogates on each dataset. Each plot shows the error for a different dataset. Each plot has three different lines, respectively showing the error of each surrogate training distribution (complexity-guided, frequency-based, and uniform). Each x axis is the total training data budget. Each y axis is the error of the resulting stratified surrogate.

For a given dataset budget, the complexity-guided sampling approach results in lower error than baseline sampling approaches. Generally, increasing this dataset budget also results in lower error for all approaches. These two approaches to decreasing surrogate error (better sampling techniques and sampling more data) are not in conflict with each other.

In Figure 5.18, the sampling approaches converge in error with large dataset budgets. This convergence is due to the evaluation methodology: following the convention of prior work which established these bounds (Arora et al., 2019; Agarwala et al., 2021), I use a fixed width for all neural networks, resulting in neural networks that saturate in error with large datasets. An alternative methodology would be to grow the width of the neural network along with

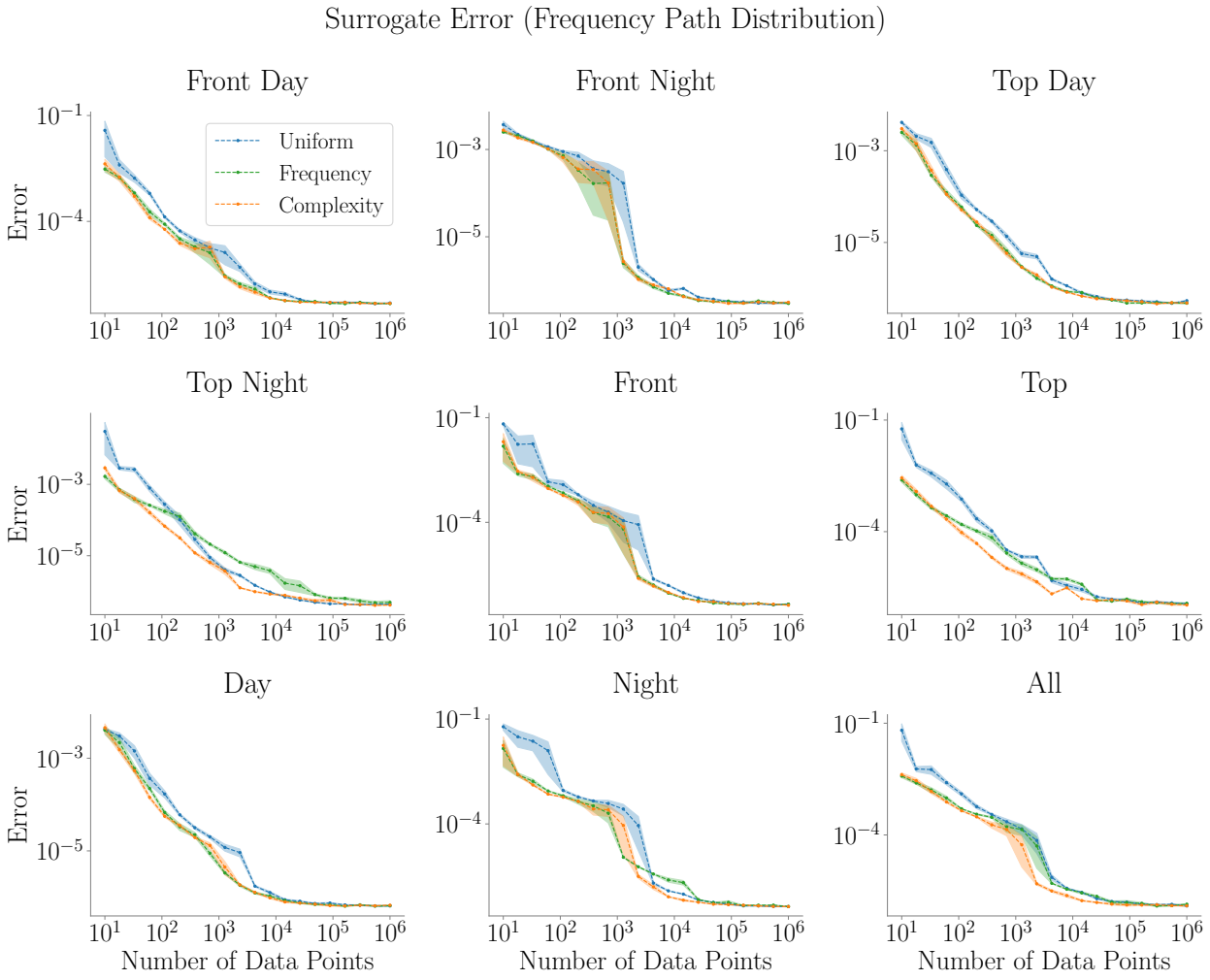


Figure 5.18: Errors of stratified surrogates of each dataset.

the size of the dataset, requiring a full hyperparameter search at each network scale. With such a methodology, the errors would not plateau in the way that they do in Figure 5.18.

Visualization

Figure 5.14 presents the renderings generated by the surrogates for the Front Day and the Top Night scene. These budgets correspond to the smallest budget that leads to a validation error less than 2%, which was qualitatively chosen as a threshold around which surrogate renders converge on the ground truth (i.e., the rendered scenes visually approach the quality of the original scene).

The top row shows the Front Day scene using surrogates trained on the Front Day dataset. In this scene, the complexity-guided and frequency-based surrogates result in similarly accurate renders, with the primary difference being that the frequency-based surrogate rendering has slightly darker green shadows on the front of the house. This similarity is expected given the similar errors observed in Table 5.4. Uniform sampling results in an inaccurate render, as expected given its high error. The bottom row shows the Top Night scene using surrogates trained on the dataset combining all scenes. In this scene the complexity-guided surrogate has the most accurate render, as expected given the errors observed in Table 5.4. The frequency-trained surrogate colors everything much darker purple. The uniform-trained surrogate in contrast colors everything much more tan. In sum, the error improvements in Table 5.4 correspond with improvements in the rendered images.

An alternative framing of these results is to consider the additional samples required to achieve the same error as complexity-guided sampling. For the Front Day scene, the frequency-based surrogate requires 1.1 times as many samples as the complexity-guided surrogate to achieve the same error, and the uniform-based surrogate requires 1.9 times as many samples. For the Top Night scene, the frequency-based surrogate requires 2.0 times as many samples as the complexity-guided surrogate to achieve the same error, and the uniform-based surrogate requires 2.6 times as many samples.

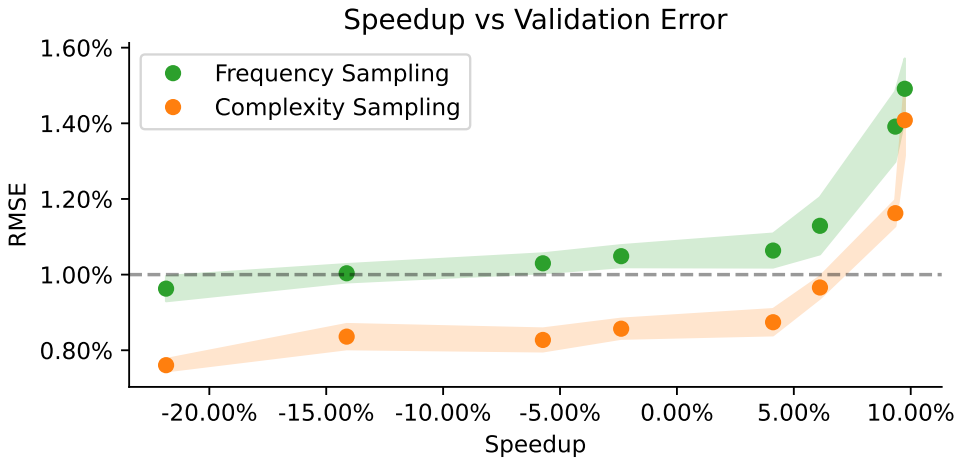


Figure 5.19: The validation error and speedup of different sizes of neural networks trained according to different data distributions.

Speedup

Figure 5.19 shows that complexity-guided sampling can result in faster-to-execute surrogates than baseline sampling approaches for a given error threshold and dataset budget. This plot shows the validation error and speedup of different sizes of neural networks trained according to different data distributions. Each surrogate is a monolithic (not stratified) surrogate trained on the front day dataset, ranging from a 1-layer 4-hidden-unit neural network to a 2-layer 32-hidden-unit neural network. The x axis is the end-to-end speedup of the renderer with the corresponding surrogate compared to the renderer without a surrogate. The y axis is the validation error of the corresponding surrogate. The dashed line at 1% error represents the error threshold for surrogate compilation in this example.

The fastest-to-execute neural network trained according to the frequency distribution of paths that meets this error threshold does not speed the program up, and instead slows it down by 14%. The fastest-to-execute neural network trained according to the complexity-guided distribution of paths that meets this error threshold speeds the program up by 6.12%.

5.5 Related Work

In this section I survey related work for each contribution.

Optimal stratified sampling. Optimal stratified sampling is a classic area in statistics (Thompson, 2012). Most work in this domain focuses on optimal parameter estimation, and uses stratified sampling to reduce the variance of estimates by ensuring sufficient independent samples are taken from each subpopulation. My approach is novel in the assumptions I make for training stratified surrogates of programs, and in the specific sample complexity bounds I base the results on.

Santner et al. (2018) survey sampling techniques for computer experiments. Chapter 5.2.3 discusses stratified random sampling in particular, showing optimality criteria for sampling for unbiased estimators. These approaches are generic for minimizing the variance of estimators, and do not consider specifically training a neural network. These approaches also do not consider the different complexity of different strata.

Cortes et al. (2019) present an active learning approach for learning in the regime where the input space is partitioned into separate regions (strata, using my terminology) and a separate hypothesis (surrogate) is trained of each, and derive a similar allocation of data points. This approach has several differences from my approach. First, it assumes a different form for sample complexity and derives correspondingly different sampling bounds than mine. The definition of complexity (ζ in my formalism) that Cortes et al. use is a function of the number of hypotheses in the hypothesis class, the total number of data points used, and the number of data points for a given stratum that have been queried thus far; it is not a function of any complexity metric of the function being learned. More concretely, Cortes et al.'s approach assumes a small, finite hypothesis class (the set of possible outputs of the training algorithm) of binary classifiers, and has runtime proportional to the size of the hypothesis class, requires samples proportional to the log of the size of the hypothesis class, and bounds the error of the

result as a function of the log of the size of the hypothesis class. In their evaluation, Cortes et al. use a hypothesis class of a set of 3000 random hyperplanes. However, this approach is not tractable when using a neural network as the hypothesis class: a neural network with 43,000 32-bit floating point weights (as in the case study in Section 5.4.2) induces a hypothesis class of size $10^{414,217}$. This results in intractable runtime and large or meaningless bounds. Beyond these distinctions, Cortes et al.'s approach is also an active learning approach that determines whether or not to query a label of a given data point for an input stream of data points, whereas my approach operates offline. Cortes et al.'s approach is thus a better fit when learning stratified functions of unknown complexity (e.g., non-analytic functions) using a finitely sized hypothesis class (not a neural network), and is targeted at the online setting when given a sampler of the overall data distribution but not one for each stratum. My approach is a better fit when learning *a priori* known stratified analytic functions with neural networks.

Sample complexity program analysis. Program analysis is a broad set of techniques to determine properties of programs (Nielson et al., 1999; Cousot and Cousot, 1977). The analysis in Section 5.3.2 is a novel nonstandard interpretation calculating the tilde, combined with a standard implementation of forward-mode automatic differentiation (Wengert, 1964; Griewank and Walther, 2008) and a standard symbolic execution which executes all paths in the program (King, 1976; Cadar et al., 2008).

Bao et al. (2012) present a program analysis that decomposes programs into continuous regions, with the goal of characterizing the sensitivity of each continuous region to input noise. This analysis computes a different notion of complexity than my approach does, and does not represent the sample complexity of learning a surrogate of each region.

Hoffmann and Hofmann (2010) present a program analysis that calculates the algorithmic complexity of a program. This complexity again does not lead to bounds on the sample complexity of learning a surrogate of the program.

5.6 Discussion

I present an approach to allocating samples among strata to train stratified neural network surrogates of stratified functions. I also present a programming language, TURACO, in which all programs are learnable stratified functions and a program analysis to determine the complexity of learning surrogates of those programs. Here I document these assumptions and note possible failure modes for the techniques, and discuss directions for future work.

