Neural Surrogates of Programs

Anonymous Author(s)

Abstract

Neural surrogates, neural networks that are trained to compute the same function as a given program, provide alternative representations of classical programs with different properties that are useful for a variety of programming tasks. With neural compilation, programmers develop a neural surrogate that replicates the behavior of the original program to deploy to end-users in place of the original program. With neural adaptation, programmers first develop a neural surrogate of a program then continue to train the neural surrogate on a different task. With neural surrogate optimization, programmers develop a neural surrogate of the original program, optimize input parameters of that neural surrogate using gradient descent, then plug the optimized input parameters back into the original program. Compared to standard programming approaches on these tasks, neural surrogates are more efficient and result in higher accuracy.

However, the approaches in the literature for developing neural surrogates are disparate. We identify the *neural surrogate programming methodology* common to these approaches, consisting of the *specification* of the problem, the *design* of the neural network architecture, the *training* of the neural network, and the *deployment* considerations of the system.

Keywords: keyword1, keyword2, keyword3

1 Introduction

There is an emerging body of work around developing *neural surrogates* of programs, neural networks that are trained to mimic the behavior of a program [19, 33, 49, 52, 59, 62]. Neural surrogates are used for a variety of tasks including accelerating computational kernels in numerical programs [19], replacing physical simulators wholesale with more accurate versions [59], and tuning parameters of complex simulators [49, 62]. Compared to standard programming approaches, neural surrogates require lower development and execution costs and result in higher accuracy.

Neural surrogate applications. We first classify neural surrogate uses into three classes of application: *neural compilation*, *neural adaptation*, and *neural surrogate optimization*.

With neural compilation, programmers develop a neural 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 neural surrogate on different hardware and to bound or accelerate the execution time of the neural surrogate [19, 41].

Onward! '21, October 17–22, 2021, Chicago, IL, USA
2021.

With neural adaptation, programmers first develop a neural surrogate of a program then continue to train the neural surrogate on a different task. The key benefit of this approach is that neural adaptation makes it possible to alter the semantics of the program to achieve a task it is otherwise unable to perform [59, 64].

With neural surrogate optimization, programmers develop a neural surrogate of the original program, optimize input parameters of that neural surrogate using gradient descent, then plug the optimized input parameters back into the original program. Key benefits of this approach include that it can optimize input parameters of programs that operate over discrete inputs or are discontinuous, and that it does not require the original program to be written in a differentiable programming language [49, 52, 62].

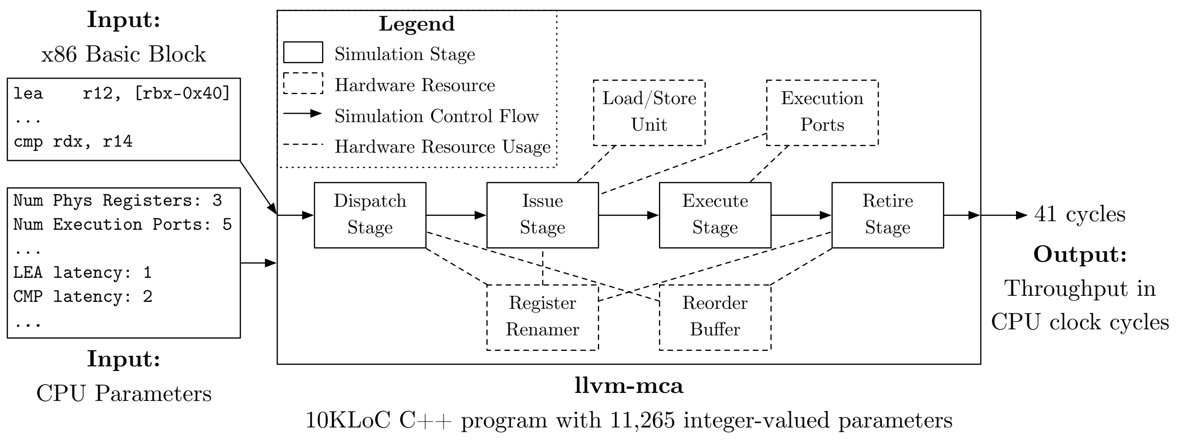
Neural surrogate programming methodology. The development methodologies common to these applications induce what we term the *neural surrogate programming methodology*, consisting of the *specification* of the neural surrogate optimization problem, the *design* of the neural network architecture, the *training* process for the neural network, and the *deployment* considerations of the system.

We focus specifically on two types of questions about the neural surrogate programming methodology. We address questions that arise from the fact we study neural surrogates *of programs* (e.g., selecting a neural surrogate architecture that can represent the original program with high accuracy). We also address questions that arise from the fact that neural surrogate development is itself a form of *programming*, constructing a function to meet a correctness specification while trading off among other objectives (e.g., minimizing execution costs while satisfying an accuracy constraint).

We present the programming methodology as a catalog of specifications of the end-to-end behavior of neural surrogates, followed by a set of design questions and answers to those questions that guide the design, training, and deployment process of the neural surrogate.

Contributions.

- We identify and define three classes of application of neural surrogates of programs: neural compilation, neural adaptation, and neural surrogate optimization. We provide detailed case studies of each of these classes of application.
- We identify elements of the neural surrogate programming methodology in the form of specifications and design questions that unify the three classes of application. We discuss answers to each of these design



```
lea    r12, [rbx-0x40]
...
cmp rdx, r14
```

```
Num Phys Registers: 3
Num Execution Ports: 5
...
LEA latency: 1
CMP latency: 2
...
```

Figure 1. Input-output specification and design of llvm-mca.

questions, showing the trade-offs that programmers must consider when developing neural surrogates.
- We lay out future directions towards the goal of further systematizing the development methodology of neural surrogates.

Neural surrogates are an important emerging frontier of programming with a wealth of potential use cases. By identifying the different classes of application and describing the programming methodology used to develop neural surrogates, we shed light on a set of tools for writing and reasoning about the behavior of complex programs that execute in complex or uncertain environments.

2 Example

We first demonstrate how developing neural surrogates of an x86 CPU simulator makes it possible to solve three programming tasks: (1) replacing the simulator with a neural surrogate that has lower average execution time than the simulator, (2) simulating the execution behavior of a real-world processor that is not well-modeled by the simulator itself, and (3) optimizing the microachitectural parameters of the simulator to produce better parameters than the original, expert-provided parameters.

Program under study. Following Renda et al. [49] we study llvm-mca [18], an x86 basic block throughput estimator included in the LLVM compiler infrastructure [37].

Figure 1 presents llvm-mca's input-output specification and design. llvm-mca takes as input an *x86 basic block*, a sequence of x86 assembly instructions with no jumps or loops, and a set of *CPU parameters*, integers that describe physical properties of the x86 CPU being modeled. llvm-mca outputs a prediction of the *throughput* of the basic block on the CPU, a prediction of number of CPU clock cycles taken to execute the block when repeated for a fixed number of iterations. LLVM contains default expert-set CPU parameter settings for llvm-mca that target common x86 hardware architectures.

llvm-mca does not emulate the precise behavior of the CPU under study. Instead, llvm-mca makes several modeling assumptions about the behavior of the CPU, and simulates basic blocks using an abstract execution model of the CPU.

llvm-mca simulates a processor in four main stages: *dispatch*, *issue*, *execute*, and *retire*. Instructions pass through each these four stages in turn. When all instructions of the basic block have passed through the simulation pipeline, the simulation terminates and the final throughput prediction is the number of simulated CPU clock cycles.

