Programming with Surrogates of Programs

Alex Renda
MIT CSAIL
Cambridge, MA, USA
renda@csail.mit.edu

Yi Ding
MIT CSAIL
Cambridge, MA, USA
ding1@csail.mit.edu

Michael Carbin
MIT CSAIL
Cambridge, MA, USA
mcarbin@csail.mit.edu

Abstract

Surrogates, models that mimic the behavior of a given program, form the basis of a variety of programming techniques. We define three programming techniques that use surrogates. With *surrogate compilation*, programmers a develop a surrogate that replicates the behavior of a program to deploy to end- users in place of the original program. With *surrogate adaptation*, programmers develop a surrogate of a program then retrain that surrogate on a different task. With *surrogate optimization*, programmers develop a surrogate of a program, optimize input parameters of that surrogate, then plug the optimized input parameters back into the original program.

These techniques share a common programming methodology for developing and deploying the surrogate of the program. Using this programming methodology in a case study of a large-scale CPU simulator, surrogate compilation accelerates the program by 1.6×, surrogate adaptation decreases the simulation error by up to 50%, and surrogate optimization finds simulation parameters that decrease simulation error by 18% compared to expert-set parameters.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; *Software evolution*; • **Computing methodologies** → *Machine learning*.

Keywords: programming languages, machine learning, surrogate models, neural networks

ACM Reference Format:
Alex Renda, Yi Ding, and Michael Carbin. 2021. Programming with Surrogates of Programs. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '21), October 20–22, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3486607.3486748

1 Introduction

Programmers and researchers are increasingly developing *surrogates* of programs, models that approximately mimic a subset of the observable behavior of a given program [22, 39, 66, 72, 80, 83]. Surrogates are constructed from measurements of the behavior of the program on a dataset of input examples [25, 27, 58, 68]. Programmers use surrogates for a variety of tasks including accelerating computational kernels in numerical programs [22], replacing physical simulators with more accurate versions [80], and tuning parameters of complex simulators [66, 83]. Compared to standard programming techniques, programming with surrogates requires lower development costs and results in programs with lower execution cost and higher result quality.

Programming techniques. We first classify uses of surrogates into three programming techniques: *surrogate compilation*, *surrogate adaptation*, and *surrogate optimization*.

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 [22, 54].

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 perform a task of interest [80, 87].

With surrogate optimization, programmers develop a surrogate of the original program, optimize input parameters of that surrogate, then plug the optimized input parameters back into the original program. 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) [66, 72, 83].

Neural surrogates. In this paper we focus specifically on *neural surrogates*, surrogate models instantiated with neural networks. This focus is for three primary reasons. First, neural networks are able to take inputs of arbitrary structure and complexity, from floating point vectors [9, Chapter 5; 39] to strings of tokens [51, 55] to more complex graph structures [56, 93]. Second, for many tasks of interest neural networks are state-of-the-art models that lead to high accuracy [20, 44]. Third, the design and training procedures for neural networks share a common design methodology

Onward! '21, October 20–22, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9110-8/21/10.
https://doi.org/10.1145/3486607.3486748

distinct from that of other surrogate models, which allows for explicitly trading off among competing metrics of interest [25]. Due to these factors, neural surrogates have emerged as a popular surrogate model architecture in the literature [22, 39, 66, 72, 80, 83].

Programming methodology. The development methodologies common to the three surrogate programming techniques instantiated with neural networks induce what we 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 neural network, and the *deployment* considerations of the system.

We focus on two aspects of the development methodology of neural surrogates of programs. We address questions that arise from the fact we study neural 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). We also address questions 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).

We present the programming methodology as a set of design questions that guide the specification, design, training, and deployment process of the neural surrogate.

Case study. Using the neural surrogate programming methodology in a case study of llvm-mca [21], a 10,000 line-of-code CPU simulator, we find that surrogate compilation accelerates the program by 1.6×, surrogate adaptation decreases the simulation error by up to 50%, and surrogate optimization finds simulation parameters that decrease simulation error by 18% compared to expert-set parameters.

Contributions.

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

Surrogates are an important emerging frontier of programming with a wealth of potential use cases for writing and reasoning about complex programs that execute in complex or uncertain environments. By identifying the different classes of application and describing the programming methodology used to develop surrogates, our work provides a taxonomy for reasoning about and developing surrogates. Our work offers a foundation on which the programming languages community can build new tools that aid in the construction and analysis of surrogates of programs.

2 Example

We first demonstrate how developing surrogates of an x86 CPU simulator makes it possible to solve three programming tasks: (1) replacing the simulator with a 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, and (3) optimizing simulation parameters of the simulator to produce better parameters than the original expert-provided parameters. We present Renda et al. [66]'s surrogate optimization case study and further demonstrate two new case studies of surrogate compilation and surrogate adaptation using the same x86 CPU simulator under study.

Program under study. Following Renda et al. [66] we study llvm-mca [21], an x86 basic block throughput estimator included in the LLVM compiler infrastructure [47].

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 CPU being modeled. llvm-mca 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. 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 of 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. llvm-mca's input CPU parameters specify what resources are available on the hardware and which resources to reserve for each instruction.

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

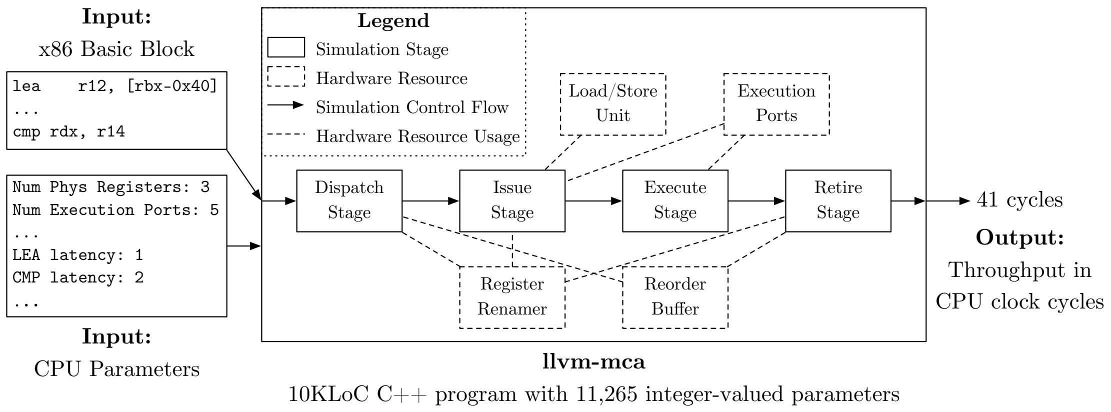

Figure 1. Input-output specification and design of llvm-mca.