5.6.1 Limitations

The contributions in Sections 5.2 and 5.3 make assumptions about the programs under study, the functions that those programs denote, and the surrogate training algorithms.

Assumptions imported from prior work. The sample complexity results are subject to all assumptions from the prior work that gives the sample complexity bounds for neural networks that I use (Agarwala et al., 2021). These sample complexity bounds only apply to analytic functions, which is a critical limitation which limits the applicability of the approach. They further assume that inputs come from the unit sphere; this does not match many practical applications, including those in Section 5.4. Finally, these sample complexity bounds assume that the neural network under study is a 2-layer, sufficiently wide neural network trained with SGD with an infinitely small step size, using a 1-Lipschitz loss function.

Despite these assumptions, Agarwala et al. (2021, Appendix B.2) empirically verify that the sample complexity bounds hold. I also show in Sections 5.1 and 5.4.2 that the theoretical sample complexity bounds correlate with empirical sample complexity results.

Complexity-guided sampling. The first assumption is that developers know the distribution of inputs ahead of time $D(x)$, both in terms of the distribution of strata $\int_{x \in s_i} D(x)\mathrm{d}x$ and the distribution of inputs within a given stratum $D(x|s_i)$. The second assumption is

that optimizing the upper bound of the per-stratum loss results in a reasonable optimum for the combined surrogate. The third is the assumption I make that $\forall i, j.\ \delta_i = \delta_j$, which I make to ensure a closed-form solution; this is not guaranteed to be optimal.

Convergence in the limit of strata. In the limit of infinite strata ($\lim_{c\to\infty}$), the complexity-guided sampling approach induced by Theorem 2 converges to sampling each stratum with probability proportional to $\left(\int_{x\in s_i} D(x)\mathrm{d}x\right)^{\frac{2}{3}}$ (see Note 5.2.1). In the limit, this distribution does not account for complexity. However in practice the complexity still guides sampling. Section 5.4.2 evaluates an example with all complexities $\zeta(f_i) \geq 5899$ and $\delta = 0.01$. For the $\log(\delta_i^{-1})$ term to match the contribution of the complexity term there would need to be $\approx 10^{2600}$ strata; this example only has 9. Thus while in the infinite limit of strata the approach is complexity-agnostic, in practice it is dominated by the complexity.

Note that this property necessarily occurs with any underlying PAC-style bound with a term that sums complexity and $\log\left(\frac{1}{\delta}\right)$ (e.g., those of Vapnik and Chervonenkis (1971) and Valiant (1984)): almost surely, paths are sampled with probability that does not depend on their complexity (see Note 5.2.2).

Program analysis. The main assumption here is that Agarwala et al. (2021)'s algebra on tilde functions results in a sufficiently precise upper bound on the tilde. This is not always the case, as discussed in Section 5.3.4. I also note that TURACO's program analysis could be made more precise in multiple well-known ways, for instance performing constant propagation, algebraic simplification, or automatic inference of constraints (which would be useful for `log` expressions). I have excluded such extensions for the sake of simplicity of presentation.

Analysis compute cost. For a given path, computing the tilde and its derivative has essentially the same cost as executing the path twice. Thus in the most pessimistic case this would allocate 2 more samples per path to the baseline approaches. However, this pessimistic case assumes that sampling a program input is free, which it may not be: for example the

renderer case study in Section 5.4.2 requires executing the video game engine (including running physics simulations) to get a program input for the shader of which I learn a surrogate.

Stratified surrogates. I provide sample complexity bounds for constructing stratified surrogates, assuming that for a given program every path is a different function. This assumes both that it is tractable to compute which stratum a given input resides in before applying the surrogate. This evaluation-time stratum check must not preclude the use of the surrogate for its downstream task. I therefore adopt a standard modeling assumption in the approximate computing literature: that precisely determining paths is an acceptable cost during approximate program execution (Sampson et al., 2011; Carbin et al., 2013).[6,7]

This also assumes that there are a tractable number of paths, which excludes programs with a large number of if statements or loops. The assumption that there are a tractable number of paths is a common assumption among techniques like concolic testing (King, 1976; Sen et al., 2005; Cadar et al., 2008). Similar to prior work, I find that in practice the evaluated programs only use a fraction of the syntactically possible paths (e.g., the Jmeint benchmark in Section 5.4 uses 18 out of 1728 possible paths).

Empirical evaluation limitations. I note two limitations in the empirical evaluations. The first is that some evaluations are in the ultra-low-data regime, where rounding to an integer number of data samples affects the accuracy. The second is that the δ parameter is set to an arbitrary value.

5.6.2 Future Work

Beyond verifying or weakening the above assumptions, I see several paths for future work.

[6] "EnerJ ... prohibit[s] approximate values in conditions that affect control flow." (Sampson et al., 2011).
[7] "Rely assumes that ... control flow branch targets are computed reliably." (Carbin et al., 2013).

There is no unique stratification for a given function. Different stratifications may have different properties, such as being more sample efficient or simpler to compute membership. Future work can explore automatically selecting the optimal stratification for a given function.

The complexity measure I use does not directly give the complexity of learning an approximate version of a function on a given domain. For instance, a function may be locally linear in a high-frequency region, but complex outside of this region. An approximation-aware stratification that splits the program up into functions that are essentially linear in a common stratum and complex in infrequent stratum may give better accuracy, and is an important direction for future work.

Similarly, such an approximation-aware stratification must be integrated into the optimal sampling results and the complexity analysis. For instance, it may take few samples to learn an approximately linear function (like $\sin(x)$ around 0) to moderate error, but a significant number of samples to learn it to low error (when it can no longer be treated as linear). Future work can incorporate this error-dependent analysis into the optimal sampling results and also the program analysis.

Some programs operate over strata with finite domains. For instance, a program may check whether an integer value is exactly equal to another integer, or whether it is in a small range. The neural network sample complexity bounds I use do not give reasonable sample complexity results in this regime. Future work can integrate finite domains into the sample complexity bounds.

5.6.3 Conclusion

In sum, the results in this chapter take a step towards a cohesive, end-to-end methodology for programming using surrogates of programs. These results further demonstrate that we can use facts about the modeled program to guide surrogate training to achieve better performance on downstream tasks.

Chapter 6

Renamer: A Transformer Architecture Invariant to Variable Renaming

In the previous chapter, I demonstrated that we can use facts about the modeled program to guide surrogate training data selection to achieve better performance on downstream tasks. In this chapter I present additional evidence to support the underlying hypothesis, showing that we can use facts about the program to guide the surrogate architecture design process.

In this chapter I identify a specific challenge in surrogate programming: developing surrogates that are *invariant* to input transformations that the underlying program is invariant to. Mirroring the program's invariance in the surrogate has several benefits. First, it guarantees that the surrogate's predictions are not biased by the choice of input representation. Second, it can lead to better in-distribution error, the error on inputs from the same distribution as the training distribution. Third, it can lead to better out-of-distribution generalization error, the error on inputs from a different distribution than the training distribution (LeCun and Bengio, 1995; Cohen and Welling, 2016; Lee et al., 2019; Keriven and Peyré, 2019; Wang et al., 2022). All are desirable properties in a surrogate model (Chapters 3 and 4).

Invariance in surrogates. While enforcing invariance is a common theme in general machine learning (Snavely, 2019; Bianchi et al., 2022), it has particular relevance in surrogate

programming. Programs often take structured inputs (e.g., programs in formal languages (Di Biagio and Davis, 2018), mathematical expressions (Pestourie et al., 2020), graphs (Mendis et al., 2019b), etc.) with multiple possible choices of representation for these inputs; while the program is invariant to these choices of representation, most neural network architectures are not (Biscione and Bowers, 2021; Lee et al., 2019). Further, in contrast to the soft invariances that are often studied in machine learning (e.g., invariance to translation or rotation which break down under large perturbations (Shorten and Khoshgoftaar, 2019)), the invariances I study in this chapter are hard (in that they apply to all possible perturbations) and easy to formalize. Finally, we can identify and prove the presence of invariances in modeled programs.

Renaming invariance. In this chapter I study *renaming invariance*, a particular type of invariance in sequence processing tasks which arises when reasoning about formal languages including programming languages (Chen et al., 2021; Alon et al., 2019), mathematics (Lample and Charton, 2020; Polu et al., 2022), and synthetic grammars of natural languages (Marzoev et al., 2020; Berant and Liang, 2014). Renaming invariance is defined over the *tokens* of the input sequence, the discrete elements that the input is broken into (e.g., keywords in a programming language input, words or subwords in a natural language input). Renaming invariance is then invariance to bijective transformations of the input tokens that preserve the semantics of the input. An example of renaming invariance is in llvm-mca, in which the predicted execution time of a piece of input code is invariant to the particular register names chosen in the basic block (as long as the code semantics are preserved modulo renaming).

Renaming sensitivity. General-purpose neural network architectures like LSTMs (Hochreiter and Schmidhuber, 1997) and Transformers (Vaswani et al., 2017) have shown impressive results on learning functions with renaming invariance (Chapter 2; Alon et al., 2019). However, these neural networks do not themselves demonstrate renaming invariance. For example, Alon et al. (2019) note sensitivity to "uninformative, obfuscated, or adversarial variable names". This sensitivity presents a challenge to deploying neural networks in this context as their pre-

dictions are not robust to semantics-preserving input transformations, hurting both the error of the networks as well as their ability to generalize to inputs with different naming patterns.

A common approach to learning models that are invariant to a given transformation is to train models using data augmentations that exhibit the transformation (and corresponding invariance) under study (Shorten and Khoshgoftaar, 2019; Feng et al., 2021). However, such approaches give no formal guarantees that the resulting models are always perfectly invariant to the transformation. Further, there is evidence that baking the inductive bias of the invariance into the model leads to accuracy improvements (LeCun and Bengio, 1995; Cohen and Welling, 2016; Lee et al., 2019; Keriven and Peyré, 2019; Wang et al., 2022).

Approach. I present an approach to enforcing renaming invariance in Transformers. The first key contribution that enables this approach is a formal definition of renaming invariance.

Renaming invariance is a property of functions that take sequences of tokens as input. The definition of renaming invariance involves two utility definitions. I first define a *view mapping* as a mapping from an input sequence to each token's *view*, representing the semantic information about each token that is salient to the function. I then define a *referent relation* for a given sequence as a binary relation that holds between tokens that refer to the same underlying entity.[1] Two tokens are *coreferential* if they refer to the same referent. A renaming invariant function is a function which generates the same output for any *semantics-preserving renaming* of the input sequence, any bijection of tokens that does not change tokens' views and preserves the coreferentiality of all tokens. In other words, renaming invariance is invariance to transformations which preserve the semantic meaning of each token and preserve the relationships between tokens.

I present two architecture changes that together enforce renaming invariance in Transformers. I refer to the resulting architecture as a Renamer.

[1]The term "referent" is borrowed from the literature on the philosophy of language (Frege, 1892; Yablo, 2011). I use the term "view", rather than Frege's similar term "sense", since the meanings do not quite align: two distinct tokens may have the same view, even if in Frege's terminology they would have different senses.

View projection. The first change, *view projection*, effectively replaces each token with a token that describes only its view. This enforces that the network is renaming invariant, because the network cannot make different predictions for different views. However, view projection alone reduces the representational capacity of the network, since the network can no longer distinguish whether tokens are coreferential.

Referent binding. To recover the representational capacity I introduce a novel modification to the attention layer in the first layer of the Transformer, which I call *referent binding*. Referent binding restricts the attention in the first layer of the Transformer, allowing tokens to only attend to other tokens that are coreferential with them. Together with positional embeddings, referent binding breaks the symmetry between tokens with the same view but which refer to different underlying entities, restoring the representative capacity of Renamer while maintaining that Renamer is renaming invariant.

Results. I evaluate Renamer on two case studies, developing a surrogate of llvm-mca and developing a surrogate of a symbolic differentiation engine. I find three primary results. First, Renamer's output is certified to be renaming invariant, guaranteeing that the output is not biased by the choice of names. Second, Renamer results in equivalent or lower in-distribution error than baseline models on tasks which are themselves renaming invariant. Third, Renamer is more robust to out-of-distribution variable name choices than baseline models.