Instructions first enter into the *dispatch* stage. The dispatch stage reserves physical resources like registers and slots in the reorder buffer for the instruction in the abstract execution model of the CPU. The specification of what resources are available and which resources to reserve are set by the CPU parameters that are input into llvm-mca.

Once dispatched, instructions wait in the *issue* stage until they are ready to be executed. The issue stage blocks an instruction until its input operands are available and until all of its required execution ports, another class of resources specified by the CPU parameters, are available.

Instructions then enter the *execute* stage, which reserves the instruction's execution ports and holds them for the duration specified by the CPU parameters for the instruction.

Finally, once an instruction has executed for its duration, it enters the *retire* stage, which frees the physical resources that were acquired for each instruction in the dispatch phase.

llvm-mca implementation. llvm-mca is a C++ program implemented as part of the LLVM compiler infrastructure, comprised of around 10,000 lines of code. llvm-mca's CPU parameters are comprised of 11,265 integer-valued parameters, inducing a configuration space with $10^{19,336}$ possible configurations. These parameters must be set for each different physical CPU architecture that llvm-mca targets.

Notation	Type	Definition
x	$\mathcal{X}$	Program inputs.
y	$\mathcal{Y}$	Program outputs.
f	$\mathcal{X} \to \mathcal{Y}$	Original program.
$\hat{f}$	$\mathcal{X} \to \mathcal{Y}$	Neural surrogate.
err	$\mathcal{Y} \times \mathcal{Y} \to \mathcal{R}$	Error metric when comparing neural surrogate outputs to program outputs.
tc	$\forall \mathcal{T}. \mathcal{T} \to \mathcal{R}$	Cost to train the given object.
ec	$(\mathcal{X} \to \mathcal{Y}) \to \mathcal{R}$	Cost to execute the given function.
obj	$\mathcal{R}^n \to \mathcal{R}$	Task-specific objective function to be minimized in the neural surrogate optimization problem.
con	$\mathcal{R}^n \to \mathcal{R}^m$	Task-specific constraint function to be satisfied in the neural surrogate optimization problem.
ℓ	$\mathcal{Y} \to \mathcal{R}$	Objective function on neural surrogate outputs that is minimized in neural adaptation and neural surrogate optimization.

Table 1. Notation used in formalism for Sections 2 to 5.

$$\hat{f}^* = \underset{\hat{f}}{\arg\min}\; obj\left(\begin{matrix}\frac{1}{|X|}\sum_{x \in \mathcal{X}} err\left(\hat{f}(x), f(x)\right), \\ ec\left(\hat{f}\right)\end{matrix}\right) \text{ such that } con\left(\begin{matrix}\frac{1}{|X|}\sum_{x \in \mathcal{X}} err\left(\hat{f}(x), f(x)\right), \\ tc\left(\hat{f}\right), \\ ec\left(\hat{f}\right)\end{matrix}\right) \leq 0$$

Figure 2. Optimization problem for neural compilation, which learns a neural surrogate $\hat{f}^*$ of the original program f.

llvm-mca validation and accuracy. llvm-mca implements an approximate simulation which makes simplifying assumptions about the real-world system that it models, ultimately leading to approximate predictions of the ground-truth throughput of basic blocks. Chen et al. [12] validate the accuracy of llvm-mca by collecting BHive, a dataset of x86 basic blocks from a variety of end-user programs, and collecting ground-truth throughput measurements for each basic block in BHive by timing them on real CPUs. To calculate llvm-mca's accuracy, Chen et al. define a error metric *err* over llvm-mca's throughput predictions. This error metric is the absolute percentage error, which is the normalized difference between llvm-mca's predicted throughput y_{pred} and the ground-truth measured throughput y_{true}:

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

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%.

Neural surrogate programming methodology. In Sections 3 to 5 we demonstrate the solution to each of the three programming tasks on llvm-mca introduced above. The set of solutions to these three programming tasks induces what we term the *neural surrogate programming methodology*, consisting of the *specification*, *design*, *training*, and *deployment* steps for developing the neural surrogate. We demonstrate the neural surrogate programming methodology on each of these programming tasks in turn.

3 Neural Compilation

With neural compilation, programmers develop a neural 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 neural surrogate on different hardware and to bound or accelerate the execution time of the neural surrogate [19, 41].

Figure 2 presents the optimization problem that defines neural compilation, using the notation developed in Table 1. Neural compilation is an optimization problem that minimizes a task-dependent objective function *obj* of the average error *err* between the outputs of the neural surrogate $\hat{f}$ and the original program f over the training dataset $\mathcal{X}$ (i.e., the error of the neural surrogate) and the execution cost *ec* of the neural surrogate, subject to a task-dependent constraint function *con* of the average error *err* between the outputs of the neural surrogate $\hat{f}$ and the original program f over the training dataset $\mathcal{X}$, the training cost *tc* of the neural surrogate, and the execution cost *ec* of the neural surrogate.

3.1 Case Study

We evaluate the *execution throughput* of llvm-mca, the CPU simulator described in Section 2, when llvm-mca is instantiated with LLVM's default Haswell CPU parameters. We define the execution throughput as the number of basic blocks per second that llvm-mca is able to generate throughput predictions for when given the binary representing the assembled basic block, generating one prediction at a time on a single CPU. llvm-mca's execution throughput on an Intel Xeon Haswell CPU at 2.3GHz is 176 blocks per second.[1]

Approaches in the literature for accelerating llvm-mca's execution throughput include rewriting the simulation software to be faster [25], and applying compiler optimizations not included in llvm-mca's default compiler's optimization set like superoptimization [40, 50].

An alternative approach for accelerating llvm-mca is neural compilation. With neural compilation a programmer develops a neural surrogate to match the outputs of the original program while accelerating the execution throughput of the program. The neural surrogate is a neural network that takes as input a basic block and predicts the throughput of that basic block in simulated CPU clock cycles (i.e., the result of llvm-mca on that basic block when instantiated with LLVM's default Haswell CPU parameters).

Using neural compilation we learn a neural surrogate of llvm-mca that has an execution throughput of 222 blocks per second, a speedup of 1.23× over llvm-mca's execution throughput of 176 blocks per second. This neural surrogate has a mean absolute percentage error (MAPE) of 8.7% compared to llvm-mca's predictions, within a standard error rate of less than 10% for approximate applications [55]. Against BHive's ground-truth measured data on a real Haswell CPU, the neural surrogate has an error rate of 25.1%, compared to an error rate of 25.0% for llvm-mca.

3.2 Programming Methodology

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

3.2.1 Specification. The primary concern with any programming task is its specification. In the neural surrogate programming methodology, the specification comes in the form of specific instantiations of the objective function *obj*, the constraint function *con*, the error function *err*, and the training cost and execution cost functions *tc* and *ec* in the neural surrogate's optimization problem.

For this neural compilation example, the specification for the neural surrogate is to minimize the execution throughput of the neural surrogate while also constraining the error of the neural surrogate compared to llvm-mca to be less than 10% as measured by the MAPE of the neural surrogate:

$$\hat{f}^* = \underset{\hat{f}}{\arg\max}\ \text{execution-throughput}(\hat{f})$$

$$\text{such that } \frac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} \frac{\left|\hat{f}(x) - f(x, \text{haswell-params})\right|}{f(x, \text{haswell-params})} \leq 10\%$$

[1]Full methodological details on this evaluation are presented in Appendix A.

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

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

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

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

The neural network architecture topology is the connection pattern of the neurons in the neural network [22]. The topology determines the types of inputs that can be processed (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.

