Programming with Neural 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 programs, form the basis of a variety of development workflows. We study three surrogate-based design patterns, evaluating each in case studies on a large-scale CPU simulator.

With *surrogate compilation*, programmers develop a surrogate that mimics the behavior of a program to deploy to end-users in place of the original program. Surrogate compilation accelerates the CPU simulator under study by 1.6×. With *surrogate adaptation*, programmers develop a surrogate of a program then retrain that surrogate on a different task. Surrogate adaptation decreases the simulator's error by up to 50%. With *surrogate optimization*, programmers develop a surrogate of a program, optimize input parameters of the surrogate, then plug the optimized input parameters back into the original program. Surrogate optimization finds simulation parameters that decrease the simulator's error by 5% compared to the error induced by expert-set parameters.

In this paper we formalize this taxonomy of surrogate-based design patterns. We further describe the programming methodology common to all three design patterns. Our work builds a foundation for the emerging class of workflows based on programming with surrogates of programs.

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 Neural 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

Onward! '21, October 20–22, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9110-8/21/10.
https://doi.org/10.1145/3486607.3486748

1 Introduction

Programmers and researchers are increasingly developing *surrogates* of programs, models of a subset of the observable behavior of a given program, to solve a variety of software development challenges [23, 42, 48, 49, 60, 67, 71, 77, 84, 87].

Programmers train surrogates from measurements of the behavior of a program on a dataset of input examples [27, 29, 61, 73]. Typical examples of surrogates include neural networks [27, 71], Gaussian processes [3, 69], linear models [22, 25], and random forests [38, 62]. Of these model architectures, *neural surrogates* have emerged as a popular design for surrogates in the literature [23, 42, 71, 77, 84, 87] because for many tasks neural networks are state-of-the-art models that lead to high accuracy [20, 47].

Programmers use surrogates for a variety of tasks including accelerating computational kernels in numerical programs [23], replacing physical simulators with more accurate versions [84], and tuning parameters of complex simulators [71, 87]. Compared to standard development workflows, programming with surrogates requires lower development costs [45, 49, 71, 77, 87] and results in programs with lower execution cost [23, 57, 60, 67] or higher result quality [48, 71, 84, 87]. However, the approaches in the literature for both applying and developing surrogates are disparate, with no unifying taxonomy or development methodology.

1.1 Surrogate-Based Design Patterns

In this paper we contribute a taxonomy that classifies the workflows above into three different design patterns: *surrogate compilation*, *surrogate adaptation*, and *surrogate optimization*. We concretize these design patterns by demonstrating how to use each to solve one of three development tasks for llvm-mca [21], a 10,000 line-of-code CPU simulator that predicts the execution time of code snippets.

Surrogate compilation. With surrogate compilation, programmers develop a surrogate that replicates the behavior of a program to deploy to end-users in place of that program. Key benefits of this approach include the ability to execute the surrogate on different hardware and the ability to bound or to accelerate the execution time of the surrogate [23, 57].

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

Surrogate adaptation. With surrogate adaptation, programmers first develop a surrogate of a program then further train that surrogate on data from a different task. Key benefits of this approach include that surrogate adaptation makes it possible to alter the semantics of the program to perform a different task of interest and that it may be more data-efficient or result in higher accuracy than training a model from scratch for the task [48, 84].

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

Surrogate optimization. With surrogate optimization, programmers develop a surrogate of a program, optimize input parameters of that surrogate, then plug the optimized 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) [71, 77, 87].

We train a neural network to replicate llvm-mca's prediction when llvm-mca is parameterized with different sets of simulation parameters, then optimize against that network to find parameters that lead the network to accurately predict ground-truth timings. We then plug these parameters back into llvm-mca. These parameters improve llvm-mca's accuracy by 5% relative to expert-selected parameters.

1.2 Programming Methodology

The development methodologies common to these surrogate-based design patterns when 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 network, and the *deployment* of the system.

We present the programming methodology as a set of questions that guide development of the surrogate. A complete set of answers to these questions constitutes a concrete plan for the development and deployment of a neural surrogate.