Contributions. In this chapter I present the following contributions:

- I introduce and formally characterize the renaming invariance problem.
- I propose the two-step process of view projection and referent binding to enforce renaming invariance while maintaining representational power. I implement these in Renamer, a renaming invariant Transformer model architecture.

- I evaluate Renamer as a surrogate of llvm-mca, a renaming invariant x86 assembly processing task. Renamer reduces in-distribution error compared to a vanilla Transformer model by between 27.58% and 52.80% and out-of-distribution generalization error by between 54.56% and 79.58%.

- I evaluate Renamer as a surrogate of a renaming invariant symbolic differentiation engine. Renamer reduces out-of-distribution generalization error compared to a vanilla Transformer model by 49.1%.

By identifying and defining renaming invariance and proposing a Transformer model invariant to renaming invariance, this work takes a key step towards the goal of providing low-error surrogate with provable guarantees for modeling programs with input invariances.

6.1 Renaming Invariance in x86 Assembly

This section discusses how renaming invariance manifests in llvm-mca and shows how Renamer's mirroring of llvm-mca's invariance leads to a better surrogate. I first describe how x86 basic block throughput prediction is a renaming-invariant task. I then describe renaming invariant permutations for this task, and show that the task's labels are invariant to these permutations, but are not invariant to other permutations. I finally demonstrate that Renamer generates accurate and renaming invariant predictions for this task, while baseline models are not renaming invariant and are therefore less accurate.

Task under study. As in prior chapters, the task in this section is to create a neural network surrogate of llvm-mca, a CPU simulator included in the LLVM compiler infrastructure (Lattner and Adve, 2004). As input llvm-mca takes a *basic block* of x86-64 assembly language, a sequence of assembly instructions with no jumps or loops. It then outputs a prediction of the *throughput* of the basic block on the simulated CPU, a prediction of the number of CPU clock cycles taken to execute the block when repeated for a fixed number

of iterations. Learning a surrogate of llvm-mca results in faster or more accurate throughput estimation than using llvm-mca itself (Section 3.1 and Chapter 2).

Input specification. I evaluate using the BHive dataset of AT&T-syntax x86-64 basic blocks (Chen et al., 2019). Figure 6.1 presents three such basic blocks. To reintroduce terminology relevant for this chapter: AT&T syntax basic blocks are sequences of instructions, where each instruction consists of an opcode (e.g., `mov`), a source operand (e.g., `64(%rsp)`), and a destination operand (e.g., `%rax`). Each operand may be a constant (e.g., `$1`), a register (e.g., `%rax`), or a memory address (e.g., `64(%rsp)`). In AT&T syntax x86, the final operand of an instruction is the destination to which the instruction writes (and may also be read from).

Registers are the variables of x86-64 assembly. A given register operand consists of a *bitwidth* and a *base register*. The bitwidth is how many bits of the register data are addressed by the register. As an example, `%rax` addresses all 64 bits of the register data, `%eax` addresses the lowest 32 bits, and `%ax` addresses the lowest 16 bits. The base register is the location where register data is stored; this is typically indicated by the final several characters of the register (e.g., `ax` in `%rax`).

Simulation model. When executing an instruction, llvm-mca first waits for all of its source operands to be ready. An operand is ready when all previous instructions that have destination with the same base register (i.e., all predecessors in the *register dependency graph*) have finished executing. Once an instruction starts executing, its execution time is a function of the simulator state, the instruction's opcode, and the bitwidth of the instruction's operands.

Renaming invariance in llvm-mca. Renaming invariance manifests in llvm-mca as invariance to register renaming. When the base register names are renamed in a given block such that neither the register bitwidths nor the register dependency graph change, llvm-mca generates an identical prediction for this block. Thus, llvm-mca is invariant to this class of variable renaming transformations.

```
mov  64(%rsp), %rax
sub  $1,    56(%rbp)
mov  16(%rax), %eax
```

llvm-mca: 1.68 cycles

(a) Original block.

```
mov  64(%rsp), %rbx
sub  $1,    56(%rbp)
mov  16(%rbx), %ebx
```

llvm-mca: 1.68 cycles

(b) Invariant renaming.

```
mov  64(%rax), %rax
sub  $1,    56(%ebp)
mov  16(%rax), %eax
```

llvm-mca: 10.03 cycles

(c) Non-invariant renaming.

Figure 6.1: Example of an x86-64 basic block and invariant and non invariant renaming. The registers may be renamed, as long as each register is renamed to a register with the same bitwidth, and the register dependency graph is preserved.

To formalize this, I say that the *view* of a given register is its bitwidth: this is the semantic information about the register that is used by llvm-mca. I then define the *referent relation* as a binary relation that holds between two registers if they have the same base register (e.g., as `%rax` and `%eax` do); we say that any registers related by the referent relation are *coreferential*. Any transformation of registers that maintains both register views and the coreferentiality of all pairs of registers is a renaming invariant transformation.

Example. Figure 6.1 presents three x86 basic blocks in AT&T syntax. Figure 6.1a shows the original block. This basic block has a throughput of 1.6 cycles per iteration in llvm-mca's model. There are four unique registers in the basic block: `%rsp`, `%eax`, `%rax`, and `%rbp`. The registers `%rsp`, `%rax`, and `%rbp` all have the view `64-bit` (their bitwidth). `%eax` has the view `32-bit`. The registers `%eax` and `%rax` are coreferential as they both share the `ax` base register. The registers `%rsp` and `%rbp` are not coreferential with any other registers in Figure 6.1a as they refer to the `sp` and `bp` base registers respectively.

Figure 6.1b shows a semantically equivalent version of the block as all registers are renamed in a bijective manner that preserves each token's view and preserves the coreferentiality of all pairs of registers. To generate this new block, `%rax` was renamed to `%rbx`, `%eax` was renamed to `%ebx`, and all other registers remained fixed. Since `%rbx` has the view `64-bit` and `%ebx` has the view `32-bit`, this transformation preserves the views of all registers. Since `%rbx` and `%ebx` are coreferential (and are not coreferential with any other registers), this transformation

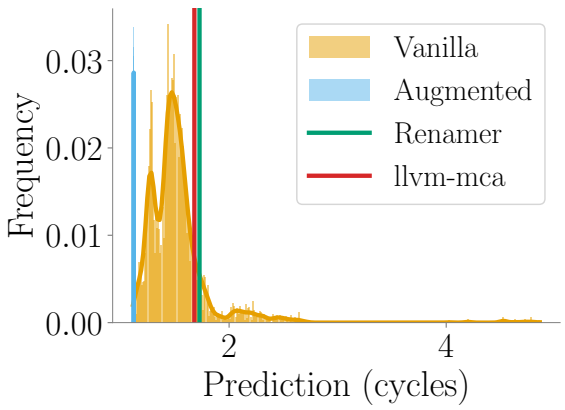
preserves the referent relation. Thus, because the permutation preserves the semantics of the original block, llvm-mca outputs the same timing for the renamed block as the original block.

Figure 6.1c shows a version of the block with registers renamed in manner that is not semantically equivalent, as it preserves neither the views nor the coreferentiality of registers. First, views are not preserved: `%rbp`, which has view `64-bit`, is renamed to `%ebp`, which has view `32-bit`. Such a transformation changes the execution time of the instruction, changing the semantics of the block. Second, coreferentiality is not preserved: `%rsp` is renamed to `%rax` on the first line while `%rax` and `%eax` are not renamed. This creates a new dependency in the renamed block as registers which originally referred to different base registers (and thus were not coreferential) were mapped to registers which do share the same base register (and are thus coreferential). Because the permutation does not preserve the semantics of the original block, llvm-mca outputs a different timing for the renamed block compared to the original block.

Renaming invariance in Transformers. In Section 3.1 I use a Transformer (Vaswani et al., 2017) as the surrogate architecture for llvm-mca. This Transformer model takes the basic block as input, and outputs a prediction of what llvm-mca would output on that basic block.

Figure 6.2 presents a case study of each model's predictions on the basic block presented in Figure 6.1a. The figure on the left is a histogram and corresponding density plot of predictions by a range of models on semantically equivalent renamings of the basic block. The ground-truth timing for this basic block as output by llvm-mca is 1.68 cycles. To generate the distribution of predictions, I uniformly sample 100,000 valid semantics-preserving register permutations, apply the permutation to the original block, then evaluate each model on the permuted block.

The models under study are single trials of the best-performing BERT-Tiny models, developed with different approaches. The *vanilla* model is a BERT-Tiny model trained on the original llvm-mca task. The *augmented* model is a BERT-Tiny model trained on the original llvm-mca task, but with input blocks randomly permuted with a semantics-preserving




```
mov  64(%rsp), %rax
sub  $1,    56(%rbp)
mov  16(%rax), %eax
```

llvm-mca: 1.68 cycles Vanilla: 1.51 cycles
Augmented: 1.12 cycles Renamer: **1.72** cycles

```
mov  64(%r8),  %rax
sub  $1,    56(%rbp)
mov  16(%rax), %eax
```

llvm-mca: 1.68 cycles Vanilla: 2.32 cycles
Augmented: 1.12 cycles Renamer: **1.72** cycles

Figure 6.2: The range of generated predictions for renamings of the basic block in Figure 6.1a.

permutation during training. The *Renamer* model is a Transformer model variant that I propose that is invariant to renaming transformations.

The vanilla model generates a range of predictions for permutations for this block, ranging from 1.11 cycles to 4.86 cycles. The predictions generated by the vanilla model are multimodal, though there are no clear indicators for which mode a given block will induce. The augmented model generates a significantly smaller range of predictions – though there is some variation on the order of one thousandth of a cycle, the predictions are essentially constant at 1.12 cycles. By construction, Renamer generates constant predictions for this basic block of 1.72 cycles.

6.2 Renaming Invariance

In this section I formally define what it means for a function to be renaming invariant, towards the goal of describing an architecture that is itself renaming invariant.

Formalism. Let $x \in \mathcal{X}$ be an input token from the set of input tokens and let $\vec{x} \in \mathcal{X}^n$ be a length n sequence of tokens. Let $f \in \mathcal{X}^n \to \mathcal{O}$ be a function over a sequence of tokens, where $\mathcal{O}$ is the set of possible outputs. Let $\mathcal{V}$ be a set of *views*. A view represents the semantic information about a token that is relevant to the function output. Each token in a sequence is associated with a view by the *view mapping*: $m_v \in \mathcal{X}^n \to \mathcal{V}^n$.

Let $R \in \mathcal{R}$ be a *referent relation*, a reflexive and symmetric binary relation on $\mathcal{X}$. The referent relation is a relation which holds between two tokens if they refer to the same object. Each sequence of tokens is associated with a referent relation by the *referent mapping*: $m_r \in \mathcal{X}^n \to \mathcal{R}$. I use the notation $\vec{x}_i =_{m_r} \vec{x}_j$ to denote that two tokens in $\vec{x}$ are related by the referent relation induced by the referent mapping m_r. I also refer to such tokens as *coreferential*.

Note that both the view mapping and the referent mapping are functions over sequences of tokens. This is because the view of a token and coreferentiality of a pair of tokens may depend on the other tokens in the sequence – these are defined sequencewise over tokens-in-context rather than pointwise over tokens-in-vocabulary.

Let $\sigma \in \mathcal{X}^n \to \mathcal{X}^n$ be a function over token sequences. I call σ *m_v-view-constrained* if:

$$\forall \vec{x}_i \in \vec{x}.\, m_v(\vec{x}_i) = m_v(\sigma(\vec{x})_i) \tag{6.1}$$

That is, σ is m_v-view-constrained if it preserves the view mapping for all tokens in the sequence. I call σ *m_r-referent-constrained* if:

$$\forall \vec{x}_i, \vec{x}_j \in \vec{x}.\, \vec{x}_i =_{m_r} \vec{x}_j \Leftrightarrow \sigma(\vec{x})_i =_{m_r} \sigma(\vec{x})_j \tag{6.2}$$

That is, σ is m_r-referent-constrained if it preserves (neither reducing nor augmenting) the coreferentiality of all pairs of tokens in the sequence. Finally I call σ *m_v-m_r-semantics-preserving* if it is both m_v-view-constrained and m_r-referent-constrained. A function f is *m_v-m_r-renaming invariant* if for all m_v-m_r-semantics-preserving permutations σ, $f(\vec{x}) = f(\sigma(\vec{x}))$. When the view mapping and referent mapping are clear from context, I refer to a function just as renaming invariant.