Following Renda et al. [49], we use the neural network architecture proposed by Mendis et al. [42], which is a hierarchical pair of *LSTMs* [30], which are a neural network topology that process sequences of inputs and output an *embedding* for the sequence, a fixed-length vector of real numbers that represents the sequence. In Mendis et al.'s hierarchical LSTM design, the bottom LSTM processes each instruction (e.g., opcode and operands) independently to generate an embedding per-instruction. The top LSTM operates over the sequence of instruction embeddings generated by the lower-level LSTM to generate an embedding for the overall basic block. The final embedding is then dot-producted with a learned vector to produce a final basic block throughput prediction. Mendis et al. [42] validate that this architecture learns to accurately predict the throughput of basic blocks on physical Intel CPUs, a similar problem to learning to predict the throughput predicted by llvm-mca.

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

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

We search among candidate capacities of the neural surrogate to find the smallest-capacity network that meets the accuracy specification. To control the capacity we consider using different sizes for the embeddings generated by the LSTMs; a larger embedding width gives the LSTM higher capacity. We search among widths of the LSTMs in the neural surrogate that are powers of 2 between 1 and 256. We find that a width of 32 leads to the fastest-to-execute network that achieves a MAPE of less than 10% error.

3.2.3 Training.

In addition to designing the neural surrogate's architecture, the programmer must determine how to train the neural surrogate.

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

The training data distribution is the distribution of inputs on which the neural surrogate is expected to perform well.

We use basic blocks from the BHive dataset [12] to train the neural surrogate.

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

The *loss function*, the objective in a neural network's optimization process, is a differentiable, continuous relaxation of the error metric *err* from the specification (which may not necessarily be differentiable), with different relaxations having different properties [8, pp. 337–338].

The loss function for training the neural surrogate for neural compilation is the MAPE of the neural surrogate's prediction of llvm-mca's prediction of throughput, as specified in the specification.

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

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

In this neural compilation example minimizing or constraining training time is not a part of the specification, so the neural surrogate is trained until convergence (when the loss function stops decreasing). On the neural surrogate with width of 32, this occurs after 22 passes over the training set.

3.2.4 Deployment.

Once the neural 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 neural surrogate: whether the downstream task requires low-latency or high-throughput execution, whether the neural 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 neural surrogate.

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

For fairness of comparison with llvm-mca the neural surrogate is deployed on identical hardware to llvm-mca, which in this case is a single Intel Xeon Haswell CPU at 2.3GHz.

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

The neural surrogate requires preprocessing of the input assembled basic block binary, which is performed in C. To execute the neural surrogate we use the ONNX runtime [17], a runtime environment that accelerates neural network execution while also being portable across devices and programming languages.

3.3 Other Examples of Neural Compilation

Compiling to different hardware. Esmaeilzadeh et al. [19] learn neural surrogates of small computational kernels, then deploy the neural surrogates on a custom-designed hardware accelerator which reduces the latency and energy cost of executing the neural surrogate relative to the original program. More generally, neural surrogates can be deployed on any hardware that supports executing neural networks, resulting in different trade-offs compared to the CPU architectures that many programs execute on.

Different algorithmic complexity. Algorithmic complexity can differ between a neural surrogate and its corresponding original program: for example, while a classical algorithm may require an exponential number of operations in the size of the input, a neural surrogate may require linear or even a constant number of operations to approximate the original program to satisfactory accuracy [35, 43].

4 Neural Adaptation

With neural adaptation, programmers first develop a neural surrogate of a program then continue to train the neural surrogate on a different task. The key benefit of this approach is that neural adaptation makes it possible to alter the semantics of the program to achieve a task it is otherwise unable to perform [59, 64].

Figure 3 shows the two optimization problems that define neural adaptation, using the notation developed in Table 1.

The first optimization problem finds a neural surrogate $\hat{f}_1$ that minimizes a task-dependent objective function *obj* of the average error *err* between the outputs of the neural surrogate $\hat{f}$ and the original program f over the training dataset $\mathcal{X}$ (i.e., the error of the neural surrogate), subject to a task-dependent constraint function *con* of the training cost *tc* of the neural surrogate and the execution cost *ec* of the neural surrogate.

The second optimization problem finds a neural surrogate that a different task-dependent objective function *obj′* of the average value of a downstream objective ℓ of the outputs of the neural surrogate over the training dataset $\mathcal{X}$ (i.e., an error metric beyond just mimicking the original program) and the average error *err* between the outputs of the neural surrogate $\hat{f}$ and the neural surrogate $\hat{f}_1$ from the first optimization problem over the training dataset $\mathcal{X}$, subject to a task-dependent constraint function *con′* of the average value of a downstream objective ℓ of the outputs of the neural surrogate over the training dataset $\mathcal{X}$, the average error *err* between the outputs of the neural surrogate $\hat{f}$ and the neural surrogate from the first optimization problem $\hat{f}_1$ over the training dataset $\mathcal{X}$, the training cost *tc* of the neural surrogate, and the execution cost *ec* of the neural surrogate. The second optimization problem is seeded with the neural surrogate $\hat{f}_1$ resulting from the first.

$$\hat{f}_1 = \underset{\hat{f}}{\arg\min}\ obj\left(\tfrac{1}{|\mathcal{X}|}\textstyle\sum_{x\in\mathcal{X}} err\left(\hat{f}(x), f(x)\right)\right) \text{ such that } con\begin{pmatrix} tc\left(\hat{f}\right), \\ ec\left(\hat{f}\right) \end{pmatrix} \le 0$$

$$\hat{f}^* = \underset{\hat{f}}{\arg\min}\ obj'\begin{pmatrix} \tfrac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} \ell\left(\hat{f}(x)\right), \\ \tfrac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} err\left(\hat{f}(x), \hat{f}_1(x)\right) \end{pmatrix} \text{ such that } con'\begin{pmatrix} \tfrac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} \ell\left(\hat{f}(x)\right), \\ \tfrac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} err\left(\hat{f}(x), \hat{f}_1(x)\right), \\ tc\left(\hat{f}\right), \\ ec\left(\hat{f}\right) \end{pmatrix} \le 0$$

Figure 3. Optimization problems for neural adaptation, which first learns a neural surrogate $\hat{f}_1$ of the original program f then re-trains that neural surrogate to find another neural surrogate $\hat{f}^*$ with higher accuracy against a downstream objective ℓ.

4.1 Case Study

When instantiated with LLVM's default Haswell CPU parameters, llvm-mca has a mean absolute percentage error (MAPE) of 25.0% when predicting the ground-truth throughput for the basic blocks in the BHive dataset that are measured on a Haswell processor [12]. This error is in part due to simplifying modeling assumptions that llvm-mca makes that do not accurately reflect real-world CPUs.

To accurately model behaviors observed in real-world processors, a programmer must design and implement a model of that behavior and tune it to match the observed behavior of the processor. The programmer could also train a machine learning model from scratch based on the observed behavior of the processor.

An alternative approach for increasing the accuracy of llvm-mca is neural adaptation. With neural adaptation a programmer first develops a neural surrogate to match the outputs of the original program. The programmer then further trains the neural surrogate on observed ground-truth data, adapting it to the observed ground-truth behavior regardless of the set of behaviors modeled by the original program.