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.

llvm-mca validation and accuracy. Chen et al. [13] 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 an 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 output o_{pred} and the ground-truth measured throughput o_{true}:

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

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 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 [22, 54].

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. llvm-mca's execution throughput on an Intel Xeon Skylake CPU at 3.1GHz is 1742 blocks per second.[1]

Approaches in the literature for accelerating llvm-mca's execution throughput include rewriting the simulation software to be faster [29], and applying compiler optimizations not included in llvm-mca's default compiler's optimization set like superoptimization [52, 69].

An alternative approach for accelerating llvm-mca is surrogate compilation. With surrogate compilation a programmer develops a surrogate with faster execution throughput that matches the outputs of the original program. The surrogate is a model 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 surrogate compilation we learn a neural surrogate of llvm-mca that has an execution throughput of 2820 blocks per second on the same hardware, a speedup of 1.6× over llvm-mca's execution throughput of 1742 blocks per second. This surrogate has a mean absolute percentage error (MAPE) of 9.1% compared to llvm-mca's predictions, within a standard error rate of less than 10% for approximate applications [75]. Against BHive's ground-truth measured data on a real Haswell CPU, the surrogate has an error rate of 27.1%, compared to an error rate of 25.0% for llvm-mca.

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

3.2 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. 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 surrogate programming methodology, the specification comes in the form of an optimization problem with an objective and constraints for the surrogate.

The specification for the surrogate in this example is to minimize the execution throughput 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:

$$s^* = \arg\max_s \ \text{execution-throughput}(s)$$
$$\text{such that } \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \frac{|s(i) - p(i, \text{haswell-params})|}{p(i, \text{haswell-params})} \leq 10\%$$

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

The remainder of the 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.

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 surrogate, as well as scale the capacity appropriately to the complexity of the problem. 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 throughput 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 [25]. 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.

We use a BERT encoder [20], a type of Transformer [86], as the neural network topology for surrogate compilation of llvm-mca. Though many architectures could provide an acceptable solution to the task, we select and evaluate the BERT architecture due to its popularity [67], expressive power [96], and relative ease of use [91] for arbitrary sequence modeling tasks (though programmers should in general choose the most appropriate neural network architecture to the model the program depending on the domain). Our 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 the network can represent. Higher capacity neural networks better fit the training data [7], but have higher execution cost [79]. Scaling the capacity involves adding more layers or increasing the width of each layer.

We search among candidate capacities of the surrogate to find the smallest-capacity BERT architecture that meets the accuracy specification. We present more details on this hyperparameter search in Appendix B.1.

3.2.3 Training. In addition to designing the surrogate's architecture, the programmer must determine how to 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 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. We use basic blocks from the BHive dataset [13] to train the surrogate for consistency with the case studies in Sections 4 and 5.

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 from the specification (which may not necessarily be differentiable), with different relaxations having different properties [9, pp. 337–338].

The loss function for training the neural surrogate for this surrogate compilation example is the MAPE of the surrogate's prediction of 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 by either an acceptability threshold on the error, or a fixed training budget. Because the training procedure may be ran multiple times when designing the surrogate, the threshold or budget should be set appropriately to account for the full cost of design and training.

We train the surrogate for 500 passes over the training set (500 epochs), recording the loss over a validation set after each epoch. At the end of training, we select the model with the best validation loss as the final model from training. We present more details on the training in Appendix B.

3.2.4 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 BERT-based surrogate does not require any preprocessing of the input assembly. To execute the surrogate we use the ONNX runtime [19], a runtime environment that accelerates neural network execution while also being portable across devices and programming languages.

3.3 Other Benefits of Surrogate Compilation

Compiling to different hardware. Esmaeilzadeh et al. [22] learn neural 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, 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 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 [41, 56].

4 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 perform a task of interest [80, 87].

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 [13]. 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 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 alternatively train a machine learning model from scratch based on observations of the behavior of the processor.

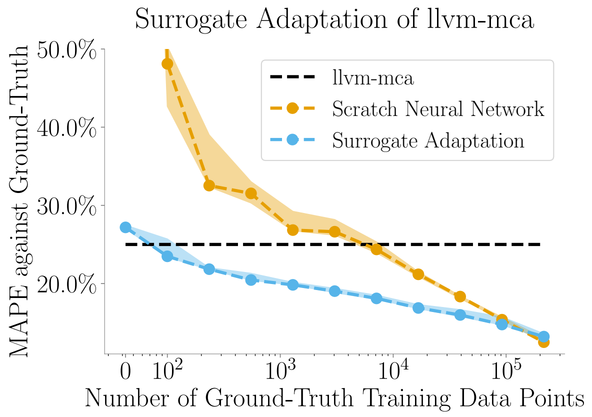

Figure 2. Error on ground-truth data of llvm-mca (black), a neural network trained from scratch (orange), and surrogate adaptation of llvm-mca (blue).

Another approach for increasing the accuracy of llvm-mca is surrogate adaptation. With surrogate adaptation a programmer first develops 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 observed ground-truth behavior regardless of the set of behaviors modeled by the original program.

Figure 2 presents the MAPE of several approaches to accurate CPU modeling against ground-truth throughputs on those CPUs, as a function of the size of the training dataset of the approach. llvm-mca's error rate (black dashed line) is not a function of the amount of ground-truth training data available, and is constant at 25.0%. Surrogate adaptation's error rate (blue dots) is upper bounded by llvm-mca's, as surrogate 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 (orange dots) 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.

4.2 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, and a deployment configuration.

4.2.1 Specification. Surrogate adaptation requires two steps, finding the original surrogate then adapting the surrogate to the downstream task. This is represented as two sequential optimization problems.

In the first optimization problem for this surrogate adaptation example, finding a surrogate that mimics llvm-mca, we find a surrogate that minimizes the error against llvm-mca without any other constraints.

$$s_1^* = \arg\min_s \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \frac{|s(i) - p(i, \text{haswell-params})|}{p(i, \text{haswell-params})}$$

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

In the second optimization problem, we optimize for accuracy on the ground-truth data, while minimizing the change from the intermediate surrogate s_1^*.

$$s^* = \arg\min_s \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \frac{|s(i) - \mathrm{g}(i)|}{\mathrm{g}(i)}$$
$$\text{such that } \left\|s - s_1^*\right\|_1 \leq \Delta$$

where $\mathcal{D}$ is the dataset of basic blocks i from BHive, g is the ground-truth measured timing of the basic block on a Haswell CPU from BHive, and Δ is a threshold constraining the L1 distance in the weights between the s and s_1^* (corresponding to the max number of steps of gradient descent).