6.3 Renamer Architecture

In this section I present Renamer, a Transformer architecture that is itself renaming invariant. For any semantics-preserving renaming, Renamer's output is identical for the original and renamed input. Renamer otherwise preserves the full expressive power of the Transformer architecture, by modifying only the first layer of the Transformer.

6.3.1 Transformers Background

Renamer modifies the embeddings and self-attention mechanisms of Transformer networks. A Transformer first takes a length-n sequence of tokens $\vec{x}$ as input and converts each to a vector *embedding* in $\mathbb{R}^d$ (a d-dimensional vector of real numbers $\mathbb{R}$). A Transformer then computes a sequence of L layers of embeddings $E_l \in \mathbb{R}^{n \times d}$. Each layer consists of a *self-attention* layer followed by a *feed-forward* layer.

Embeddings. The embedding for a token sequence is composed of the combination of a *content embedding* and a *positional embedding* for each token. For a given token $\vec{x}_i$ in the input sequence, the content embedding $C_i \in \mathbb{R}^d$ is a function of the token, and the positional embedding $P_i \in \mathbb{R}^d$ is a function of the token's position in the input sequence. Given an input $\vec{x}$, the Transformer computes an initial embedding $E_0 \in \mathbb{R}^{n \times d}$ as $E = C + P$, where $+$ denotes element-wise addition of the respective content and positional embedding matrices.

Self-attention. Each layer first applies self-attention to the embeddings. As a function of the layer's input embedding E_l, the self-attention layer computes a *query* matrix $Q_l \in \mathbb{R}^{n \times d'}$, a *key* matrix $K_l \in \mathbb{R}^{n \times d'}$, and a *value* matrix $V_l \in \mathbb{R}^{n \times d}$. The Transformer also computes a boolean *attention mask* $M_l \in \mathbb{B}^{n \times n}$ from $\vec{x}$. The output of the self-attention layer is then

$E'_l = \text{softmax}\big(M_l \odot Q_l K_l^T\big) V_l$, where $\odot$ applies the following operator:

$$(M \odot X)_{i,j} = \begin{cases} X_{i,j}, & \text{when } M_{i,j} \\ -\infty, & \text{otherwise} \end{cases}$$

Feed-forward. The feed-forward layer is a multi-layer perceptron (MLP) applied elementwise to the output of the self-attention layer. The output of the feed-forward layer is then passed to the next layer of the Transformer: $E_{l+1} = \text{MLP}(E'_l)$.

6.3.2 Renamer Architecture Modifications

This section presents the two key architectural modifications that enable Renamer to be renaming invariant for a given view mapping m_v and referent mapping m_r: *view projection* and *referent binding*. Renamer includes these modifications only through the first layer (computing E_1); the remainder of Renamer is identical to a vanilla Transformer.

View projection. So that all input tokens which share the same view share the same representation, all tokens which share the same view must share the same content embedding:

$$\forall \vec{x}_i, \vec{x}_j \in \vec{x}.\, m_v(x_i) = m_v(x_j) \Leftrightarrow C_i = C_j$$

The result is that C is invariant to the application of m_v-view-constrained σ.

Then $\forall x_j \in [x_i].\, C(x_j) = C(\sigma(x_j)) \Rightarrow C^{[x_i]} = C^{\sigma^n([x_i])}$. Thus the content embeddings of the input sequence and the renamed input sequence are identical, meaning that after view projection Renamer is renaming invariant. Concretely, Renamer implements view projection by mapping each token in the input to a token that represents only its view.

Referent binding. View projection alone limits the class of functions the Transformer can represent: all tokens with the same view share the same content embedding, so the network can't differentiate which tokens in the input are coreferential and which are not.

To allow this differentiation, the first layer of Renamer only applies self-attention between coreferential tokens. Together with positional embeddings, this breaks symmetry between tokens that share the same view but that are not coreferential. Formally, given an input $\vec{x}$, the Renamer computes an attention mask $M_1^r \in \mathbb{R}^{n \times n}$ as follows:

$$M_1^r = \begin{cases} 1, & \text{when } \vec{x}_i =_{m_r} \vec{x}_j \\ 0, & \text{otherwise} \end{cases}$$

The computation of Q, K, V, and the feed-forward layer are otherwise identical to a vanilla Transformer. The representation after referent binding is thus $E_1 = \text{MLP}\big(\text{softmax}\big(M_1^r \odot Q_1 K_1^T\big)V_1\big)$.

Applying the referent attention mask gives the Renamer the capacity to differentiate between tokens which share the same view but which are not coreferential.

6.4 Evaluation on llvm-mca

I first evaluate Renamer by learning a surrogate of llvm-mca. I show that on this task, the Renamer results in lower in-distribution surrogate error than a suite of baseline approaches. I also show that Renamer is able to generalize to inputs with different distributions of names than those seen during training.

6.4.1 Task

The task under study is to take an x86-64 basic block as input, and output a prediction of the timing that llvm-mca would output for this basic block.

Register renaming invariance in llvm-mca. The views for this task are the set of possible bit-widths: {`8-bit`, `16-bit`, `32-bit`, `64-bit`}, combined with the set of register classes {`general-purpose`, `floating-point`, `vector`}. For example, the view mapping m_v has:

$$m_v(\%\text{rax}) = (\texttt{64-bit}, \texttt{general-purpose})$$
$$m_v(\%\text{rbx}) = (\texttt{64-bit}, \texttt{general-purpose})$$
$$m_v(\%\text{eax}) = (\texttt{32-bit}, \texttt{general-purpose})$$
$$m_v(\%\text{ebx}) = (\texttt{32-bit}, \texttt{general-purpose})$$
$$m_v(\%\text{ymm0}) = (\texttt{256-bit}, \texttt{vector})$$
$$m_v(\%\text{xmm0}) = (\texttt{128-bit}, \texttt{vector})$$

All other tokens have unique singleton views. The view mapping is created by applying this function pointwise to each token in the basic block.

The referent relation holds between any two register tokens that share the same base register (i.e., that point to the same data). For example, the referent mapping m_r has:

$$\%\text{rax} =_{m_r} \%\text{eax}$$
$$\%\text{rbx} =_{m_r} \%\text{ebx}$$
$$\%\text{ymm0} =_{m_r} \%\text{xmm0}$$

All other tokens are only coreferential with themselves. Note that as this is defined over tokens-in-vocabulary, this has the effect of enforcing that registers in false dependency chains are coreferential. This is a sound but not precise assumption for llvm-mca.

With the exception of instructions with *implicit operands*, those instructions which read from or write to specific hard-coded registers, llvm-mca is renaming invariant with this view mapping and referent relation. Such instructions with implicit operands include `push`, `pop`, `mul`, `div`, and others.

Dataset. I evaluate the Renamer on the BHive dataset (Chen et al., 2019), which is a collection of x86-64 basic blocks from a variety of real-world programs. The full BHive dataset consists of 287,639 basic blocks. I first remove inputs which have implicit operands from this dataset.[2] I then perform a random 70/10/20 split on the original dataset, resulting in 185,773 blocks in the training set, 26,107 blocks in the validation set, and 52,278 blocks in the test set.

6.4.2 Models

For all experiments, I use a BERT model (Devlin et al., 2019) as the backbone of the Renamer architecture. I evaluate on BERT-Tiny (2,923,777 parameters), BERT-Mini (8,639,489 parameters), and BERT-Small (25,274,369 parameters) (Turc et al., 2019).

Vanilla Transformer. The vanilla Transformer, which serves as a backbone for all evaluated models, is an encoder-only BERT (Devlin et al., 2019) with absolute positional encodings.

Augmented Transformer. In addition to the vanilla Transformer, I evaluate against an augmented training baseline. The architecture for the augmented baseline is identical to a vanilla Transformer. When training the augmented Transformer, I apply random semantics-preserving register permutations to the registers of input basic blocks (as described in Section 6.1). While the resulting model is not guaranteed to be invariant to variable renaming, this training paradigm removes any bias towards specific registers in the dataset.

Canonicalized Transformer. I also evaluate a Transformer model which *canonicalizes* basic blocks before using them as input to the vanilla Transformer. Canonicalization takes each basic block and maps it to a unique *canonical* basic block; canonicalization maps all semantics-preserving transformation of a basic block to the same canonical block. This preprocessing scheme ensures that the Transformer is invariant to variable renaming.

[2]This limitation can be addressed by making inputs with implicit operands coreferential with their operands, but I do not evaluate this approach.

Renamer. Renamer models have the same architecture as the vanilla model, except for the first encoder block which employs view projection and referent binding as described in Section 6.3. View projection and referent binding do not change the number of parameters of the Transformer, and maintain or reduce the number of FLOPs.

6.4.3 Evaluation Methodology

System. I evaluate using PyTorch-1.2.0 (Paszke et al., 2019), HuggingFace 4.17.0 (Wolf et al., 2020). Training is performed using an NVIDIA Tesla-V100.

Each reported metric is the mean and the standard error of that metric across five trials with different random seeds. In each table of reported results, I use a Kruskal-Wallis test to determine if there is a significant difference between the means of the results of the models (with $p = 0.05$), and then a post-hoc Conover test to determine which models have the best error (again with $p = 0.05$), which are then bolded.[3]

Hyperparameters. I train all models with the AdamW optimizer with a β_1, β_2 of 0.9 and 0.999 respectively. I empirically determine the learning-rate, weight-decay, and dropout through a grid search over the hyperparameters, selecting the hyperparameter configuration which has the lowest validation error for the vanilla model. The hyperparameters swept over are: learning-rate $\{3 \times 10^{-4}, 1 \times 10^{-4}, 5 \times 10^{-5}, 1 \times 10^{-5}\}$, weight-decay $\{0.0, 0.01\}$, and dropout $\{0.0, 0.1\}$. Based on this sweep, the Tiny and Mini models use a learning rate of 3×10^{-4} and the Small models use 1×10^{-4}. All models have a weight decay of 0.01, a dropout of 0, a batch size of 64, max sequence length of 128, and are trained for 500 epochs following the methodology in Chapter 3.

Objective. Following Chen et al. (2019) the loss and error metric are identical and are defined as the mean absolute percentage error (MAPE): $\mathcal{L}_{\text{MAPE}} = \sum_{x,y \in D} \frac{|f(x)-y|}{y}$

[3] https://github.com/maximtrp/scikit-posthocs/blob/f739c7aff6973f29a3e42f07af07caab9b08cef7/docs/source/tutorial.rst#non-parametric-anova-with-post-hoc-tests

Table 6.1: llvm-mca: MAPE on original test set

	Model size		
Model	BERT-Tiny	BERT-Mini	BERT-Small
Vanilla	3.30% ±0.16%	1.13% ±0.17%	1.25% ±0.43%
Augmented	3.34% ±0.11%	2.25% ±0.02%	2.36% ±0.02%
Canonicalized	3.03% ±0.33%	**0.96% ±0.06%**	0.76% ±0.04%
Renamer	**2.39% ±0.07%**	**0.85% ±0.07%**	**0.59% ±0.06%**

Table 6.2: llvm-mca: MAPE on test set with renamed registers

	Model size		
Model	BERT-Tiny	BERT-Mini	BERT-Small
Vanilla	5.26% ±0.37%	2.76% ±0.42%	2.89% ±0.52%
Augmented	3.44% ±0.11%	2.55% ±0.02%	2.36% ±0.03%
Canonicalized	3.03% ±0.33%	0.96% ±0.06%	0.76% ±0.04%
Renamer	**2.39% ±0.07%**	**0.85% ±0.07%**	**0.59% ±0.06%**

6.4.4 Results

In this section I evaluate the best performance of the vanilla, augmented, canonicalized, and Renamer models. This is defined as the test error of the epoch with the lowest validation error.