Figure 4 presents the MAPE error rate of several approaches against ground-truth throughputs, as a function of the size of the training dataset of the approach. llvm-mca's error rate (orange dashed line) is not a function of the amount of ground-truth training data available, and is constant at 25.0%. Neural adaptation's error rate (blue dots) is upper bounded by llvm-mca's, as neural adaptation is first trained to mimic llvm-mca, then decreases with the amount of training data available. In contrast, training a neural network from scratch (green dots) results in a large error rate when trained with a small number of examples, only matching neural adaptation when it is trained on the entire BHive training data set.

These results suggest that neural 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 the ground-truth data is expensive), but provides no benefit when ground-truth data is plentiful.

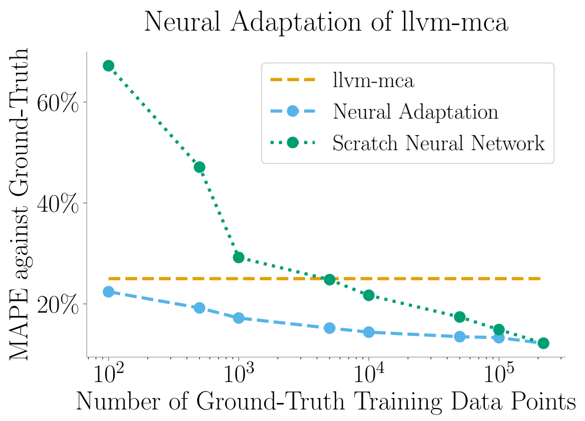


Figure 4. Error on ground-truth data of llvm-mca (orange), neural adaptation of llvm-mca (blue), and a neural network trained from scratch (green).

4.2 Programming Methodology

Developing the neural surrogate for neural adaptation requires detailed thinking about the specification of both optimization problems, the design of the neural network architecture, the training process for the neural network, and the deployment considerations of the system.

4.2.1 Specification. In the first optimization problem for neural adaptation, finding a neural surrogate that mimics llvm-mca, we find a neural surrogate that minimizes the error against llvm-mca without any other constraints on the neural surrogate:

$$\hat{f}_1 = \underset{\hat{f}}{\arg\min}\ \frac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} \frac{\left|\hat{f}(x) - f(x, \text{haswell-params})\right|}{f(x, \text{haswell-params})}$$

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

In the second optimization problem, we optimize for predictive accuracy on the ground-truth data, while minimizing the change from the intermediate neural surrogate $\hat{f}_1$ by constraining the training time (i.e., the number of steps of gradient descent) to be less than 5 epochs (passes over the training set):

$$\hat{f}^* = \arg\min_{\hat{f}} \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \frac{\left|\hat{f}(x) - g(x)\right|}{g(x)}$$
$$\text{such that training-epochs}(\hat{f}) \leq 5$$

where $\mathcal{X}$ is the dataset of basic blocks x from BHive, g is the ground-truth measured timing of the basic block on a Haswell CPU from BHive, and the optimization problem is seeded with $\hat{f} = \hat{f}_1$ and proceeds for 5 training epochs.

4.2.2 Design. In this neural adaptation example, the neural network architecture and capacity must maximize accuracy. There are no other objectives or constraints on the neural surrogate design.

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

Following Renda et al. [49], we use the hierarchical LSTM architecture proposed by Mendis et al. [42].

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

We find that a width of 32 neurons for the LSTM results in sufficient accuracy when mimicking the original program llvm-mca and when re-training on the ground-truth data.

4.2.3 Training. In addition to designing the neural surrogate's architecture, the programmer must determine how to train the neural surrogate.

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

We use the BHive dataset [12] as the set of basic blocks used to train the neural surrogate. In the first optimization problem, the throughputs to predict are llvm-mca's predictions on these basic blocks. In the second optimization problem, the throughputs to predict are the ground-truth measured timings on a Haswell CPU from BHive.

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

The loss function for training the neural surrogate in both optimization problems of this neural adaptation example is the MAPE as specified in the specification.

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

In the first optimization problem of neural adaptation, minimizing or constraining training time is not a part of the specification, so the neural surrogate is trained until convergence (when the loss function stops decreasing) after 22 epochs. In the second optimization problem of neural adaptation, the specification dictates that the neural surrogate is trained for no more than 5 epochs.

4.2.4 Deployment. Once the neural surrogate has been designed and trained, it is deployed for its downstream task.

The specification for this neural adaptation example does not specify objectives or constraints on the deployment for the neural surrogate resulting from neural adaptation.

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

The neural surrogate is executed on a GPU, which provides sufficient throughput (over 2000 training examples per second) when training the neural surrogate for each optimization problem.

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

The neural surrogate is trained and deployed in PyTorch [46], which automatically calculates the gradient of the neural surrogate when training it in both optimization problems.

4.3 Other Examples of Neural Adaptation

Neural adaptation for the physical sciences. Tercan et al. [59] train neural surrogates of computer simulations of plastic injection molding, then adapt the neural surrogates on results from real-world experiments of the injection molding process to close the gap between results predicted by simulation and those of the physical process.

Computer simulation of physical processes reduces the cost of running physical experiments in terms of time, money, or resource availability [14]. For instance, in the plastic injection molding case, computer simulation of the injection molding dynamics can help select machine parameters such as pressure or temperature without having to run physical trials. However, computer simulation is inaccurate, due to simplifying or inaccurate assumptions, limited computational budgets, or other discrepancies between the simulation and physical environment.

Tercan et al. evaluate the effectiveness of neural adaptation for neural surrogates as a method of improving handwritten simulators, showing that it leads to a neural surrogate that is more accurate than the original simulator, while also requiring less training data than a neural network trained from scratch.

Neural adaptation in other optimization problems. Neural adaptation can be useful even when the resulting neural surrogate is not deployed in place of the original program. Verma et al. [64] use neural adaptation to learn neural surrogates as an intermediate artifact in learning a programmatic reinforcement learning policy [58]. She et al. [52] develop a neural surrogate-based fuzzing tool to predict whether or not branches in a program are taken, then incrementally adapt the neural surrogate as the training dataset changes when She et al.'s fuzzing technique discovers new paths through the program.

$$\hat{f}_1 = \underset{\hat{f}}{\arg\min}\; obj\left(\tfrac{1}{|\mathcal{X}|}\textstyle\sum_{x\in\mathcal{X}} err\left(\hat{f}(x), f(x)\right)\right) \text{ such that } con\begin{pmatrix} tc\left(\hat{f}\right), \\ ec\left(\hat{f}\right)\end{pmatrix} \leq 0$$

$$x^* = \underset{x}{\arg\min}\; \ell\left(\hat{f}_1(x)\right) \text{ such that } con'\left(tc(x)\right) \leq 0$$

Figure 5. Optimization problems for neural surrogate optimization, which first learns a neural surrogate $\hat{f}_1$ of the original program f then optimizes inputs x of the neural surrogate to minimize an objective function ℓ on the neural surrogate's output.

$$\forall x \in \mathcal{X}.\left(\exists\epsilon.\forall x' \in \mathcal{X}.\, d(x,x') \leq \epsilon \Rightarrow \ell\left(\hat{f}(x)\right)\right) \leq \ell\left(\hat{f}(x')\right) \Rightarrow \left(\exists\epsilon.\forall x' \in \mathcal{X}.\, d(x,x') \leq \epsilon \Rightarrow \ell\left(f(x)\right) \leq \ell\left(f(x')\right)\right)$$

Figure 6. The property that makes a neural surrogate a good candidate for neural surrogate optimization: local minima of inputs to the neural surrogate are also local minima of the original program against the downstream objective function ℓ.