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

1.3 Contributions

In this paper we present the following contributions:

- We provide three detailed case studies of programming with surrogates on a large-scale CPU simulator.
- We formally define three design patterns that use surrogates of programs: surrogate compilation, surrogate adaptation, and surrogate optimization. We demonstrate that this taxonomy captures examples of surrogate programming from the literature.
- We identify elements of the neural surrogate programming methodology in the form of specifications and design questions that unify these surrogate-based design patterns. 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 use cases for developing complex programs. By identifying the three surrogate-based design patterns 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 of programs.

2 Case Study: Overview

We first demonstrate how developing surrogates of a CPU simulator makes it possible to solve three development tasks: (1) increasing the speed of the simulation, (2) simulating the execution behavior of a real-world processor that is not well-modeled by the simulator, and (3) finding simulation parameters that lead the simulator to accurate simulation of the behavior of a real-world processor. We present a surrogate optimization case study from our prior work [71] and present two new case studies of surrogate compilation and surrogate adaptation on the same simulator under study.

Program under study. We study llvm-mca [21], a CPU simulator included in the LLVM compiler infrastructure [50].

Figure 1 presents llvm-mca's input-output specification and design. As input, llvm-mca takes a *basic block*, a sequence of assembly instructions with no jumps or loops, and a set of *CPU parameters*, integers that describe properties of the CPU being modeled. It then outputs a prediction of the *throughput* of the basic block on the CPU, a prediction of the number of CPU clock cycles taken to execute the block when repeated for a fixed number of iterations.

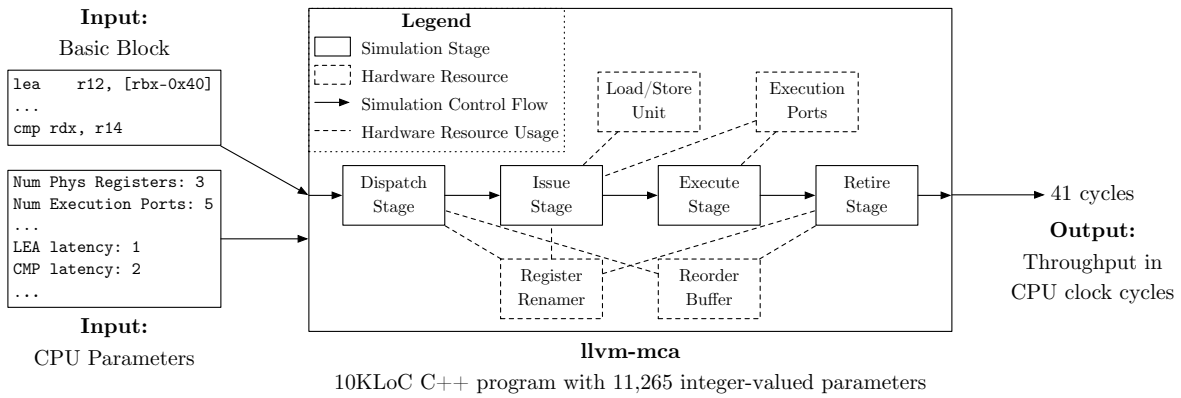

Figure 1. Input-output specification and design of llvm-mca.

Rather than precisely emulating the behavior of the CPU under study, llvm-mca makes several modeling assumptions about the behavior of the CPU, and simulates basic blocks using an abstract execution model of that CPU. The llvm-mca system simulates a processor in four main simulation stages: *dispatch*, *issue*, *execute*, and *retire*. Instructions pass through each of these four stages in turn. Each stage is bottlenecked by the availability of *hardware resources* in the simulation model. The input CPU parameters specify what resources are available on the hardware and what resources to reserve for each instruction. When all instructions of the basic block have passed through the full 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 the hardware resources needed to track the execution of the instruction in the simulation model.

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

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

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

Implementation. The llvm-mca system is a C++ program implemented as part of the LLVM compiler infrastructure, comprised of around 10,000 lines of code. The CPU parameters are comprised of 11,265 integer-valued parameters, inducing a configuration space with $10^{19,336}$ possible configurations. LLVM contains expert-set CPU parameter settings for llvm-mca that target common x86 hardware architectures.