Standard test set. Table 6.1 shows the error of each model across BERT sizes on the standard test set. I find that across all model sizes, the Renamer model outperforms the vanilla, augmented, and canonicalized models on the original test set (except that the canonicalized model nearly matches on BERT-Mini). Renamer has a relative decrease in error compared to the vanilla model of 27.58%, 24.79%, and 52.80% for the Tiny, Mini, and Small BERT variants respectively. Additionally, the Renamer has a decrease in error as compared to the augmented model of 28.44%, 62.22%, and 75.00% for the Tiny, Mini, and Small BERT variants respectively. Finally, Renamer has a relative decrease in error compared to the canonicalized model of 21.12%, 11.46%, and 23.07% for the Tiny, Mini, and Small BERT variants respectively.

Additionally, I find that as model size is increased, Renamer suffers less from diminishing returns as compared to the vanilla and augmented models. For the vanilla model, the

relative decrease in error between the Tiny and Mini model is 66.76%, while error increases by 10.62% between the Mini and Small model. Likewise for the augmented model, the error decreases by 32.63% for Tiny to Mini and increases by 4.89% for Mini to Small. In contrast, the Renamer decreases in error between both size variants by 64.44% and 30.59%. While both the vanilla and augmented models have a relative increase in error between BERT-Mini and BERT-Small, Renamer has a significant decrease in error.

Renamed test set. In this section I evaluate the error of the best performing vanilla, augmented, canonicalized, and Renamer models on a semantics-preserving register renamed version of the test set. This experiment tests the hypothesis that the Renamer generalizes better to out-of-distribution variable name choices than the baseline models.

For each basic block in the original test set, a random semantics-preserving permutation is applied to the registers of the basic block. The checkpoint of the model with the lowest error on the unperturbed validation set is selected and then evaluated on the renamed test set.

Table 6.2 shows the error of each model across BERT sizes on the permuted version of the test set. The performance of the vanilla model is significantly affected by permuting the registers. Compared to the original, unperturbed test set, the error of the vanilla model increases by 59.39%, 144.25%, and 131.20% for the Tiny, Mini, and Small BERT variants respectively. This increase in error further empirically demonstrates the renaming sensitivity of the vanilla network architecture.

In contrast to the vanilla model, Renamer is provably invariant to register perturbations. Accordingly, on the renamed test set, Renamer outperforms the vanilla model by 54.56%, 69.20%, and 79.58% for the Tiny, Mini, and Small BERT variants respectively.

While the augmented training model is also less sensitive to register permutations and the canonicalized model is also guaranteed to be invariant, Renamer still outperforms both the augmented training and canonicalized models on the renamed test set.

6.5 Evaluation on Symbolic Differentiation Engine

I next evaluate the Renamer as a surrogate of a symbolic differentiation engine. I show that on this task, Renamer matches the in-distribution error, and is able to generalize to out-of-distribution inputs better than a suite of baseline models.

6.5.1 Task

I evaluate the Renamer on an invariant modification of Lample and Charton (2020)'s Backward dataset. Each input in the dataset is composed of a pair of expressions in prefix notation and the corresponding label is whether one expression is the partial derivative of the other with respect to the variable `x`. I present two examples below:

$$(\frac{\partial}{\partial x}\texttt{sin x} \stackrel{?}{=} \texttt{cos x}, \texttt{true})$$
$$(\frac{\partial}{\partial x}\texttt{mul a0 cos x} \stackrel{?}{=} \texttt{add a0 x}, \texttt{false})$$

Tokens in the dataset include standard mathematical operators (`add`, `sub`, `mul`, `pow`, `sin`, `cos`, etc.), *coefficient variables* (`a0`, `a1`, and `a2`), and *input variables* (`x`, `y`, and `z`).

Variable renaming invariance in Backward dataset. For this task, coefficient variables can be renamed to any other coefficient variable. Input variables other than `x` can be renamed to any other input variable other than `x`. The variable `x` and operators cannot be renamed.

Thus view mapping m_v maps coefficient variables to the view `coefficient`, input variables other than `x` to the view `input-nonx`, the variable `x` to the view `input-x`, and operators to unique views (applied pointwise across the sequence). Two tokens are coreferential if they are the same variable – that is, every token is only coreferential with itself.

Dataset. I use the same dataset generation technique as Lample and Charton (2020)'s Backward dataset, but randomly pair each expression with its derivative with probability 0.5,

and a random other expression from the dataset with probability 0.5. This turns the task into an invariant classification task, rather than the equivariant generation task it is in Lample and Charton (2020). The generated modified Backward dataset is composed of a training set of 300,000 examples, and validation and test set of 9128 and 9139 examples respectively.

6.5.2 Evaluation Methodology

Models. The details of the models used are the same as those defined in Section 6.4.2. However, for this symbolic algebra task I only use the BERT-Small model size (the largest model evaluated in Section 6.4).

System. The system details are the same as those in Section 6.4.3.

Hyperparameters. I use the same hyperparameters as those for BERT-Small reported in Section 6.4.3, with the exception of training for 50 epochs (rather than 500).

Objective. I train the model using the cross entropy loss, and report classification error.

6.5.3 Results

I again present results on the vanilla, canonicalized, and Renamer models on the original test set and an extended version of the test set. I again define test error as the test error of the epoch with the lowest validation error.

Standard test set. Table 6.3 presents the test errors of all models on the standard test set, achieving between 0.71% and 0.80% error. I find that all models perform similarly well on the original test set: the statistical test discussed in Section 6.4.3 does not distinguish between the errors from any model.

Table 6.3: Symbolic Algebra: Error of different model variants on the original test set.

	Model size
Model	BERT-Small
Vanilla	**0.79% ±0.13%**
Canonicalized	**0.71% ±0.14%**
Renamer	**0.80% ±0.06%**

Table 6.4: Symbolic Algebra: Error of different model variants on the augmented test set.

	Model size
Model	BERT-Small
Vanilla	4.68% ±0.69%
Canonicalized	4.98% ±0.56%
Renamer	**2.38% ±0.16%**

Extended test set. I also evaluate on a version of the test set extended with an additional coefficient variable, labeled `a3`. This experiment tests the hypothesis that the Renamer generalizes better to unseen variable names than all other models.

Table 6.4 presents the test errors of all models on the extended test set. While the performance of all models decreases on the extended test set, I find that on the extended test set Renamer significantly outperforms all other models. The vanilla model achieves an error of 4.68% on the extended test set while Renamer reaches an error of 2.38%, a 49.1% decrease in error compared to the vanilla model. Similarly, the canonicalized model achieves an error of 4.98%, meaning the Renamer has a 52.2% decrease in error compared to the canonicalized model.

6.6 Related Work

In this section I survey related work to the Renamer's approach to invariance.

Anonymization and canonicalization. Anonymization and canonicalization of training data is an area of much focus in fields ranging from ethical AI to privacy-preserving AI. In an effort to reduce gender and region bias in gendered pronoun resolution Liu (2019) mask individual names by drawing from a set of canonical names. Similarly, for debiasing and preserving privacy in clinical ML, de-identification of data is a prevalent technique (Dernoncourt et al., 2016; Liu et al., 2017; Johnson et al., 2020; Minot et al., 2022). Most work, however, is focused on the process of automatic de-identification and not on the result of training

on de-identified data. Minot et al. (2022) investigate the result of training on a canonicalized version of medical records. While canonicalization and de-identification reduce the bias of the model, they suffer from the fact that not every canonical representation of the same entity is guaranteed to have the same representation, and that inputs which have more entities than the number of canonical representations trained can't be represented. Furthermore, training on de-identified data is often associated with a degradation to network performance.

Transformer invariances. A wide variety of invariances and equivariances have been encoded into Transformer architectures. Lee et al. (2019) propose Set Transformer, which is invariant to permutations of the ordering of the input sequence. Fuchs et al. (2020) propose SE (3), which is equivariant to 3D translations and rotations, and evaluate on a variety of domains ranging from n-body simulations to point-cloud object classification. Su et al. (2021); Wennberg and Henter (2021) explore translation invariance in the context of natural language tasks. While these works enforce invariances, they all deal with spatial or positional invariances. To my knowledge, there is limited prior work on encoding invariances regarding the content of individual tokens.

6.7 Discussion

In this section I analyze the results in more detail and survey future work for renaming invariance.

Source of improvement. I have demonstrated that the Renamer achieves matching or lower error than baseline networks on the original test set, despite having restricted capacity in the first layer. I hypothesize that this is because the Renamer only represents the subset of functions that are renaming invariant. Because both the llvm-mca and the symbolic algebra tasks are renaming invariant, this leads to a smaller search space for SGD, along with the guarantee that all solutions match the tasks' renaming invariance.

```
add  $1 %rax
xor  %rax, %rax
add  $1 %rax
xor  %rax, %rax
xor  %rbx, %rbx
```

(a) Original block with a false dependency.

```
add  $1 %rax
xor  %rax, %rax
add  $1 %rbx
xor  %rax, %rax
xor  %rbx, %rbx
```

(b) Renamed block (breaking the false dependency).

Figure 6.3: An example of a false dependency and how it is broken by renaming.

Spurious correlations. Though the invariant model improves matches or reduces test error on all evaluated tasks, this may not happen on all renaming invariant tasks or datasets. Specifically, I hypothesize that renaming invariance can hurt error on datasets with a spurious correlation between referents and labels, even when the underlying task is renaming invariant. For instance, a version of BHive that only used the `%rax` register when the label is less than 100 cycles would have this property, even though the function being modeled is still renaming invariant. Though the in-distribution error of such a dataset may suffer with the Renamer, the transformed register evaluation performed in Section 6.4 would still result in the invariant model having better error.

More advanced view and referent mappings. The view and referent mappings in Sections 6.4 and 6.5 are defined over tokens-in-vocabulary and applied pointwise. It would be possible to define view and referent mappings over tokens-in-context, which would give a more precise statement of the invariance for llvm-mca in particular. For example, the blocks in Figure 6.3 have the same semantics, but would be considered different by the view and referent mappings defined in Section 6.4. In practice, Renamer still achieves low error on llvm-mca despite this false dependency, but other tasks may require a more precise definition of the view and referent mappings.

Conclusion. Renaming invariance is an important property of a range of tasks, from x86 assembly throughput prediction to symbolic differentiation. I formalize the concept of renaming invariance, and present the Renamer, a renaming invariant Transformer architecture. I

find that the Renamer results in matching or lower in-distribution error than baseline models on tasks which are themselves renaming invariant, and is more robust to out-of-distribution variable names than baseline models. my work takes a key step towards the goal of providing low-error models with provable guarantees for tasks with input invariances. Together, these results demonstrate that we can use facts about the modeled program to guide surrogate architecture selection to achieve better performance on downstream tasks.

Chapter 7

Conclusion and Future Directions

Throughout this thesis I have demonstrated the viability and effectiveness of surrogate programming, a programming methodology that uses surrogates to solve large-scale programming tasks. In Chapter 2 I demonstrated that surrogate programming can lead to state-of-the-art performance on large-scale programming tasks. In Chapter 3 I generalized these findings to show that surrogate programming is a coherent set of design patterns with a shared underlying methodology. Finally in Chapters 5 and 6 I demonstrated that we can guide surrogate design choices based on the semantics of the original program to achieve better performance on downstream tasks. Through these contributions, this thesis advances the understanding of surrogate programming for both researchers and practitioners. Nonetheless, there are still several open problems related to the development and application of surrogates.

Defining the scope of applicability. In Chapters 2 and 3 I demonstrated that surrogates provide state-of-the-art solutions to large-scale programming problems. However, certain programs and tasks do not admit surrogate programming as a viable solution. Some of these examples are clear: for example, programs in which neural networks cannot interpolate inputs, such as hash functions, or tasks for which the approximation error of a surrogate is not acceptable, such as in safety-critical systems. However, while the prevailing wisdom is that approximate programming techniques like surrogate programming are exclusively

applicable to such tasks that accept approximation (Stanley-Marbell et al., 2020), surrogate programming (specifically, surrogate adaptation and optimization) can be applied to tasks to result in programs that maintain or increase their reliability. There is an opportunity for future work in this domain to more precisely characterize when surrogate programming is and is not the most appropriate solution to a given programming task.

Broadening to other surrogate models. Though the design patterns in Chapter 3 are general to all types of surrogate models, the neural surrogate programming methodology in Sections 3.3.1 to 3.3.3 and the contributions in Chapter 5 are specific to when using neural networks as surrogate models. However, other surrogate models are popular in the literature, including surrogates based on Gaussian processes (Rasmussen and Williams, 2005; Alipourfard et al., 2017), linear models (Gelman and Hill, 2006; Ding et al., 2021), and random forests (Ho, 1995; Nardi et al., 2019). Future work in this direction includes the study of the extent to which other surrogate models can serve as backbones to the surrogate programming design patterns, and how to extend the programming methodology presented in this thesis to other classes of surrogate models beyond just neural networks.