5 Neural Surrogate Optimization

With neural surrogate optimization, programmers develop a neural surrogate of the original program, optimize input parameters of that neural surrogate using gradient descent, then plug the optimized input parameters back into the original program. Key benefits of this approach include that it can optimize input parameters of programs that operate over discrete inputs or are discontinuous and that it does not require the original program to be written in a differentiable programming language [49, 52, 53, 62].

Figure 5 presents the two optimization problems that constitute neural surrogate optimization, using the notation developed in Table 1.

The first optimization problem finds a neural surrogate that minimizes a task-dependent objective function *obj* of the average error *err* between the outputs of the neural surrogate $\hat{f}$ and the original program f over the training dataset $\mathcal{X}$ (i.e., the error of the neural surrogate), subject to a task-dependent constraint function *con* that is a function of the training cost *tc* of the neural surrogate and the execution cost *ec* of the neural surrogate.

The second optimization problem finds input parameters x to the neural surrogate $\hat{f}_1$ from the first optimization problem that minimize a downstream objective ℓ of the outputs of the neural surrogate, subject to a task-dependent constraint function *con′* that is a function of the training cost *tc* of the input parameters.

Figure 6 present the property that makes a neural surrogate resulting from the first optimization problem a good candidate for optimizing input parameters in the second optimization problem. The ideal is a neural surrogate such that local minima of inputs to the neural surrogate (as measured by a distance metric d) with respect to the downstream objective ℓ (i.e., inputs x such that all nearby points x' have a higher value of the downstream objective ℓ) are also local minima of the original program with that same objective ℓ.

5.1 Case Study

In this section we study a case study of neural surrogate optimization drawn from Renda et al. [49].

When instantiated with LLVM's default Haswell CPU parameters, llvm-mca has a mean absolute percentage error (MAPE) of 25.0% when predicting the ground-truth throughput for the basic blocks in the BHive dataset that are measured on a Haswell processor [12]. This error is in part due to the difficulties of setting llvm-mca's CPU parameters to values that lead llvm-mca to low prediction error.

llvm-mca's Haswell CPU parameters are comprised of 11,265 integer-valued parameters, inducing a configuration space with $10^{19{,}336}$ possible configurations. These parameters must be set for each CPU architecture that llvm-mca targets.

One class of approaches for setting llvm-mca's parameters is to gather fine-grained measurements of each individual parameter's physical realization in the CPU architecture that llvm-mca targets [2, 20, 49].

Another class of approaches is to gather coarse-grained measurements of entire basic blocks, and optimize llvm-mca's parameters to best fit the timings of the basic blocks. Because llvm-mca is not written in a differentiable programming language [6] and operates over discrete values, it is not possible to calculate its gradient exactly or to optimize its parameters with gradient descent. Due to the size of the parameter space, this is an optimization problem for which gradient-free optimization techniques [3] are intractable.

An alternative approach for optimizing parameters of the program using coarse-grained measurements is neural surrogate optimization. With neural surrogate optimization, a programmer develops a neural surrogate of the program then optimizes input parameters using gradient descent through the neural surrogate to maximize performance on a downstream task. Gradient descent converges faster to local minima than gradient-free optimization with the original program [29].

Using neural surrogate optimization, Renda et al. [49] find parameters from scratch that lead llvm-mca to an error of 23.7% on the ground-truth Haswell basic blocks. In contrast, the expert-tuned default Haswell parameters gathered with fine-grained measurement lead llvm-mca to an error of 25.0%. OpenTuner [3], a gradient-free optimization technique, is not able to find parameters that lead llvm-mca to lower than 100% error when given an equivalent computational budget to neural surrogate optimization.

5.2 Programming Methodology

Developing the neural surrogate for the neural surrogate optimization example requires detailed thinking about the specification of the problem, the design of the neural network architecture, the training process for the neural network, the deployment considerations of the system, and the parameter optimization process with a trained neural surrogate.

5.2.1 Specification. In the first optimization problem for neural surrogate optimization, the objective is to find a neural surrogate that minimizes the error against llvm-mca's behavior for any given input basic block and set of CPU parameters, subject to a limited training time (to compare against other search techniques with bounded budgets):

$$\hat{f}_1 = \underset{\hat{f}}{\arg\min} \sum_{\substack{x_{block} \in \mathcal{X}_{block} \\ x_{params} \in \mathcal{X}_{params}}} \frac{\left|\hat{f}(x_{block}, x_{params}) - f(x_{block}, x_{params})\right|}{f(x_{block}, x_{params})}$$
$$\text{such that training-epochs}(\hat{f}) \leq 60$$

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

In the second optimization problem, the objective is to find input parameters that optimize predictive accuracy against the ground-truth data, again subject to a limited training budget on the inputs:

$$x^*_{params} = \underset{x_{params}}{\arg\min} \sum_{x_{block} \in \mathcal{X}_{block}} \frac{\left|\hat{f}_1(x_{block}, x_{params}) - g(x_{block})\right|}{g(x_{block})}$$
$$\text{such that training-epochs}(x_{params}) \leq 1$$

where $\mathcal{X}_{block}$ is the dataset of basic blocks x_{block} from BHive, g is the ground-truth measured timing of the basic block on a Haswell CPU from BHive, and the parameters x_{params} are trained for no more than 1 epoch.

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

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

Renda et al. [49] use the neural network architecture proposed by Mendis et al. [42]. To incorporate the CPU simulator parameters x_{params}, Renda et al. [49] concatenate relevant parameters that llvm-mca uses to each instruction's generated embedding, to use as input to the second LSTM in the hierarchical LSTM architecture.

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

Renda et al. use a stack of 4 LSTMs in place of each original LSTM in the architecture, each with a width of 256 neurons. Stacking LSTMs increases their capacity, which is appropriate due to the complexity induced by adding the parameters x_{params} as input to the neural surrogate.

5.2.3 Training. The programmer must also determine how to train the neural surrogate.

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

Renda et al. use the BHive dataset [12] to train the neural surrogate, which is designed to validate x86 basic block throughput estimators and contains basic blocks from a variety of end-user programs. In the first optimization problem, the throughputs to predict are llvm-mca's predictions on these basic blocks. In the second optimization problem, the throughputs to predict are the measured timings from BHive.

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

The loss function for training the neural surrogate in both optimization problems of this neural surrogate optimization example is the MAPE of the prediction of llvm-mca's prediction of throughput, as specified in the specification.

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

The specification explicitly limits the first training phase to 60 epochs and the second to 1 epoch.

5.2.4 Deployment. Once the neural surrogate has been designed and trained, it is deployed for its downstream task.

In this neural surrogate optimization example the neural surrogate is used entirely as an intermediate artifact in the gravity optimization process, not being deployed to end-users.

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

The neural surrogate is executed on a GPU, which provides sufficient throughput for optimizing the input parameters.

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

The neural surrogate and the input parameters are trained in PyTorch [46]. Once found, the input parameters x^*_{params} are then plugged back into llvm-mca.

5.3 Other Examples

Extending output domain of programs. She et al. [52] 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 neural surrogate, which then allows She et al. to use gradient descent to find inputs that induce a specific execution trace on the original program.

Extending input domain of programs. Grathwohl et al. [24] use learned neural surrogates to approximate the gradient of black-box or non-differentiable functions, in order to reduce the variance of gradient estimators of random variables. Of particular relevance, Grathwohl et al. learn neural surrogates of functions of discrete variables, which they are then able to use to optimize the values of continuous relaxations of these discrete variables.