4.2.2 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, we use a BERT Transformer architecture. 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, we reuse the capacity for the neural surrogate from Section 3, which has less than 10% error against llvm-mca.

4.2.3 Training.

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

As in Section 3, we 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 throughputs that the surrogate predicts are llvm-mca's predictions on these basic blocks. In the second optimization problem, the throughputs that the surrogate predicts 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 of this surrogate adaptation example is the MAPE, as specified in the specification. In general, the loss functions for the two optimizations problems do not have to be the same, if the surrogate is being adapted to a substantially 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; we therefore reuse the neural surrogate trained in Section 3, which is the minimum-validation-loss surrogate within 500 epochs of training. In the second optimization problem, the specification dictates that the L1 distance between the surrogate resulting from the first optimization problem and the surrogate being optimized in the second is constrained to be less than a constant threshold Δ. We constrain this distance by limiting the training budget for the surrogate. We therefore use the minimum-validation-loss surrogate within 500 epochs of training. The training curves for the models are presented in Appendix B.2.

4.2.4 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 executed on a GPU, which provides sufficient throughput (over 512 training examples per second) to train the surrogate for each optimization problem.

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

The neural surrogate is trained in PyTorch [61], which automatically calculates the gradient of the surrogate for both optimization problems. Since the specification does not impose constraints on the deployment of the surrogate, we also deploy it in PyTorch for simplicity.

4.3 Other Benefits of Surrogate Adaptation

Surrogate adaptation for the physical sciences. Computer simulation of physical processes reduces the cost of running physical experiments in terms of time, money, or resource availability [16]. 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. [80] train neural surrogates of computer simulations of plastic injection molding, then adapt the 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. Tercan et al. show that surrogate adaptation leads to a surrogate that is more accurate than the original simulator, while also requiring less training data than a neural network trained from scratch.

Surrogate adaptation in other optimization problems. Surrogate adaptation can be useful even when the resulting surrogate is not deployed in place of the original program. Verma et al. [87] use surrogate adaptation to learn neural surrogates as an intermediate artifact in learning a programmatic reinforcement learning policy [78]. She et al. [72] develop a neural-surrogate-based fuzzing tool to predict whether or not branches in a program are taken, then incrementally adapt the surrogate as the training dataset changes when She et al.'s fuzzing technique discovers new paths through the program.

5 Surrogate Optimization

With surrogate optimization, programmers develop a surrogate of the original program, optimize input parameters of that surrogate, then plug the optimized input parameters back into the original program. 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) [66, 72, 83].

5.1 Case Study

In this section we study a case study of surrogate optimization drawn from Renda et al. [66].

As noted before, when instantiated with LLVM's default Haswell CPU parameters, llvm-mca has a MAPE of 25.0% against the ground-truth. Though some of this error is attributable to simplifying assumptions in the simulator, the error is also attributable in part 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 measurements of each parameter's realization in the CPU architecture that llvm-mca targets [2, 23, 66].

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 surrogate optimization. With surrogate optimization, a programmer develops a surrogate of the program then optimizes input parameters to maximize performance on a downstream task. A common choice is to optimize input parameters with gradient descent, which is possible if the surrogate model is differentiable regardless of whether or not the original program is differentiable [66, 83]. Gradient descent converges more quickly to local minima than gradient-free optimization with the original program [35].

Using surrogate optimization, Renda et al. find parameters that lead llvm-mca to an error of 23.7% on the Haswell basic blocks. In contrast, the expert-tuned default Haswell parameters gathered with 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 surrogate optimization.

5.2 Programming Methodology

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

5.2.1 Specification. Surrogate optimization requires two steps, finding the original surrogate then optimizing inputs to the surrogate. As in 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 behavior for any given input basic block and set of CPU parameters:

$$s_1^* = \arg\min_{s} \sum_{\substack{i_{\text{block}} \in \mathcal{D}_{\text{block}} \\ i_{\text{params}} \in \mathcal{D}_{\text{params}}}} \frac{\left|s\left(i_{\text{block}}, i_{\text{params}}\right) - p\left(i_{\text{block}}, i_{\text{params}}\right)\right|}{p\left(i_{\text{block}}, i_{\text{params}}\right)}$$

where s is the surrogate, $\mathcal{D}_{\text{block}}$ is the dataset of basic blocks i_{block} from BHive, $\mathcal{D}_{\text{params}}$ is a uniform distribution over parameter values, 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:

$$i^*_{\text{params}} = \arg\min_{i_{\text{params}}} \sum_{i_{\text{block}} \in \mathcal{D}_{\text{block}}} \frac{\left|s^*_1(i_{\text{block}}, i_{\text{params}}) - \mathrm{g}(i_{\text{block}})\right|}{\mathrm{g}(i_{\text{block}})}$$

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

5.2.2 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 the inclusion of i_{params} as input to the surrogate, the BERT architecture presented in Sections 3 and 4 which consumes x86 basic blocks as input is not sufficient for this task. Renda et al. use the neural network architecture proposed by Mendis et al. [55], which is a hierarchical pair of LSTMs [36], one of which processes elements of each instruction independently (i.e., opcode and operands) and the other which processes embeddings generated by each instruction in order. Mendis et al. [55] validate that this architecture learns to predict the throughput of basic blocks on physical Intel CPUs with low error, a similar problem to learning to predict the throughput predicted by llvm-mca. To incorporate the CPU simulator parameters i_{params}, Renda et al. 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 2: *How do you scale the 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 i_{params} as input to the surrogate.

5.2.3 Training.

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

Renda et al. use the BHive dataset [13] to train the 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 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 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?*

Renda et al. 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 training phase.

5.2.4 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 i^*_{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 the input parameters are trained in PyTorch [61], which automatically calculates the gradients for both the surrogate and the input optimization. Again, once found,the input parameters i^*_{params} are then plugged back into llvm-mca.

5.3 Other Benefits of Surrogate Optimization

Extending output domain of programs. She et al. [72] 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.

Extending input domain of programs. Grathwohl et al. [28] use 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 Surrogate Programming Formalism

This section formalizes the definition of each of the surrogate programming techniques. 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 form an algorithm sketch for each surrogate programming technique.