Broadening Turaco and Renamer to larger scale programs. In Chapters 5 and 6 I have shown that we can use the semantics of the original program to guide surrogate design choices to achieve better performance on downstream tasks. However, these approaches lie in tension with the scalability of surrogate programming: the more we rely on precise analysis of the original program, the more we limit the applicability of surrogate programming to large-scale programs. This is because modern program analysis techniques often struggle to scale to large programs or those in languages less amenable to static analysis like C and C++. These approaches would benefit both from advances in program analysis to scale to larger programs, and from advances in surrogate programming to relax the reliance on precise program analysis (e.g., for Turaco by determining complexity dynamically rather than statically).

Generalization and robustness. Large-scale neural networks struggle to generalize outside of their training dataset (Barnard and Wessels, 1992; Ilyas et al., 2019; Xu et al., 2021). On the other hand, formal program reasoning techniques can prove properties about the behavior of programs on entire classes of inputs (Platzer, 2010). To address situations where the neural surrogate is expected to extrapolate outside of its training data, neural surrogate programmers must develop new approaches to recognizing and addressing generalization issues. This may be easier for surrogates of programs than for neural networks in general, because programmers still have access to the original program when developing a surrogate of that program.

Interpretability. Neural networks do not generate explanations for predictions (Gilpin et al., 2018), leading to difficulties when reasoning about neural surrogates' predictions. Future work can address these issues by better characterizing what interpretability means for different domains, developing interpretability tools for neural surrogates specifically (again aided by access to the original program), and characterizing when interpretability is and is not a relevant concern for neural surrogates. For example, surrogate optimization uses surrogates as an intermediate artifact to aid another optimization process, where interpretability is less of a concern.

Large language models of code. Large language models of code (Chen et al., 2021; OpenAI, 2023) are used for a range of purposes from code autocomplete, to end-to-end program synthesis, to program analysis (Chen et al., 2021; Nye et al., 2021; Olausson et al., 2024). While their immediate mode of operation, next token prediction, is different from the surrogate programming design patterns, in certain use cases they must serve as surrogates of programs. For example, Nye et al. (2021) use language models to execute programs. Inala et al. (2022) use LLMs to predict whether a program will produce correct examples, doing which correctly requires the language model to reason about program execution. Zhou et al. (2022) use LLMs to learn simple algorithms from examples, again requiring the LLM to reason about program execution. Future work can study the extent to which LLMs of code can serve

as effective surrogates of programs, in both their ability to model programs and their ability to successfully serve as backbones for surrogate compilation, adaptation, and optimization.

Future directions. I have at best scratched the surface of the applications and methodologies of surrogate programming. The three design patterns listed in Chapter 3 cover what I have observed in the literature, but there may be design patterns in use that I have not yet encountered, or design patterns that have not yet been invented. Further, there is much work left to do in fleshing out the methodologies underlying surrogate programming, from better understanding of how to trade off between different desiderata of the surrogate, to further exploiting the semantics of the original program to guide surrogate design and training.

Surrogate programming is one instance of the emerging paradigm of *neurosymbolic software systems*, systems which combine classical symbolic reasoning with modern machine learning techniques (Sun et al., 2022). In this future software landscape, surrogate programming will be no more exotic than currently standard program evolution techniques like refactoring. For example, it should be possible to simply highlight a block of code in an IDE, input a specification consisting of an objective and constraints, and have a system automatically design, train, and deploy a surrogate of that block of code that meets the specification. The ability to do this would significantly change the ways in which developers interact with and program large systems – it would be less important to be a domain expert in the system or task that the surrogate will replace. This would allow us to more easily design and evolve hybrid neurosymbolic computer systems, programs in which components are sketched out and replaced with surrogates, parameters are synthesized using surrogate optimization, and other chunks of our program may be entirely neural. This is also synergistic with a world in which more and more code is automatically written or synthesized: for example, code generated by a synthesizer (e.g., an LLM) might not do quite what the programmer wants, but developers can use surrogate programming to adapt the LLM's generated code to better meet their needs. This thesis is a concrete step towards that future, but there remains much work to be done.

Appendix A

Turaco

A.1 Evaluation Programs

This appendix presents the longer programs in the evaluation that were not presented in Section 5.4. Figure A.1 presents the Camera benchmark. Figure A.2 presents the EQuake benchmark. Figure A.3 presents the Jmeint benchmark.

```
fun(T, x, y, invKiloK) {
 invKiloK = invKiloK / 5;

 // chromaticity x coefficients for T <= 4000K
 A_x00 = -0.2661239;
 A_x01 = -0.2343580;
 A_x02 = 0.8776956;
 A_x03 = 0.179910;

 // chromaticity x coefficients for T > 4000K
 A_x10 = -3.0258469;
 A_x11 = 2.1070379;
 A_x12 = 0.2226347;
 A_x13 = 0.24039;

 // chromaticity y coefficients for T <= 2222K
 A_y00 = -1.1063814;
 A_y01 = -1.34811020;
 A_y02 = 2.18555832;
 A_y03 = -0.20219683;

 // chromaticity y coefficients for 2222K < T <= 4000K
 A_y10 = -0.9549476;
 A_y11 = -1.37418593;
 A_y12 = 2.09137015;
 A_y13 = -0.16748867;

 // chromaticity y coefficients for T > 4000K
 A_y20 = 3.0817580;
 A_y21 = -5.87338670;
 A_y22 = 3.75112997;
 A_y23 = -0.37001483;

...
```

```
...

 if (T < .4000) {
  xc = A_x00*invKiloK*invKiloK*invKiloK +
  A_x01*invKiloK*invKiloK +
  A_x02*invKiloK +
  A_x03;
 } else {
 xc = A_x10*invKiloK*invKiloK*invKiloK +
  A_x11*invKiloK*invKiloK +
  A_x12*invKiloK +
  A_x13;
}


if (T < .2222) {
 yc = A_y00*xc*xc*xc +
  A_y01*xc*xc +
  A_y02*xc +
  A_y03;
 } else {
  if (T < .4000) {
   yc = A_y10*xc*xc*xc +
    A_y11*xc*xc +
    A_y12*xc +
    A_y13;
  } else {
   yc = A_y20*xc*xc*xc +
    A_y21*xc*xc +
    A_y22*xc +
    A_y23;
  }
 }

 x = xc;
 y = yc;

 return x, y;
}
```

Figure A.1: Camera benchmark, which performs a part of the conversion from blackbody radiator color temperature to the CIE 1931 x,y chromaticity approximation function.

```
fun(t, disptminus, dispt, disptplus, M, C, V, M23, C23, V23) {
 t0 = 0.6;
 dt = 0.0024;

 disptminus = disptminus * dt * dt;
 dispt = dispt * dt * dt;

 if (t > 0.5) {
  t = t * 1.2;

  phi0 = 1;
  phi1 = 0;
  phi2 = 0;
 } else {
  t = t * 0.6;

  phi0 = 0.5 / pi * (0.0 + t / t0 - sin(0.0 + t / t0));
  phi1 = (1.0 - cos(0.0 + t / t0)) / t0;
  phi2 = 2.0 * pi / t0 / t0 * sin(0.0 + t / t0);
 }

 disptplus = disptplus * -dt * dt
      + 2.0 * M * dispt
      - (M - dt / 2 * C) * disptminus
      - dt * dt * (M23 * phi2 / 2
             + C23 * phi1 / 2
             + V23 * phi0 / 2);

 disptplus = disptplus / dt / dt;

 return disptplus;
}
```

Figure A.2: EQuake benchmark, which computes the displacement of an object after one timestep in an earthquake simulation.

```
fun(v00, v01, v02, v10, v11, v12, v20,
   ↪  v21, v22, u00, u01, u02, u10, u11, u12, u20, u21, u22) {
 e1[3]; e2[3]; n1[3]; n2[3]; d[3];
 isect1[2];
 isect2[2];

 // Compute plane equation of triangle (v0,v1,v2)
 e1[0] = v10 - v00;
 e1[1] = v11 - v01;
 e1[2] = v12 - v02;
 e2[0] = v20 - v00;
 e2[1] = v21 - v01;
 e2[2] = v22 - v02;

 // Cross product: n1 = e1 x e2
 n1[0] = (e1[1] * e2[2]) - (e1[2] * e2[1]);
 n1[1] = (e1[2] * e2[0]) - (e1[0] * e2[2]);
 n1[2] = (e1[0] * e2[1]) - (e1[1] * e2[0]);

 // Plane equation 1: n1.X + d1 = 0
 d1 = -(n1[0] * v00 + n1[1] * v01 + n1[2] * v02);

 // Put u0,u1,u2 into plane
    ↪  equation 1 to compute signed distances to the plane
 du0 = (n1[0] * u00 + n1[1] * u01 + n1[2] * u02) + d1;
 du1 = (n1[0] * u10 + n1[1] * u11 + n1[2] * u12) + d1;
 du2 = (n1[0] * u20 + n1[1] * u21 + n1[2] * u22) + d1;

 du0du1 = du0 * du1;
 du0du2 = du0 * du2;

 // Compute plane equation of triangle (u0,u1,u2)
 e1[0] = u10 - u00;
 e1[1] = u11 - u01;
 e1[2] = u12 - u02;
 e2[0] = u20 - u00;
 e2[1] = u21 - u01;
 e2[2] = u22 - u02;

 // Cross product: n2 = e1 x e2
 n2[0] = (e1[1] * e2[2]) - (e1[2] * e2[1]);
 n2[1] = (e1[2] * e2[0]) - (e1[0] * e2[2]);
 n2[2] = (e1[0] * e2[1]) - (e1[1] * e2[0]);
 ...
```

```
...
// Plane equation 2: n2.X + d2 = 0
d2 = -(n2[0] * u00 + n2[1] * u01 + n2[2] * u02);

// Put v0,v1,v2 into plane
   ↪  equation 2 to compute signed distances to the plane
dv0 = (n2[0] * v00 + n2[1] * v01 + n2[2] * v02) + d2;
dv1 = (n2[0] * v10 + n2[1] * v11 + n2[2] * v12) + d2;
dv2 = (n2[0] * v20 + n2[1] * v21 + n2[2] * v22) + d2;

dv0dv1 = dv0 * dv1;
dv0dv2 = dv0 * dv2;

d[0] = (n1[1] * n2[2]) - (n1[2] * n2[1]);
d[1] = (n1[2] * n2[0]) - (n1[0] * n2[2]);
d[2] = (n1[0] * n2[1]) - (n1[1] * n2[0]);

// Compute and index to the largest component of d
index = 0;

if (d[0] > 0) { max = d[0]; } else { max = -d[0]; }
if (d[0] > 0) { bb = d[0]; } else { bb = -d[0]; }
if (d[0] > 0) { cc = d[0]; } else { cc = -d[0]; }

if (bb > max) { max = bb; index = 1; } else { skip; }

if (cc > max) {
 max = cc;
 vp0 = v02;
 vp1 = v12;
 vp2 = v22;
 up0 = u02;
 up1 = u12;
 up2 = u22;
} else {
 if (index > 0) {
  vp0 = v01;
  vp1 = v11;
  vp2 = v21;
  up0 = u01;
  up1 = u11;
  up2 = u21;
} else {
...
```

```
 ...
   vp0 = v00;
   vp1 = v10;
   vp2 = v20;
   up0 = u00;
   up1 = u10;
   up2 = u20;
  }
}

 vv0 = vp0; vv1 = vp1; vv2 = vp2; d0 = dv0; d1 = dv1;
    ↪ d2 = dv2; d0d1 = dv0dv1; d0d2 = dv0dv2; abc[3]; x0x1[2];
 is_coplanar_1 = 0;

 if (d0d1 > 0.0) {
  // d0d2 <= 0 --> i.e. d0, d1
     ↪  are on the same side, d2 on the other or on the plane
  abc[0] = vv2;
  abc[1] = (vv0 - vv2) * d2;
  abc[2] = (vv1 - vv2) * d2;
  x0x1[0] = d2 - d0;
  x0x1[1] = d2 - d1;
 } else { if (d0d2 > 0.0) {
  // d0d1 <= 0
  abc[0] = vv1;
  abc[1] = (vv0 - vv1) * d1;
  abc[2] = (vv2 - vv1) * d1;
  x0x1[0] = d1 - d0;
  x0x1[1] = d1 - d2;
 } else { if (d1 * d2 > 0.0 ) { // || d0 != 0.0f
  // d0d1 <= 0 or d0 != 0
  abc[0] = vv0;
  abc[1] = (vv1 - vv0) * d0;
  abc[2] = (vv2 - vv0) * d0;
  x0x1[0] = d0 - d1;
  x0x1[1] = d0 - d2;
 } else { if (d1 > 0.0) {
  abc[0] = vv1;
  abc[1] = (vv0 - vv1) * d1;
  abc[2] = (vv2 - vv1) * d1;
  x0x1[0] = d1 - d0;
  x0x1[1] = d1 - d2;
 } else { if (d2 > 0.0) {
 ...
```

```
...
 abc[0] = vv2;
 abc[1] = (vv0 - vv2) * d2;
 abc[2] = (vv1 - vv2) * d2;
 x0x1[0] = d2 - d0;
 x0x1[1] = d2 - d1;
} else { is_coplanar_1 = 1; }}}}}

vv0 = up0; vv1 = up1; vv2 = up2; d0 = du0; d1 = du1;
   ↪ d2 = du2; d0d1 = du0du1; d0d2 = du0du2; def[3]; y0y1[2];
is_coplanar_2 = 0;

if (d0d1 > 0.0) {
 // d0d2 <= 0 --> i.e. d0, d1
    ↪  are on the same side, d2 on the other or on the plane
 def[0] = vv2;
 def[1] = (vv0 - vv2) * d2;
 def[2] = (vv1 - vv2) * d2;
 y0y1[0] = d2 - d0;
 y0y1[1] = d2 - d1;
} else { if (d0d2 > 0.0) {
 // d0d1 <= 0
 def[0] = vv1;
 def[1] = (vv0 - vv1) * d1;
 def[2] = (vv2 - vv1) * d1;
 y0y1[0] = d1 - d0;
 y0y1[1] = d1 - d2;
} else { if (d1 * d2 > 0.0 ) { // || d0 != 0.0f
 // d0d1 <= 0 or d0 != 0
 def[0] = vv0;
 def[1] = (vv1 - vv0) * d0;
 def[2] = (vv2 - vv0) * d0;
 y0y1[0] = d0 - d1;
 y0y1[1] = d0 - d2;
} else { if (d1 > 0.0) {
 def[0] = vv1;
 def[1] = (vv0 - vv1) * d1;
 def[2] = (vv2 - vv1) * d1;
 y0y1[0] = d1 - d0;
 y0y1[1] = d1 - d2;
} else { if (d2 > 0.0) {
...
```

```
...
 def[0] = vv2;
 def[1] = (vv0 - vv2) * d2;
 def[2] = (vv1 - vv2) * d2;
 y0y1[0] = d2 - d0;
 y0y1[1] = d2 - d1;
} else { is_coplanar_1 = 1; }}}}}

return abc
   ↪ [0], abc[1], abc[2], x0x1[0], x0x1[1], def[0], def[1],
   ↪  def[2], y0y1[0], y0y1[1], is_coplanar_1, is_coplanar_2;
```