Validation and accuracy. In our prior work [14], we validate the accuracy of llvm-mca by collecting BHive, a dataset of x86 basic blocks from a variety of end-user programs. For each basic block in BHive we also collect ground-truth throughput measurements of the block by timing them on real CPUs. We calculate the mean absolute percentage error (MAPE) of llvm-mca's throughput predictions, which is the normalized difference between llvm-mca's output y_{pred} and the ground-truth measured throughput y_{true}:

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

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

3 Case Study: Surrogate Compilation

To quickly generate throughput predictions for basic blocks, programmers must develop fast CPU simulation models.

The standard approach, used by llvm-mca, is to manually implement a fast and sufficiently accurate simulation model, then use compiler optimizations to accelerate the *execution speed* of the simulation code. We define llvm-mca's execution speed as the number of basic blocks per second that llvm-mca is able to generate throughput predictions for.

Other approaches in the literature for accelerating llvm-mca's execution speed include rewriting the simulation software to be faster [31] and applying compiler optimizations not included in llvm-mca's default compiler's optimization set, such as superoptimization [55, 74].

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

Results. When we instantiate llvm-mca with its default set of Haswell CPU parameters, llvm-mca's execution speed on an Intel Xeon Skylake CPU at 3.1GHz is 1742 blocks per second.[1] Using surrogate compilation we learn a neural surrogate of llvm-mca that has an execution speed of 2820 blocks per second on the same hardware, a speedup of 1.6× over llvm-mca. This surrogate has a mean absolute percentage error (MAPE) of 9.1% compared to llvm-mca's predictions. Against BHive's ground-truth measured data on a real Haswell CPU, the surrogate has an error rate of 27.1%, compared to an error rate of 25.0% for llvm-mca.

3.1 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.

3.1.1 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.

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

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

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

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

3.1.2 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 network's capacity appropriately. These choices must be informed by the specification of the surrogate and by the semantics of the program that the surrogate models.

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

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

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 [27]. The topology determines the types of inputs that the network can process (e.g., fixed-size inputs or arbitrary length sequences) and the *inductive biases* of the network, the assumptions about the task that are baked into the neural network.

We use a BERT encoder [20], a type of Transformer [90], 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 BERT due to its popularity [72], expressive power [100], and relative ease of use [96] for arbitrary sequence modeling tasks (though programmers should in general choose the most appropriate neural network architecture to 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 that the surrogate can represent. Higher capacity neural networks better fit the training data [8], but have higher execution cost [83]. 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.

3.1.3 Training. With the architecture in hand, 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 [14] 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 and constraints from the specification (which may not themselves be differentiable), with different relaxations having different properties [10, pp. 337–338].

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

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

The number of training iterations for the neural surrogate determines the trade-off between the training cost of the surrogate and the accuracy of the surrogate. In general, the cost of training is limited either by an acceptability threshold on the error or by a fixed training budget. Because the training procedure may be run multiple times when designing the surrogate, the threshold or budget should be set appropriately to account for the full cost of design and training.

We train the BERT model for 500 passes over the training set (500 epochs), recording the loss over a validation set after each epoch. At the end of training, 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.

3.1.4 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.

4 Case Study: Surrogate Adaptation

Beyond just being fast, CPU simulators must be accurate. To accurately model behaviors observed in real-world processors, a programmer must develop a model that matches the behavior of that processor.

The standard approach, exemplified by llvm-mca, is to manually design, implement, and tune an abstract execution model of the processor. This approach takes significant development effort, and can still result in inaccurate simulation, in part due to simplifying modeling assumptions that programmers must make that do not accurately reflect real CPUs.

Alternatively to hand-tuning a model, programmers can train a machine learning model from scratch based on observations of the ground-truth behavior of the processor. Though it requires less development effort, this approach requires a significant amount of data to train an accurate model.

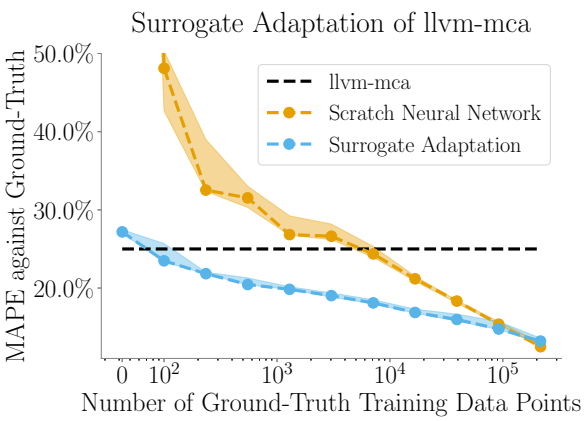

Figure 2. Error on ground-truth data of llvm-mca (black), a neural network trained from scratch (orange), and surrogate adaptation of llvm-mca (blue). The rightmost point corresponds to training on the entire BHive dataset.