6 Specifications

In this section we catalog specifications that are used to define the neural surrogate optimization problems, characterizing the design considerations and trade-offs that must be considered when creating a neural surrogate.

6.1 Error

A neural surrogate must compute a similar function to that computed by its source program. When the neural surrogate is deployed to end-users as in neural compilation and neural adaptation, the error metric for the neural surrogate is that of the domain [19]. When the neural surrogate is used as an intermediate artifact as in neural surrogate optimization, other error metrics may help to learn a neural surrogate that allows for successful downstream optimization [62].

6.2 Training and Execution Cost

Regardless of the intended use case, a neural surrogate must be efficient, not exceeding resource budgets to acquire or use.

Training cost. The training cost of a neural surrogate measures the resources required to develop the neural surrogate. This cost includes the expert time required to design the neural surrogate's network architecture, the cost of training data acquisition, and the resources required to train the neural surrogate on the training data, including any hyperparameter or network architecture search that the neural surrogate programmer performs.

Execution cost. The execution cost of a neural surrogate measures the resources required to execute the neural surrogate for its downstream task. This cost includes the storage requirements for the neural surrogate along with the costs of executing the neural surrogate in its execution environment.

6.3 Local Minima Acceptability

Neural surrogate optimization requires the neural surrogate to submit to gradient descent, and to find inputs that lead to good performance in the original program. Low error against the original program is not sufficient to satisfy these conditions, as the error of the neural surrogate's outputs is not the same as the error of the neural surrogate's gradients. Figure 6 presents the primary criterion, which is that local minima of the neural surrogate are also local minima of the program.

7 Design

In this and the following sections, we detail the design questions driving the neural surrogate programming methodology. We discuss possible answers to each of these design questions, showing the trade-offs that programmers must navigate when developing neural surrogates.

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

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

Domain-agnostic architectures. One design methodology is to use a domain-agnostic architecture for the neural surrogate, a neural network architecture designed independently of the domain of application of the neural surrogate and of the behavior of the program under consideration. A common choice of domain-agnostic architectures for neural surrogates are multilayer perceptrons (MLPs) [33, 52]. While simple to design, such domain-agnostic architectures may have high training costs or low accuracy [45, 63].

Domain-specific architectures. An alternative is to design the neural surrogate architecture specifically based on the target domain and semantics of the program [49, 62]. For instance, Renda et al. [49] use a slight modification of the neural network architecture proposed by Mendis et al. [42], a hierarchical LSTM architecture validated to have state-of-the-art accuracy on the same task as the program for which Renda et al. develop a neural surrogate. This neural surrogate architecture also exploits sparsity in the original program's execution, pre-processing inputs to the neural surrogate to remove aspects of the input that do not affect the original program's output. However, designing such architectures requires significant manual effort and expertise in the domain and in the specific program being modeled.

Question: *How do you scale the neural 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 [56]. Possible approaches include manually selecting the architecture based on expert reasoning about the complexity of the program [49] and automatically searching among possible capacities, selecting the capacity that leads to the optimal trade-offs among the components of the neural surrogate's specification [19].

8 Training

This section describes the training methodologies of neural surrogates in the literature.

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

The training data of the neural surrogate defines the distribution of inputs on which the neural surrogate is expected to perform well. This data must reflect the downstream task for which the neural surrogate is deployed, the behavior of the program that the neural surrogate models, and the architecture selected for the neural surrogate.

Instrumenting the program. One approach to collect training data reflective of the downstream task is to instrument execution of the original program and record observed inputs to the program [12, 19]. This approach is prevalent in neural compilation. An underlying challenge is that it may not be possible to guarantee that the workload in the downstream task is reflective of the training workload, especially when the neural 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 some hand-defined sampling distribution [49, 59, 62].

Neural surrogate and program symmetries. The training data must also reflect the symmetries enforced in the program and the neural 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 neural surrogate to be invariant to that symmetry [54].

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

The *loss function* is the objective in a neural network's optimization process which describes how bad a neural network's prediction is compared to the ground truth. Appropriate selection of the loss function for the neural surrogate is a programming task that should reflect the downstream specification for the neural surrogate (i.e., so that a reduction in the loss function results in a better neural surrogate for the downstream task) while also resulting in a differentiable function that is possible to optimize with gradient descent.

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

Once the training data has been collected and loss function determined, the programmer must train the neural surrogate. This results in a trade-off between accuracy and training cost. One approach is training for a fixed training time, typically determined via experiments on a validation set [19, 49]. Another approach is training until an acceptable accuracy is reached, whether via a plateau of the training loss [62] or via reaching a minimum acceptable accuracy [59].

9 Deployment

Once the neural surrogate has been designed and trained, it must deployed into its execution context. Neural networks can execute on different hardware, use different runtimes, and require different data formats.

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

The hardware that the neural surrogate is deployed on impacts the neural surrogate's execution time properties, efficiency, and available optimization opportunities. When a neural 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 neural surrogate.

GPUs. Modern large-scale deep neural networks can be executed on GPUs [13], which achieve high throughput (the number of inputs that can be processed per unit time) at the cost of high latency (the end-to-end time to process a single input) and energy usage [27].

CPUs. Other neural surrogate approaches deploy the neural surrogate on CPUs [33]. CPUs typically result in lower latency and energy use than GPUs, at the cost of reduced throughput [28, 39] (though recent work challenges some of these assumptions [16]). CPUs are also more widely available than GPUs, including on edge devices [67].

Machine learning accelerators. Esmaeilzadeh et al. [19] design and deploy a custom neural processing unit (NPU) to accelerate neural surrogates with low latency and energy cost, similar to the trade-offs CPUs have relative to GPUs. Other machine learning accelerators offer different trade-offs, such as TPUs increasing throughput even further [34], or the Efficient Inference Engine decreasing energy costs while approximating the neural surrogate [26].

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

Neural networks require specialized software runtime environments, which requires navigating concerns about the implementation of the program using the neural surrogate along with concerns about deployment of the neural surrogate across varying devices. Software execution environments include custom frameworks and runtimes which provide bespoke trade-offs for specific applications [19].

Choice of software environment can also impact availability of the neural surrogate across different hardware devices. Certain software runtimes are only available for certain devices (e.g., CPUs), some devices are only supported by certain software runtimes (e.g., TPUs are only supported by TensorFlow), and some runtimes are specialized for devices with resource constraints (e.g., TensorFlow Lite).

Normalization. *Data normalization*, which involves pre- and post-processing the inputs and outputs to be suitable for neural networks [38], induces complexity into the program that deploys the neural surrogate, with normalization and denormalization requiring additional code when integrating the neural surrogate into the original program's execution context. Data pre- and post-processing bugs in such code are difficult to diagnose, leading to reduced accuracy [51]. Esmaeilzadeh et al. [19] address these issues by integrating the normalization and denormalization steps into the custom hardware (the NPU), eliminating the opportunity for software bugs entirely.

Batching. *Batching*, determining the number of input examples to input to the neural surrogate at a time, induces a trade-off between latency and throughput when generating predictions of the neural surrogate, which various approaches handle in different ways. Parrot [19], a neural compilation approach which deploys the neural surrogate to end-users, focuses entirely on latency and therefore uses a single data item in each batch, sacrificing throughput for decreased latency. DiffTune [49], a neural surrogate optimization approach, has no explicit latency requirements and focuses entirely on throughput, increases throughput by batching large numbers of training examples into single invocations of the neural surrogate.

10 Open Problems

While we have presented a programming methodology that details the questions and trade-offs that must be addressed when developing a neural surrogate, there are still several open problems related to the development and application of neural surrogates. This section details open problems and future work not addressed in this paper.