Figure A.3: Jmeint benchmark, which calculates whether two 3D triangles intersect, and several auxiliary variables related to their intersection.

References

Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. The frankencamera: An experimental platform for computational photography. In *ACM SIGGRAPH*, 2010.

Atish Agarwala, Abhimanyu Das, Brendan Juba, Rina Panigrahy, Vatsal Sharan, Xin Wang, and Qiuyi Zhang. One network fits all? modular versus monolithic task formulations in neural networks. In *International Conference on Learning Representations*, 2021.

A. Akram and L. Sawalha. Validation of the gem5 simulator for x86 architectures. In *IEEE International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2019.

Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *USENIX Conference on Networked Systems Design and Implementation*, 2017.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2019.

Zack Ankner, Alex Renda, and Michael Carbin. Renamer: A transformer architecture invariant to variable renaming. In Preparation, 2023.

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, 2014.

Sanjeev Arora, Simon Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. In *International Conference on Machine Learning*, 2019.

Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E. Hassan, Juergen Dingel, and James R. Cordy. Analyzing a decade of linux system calls. *Empirical Software Engineering*, 23(3), 2018.

Tao Bao, Yunhui Zheng, and Xiangyu Zhang. White box sampling in uncertain data processing enabled by program analysis. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

E. Barnard and L.F.A. Wessels. Extrapolation and interpolation in neural network classifiers. *IEEE Control Systems Magazine*, 12(5):50–53, 1992.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *International Conference on Neural Information Processing Systems*, 2018.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32), 2019.

Jonathan Berant and Percy Liang. Semantic parsing via paraphrasing. In *Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2014.

Federico Bianchi, Debora Nozza, and Dirk Hovy. Language invariant properties in natural language processing. In *NLP Power! The First Workshop on Efficient Benchmarking in NLP*, 2022.

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2), 2011.

C. Bischof, P. Khademi, A. Mauer, and A. Carle. Adifor 2.0: automatic differentiation of fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3):18–32, 1996.

Valerio Biscione and Jeffrey S. Bowers. Convolutional neural networks are not invariant to translation, but they can learn to be. *Journal of Machine Learning Research*, 22(1), 2021.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, 2009.

Ethan Bommarito and M. Bommarito. An empirical analysis of the python package index (pypi). *Software Engineering eJournal*, 2019.

Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2015.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

Michael Carbin, Sasa Misailovic, and Martin Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2013.

Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. Machine learning and the physical sciences. *Reviews of Modern Physics*, 91, Dec 2019.

Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018.

Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondrej Sykora, Saman Amarasinghe, and Michael Carbin. BHive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *IEEE International Symposium on Workload Characterization*, 2019.

Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. Renderman: An advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics*, 37(3), 2018.

Alexandra Chronopoulou, Christos Baziotis, and Alexandros Potamianos. An embarrassingly simple approach for transfer learning from pretrained language models. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *International Joint Conference on Artificial Intelligence*, 2011.

Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International Conference on Machine Learning*, 2016.

Quentin Colombet. [patch][x86][haswell][schedmodel] add exceptions for instructions that diverge from the generic model, 2014. URL https://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20140811/230499.html.

Corinna Cortes, Giulia DeSalvo, Claudio Gentile, Mehryar Mohri, and Ningshan Zhang. Region-based active learning. In *International Conference on Artificial Intelligence and Statistics*, 2019.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

Shabnam Daghaghi, Nicholas Meisburger, Mengnan Zhao, Yong Wu, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. Accelerating SLIDE deep learning on modern cpus: Vectorization, quantizations, memory optimizations, and more. In *Conference on Machine Learning and Systems*, 2021.

Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. Tensorflow lite micro: Embedded machine learning for tinyml systems. In *Conference on Machine Learning and Systems*, 2021.

Alexandre Decan, Tom Mens, and Maelick Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *European Conference on Software Architecture Workshops*, 2016.

Seyyed Mehdi Dehaghani and Neda Hajrahimi. Which factors affect software projects maintenance cost more? *Acta Informatica Medica*, 21(1):63–66, 2013.

Franck Dernoncourt, Ji Young Lee, Ozlem Uzuner, and Peter Szolovits. De-identification of patient notes with recurrent neural networks. *Journal of the American Medical Informatics Association*, 24(3), 2016.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

Andrea Di Biagio. [llvm-dev] [llvm-mca] resource consumption of procresgroups. http://lists.llvm.org/pipermail/llvm-dev/2020-May/141486.html, 2020.

Andrea Di Biagio and Matt Davis. llvm-mca, 2018. URL https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html.

Yi Ding, Ahsan Pervaiz, Michael Carbin, and Henry Hoffmann. Generalizable and interpretable learning for configuration extrapolation. In *ACM Joint Meeting on European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021.

Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient sampling of SAT solutions for testing. In *International Conference on Software Engineering*, 2018.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *IEEE/ACM International Symposium on Microarchitecture*, 2012.

Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. A survey of data augmentation approaches for NLP. In *Findings of the Association for Computational Linguistics*, 2021.

Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Technical report, Technical University of Denmark, 1996.

Gottlob Frege. On sense and reference. *Zeitschrift für Philosophie und philosophische Kritik*, 100:25–50, 1892. Translated from: "Über Sinn und Bedeutung".

Fabian Fuchs, Daniel Worrall, Volker Fischer, and Max Welling. Se(3)-transformers: 3d roto-translation equivariant attention networks. In *Advances in Neural Information Processing Systems*, 2020.

Andrew Gelman and Jennifer Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge University Press, 2006.

Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. In *IEEE International Conference on Data Science and Advanced Analytics*, 2018.

Jesus M. Gonzalez-Barahona, G. Robles, Martin Michlmayr, Juan José Amor, and D. Germán. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14, 2008.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Uncertainty in Artificial Intelligence*, 2008.

Robert B. Gramacy. *Surrogates: Gaussian Process Modeling, Design and Optimization for the Applied Sciences*. Chapman Hall/CRC, 2020.

Will Grathwohl, Dami Choi, Yuhuai Wu, Geoff Roeder, and David Duvenaud. Backpropagation through the void: Optimizing control variates for black-box gradient estimation. In *International Conference on Learning Representations*, 2018.

Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2nd edition, 2008.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2), 2017.

Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor N. Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel C. Paver. Sources of error in full-system simulation. *IEEE International Symposium on Performance Analysis of Systems and Software*, 2014.

Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., 2010.

Song Han. *Efficient Methods and Hardware for Deep Learning*. PhD thesis, Stanford University, 2017.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture*, 2016a.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations*, 2016b.

Jussi Hanhirova, Teemu Kämäräinen, Sipi Seppälä, Matti Siekkinen, Vesa Hirvisalo, and Antti Ylä-Jääski. Latency and throughput characterization of convolutional neural networks for mobile computer vision. In *ACM Multimedia Systems Conference*, 2018.

Milad Hashemi, Kevin Swersky, J. A. Smith, G. Ayers, H. Litz, Jichuan Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In *International Conference on Machine Learning*, 2018.

Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture*, 2018.

John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7), 2000.

Michiel Hermans and Benjamin Schrauwen. Training and analysing deep recurrent neural networks. In *Advances in Neural Information Processing Systems*. 2013.

Jason Hicken, Juan Alonso, and Charbel Farhat. Lecture notes in aa222 - introduction to multidisciplinary design optimization, 2020. URL http://adl.stanford.edu/aa222/Lecture_Notes_files/chapter6_gradfree.pdf.

Tin Kam Ho. Random decision forests. In *International Conference on Document Analysis and Recognition*, 1995.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), 1997.

Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *Programming Languages and Systems, European Symposium on Programming*, 2010.

Fu Jie Huang and Yann LeCun. Large-scale learning with svm and convolutional nets for generic object categorization. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2006.

Ling Huang, Jinzhu Jia, Bin Yu, Byung gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in Neural Information Processing Systems*. 2010.

Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. In *Advances in Neural Information Processing Systems*, 2019.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madan Musuvathi, and Jianfeng Gao. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems*, 2022.

Intel. Intel architecture code analyzer, 2017. URL https://software.intel.com/en-us/articles/intel-architecture-code-analyzer.

Engin İpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

Alistair E. W. Johnson, Lucas Bulgarelli, and Tom J. Pollard. Deidentification of free-text medical records using pre-trained bidirectional transformers. In *ACM Conference on Health, Inference, and Learning*, 2020.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture*, 2017.

Bongsoon Kang, Ohak Moon, Changhee Hong, Honam Lee, Bonghwan Cho, and Youngsun Kim. Design of advanced color: Temperature control system for HDTV applications. *Journal of the Korean Physical Society*, 41(6), 2002.

Andrej Karpathy. Software 2.0, November 2017. URL https://medium.com/@karpathy/software-2-0-a64152b37c35.

Mine Kaya and Shima Hajimirza. Using a novel transfer learning method for designing thin film solar cells with enhanced quantum efficiencies. *Scientific Reports*, 9(5034), 2019.