Surrogate adaptation. Another approach for developing an accurate simulation model is surrogate adaptation. With surrogate adaptation, programmers first develop a surrogate of a program then further train that surrogate on data from a different task. Key benefits of this approach include that surrogate adaptation makes it possible to alter the semantics of the program to perform a different task of interest and that it may be more data-efficient or result in higher accuracy than training a model from scratch for the task [48, 84].

Results. Figure 2 presents the MAPE of several approaches to predicting ground-truth basic block throughputs, as a function of the size of the training dataset of the approach. The black dashed line shows llvm-mca's error rate, which is not a function of the amount of ground-truth training data available, and is constant at 25.0%. The blue dotted line shows surrogate adaptation's error rate, which is upper bounded by llvm-mca's, as surrogate adaptation is first trained to mimic llvm-mca, then decreases with more training data. The orange dots show the error of a neural network trained from scratch, which results in a large error rate when trained with a small number of examples, only matching surrogate adaptation when it is trained on the entire BHive training data set.

These results show that surrogate adaptation leads to more accurate simulation than training a neural network from scratch when ground-truth data is not readily available (e.g., in cases where collecting ground-truth data is expensive), but provides no benefit when ground-truth data is plentiful.

4.1 Programming Methodology

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

4.1.1 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:

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

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

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

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

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

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

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

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

To minimize hyperparameter search cost, we reuse the capacity for the neural surrogate from Section 3, which has less than 10% error against llvm-mca.

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

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

The loss function for training the surrogate in both optimization problems is the MAPE, as specified in the specification. In general, the loss functions for the two optimization problems do not have to be the same, if the programmer is adapting the surrogate 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 surrogate with minimum validation loss within 500 epochs of training. In the second optimization problem, the surrogate resulting from the first step is used as a warm starting point for optimization. We again use the minimum-validation-loss surrogate within 500 epochs of training, which constrains the surrogate in the second problem to not deviate too much from the original surrogate. The plots of training and validation loss over the course of training are presented in Appendix B.2.

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

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

The neural surrogate is trained in PyTorch [65], which automatically calculates the gradient of the surrogate for both optimization problems. Since the specification does not impose deployment constraints, we also deploy it in PyTorch.

5 Case Study: Surrogate Optimization

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

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

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

Alternatively, the parameters may be set by automatic approaches based entirely on measurement. One class of automatic approaches for setting llvm-mca's parameters is to gather measurements of each parameter's realization in the CPU architecture that llvm-mca targets [2, 24, 71].

Another class of approaches is to gather coarse-grained measurements of entire basic blocks then optimize llvm-mca's parameters to best fit the timings of the basic blocks. Due to the size of the parameter space, this is an optimization problem for which gradient-free optimization techniques [4] are intractable. Gradient descent converges to local minima more quickly than gradient-free optimization with the original program [37]. However, since llvm-mca is not written in a differentiable programming language [7] and operates over discrete values, it is also not possible to calculate its gradient or optimize its parameters with gradient descent.

Surrogate optimization. An alternative approach for optimizing parameters of the program using coarse-grained measurements is to use surrogate optimization. We present a case study drawn from our prior work [71] of using surrogate optimization to optimize llvm-mca's parameters.

With surrogate optimization, programmers develop a surrogate of a program, optimize input parameters of that surrogate, then finally 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) [71, 77, 87].

Results. Using surrogate optimization, we find parameters that lead llvm-mca to an average error of 23.7% on the Haswell basic blocks in BHive [14]. In contrast, the expert-tuned default Haswell parameters lead llvm-mca to an average error of 25.0%. OpenTuner [4], a gradient-free optimization technique, is not able to find parameters that lead llvm-mca to lower than 100% error when given a computational budget equivalent to that of surrogate optimization.

5.1 Programming Methodology

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

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

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

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

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

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

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

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

5.1.2 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 including x_{params} as input to the surrogate, the BERT architecture in Sections 3 and 4, which expects just basic blocks as input, is not sufficient for this task. We use the neural network architecture proposed by our prior work [58].