More mechanization and systematization. We have presented a framework for reasoning about the trade-offs in developing neural surrogates. However, our framework is not mechanized: a programmer of a neural surrogate still must manually navigate the trade-off space. Future work in this domain can mechanize the various aspects of neural surrogate construction, from automating the neural surrogate architecture design process based on the semantics of the original program, to automatically training the neural surrogate based on explicit specifications and objectives over a data distribution, to automatically integrating the neural surrogate into the original program's execution context. While some of these concerns have been addressed in prior work [19], fully mechanizing this process for all types of neural surrogates is an important direction for future work.

Defining the scope of applicability. In Sections 3 to 5 we have shown that neural surrogates provide state-of-the-art solutions to large-scale programming problems. However, we have not precisely characterized what problems neural surrogates are not suitable for. In addition to expanding the scope of applicability of neural surrogates through addressing questions about generalization, robustness, and interpretability, future work can more precisely characterize when neural surrogates are and are not the most appropriate solutions to a given programming task.

Generalization and robustness. Large-scale neural networks struggle to generalize outside of their training dataset [4, 32, 68]. Generalization consists of interpolation and extrapolation; while neural networks interpolate well, they struggle to extrapolate. On the other hand, formal program reasoning techniques can prove properties about the behavior of programs on entire classes of inputs [47]. To address situations where the neural surrogate is expected to extrapolate outside of its training data, such as when predicting data for a different task or when deployed to end-users without explicit guarantees about the input data distribution, neural surrogate programmers must develop new approaches to recognizing and addressing generalization issues.

Interpretability. Neural networks do not generate explanations for predictions [21], leading to difficulties when reasoning about neural surrogates' predictions. Future work can address these issues by developing better interpretability tools for neural networks, or for neural surrogates specifically by better characterizing precisely what interpretability means for a given domain, when it is and is not a relevant concern for system design, and using neural surrogates appropriately in the system context. For example, neural surrogate optimization uses neural surrogates as an intermediate artifact to aid another optimization process, where interpretability is not a concern.

11 Related Work Addressing Similar Tasks

In this section we discuss related work that provides alternative solutions than neural surrogates to the programming tasks studied in this paper.

Function approximation. Neural surrogate programming 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 or neural networks [48, 61]. The conventional wisdom is that compared to other function approximation approaches, neural networks (and therefore neural surrogates) excel at *feature extraction* [31], converting function inputs (including discrete and structured inputs) into real-valued vectors which can then be processed by machine learning algorithms. Neural networks also excel when given a large amount of training data [36]. Other function approximation approaches have different trade-offs relative to neural networks, and may be appropriate in circumstances with significantly limited execution efficiency or data constraints, or when requiring specific bounds on the behavior of the function approximation.

Differentiable programming. Differentiable programming is a set of techniques that calculates the derivatives of programs with respect to their input parameters [6]. In contrast with estimating the program's gradient with neural 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 neural 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 optimization. Differentiable programming also relies on implementing the program in a language amenable to differentiable programming such as Pytorch or TensorFlow [1, 7, 46].

In contrast, neural surrogate optimization approximates the program regardless of the provenance of its original implementation, meaning that points in the original program that were originally non-smooth, discontinuous, or had derivative 0 may not in the neural surrogate, allowing for optimizing the inputs despite challenges posed by the original program [49].

Program smoothing. Chaudhuri and Solar-Lezama [11] 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 neural 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, neural surrogate optimization estimates the gradients of arbitrary programs regardless of the constructs used in the program's implementation.

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 [15, 23]. When fitting parameters of a probabilistic program, such techniques require the program to be explicitly specified as a probabilistic program, then the parameters are optimized using inference techniques like Monte Carlo inference [44] and variational inference [9]. In contrast, when optimizing parameters with neural surrogate optimization the original program can be specified in any form, while the parameters are optimized with stochastic gradient descent.

12 Related Work Addressing Other Tasks

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

Neural surrogates of non-programs. Building neural surrogates of black-box processes beyond just programs is used across a wide variety of domains from computer systems to physical sciences [10, 42, 57]. For example, Mendis et al. [42] train a neural surrogate of the execution behavior of Intel CPUs to predict the execution time of code. This is not an example of a neural surrogate because this is performed with no *a priori* knowledge of the execution behavior of the CPU (in that the precise execution behavior of Intel CPUs is not publicly known). This paper focuses on constructing neural surrogates of programs for which we have an intensional representation of the semantics of the program (e.g., program source code) rather than developing neural 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 [60, 64, 66]. 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. [64] train neural networks that augment programmatic reinforcement learning policies. While learning such residual models is a form of programming, the neural networks are not neural surrogates of programs, and are thus out of scope for this paper.

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 [5, 64, 65]. For example, after training a residual model, Verma et al. [64] 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 paper due to the significant differences in programming methodologies when synthesizing a program that mimics a neural network and developing a neural surrogate that mimics a program.

13 Conclusion

Neural surrogates are an important emerging frontier of programming with a wealth of potential use cases. By identifying the different classes of application and describing the programming methodology used to develop neural surrogates, we aim to provide a set of powerful tools for writing and reasoning about the behavior of complex programs that execute in complex or uncertain environments.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX Symposium on Operating Systems Design and Implementation*.

[2] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O’Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*.

[4] E. Barnard and L.F.A. Wessels. 1992. Extrapolation and interpolation in neural network classifiers. *IEEE Control Systems Magazine* 12, 5 (1992), 50–53.

[5] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *International Conference on Neural Information Processing Systems*.

[6] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research* 18, 153 (2018), 1–43.

[7] C. Bischof, P. Khademi, A. Mauer, and A. Carle. 1996. Adifor 2.0: automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering* 3, 3 (1996), 18–32.

[8] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

[9] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. 2017. Variational Inference: A Review for Statisticians. *J. Amer. Statist. Assoc.* 112, 518 (2017), 859–877.

[10] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. 2019. Machine learning and the physical sciences. *Rev. Mod. Phys.* 91 (Dec 2019), 39 pages. Issue 4.

[11] Swarat Chaudhuri and Armando Solar-Lezama. 2010. Smooth Interpretation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[12] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondrej Sykora, Saman Amarasinghe, and Michael Carbin. 2019. BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. In *IEEE International Symposium on Workload Characterization*.

[13] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. 2011. Flexible, High Performance Convolutional Neural Networks for Image Classification. In *International Joint Conference on Artificial Intelligence*.

[14] Kyle Cranmer, Johann Brehmer, and Gilles Louppe. 2020. The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences* 117, 48 (2020), 30055–30062.

[15] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[16] Shabnam Daghaghi, Nicholas Meisburger, Mengnan Zhao, Yong Wu, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. 2021. Accelerating SLIDE Deep Learning on Modern CPUs: Vectorization, Quantizations, Memory Optimizations, and More. In *Conference on Machine Learning and Systems*.

[17] ONNX Runtime developers. 2021. ONNX Runtime. https://www.onnxruntime.ai. Version: 1.7.0.

[18] Andrea Di Biagio and Matt Davis. 2018. *llvm-mca*. https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html

[19] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *IEEE/ACM International Symposium on Microarchitecture*.

[20] Agner Fog. 1996. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical Report. Technical University of Denmark.

[21] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. 2018. Explaining Explanations: An Overview of Interpretability of Machine Learning. In *IEEE International Conference on Data Science and Advanced Analytics*.