6.1 Preliminaries

Let $p \in \mathcal{P}$ denote a program under study. Let $\omega \in \mathcal{P} \to \mathcal{I} \to \mathcal{O}$ denote an *interpreter*, which takes the program p and an input $i \in \mathcal{I}$ and produces an output $o \in \mathcal{O}$. 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 [90, Chapter 5]. Other interpreters may output other observations of the execution of the program, such as its execution time, memory usage, the control flow trace, or any other aspect of its denotational or operational semantics. The program is measured on a dataset of inputs $\mathcal{D} \subseteq \mathcal{I}$.

Let $s \in \mathcal{P}$ denote a surrogate of the program. The ideal is a surrogate such that for a given interpretation ω of the original program, the standard interpretation ω^* of the surrogate has the same output for all inputs:

$$\forall i \in \mathcal{D}.\, \omega(p)(i) = \omega^*(s)(i)$$

Let obj and con denote task-specific objective and constraint functions for the optimization problems. These functions allow the generic versions of the optimization problems to be instantiated for a specific task. Together with the choice of interpreters for the program and the surrogate, the task-specific objective and constraints select for what criteria to consider (e.g., error, execution cost, etc.) and how to weigh these criteria. Section 7 discusses criteria considered by examples of surrogate programming from the literature.

6.2 Surrogate Construction

The first step of each surrogate programming technique is to train a surrogate of the original program. Figure 3 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 obj of an interpretation ω_p of the original program p, a different interpretation ω_s of the surrogate s, and a training dataset $\mathcal{D}$, subject to a task-dependent constraint function con of the same terms.

6.3 Surrogate Compilation

In surrogate compilation, the programmer simply deploys the surrogate found in the surrogate construction step in Section 6.2 to the end-user: $s^* = s_1^*$.

6.4 Surrogate Adaptation

The first step of surrogate adaptation is the initial surrogate construction step of Section 6.2. The second step is to continue to train the surrogate to optimize a different objective. Figure 4 shows the generic optimization problem that defines the second step of surrogate adaptation.

This second optimization problem finds a surrogate s^* that minimizes a different task-dependent objective function obj′ of a different interpretation ω'_p of the original program p, a different interpretation ω'_s of the surrogate s, a different training dataset $\mathcal{D}'$, and the standard interpretation ω^* of the surrogate s_1^* resulting from the first optimization problem, subject to a different task-dependent constraint function con′ of the same terms.

6.5 Surrogate Optimization

The first step of surrogate optimization is the initial surrogate construction step of Section 6.2. The second step is to optimize inputs to the surrogate to optimize a different objective. Figure 5 shows the generic optimization problem that defines the second step of surrogate optimization.

This second optimization problem finds an input i^* to the surrogate that minimizes a different task-dependent objective function obj′ of a different interpretation ω'_p of the original program p, the standard interpretation ω^* of the surrogate s_1^* from the first optimization problem, a different training dataset $\mathcal{D}'$, and the inputs i, subject to a different task-dependent constraint function con′ of the same terms.

Figure 6 present the properties that make a surrogate resulting from the first optimization problem a good candidate for optimizing input parameters in the second optimization problem: that according to a distance metric dist on inputs and an objective function obj on outputs, all local minima of the surrogate are also local minima of the program (and vice versa), and that the objective values of local minima in the surrogate the and program have the same ordering. Together, these properties mean that local or global minima found by optimizing inputs against the surrogate are also local or global minima of inputs in the original program.

7 Specifications

In this section we catalog specifications from the literature that are used to define the surrogate optimization problems. These specifications correspond to concrete instantiations of interpreters ω along with terms within obj and con that evaluate the surrogate against the program and input dataset.

Tables 1 to 3 present surveys of other examples of surrogate compilation, surrogate adaptation, and surrogate optimization in the literature, showing the terms used in the optimization problem solved by each piece of related work. Sections 7.1 and 7.2 then discuss the design considerations and trade-offs that must be considered when specifying the optimization problem for training a surrogate.

$$s_1^* = \arg\min_s \text{obj}\begin{pmatrix}\omega_p(p),\\ \omega_s(s),\\ \mathcal{D}\end{pmatrix} \text{ subject to con}\begin{pmatrix}\omega_p(p),\\ \omega_s(s),\\ \mathcal{D}\end{pmatrix}$$

Figure 3. Optimization problem for learning a surrogate s_1^* of the original program p, which is the first step of all surrogate programming techniques.

$$s^* = \arg\min_s \text{obj}'\begin{pmatrix}\omega'_p(p),\\ \omega'_s(s),\\ \mathcal{D}',\\ \omega^*(s_1^*)\end{pmatrix} \text{ subject to con}'\begin{pmatrix}\omega'_p(p),\\ \omega'_s(s),\\ \mathcal{D}',\\ \omega^*(s_1^*)\end{pmatrix}$$

Figure 4. 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.

$$i^* = \arg\min_i \text{obj}'\begin{pmatrix}\omega'_p(p),\\ \omega^*(s_1^*),\\ \mathcal{D}',\\ i\end{pmatrix} \text{ subject to con}'\begin{pmatrix}\omega'_p(p),\\ \omega^*(s_1^*),\\ \mathcal{D}',\\ i\end{pmatrix}$$

Figure 5. Second optimization problem for surrogate optimization, which optimizes inputs i of a surrogate s_1^* to minimize a different objective function on the surrogate.

$$\begin{aligned}
\text{isLocalMinimum}(f,i) &\triangleq \exists\epsilon.\,\forall i' \in \mathcal{I}.\,\text{dist}(i,i') \le \epsilon \Rightarrow \text{obj}(f(i)) \le \text{obj}(f(i'))\\
\text{allLocalMinimaMatch} &\triangleq \forall i \in \mathcal{I}.\,\text{isLocalMinimum}(s,i) \Leftrightarrow \text{isLocalMinimum}(p,i)\\
\text{localMinimaOrdering} &\triangleq \forall i,i' \in \mathcal{I}.\,(\text{isLocalMinimum}(s,i) \land \text{isLocalMinimum}(s,i') \land \text{obj}(s(i)) < \text{obj}(s(i')))\\
&\qquad\qquad\qquad \Rightarrow (\text{obj}(p(i)) < \text{obj}(p(i')))
\end{aligned}$$

Figure 6. The properties that make a surrogate a good candidate for surrogate optimization: local minima of inputs to the surrogate are also local minima of the original program against the objective function, and the ordering of local minima is preserved between the surrogate and the program.

7.1 Error

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 [22]. 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 [83].