Makena Kelly. Unemployment checks are being held up by a coding language almost nobody knows. *The Verge*, April 2020.

Nicolas Keriven and Gabriel Peyré. Universal invariant and equivariant graph neural networks. In *Advances in Neural Information Processing Systems*, 2019.

James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

Diederick P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

Bogdan Kustowski, Jim A. Gaffney, Brian K. Spears, Gemma J. Anderson, Jayaraman J. Thiagarajan, and Rushil Anirudh. Transfer learning as a tool for reducing simulation bias: Application to inertial confinement fusion. *IEEE Transactions on Plasma Science*, 48(1): 46–53, 2020.

Jihye Kwon and Luca P. Carloni. Transfer learning for design-space exploration with high-level synthesis. In *ACM/IEEE Workshop on Machine Learning for CAD*, 2020.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2020.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, 2004.

J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2018.

Averill M. Law. *Simulation Modeling and Analysis*. McGraw-Hill, 5th edition, 2015.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun and Yoshua Bengio. *Convolutional networks for images, speech, and time-series*. MIT Press, 1995.

Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

B. C. Lee and D. M. Brooks. Illustrative design space studies with microarchitectural regression models. In *IEEE International Symposium on High Performance Computer Architecture*, 2007.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, 2019.

Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *International Symposium on Computer Architecture*, 2010.

M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1980.

David Lettier. 3d game shaders for beginners, 2019. URL https://github.com/lettier/3d-game-shaders-for-beginners.

Da Li, Xinbo Chen, Michela Becchi, and Ziliang Zong. Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus. In *IEEE International Conferences on Big Data and Cloud Computing, Social Computing and Networking, Sustainable Computing and Communications*, 2016.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Bo Liu. Anonymized BERT: An augmentation approach to the gendered pronoun resolution challenge. In *Workshop on Gender Bias in Natural Language Processing*, 2019.

Zengjian Liu, Buzhou Tang, Xiaolong Wang, and Qingcai Chen. De-identification of clinical notes via recurrent neural network and conditional random field. *Journal of Biomedical Informatics*, 75:S34–S42, 2017.

Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

Victor Magron, George Constantinides, and Alastair Donaldson. Certified roundoff error bounds using semidefinite programming. *ACM Transaction on Mathematical Software*, 43(4), 2017.

Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, A. Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin A. Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

Alana Marzoev, Samuel Madden, M. Frans Kaashoek, Michael J. Cafarella, and Jacob Andreas. Unnatural language processing: Bridging the gap between synthetic and natural language data. *arXiv preprint arXiv:2004.13645*, 2020.

Henry Massalin. Superoptimizer: A look at the smallest program. In *International Conference on Architectual Support for Programming Languages and Operating Systems*, 1987.

Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation - Advances in Research and Theory*, 24:109–165, January 1989.

Patrick McGeehan. He needs jobless benefits. he was told to find a fax machine. *The New York Times*, April 2020.

Charith Mendis. *Towards Automated Construction of Compiler Optimizations.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2020.

Charith Mendis and Saman Amarasinghe. GoSLP: Globally optimized superword level parallelism framework. 2018.

Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, 2019a.

Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. Compiler auto-vectorization with imitation learning. In *Advances in Neural Information Processing Systems*, 2019b.

Joshua R. Minot, Nicholas Cheney, Marc Maier, Danne C. Elbers, Christopher M. Danforth, and Peter Sheridan Dodds. Interpretable bias mitigation for textual data: Reducing genderization in patient notes while maintaining classification performance. *ACM Transactions on Computing for Healthcare*, 2022.

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2014.

Andreas Munk, Adam Ścibior, Atılım Güneş Baydin, Andrew Stewart, Goran Fernlund, Anoush Poursartip, and Frank Wood. Deep probabilistic surrogate networks for universal simulator approximation. In *Conference on Uncertainty in Artificial Intelligence*, 2022.

R.H. Myers, D.C. Montgomery, and C.M. Anderson-Cook. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley, 2009.

Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. Hypermapper: a practical design space exploration framework. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2019.

Radford M. Neal. Probabilistic inference using markov chain monte carlo methods. Technical report, University of Toronto, 1993.

Behnam Neyshabur. Towards learning convolutions from scratch. In *Advances in Neural Information Processing Systems*, 2020.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

Maxwell Nye, Anders Andreassen, Guy Gur-Ari, Henryk Witold Michalewski, Jacob Austin, David Bieber, David Martin Dohan, Aitor Lewkowycz, Maarten Paul Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations*, 2024.

ONNX Runtime developers. Onnx runtime. https://www.onnxruntime.ai, 2021. Version: 1.7.0.

OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A full system simulator for multicore x86 cpus. In *ACM/EDAC/IEEE Design Automation Conference*, 2011.

David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *ACM SIGOPS Symposium on Operating Systems Principles*, 2009.

Raphaël Pestourie, Youssef Mroueh, Thanh V. Nguyen, Payel Das, and Steven G. Johnson. Active learning of deep surrogates for PDEs: application to metasurface design. *npj Computational Materials*, 6(164), 2020.

André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics.* Springer, 1st edition, 2010. ISBN 3642145086.

Angela Pohl, Biagio Cosenza, and Ben Juurlink. Vectorization cost modeling for NEON, AVX and SVE. *Performance Evaluation*, 140-141, 2020.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

Nestor V. Queipo, Raphael T. Haftka, Wei Shyy, Tushar Goel, Rajkumar Vaidyanathan, and P. Kevin Tucker. Surrogate-based analysis and optimization. *Progress in Aerospace Sciences*, 41(1), 2005.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning).* The MIT Press, 2005. ISBN 026218253X.

Roger Ratcliff. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97 2:285–308, 1990.

Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin. Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates. In *IEEE/ACM International Symposium on Microarchitecture*, 2020.

Alex Renda, Yi Ding, and Michael Carbin. Programming with neural surrogates of programs. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2021.

Alex Renda, Yi Ding, and Michael Carbin. Turaco: Data sampling for training neural surrogates of programs. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2023.

Fabian Ritter and Sebastian Hack. PMEvo: Portable inference of port mappings for out-of-order processors by evolutionary optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3), 1951.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A primer in BERTology: What we know about how BERT works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020.

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *International Symposium on Computer Architecture*, 2013.

T. J. Santner, Williams B., and Notz W. *The Design and Analysis of Computer Experiments, Second Edition*. Springer-Verlag, 2018.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *European Software Engineering Conference / ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

Joan Serrà, Dídac Surís, Marius Miron, and Alexandros Karatzoglou. Overcoming catastrophic forgetting with hard attention to the task. In *International Conference on Machine Learning*, 2018.

Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

Dongdong She, Kexin Pei, D. Epstein, J. Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program smoothing. In *IEEE Symposium on Security and Privacy*, 2019.

Sergey Shirobokov, Vladislav Belavin, Michael Kagan, Andrey Ustyuzhanin, and Atılım Güneş Baydin. Black-box optimization with local generative surrogates. In *Advances in Neural Information Processing Systems*, 2020.

Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6:60, 2019.

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, 2011.

Noah Snavely. Lecture notes: Cs5670: Computer vision – lecture 5: Feature invariance, 2019. URL https://www.cs.cornell.edu/courses/cs5670/2019sp/lectures/lec05_invariance.pdf.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *International Symposium on Formal Methods*, 2018.

Phillip Stanley-Marbell, Armin Alaghi, Michael Carbin, Eva Darulova, Lara Dolecek, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Thierry Moreau, Mattia Cacciotti, Alexandros Daglis, Natalie Enright Jerger, Babak Falsafi, Sasa Misailovic, Adrian Sampson, and Damien Zufferey. Exploiting errors for efficiency: A survey from circuits to applications. *ACM Computing Surveys*, 53(3), 2020.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

Gang Sun and Shuyue Wang. A review of the artificial neural network surrogate modeling in aerodynamic design. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 233(16):5863–5872, 2019.

Jennifer J. Sun, Megan Tjandrasuwita, Atharva Sehgal, Armando Solar-Lezama, Swarat Chaudhuri, Yisong Yue, and Omar Costilla Reyes. Neurosymbolic programming for science. In *NeurIPS 2022 AI for Science: Progress and Promises*, 2022.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2018.

Ondrej Sykora, Clement Courbet, Guillaume Chatelet, and Nicolas Paglieri. EXEgesis Project, 2018. URL https://github.com/google/EXEgesis.

Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, 2019.

Natalya Tatarchuk. Advances in real-time rendering in 3d graphics and games i. In *ACM SIGGRAPH 2009 Courses*, 2009.

Hasan Tercan, Alexandro Guajardo, Julian Heinisch, Thomas Thiele, Christian Hopmann, and Tobias Meisen. Transfer-learning: Bridging the gap between real and simulation data for machine learning in injection molding. *CIRP Conference on Manufacturing Systems*, 72, 2018.

Steven K. Thompson. *Stratified Sampling*, chapter 11, pages 139–156. John Wiley & Sons, 2012.

George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. In *Computer Vision and Pattern Recognition*, 2017.

Craig Topper. [patch] d73844: [x86] update the haswell and broadwell scheduler information for gather instructions, 2014. URL https://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20200127/738394.html.

Craig Topper. Patch] d44644: [x86] use silvermont cost model overrides for goldmont as well, 2018. URL http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20180319/537000.html.

Lloyd N. Trefethen. *Approximation Theory and Approximation Practice (Other Titles in Applied Mathematics)*. Society for Industrial and Applied Mathematics, USA, 2012.

Ethan Tseng, Felix Yu, Yuting Yang, Fahim Mannan, Karl St. Arnaud, Derek Nowrouzezahrai, Jean-François Lalonde, and Felix Heide. Hyperparameter optimization in black-box image processing using differentiable proxies. In *ACM SIGGRAPH*, 2019.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962v2*, 2019.

United States General Accounting Office. Federal agencies' maintenance of computer programs: Expensive and undermanaged. Report to the Congress AFMD-81-25, United States General Accounting Office, February 1981. URL https://www.gao.gov/assets/afmd-81-25.pdf.

Gregor Urban, Krzysztof J. Geras, Samira Ebrahimi Kahou, Özlem Aslan, Shengjie Wang, Abdelrahman Mohamed, Matthai Philipose, Matthew Richardson, and Rich Caruana. Do deep convolutional nets really need to be deep and convolutional? In *International Conference on Learning Representations*, 2017.

Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27, 1984.

Vladimir Vapnik and Alexey Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2), 1971.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, 2018.

Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Advances in Neural Information Processing Systems*, 2019.

Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. Cachequery: Learning replacement policies from hardware caches. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

Rui Wang, Robin Walters, and Rose Yu. Data augmentation vs. equivariant networks: A theory of generalization on dynamics forecasting. In *ICML Workshop on Principles of Distribution Shift*, 2022.

Peter A. G. Watson. Applying machine learning to improve simulations of a chaotic dynamical system using empirical error correction. *Journal of Advances in Modeling Earth Systems*, 11(5):1402–1417, 2019.

Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *IEEE International Symposium on Workload Characterization*, 2008.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, 2009.

R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, August 1964.

Ulme Wennberg and Gustav Eje Henter. The case for translation-invariant self-attention in transformer-based language models. *arXiv preprint arXiv:2106.01950*, 2021.

Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers:

State-of-the-art natural language processing. In *Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020.

Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at facebook: Understanding inference at the edge. In *IEEE International Symposium on High Performance Computer Architecture*, 2019.

Keyulu Xu, Mozhi Zhang, Jingling Li, Simon Shaolei Du, Ken-Ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *International Conference on Learning Representations*, 2021.

Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2015.

Stephen Yablo. Meaning & reference. MIT OpenCourseWare, 2011. URL https://ocw.mit.edu/courses/24-251-introduction-to-philosophy-of-language-fall-2011/febcb9ac20bf0740c70a14323b9cc657_MIT24_251F11_lec02.pdf. Lecture notes for 24.251 Introduction to Philosophy of Language, Fall 2011.

Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test*, 34(2):60–68, 2017.

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems*, 2014.

M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IEEE International Symposium on Performance Analysis of Systems Software*, 2007.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*, 2020.

Hattie Zhou, Azade Nova, Hugo Larochelle, Aaron Courville, Behnam Neyshabur, and Hanie Sedghi. Teaching algorithmic reasoning via in-context learning. *arXiv preprint arXiv:2211.09066*, 2022.