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

[23] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Uncertainty in Artificial Intelligence*.

[24] Will Grathwohl, Dami Choi, Yuhuai Wu, Geoff Roeder, and David Duvenaud. 2018. Backpropagation through the Void: Optimizing control variates for black-box gradient estimation. In *International Conference on Learning Representations*.

[25] Georg Hager and Gerhard Wellein. 2010. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc.

[26] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ACM/IEEE International Symposium on Computer Architecture*.

[27] Jussi Hanhirova, Teemu Kämäräinen, Sipi Seppälä, Matti Siekkinen, Vesa Hirvisalo, and Antti Ylä-Jääski. 2018. Latency and Throughput Characterization of Convolutional Neural Networks for Mobile Computer Vision. In *ACM Multimedia Systems Conference*.

[28] 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. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *IEEE International Symposium on High Performance Computer Architecture*.

[29] Jason Hicken, Juan Alonso, and Charbel Farhat. 2020. Lecture notes in AA222 - Introduction to Multidisciplinary Design Optimization. http://adl.stanford.edu/aa222/Lecture_Notes_files/chapter6_gradfree.pdf

[30] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997).

[31] Fu Jie Huang and Yann LeCun. 2006. Large-scale learning with SVM and convolutional nets for generic object categorization. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.

[32] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. 2019. Adversarial Examples Are Not Bugs, They Are Features. In *Advances in Neural Information Processing Systems*.

[33] Engin İpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 2006. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[34] 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. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture*.

[35] Andrej Karpathy. 2017. Software 2.0. https://medium.com/@karpathy/software-2-0-a64152b37c35

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*.

[37] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*.

[38] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. 2012. *Efficient BackProp*. Springer Berlin Heidelberg, Berlin, Heidelberg, 9–48.

[39] 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. 2010. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *International Symposium on Computer Architecture*.

[40] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *International Conference on Architectual Support for Programming Languages and Operating Systems*.

[41] Charith Mendis. 2020. *Towards Automated Construction of Compiler Optimizations*. Ph.D. Thesis. Massachusetts Institute of Technology, Cambridge, MA.

[42] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *International Conference on Machine Learning*.

[43] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. 2019. Compiler Auto-Vectorization with Imitation Learning. In *Advances in Neural Information Processing Systems*.

[44] Radford M. Neal. 1993. *Probabilistic Inference Using Markov Chain Monte Carlo Methods*. Technical Report. University of Toronto.

[45] Behnam Neyshabur. 2020. Towards Learning Convolutions from Scratch. In *Advances in Neural Information Processing Systems*.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*.

[47] André Platzer. 2010. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics* (1st ed.). Springer.

[48] Carl Edward Rasmussen and Christopher K. I. Williams. 2005. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

[49] Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin. 2020. DiffTune: Optimizing CPU Simulator Parameters with Learned Differentiable Surrogates. In *IEEE/ACM International Symposium on Microarchitecture*.

[50] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[51] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High Interest Credit Card of Technical Debt. In *Software Engineering for Machine Learning (NIPS 2014 Workshop)*.

[52] Dongdong She, Kexin Pei, D. Epstein, J. Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *IEEE Symposium on Security and Privacy*.

[53] Sergey Shirobokov, Vladislav Belavin, Michael Kagan, Andrey Ustyuzhanin, and Atılım Güneş Baydin. 2020. Black-Box Optimization with Local Generative Surrogates. In *Advances in Neural Information Processing Systems*.

[54] Connor Shorten and Taghi M. Khoshgoftaar. 2019. A survey on Image Data Augmentation for Deep Learning. *J. Big Data* 6 (2019), 60.

[55] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Trade-Offs with Loop Perforation. In *ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*.

[56] 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. 2020. Exploiting Errors for Efficiency: A Survey from Circuits to Applications. *ACM Comput. Surv.* 53, 3, Article 51 (June 2020), 39 pages.

[57] Gang Sun and Shuyue Wang. 2019. 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 (2019), 5863–5872.

[58] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. The MIT Press.

[59] Hasan Tercan, Alexandro Guajardo, Julian Heinisch, Thomas Thiele, Christian Hopmann, and Tobias Meisen. 2018. Transfer-Learning: Bridging the Gap between Real and Simulation Data for Machine Learning in Injection Molding. *CIRP Conference on Manufacturing Systems* 72 (2018), 185–190.

[60] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. 2017. Full Resolution Image Compression with Recurrent Neural Networks. In *Computer Vision and Pattern Recognition*.

[61] Lloyd N. Trefethen. 2012. *Approximation Theory and Approximation Practice (Other Titles in Applied Mathematics)*. Society for Industrial and Applied Mathematics, USA.

[62] Ethan Tseng, Felix Yu, Yuting Yang, Fahim Mannan, Karl St. Arnaud, Derek Nowrouzezahrai, Jean-François Lalonde, and Felix Heide. 2019. Hyperparameter Optimization in Black-box Image Processing using Differentiable Proxies. *ACM Transactions on Graphics (Proc. SIGGRAPH)* 38, 4, Article 27 (July 2019).

[63] Gregor Urban, Krzysztof J. Geras, Samira Ebrahimi Kahou, Özlem Aslan, Shengjie Wang, Abdelrahman Mohamed, Matthai Philipose, Matthew Richardson, and Rich Caruana. 2017. Do Deep Convolutional Nets Really Need to be Deep and Convolutional?. In *International Conference on Learning Representations*.

[64] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. 2019. Imitation-Projected Programmatic Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

[65] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. In *International Conference on Machine Learning*.

[66] Peter A. G. Watson. 2019. Applying Machine Learning to Improve Simulations of a Chaotic Dynamical System Using Empirical Error

Correction. *Journal of Advances in Modeling Earth Systems* 11, 5 (2019), 1402–1417.

[67] 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. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *IEEE International Symposium on High Performance Computer Architecture.*

[68] Keyulu Xu, Mozhi Zhang, Jingling Li, Simon Shaolei Du, Ken-Ichi Kawarabayashi, and Stefanie Jegelka. 2021. How Neural Networks Extrapolate: From Feedforward to Graph Neural Networks. In *International Conference on Learning Representations.*

A Methodology for Neural Compilation Experiments

This appendix details the details of the performance evaluation experiments performed in Section 3.

All experiments were performed on a Google Cloud Platform `n1-standard-4` instance, using a single core of an Intel Xeon Haswell CPU at 2.30 GHz.

llvm-mca is compiled in release mode from version 8.0.1, using the version released by Renda et al. [49] at https://github.com/ithemal/DiffTune/tree/9992f69/llvm-mca-parametric commit hash `9992f69`. llvm-mca is invoked a single time and immediately passed a random sample of 5000 disassembled basic blocks from BHive over stdin (with each block having an average of 17.3 instructions) to predict timing values for, with the following command line invocation:

```
llvm-mca -parameters noop -march=x86-64 \
 -mtriple=x86_64-unknown-unknown -mcpu=haswell \
 --all-views=0 --summary-view -iterations=100
```

The reported execution throughput is the time from invocation to exit of the `llvm-mca` command.

The neural surrogate from neural compilation is ran on the same CPU, using a network compiled and loaded with the ONNX runtime, version 1.7.0 [17]. The neural 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 neural network architecture is designed to run with ONNX and uses a batch size of 1. The neural network is invoked repeatedly by a Python script, and is passed the same series of 5000 disassembled and tokenized basic blocks to predict timing values for. The reported execution throughput is the time from the invocation of the Python script to its exit after all predictions have been generated.