Downstream error. For surrogate adaptation and surrogate optimization, the second optimizations problems an error metric beyond that of mimicking the original program. The downstream error metric may be that of the downstream task that the original program targets [66, 80]. The downstream error metric may also be unrelated to the domain of the original program: for instance, Kwon and Carloni [46] use an error metric for surrogate adaptation that adapts the surrogate to inputs and outputs of a different domain. She et al. [72] use an error metric for surrogate optimization that measures the extent to which the discovered inputs trigger unseen control flow paths in the program.

7.2 Execution Cost

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

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 [22], high throughput [54], low storage cost [32], and minimal energy cost [22].

8 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.

Table 1. Other examples of surrogate compilation.

Citation and description	Optimization problem specification
Esmaeilzadeh et al. [22]: Training neural surrogates of small computational kernels to accelerate the kernels by executing them on different hardware.	$s_1^* = \arg\min_s \text{obj}\begin{pmatrix}\text{surrogate error,}\\ \text{execution cost}\end{pmatrix}$ subj. to $\text{con}\begin{pmatrix}\text{surrogate error,}\\ \text{execution cost}\end{pmatrix}$ • obj(surrogate error): The mean squared error between the original kernel's output and the surrogate's predictions is minimized [22, Section 4]. • obj(execution cost): The size of the surrogate (number of hidden units) is minimized to reduce execution time [22, Section 4]. • con(surrogate error): The end-to-end error of the application that uses the surrogate is constrained to be less than 10% [22, Section 7.1]. • con(execution cost): Execution of the surrogate must be faster than execution of the original program [22, Sections 7, 8].
Mendis [54, Chapter 4]: Training neural surrogates of compiler auto-vectorizers, to replace the original potentially exponential-time auto-vectorizer with a constant time surrogate.	$s_1^* = \arg\min_s \text{obj}(\text{surrogate error})$ subj. to $\text{con}(\text{execution cost})$ • obj(surrogate error): The cross entropy error between the auto-vectorizer and the surrogate is minimized [54, Chapter 4.4]. • con(execution cost): The surrogate has predictable (and not data-dependent) running time [54, Chapters 1.3.4, 4.8].
Munk et al. [57]: Training neural surrogates of stochastic simulators to accelerate simulation and inference using the simulator.	$s_1^* = \arg\min_s \text{obj}\begin{pmatrix}\text{surrogate error,}\\ \text{execution cost}\end{pmatrix}$ subj. to $\text{con}(\text{execution cost})$ • obj(surrogate error): The KL divergence between the original stochastic simulator and the surrogate is minimized [57, Section 3.1]. • obj(execution cost): The surrogate is as fast as possible to maximize execution throughput over the original simulator [57, Section 3.2]. • con(execution cost): The surrogate must have higher execution throughput than the original simulator [57, Section 3.2].
Pestourie et al. [62]: 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.	$s_1^* = \arg\min_s \text{obj}\begin{pmatrix}\text{surrogate error}\\ \text{execution cost}\end{pmatrix}$ subj. to $\text{con}(\text{execution cost})$ • obj(surrogate error): The MAPE between the PDE solver and the surrogate is minimized [62, Figure 5]. • obj(execution cost): The surrogate is as fast as possible to maximize execution latency speedup over the original solver [62, "Introduction"]. • con(execution cost): The surrogate must be faster to execute than the original PDE solver [62, "Introduction"].

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 neural surrogate, a neural network architecture designed independently of the domain of application of the surrogate and of the behavior of the program under consideration. A common choice of domain-agnostic architectures for neural surrogates are multilayer perceptrons (MLPs) [39, 72]. While simple to design, such domain-agnostic architectures may have high training costs or low accuracy [60, 85].

Domain-specific architectures. An alternative is to design the architecture based on the program under study and its target domain [66, 83]. For instance, Renda et al. [66] use a derivative of the architecture proposed by Mendis et al. [55], a hierarchical LSTM with state-of-the-art accuracy on the task performed by the program under study. Renda et al.'s architecture also exploits sparsity in the program's execution, pre-processing inputs to the surrogate to remove dimensions that do not affect the original program's output. However, designing such architectures requires manual effort and expertise both in the domain and in the original program.

Table 2. Other examples of surrogate adaptation.

Citation and description	Optimization problem specification
Tercan et al. [80]: 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.	$s_1^* = \arg\min_s \text{obj}\big(\text{surrogate error}\big)$ $\quad s^* = \begin{cases} \arg\min_s \text{obj}'\begin{pmatrix}\text{downstream error,}\\ \text{execution cost}\end{pmatrix} \\ \text{subj. to con}'\big(\text{downstream error}\big)\end{cases}$ • obj(surrogate error): The Pearson correlation coefficient between the original simulation and the surrogate is maximized [80, Section 5.2]. • obj′(downstream error): The Pearson correlation coefficient between the trained surrogate and the results of the real-world experiments is maximized [80, Section 5.2]. • obj′(execution cost): The surrogate is cheaper than physical experiments [80, Section 1]. • con′(downstream error): Training is stopped when the training error (as measured by the L1 loss) is less than 0.01 [80, Section 4].
Kustowski et al. [45]: Training neural surrogates of computer simulations of inertial containment fusion, then adapting on a small number of results from real-world experiments to close the gap between simulated and real results.	$s_1^* = \arg\min_s \text{obj}\big(\text{surrogate error}\big)$ $\quad s^* = \arg\min_s \text{obj}'\begin{pmatrix}\text{downstream error,}\\ \text{surrogate error,}\\ \text{execution cost}\end{pmatrix}$ • obj(surrogate error): The Pearson correlation coefficient between the original simulation and the surrogate is maximized [45, Section II]. • obj′(downstream error): The Pearson correlation coefficient between the surrogate and the results of the real-world experiments is maximized [45, Section II]. • obj′(surrogate error): s^* is biased to be close to s_1^* by freezing weights in most layers in the network to their values in s_1^* [45, Section III.B]. • obj′(execution cost): The surrogate is cheaper to run than real-world experiments [45, Section I].
Kwon and Carloni [46]: Training neural surrogates of computer architecture simulations of programs for design space exploration of properties of the architecture, then adapting the surrogates for accurate design space exploration of other programs.	$s_1^* = \arg\min_s \text{obj}\big(\text{surrogate error}\big)$ $\quad s^* = \arg\min_s \text{obj}'\begin{pmatrix}\text{downstream error,}\\ \text{surrogate error,}\\ \text{execution cost}\end{pmatrix}$ • obj(surrogate error): The mean squared error between the simulated running time for the training programs and the surrogate's predictions is minimized [46, Section 1]. • obj′(downstream error): The mean squared error of the surrogate on new programs not in the surrogate's original training set is minimized [46, Section 2]. • obj′(surrogate error): s^* is biased to be close to s_1^* by seeding the optimization problem with weights from s_1^* [46, Section 3]. • obj′(execution cost): The surrogate is cheaper to run than simulation [46, Section 1].
Kaya and Hajimirza [42]: Training neural 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.	$s_1^* = \arg\min_s \text{obj}\big(\text{surrogate error}\big)$ $\quad s^* = \begin{cases} \arg\min_s \text{obj}'\begin{pmatrix}\text{downstream error,}\\ \text{surrogate error,}\\ \text{execution cost}\end{pmatrix} \\ \text{subj. to con}'\big(\text{surrogate error}\big)\end{cases}$ • obj(surrogate error): The mean squared error between the simulation predictions for the base material and the surrogate's predictions on that material is minimized [42, "Results and Discussion – Base Case"]. • obj′(downstream error): The error of the surrogate on the new material is minimized [42, "Results and Discussion – Transfer Cases"]. • obj′(surrogate error): s^* is biased to be close to s_1^* by seeding the optimization problem with weights from s_1^* [42, "Introduction"]. • obj′(execution cost): The surrogate is cheaper to run than simulation [42, "Introduction"]. • con′(surrogate error): If s^* is less accurate than simulation, then the transfer learning results in low accuracy and is rejected [42, "Results and Discussion – Transfer Cases"].

Table 3. Other examples of surrogate optimization.

Citation and description	Optimization problem specification
Renda et al. [66]: 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.	$s_1^* = \arg\min_s \mathrm{obj}\big(\text{surrogate error}\big)$ $\quad i^* = \arg\min_i \mathrm{obj}'\big(\text{downstream error}\big(s_1^*(i)\big)\big)$ • obj(surrogate error): The MAPE between the CPU simulator's prediction and the surrogate's prediction for input code is minimized [66, Section III]. • downstream error$\big(s_1^*(x)\big)$: The MAPE of the simulation parameters are minimized against the ground-truth data [66, Section III].
She et al. [72]: Training neural surrogates of the branching behavior of programs, to find inputs that trigger branches that cause bugs in the program.	$s_1^* = \arg\min_s \mathrm{obj}\big(\text{surrogate error}\big)$ $\quad i^* = \arg\min_i \mathrm{obj}'\big(\text{downstream error}\big(s_1^*(i)\big)\big)$ • obj(surrogate error): The binary cross-entropy error between the predicted branching behavior and the actual branching behavior is minimized [72, Section IV.B]. • downstream error$\big(s_1^*(x)\big)$: Gradient descent tries to find an input that lead to an unseen set of branches taken in the program [72, Section IV.C].
Tseng et al. [83]: Training neural surrogates of camera pipelines, to find parameters for the pipelines that lead to the cameras producing the most photorealistic images.	$s_1^* = \arg\min_s \mathrm{obj}\big(\text{surrogate error}\big)$ $\quad i^* = \arg\min_i \mathrm{obj}'\big(\text{downstream error}\big(s_1^*(i)\big)\big)$ • obj(surrogate error): The L2 error between the image resulting from the pipeline and the surrogate's predicted image is minimized [83, Section 4.2]. • downstream error$\big(s_1^*(x)\big)$: Gradient descent tries to find parameters that lead to images being as similar as possible in L2 distance to the ground-truth [83, Section 4.2].
Shirobokov et al. [73]: Training neural surrogates of physics simulators to find input that lead to local optima.	$s_1^* = \arg\min_s \mathrm{obj}\big(\text{surrogate error}\big)$ $\quad i^* = \arg\min_i \mathrm{obj}'\big(\text{downstream error}\big(s_1^*(i)\big)\big)$ • obj(surrogate error): The error (as measured by a domain-specific loss function) between the simulation and surrogate is minimized [73, Section 2.2]. • downstream error$\big(s_1^*(x)\big)$: Gradient descent tries to find parameters that lead to local optima in the problem space [73, Section 2.2].

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 [76]. Possible approaches include manually selecting the architecture based on expert reasoning about the complexity of the program [66] and automatically searching among possible capacities, selecting the capacity that leads to the optimal trade-offs among the components of the surrogate's specification [22].

9 Training

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

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 to collect training data reflective of the downstream task is to instrument execution of the original program and record observed inputs to the program [13, 22]. 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 some hand-defined sampling distribution [66, 80, 83].

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 [74].

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 ran multiple times during hyperparameter search, the threshold or budget should be set appropriately to account for the full cost of design and training.

One approach is training for a fixed training time, typically determined via experiments on a validation set [22, 66]. Another approach is training until an acceptable accuracy is reached, whether via a plateau of the training loss [83] or via reaching a minimum acceptable accuracy [80]. Such variable-length training time approaches are discussed in more depth by Goodfellow et al. [25, Chapter 7.8].

Determining the training length for surrogate adaptation is especially important due to the challenges imposed by *catastrophic forgetting* [53, 65], 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 [14, 43, 71, 95]; in the case study in Section 4 we simply select the (relatively small) training time that results in the minimum validation error on a held-out test set.

10 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 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 [15], 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 [30, 33, 50].

CPUs. Other approaches deploy the surrogate on a CPU [39]. 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 [30, 34, 49, 50] (though recent work challenges some of these assumptions [18]). CPUs are also more widely available than GPUs, including on edge devices [92].

Machine learning accelerators. Esmaeilzadeh et al. [22] 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 [40], or the Efficient Inference Engine decreasing energy costs while approximating the surrogate [31].

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

Neural networks require specialized software runtime environments, which 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 [22].

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).

Normalization. *Data normalization*, which involves pre- and post-processing the inputs and outputs to be suitable for neural networks [48], 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 [70]. Esmaeilzadeh et al. [22] 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 input examples to input at a time, induces a trade-off between latency and throughput when generating predictions from the surrogate, which various approaches handle in different ways. Parrot [22], a surrogate 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 [66], a 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 surrogate.

11 Future Work

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 surrogates. This section details open problems and future work not addressed in this paper.

Broadening to other surrogate models. Though the surrogate programming techniques are general to all types of surrogate models, the neural surrogate programming methodology is specific to neural networks as surrogate models. However, other surrogate models are popular in the literature, including surrogates based on response surface methodology [58], Gaussian processes [27], and other machine learning techniques [9]. Future work in this direction can extend the programming methodology presented in this paper to other classes of surrogate models.

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 surrogate still must manually navigate the trade-off space. Future work in this domain can mechanize the various aspects of surrogate construction, from automating the surrogate's architecture design process based on the semantics of the original program, to automatically training the surrogate based on explicit specifications and objectives over a data distribution, to automatically integrating the surrogate into the original program's execution context. While some of these concerns have been addressed in prior work [22], fully mechanizing this process for all types of surrogates is an important direction for future work.

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

Generalization and robustness. Large-scale neural networks struggle to generalize outside of their training dataset [4, 38, 94]. 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 [63]. 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 [24], leading to difficulties when reasoning about neural surrogates' predictions. Future work can address these issues by developing better characterizing what interpretability means for different domains, developing interpretability tools for neural surrogates, 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 less of a concern.

12 Related Work Addressing Similar Tasks

In this section we discuss related work that provides alternative solutions to the surrogate programming techniques 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 [64, 82]. The conventional wisdom is that compared to other approaches, neural networks excel at *feature extraction* [37], 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 [44]. 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.

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 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 optimization. Differentiable programming also relies on implementing the program in a language amenable to differentiable programming such as Pytorch or TensorFlow [1, 8, 61].

In contrast, 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 surrogate, allowing for optimizing the inputs despite challenges posed by the original program [66].

Program smoothing. Chaudhuri and Solar-Lezama [12] 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.

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 [17, 26]. 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 [59] and variational inference [10]. 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.

Automating construction of surrogates. Munk et al. [57] 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, regardless of the number or ordering of samples output by the original simulator. Munk et al. show that the surrogate executes faster than the original simulator.

13 Related Work Addressing Other Tasks

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

Surrogates of non-programs. Building surrogates of black-box processes beyond just programs is used across a wide variety of domains from computer systems to physical sciences [11, 55, 77]. For example, Mendis et al. [55] 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 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 surrogates of programs for which we have 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 [81, 87, 89]. 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. [87] train neural networks that augment programmatic reinforcement learning policies. 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 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, 87, 88]. For example, after training a residual model, Verma et al. [87] 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 surrogate that mimics a program.

14 Conclusion

Surrogates are an important emerging frontier of programming with a wealth of potential use cases for writing and reasoning about the behavior of complex programs that execute in complex or uncertain environments. By identifying the different classes of application and describing the programming methodology used to develop neural surrogates, our work provides a taxonomy for reasoning about and developing surrogates. Our work offers a foundation on which the programming languages community can build new tools that aid in the construction and analysis of surrogates, and opens the door to further development and applications of surrogate programming.

Acknowledgments

We would like to thank Alana Marzoev, Ben Sherman, Cambridge Yang, Charith Mendis, Charles Yuan, Jesse Michel, Saman Amarasinghe, Stella Lau, and the anonymous reviewers for their helpful comments and suggestions. This work was supported in part by the National Science Foundation (CCF-1918839 and CCF-1751011), the Defense Advanced Research Projects Agency (#HR00112190046 and #HR0011-18-C-0059), a Google Faculty Research Award, and a Sloan Research Fellowship. Yi Ding's work is supported by the National Science Foundation under Grant No. 2030859 to the Computing Research Association for the CIFellows Project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

A Methodology for Surrogate 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 `c2-standard-4` instance, using a single core of an Intel Xeon Skylake CPU at 3.1 GHz.

llvm-mca is compiled in release mode from version 8.0.1, using the version released by Renda et al. [66] at https://github.com/ithemal/DiffTune/tree/9992f69/llvm-mca-parametric. llvm-mca is invoked a single time and passed a random sample of 10,000 basic blocks from BHive over stdin, 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 is ran on the same CPU, using a network compiled, optimized, and loaded with the ONNX runtime, version 1.7.0 [19]. 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 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 10,000 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.

Table 4. 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 with 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×

B BERT Hyperparameter Selection and Training Telemetry

This appendix describes the hyperparameter selection process and training methodology used for the BERT model used in Sections 3 and 4. In Appendix B.1 we describe the hyperparameters used for the model and show the hyperparameter search process used to find the hidden size of 64. In Appendix B.2 we show the ultimate training, validation, and test loss curves of the models, along with the total amount of time taken to train all models.

B.1 Hyperparameters.

We base our BERT model on the BERT-Tiny model described by Turc et al. [84], which has an embedding width of 128, 2 layers, and 2 self-attention heads. From this base architecture we 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 4 shows the results of the hyperparameter search, with the bolded row describing the selected model (with an embedding with 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%.

B.2 Training Telemetry

We report the full training curves for the case studies in Sections 3 and 4 in supplemental material available online at https://doi.org/10.6084/m9.figshare.16622839.

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.* https://doi.org/10.1145/3297858.3304062

[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.* https://doi.org/10.1145/2628071.2628092

[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. https://doi.org/10.1109/37.158898

[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).

[7] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. 2019. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences* 116, 32 (2019). https://doi.org/10.1073/pnas.1903070116

[8] Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. 1996. Adifor 2.0: automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering* 3, 3 (1996). https://doi.org/10.1109/99.537089

[9] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning.* Springer. https://doi.org/10.1117/1.2819119

[10] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. 2017. Variational Inference: A Review for Statisticians. *J. Amer. Statist. Assoc.* 112, 518 (2017). https://doi.org/10.1080/01621459.2017.1285773

[11] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. [n.d.]. Machine learning and the physical sciences. *Reviews of Modern Physics* 91 ([n. d.]). Issue 4. https://doi.org/10.1103/RevModPhys.91.045002

[12] Swarat Chaudhuri and Armando Solar-Lezama. 2010. Smooth Interpretation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* https://doi.org/10.1145/1809028.1806629

[13] 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.* https://doi.org/10.1109/IISWC47752.2019.9042166

[14] Alexandra Chronopoulou, Christos Baziotis, and Alexandros Potamianos. 2019. 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.* https://doi.org/10.18653/v1/N19-1213

[15] 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.* https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-210

[16] Kyle Cranmer, Johann Brehmer, and Gilles Louppe. 2020. The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences* 117, 48 (2020). https://doi.org/10.1073/pnas.1912789117

[17] 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.* https://doi.org/10.1145/3314221.3314642

[18] 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.*

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

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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.* https://doi.org/10.18653/v1/N19-1423

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

[22] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *IEEE/ACM International Symposium on Microarchitecture.* https://doi.org/10.1109/MICRO.2012.48

[23] 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.

[24] 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.* https://doi.org/10.1109/DSAA.2018.00018

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

[26] 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.*

[27] Robert B. Gramacy. 2020. *Surrogates: Gaussian Process Modeling, Design and Optimization for the Applied Sciences.* Chapman Hall/CRC. https://doi.org/10.1201/9780367815493

[28] 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.*

[29] Georg Hager and Gerhard Wellein. 2010. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, Inc. https://doi.org/10.1201/EBK1439811924

[30] Song Han. 2017. *Efficient Methods and Hardware for Deep Learning.* Ph.D. Dissertation. Stanford University.

[31] 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.* https://doi.org/10.1145/3007787.3001163

[32] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations* (2016).

[33] 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*. https://doi.org/10.1145/3204949.3204975

[34] 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*. https://doi.org/10.1109/HPCA.2018.00059

[35] 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

[36] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997). https://doi.org/10.1162/neco.1997.9.8.1735

[37] 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*. https://doi.org/10.1109/CVPR.2006.164

[38] 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*.

[39] 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*. https://doi.org/10.1145/1168857.1168882

[40] 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*. https://doi.org/10.1145/3079856.3080246

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

[42] Mine Kaya and Shima Hajimirza. 2019. Using a Novel Transfer Learning Method for Designing Thin Film Solar Cells with Enhanced Quantum Efficiencies. *Scientific Reports* 9, 5034 (2019). https://doi.org/10.1038/s41598-019-41316-9

[43] 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. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences* 114, 13 (2017). https://doi.org/10.1073/pnas.1611835114

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

[45] Bogdan Kustowski, Jim A. Gaffney, Brian K. Spears, Gemma J. Anderson, Jayaraman J. Thiagarajan, and Rushil Anirudh. 2020. Transfer Learning as a Tool for Reducing Simulation Bias: Application to Inertial Confinement Fusion. *IEEE Transactions on Plasma Science* 48, 1 (2020). https://doi.org/10.1109/TPS.2019.2948339

[46] Jihye Kwon and Luca P. Carloni. 2020. Transfer Learning for Design-Space Exploration with High-Level Synthesis. In *ACM/IEEE Workshop on Machine Learning for CAD*. https://doi.org/10.1145/3380446.3430636

[47] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*. https://doi.org/10.1109/CGO.2004.1281665

[48] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. 2012. *Efficient BackProp* (2nd ed.). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35289-8_3

[49] 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*.

[50] Da Li, Xinbo Chen, Michela Becchi, and Ziliang Zong. 2016. 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*.

[51] Zachary C. Lipton, John Berkowitz, and Charles Elkan. 2015. A Critical Review of Recurrent Neural Networks for Sequence Learning. arXiv:1506.00019 [cs.LG]

[52] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/36206.36194

[53] Michael McCloskey and Neal J. Cohen. 1989. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. *Psychology of Learning and Motivation* 24 (1989). https://doi.org/10.1016/S0079-7421(08)60536-8

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

[55] 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*.

[56] 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*.

[57] Andreas Munk, Adam Ścibior, Atılım Güneş Baydin, Andrew Stewart, Goran Fernlund, Anoush Poursartip, and Frank Wood. 2019. Deep Probabilistic Surrogate Networks for Universal Simulator Approximation. arXiv:1910.11950 [cs.LG]

[58] Raymond H. Myers, Douglas C. Montgomery, and Christine M. Anderson-Cook. 2016. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments* (4th ed.). Wiley.

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

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

[61] 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.

[62] Raphaël Pestourie, Youssef Mroueh, Thanh V. Nguyen, Payel Das, and Steven G. Johnson. 2020. Active learning of deep surrogates for PDEs: application to metasurface design. *npj Computational Materials* 6, 164 (2020). https://doi.org/10.1038/s41524-020-00431-2

[63] André Platzer. 2010. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics* (1st ed.). Springer. https://doi.org/10.1007/978-3-642-14509-4

[64] Carl Edward Rasmussen and Christopher K. I. Williams. 2005. *Gaussian Processes for Machine Learning*. The MIT Press.

[65] Roger Ratcliff. 1990. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review* 97, 2 (1990). https://doi.org/10.1037/0033-295x.97.2.285

[66] 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*. https://doi.org/10.1109/MICRO50266.2020.00045

[67] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2020. A Primer in BERTology: What We Know About How BERT Works. *Transactions of the Association for Computational Linguistics* 8 (2020). https://doi.org/10.1162/tacl_a_00349

[68] Thomas J. Santner, Williams Brian J., and Notz William I. 2018. *The Design and Analysis of Computer Experiments* (2nd ed.). Springer-Verlag. https://doi.org/10.1007/978-1-4939-8847-1

[69] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/2451116.2451150

[70] 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)*.

[71] Joan Serrà, Dídac Surís, Marius Miron, and Alexandros Karatzoglou. 2018. Overcoming catastrophic forgetting with hard attention to the task. In *International Conference on Machine Learning*.

[72] 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*. https://doi.org/10.1109/SP.2019.00052

[73] 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*.

[74] Connor Shorten and Taghi M. Khoshgoftaar. 2019. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data* 6 (2019). https://doi.org/10.1186/s40537-019-0197-0

[75] 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*. https://doi.org/10.1145/2025113.2025133

[76] 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. *Comput. Surveys* 53, 3 (2020). https://doi.org/10.1145/3394898

[77] 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). https://doi.org/10.1177/0954410019864485

[78] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd ed.). The MIT Press.

[79] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *International Conference on Machine Learning*.

[80] 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). https://doi.org/10.1016/j.procir.2018.03.087

[81] 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*. https://doi.org/10.1109/CVPR.2017.577

[82] Lloyd N. Trefethen. 2019. *Approximation Theory and Approximation Practice* (extended ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611975949

[83] 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 (2019). https://doi.org/10.1145/3306346.3322996

[84] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-Read Students Learn Better: On the Importance of Pre-training Compact Models. arXiv:1908.08962 [cs.CL]

[85] 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*.

[86] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*.

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

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

[89] 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). https://doi.org/10.1029/2018MS001597

[90] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.

[91] 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. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Empirical Methods in Natural Language Processing: System Demonstrations*. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[92] 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*. https://doi.org/10.1109/HPCA.2019.00048

[93] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph

Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021). https://doi.org/10.1109/TNNLS.2020.2978386

[94] 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*.

[95] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *Advances in Neural Information Processing Systems*.

[96] Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. 2020. Are Transformers universal approximators of sequence-to-sequence functions?. In *International Conference on Learning Representations*.