The architecture consists of a stacked pair of LSTMs [39]. The bottommost LSTM generates a vector representation of each instruction independently. We then concatenate each of these instruction vector representations with the relevant parameters in x_{params} that affect simulation of the instruction. The topmost LSTM then processes each of these vector representations to generate a final prediction for the basic block.

We validate that this model learns to predict the throughput of basic blocks on physical Intel CPUs with low error [58], a similar problem to developing a surrogate of llvm-mca.

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

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

5.1.3 Training.

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

In both optimization problems, we use basic blocks from the BHive dataset as input basic blocks x_{block} [14]. In the first optimization problem, we also use a bounded uniform distribution over parameter values (informed by the range of parameter values for other CPU architectures) as input parameters x_{params}. As with the surrogate adaptation example, in the first optimization problem the throughputs to predict are llvm-mca's predictions on these basic blocks, and in the second they are the measured timings from BHive.

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

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

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

We 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.

5.1.4 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 x^*_{params} are plugged back into llvm-mca, which is executed on a CPU.

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

The surrogate and input parameters are trained in PyTorch [65], which calculates the gradients for both the surrogate and the input optimization. Once found, the input parameters x^*_{params} are then plugged back into llvm-mca.

6 Surrogate-Based Design Patterns

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

6.1 Surrogates of Programs

Let $p \in \mathcal{P}$ denote a program under study. Let $\omega \in \mathcal{P} \to \mathcal{X} \to \mathcal{Y}$ denote an *interpreter*, which takes the program p and an input $x \in \mathcal{X}$ and produces an output $y \in \mathcal{Y}$. Let ω^* denote the *standard interpreter*, corresponding to the standard input-output relationship of the program according to the denotational semantics of the programming language [95, Chapter 5]. Other interpreters may output other aspects of the execution of the program, such as its execution time, memory usage, control flow trace, or any other aspect of its denotational or operational semantics. Finally, let $s \in \mathcal{P}$ denote a surrogate of the program.

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

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

6.2 Surrogate-Based Design Patterns

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

Let $d : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ measure the error between two outputs. Let $e : (\mathcal{X} \to \mathcal{Y}) \times \mathcal{X} \to \mathbb{R}$ measure the cost of executing a given interpretation of a program on a given input (measured in latency, execution cost, energy, etc.). Let $\ell : \mathcal{Y} \times \mathcal{X} \to \mathbb{R}$ measure the error on a downstream task induced by a given prediction of a given input.

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

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

Surrogate Construction. The first step of each surrogate-based design pattern is to train a surrogate of the original program. Figure 3 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

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

Figure 3. Optimization problem for learning a surrogate s_1^* of the original program p. This optimization problem is the first step of all three surrogate-based design patterns.

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

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. The surrogate s_1^* is used as a warm start for this problem.

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

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

a task-dependent objective function o over a distribution of inputs $x \sim \mathcal{D}(\mathcal{X})$ of the error d between the standard interpretation ω^* of the surrogate s on that input x and an interpretation ω_p of the original program p on the input x, and of the execution cost e of the standard interpretation ω^* of the surrogate s on the input x, subject to a constraint function c of the same terms.

6.2.1 Surrogate Compilation.

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

6.2.2 Surrogate Adaptation.

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

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

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

6.2.3 Surrogate Optimization.

The first step of surrogate optimization is the surrogate construction step. The second step is to optimize inputs to the surrogate against 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 x^* that minimizes a task-dependent objective function o of the downstream error ℓ of the standard interpretation ω^* of the surrogate from the first optimization problem s_1^* on the input x, subject to a constraint function c of the same term.

6.2.4 Specifications in the Literature.

Tables 1 to 3 respextively present surveys of surrogate compilation, surrogate adaptation, and surrogate optimization, showing the terms in the optimization problem solved by each piece of related work. These optimization problem specifications correspond to concrete instantiations of interpreters ω, error functions d, e, and ℓ, and objective functions o and c.

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

Surrogate error. A surrogate must compute a similar function to that computed by its source program. When the surrogate is deployed to end-users as in surrogate compilation and surrogate adaptation, the error metric for the surrogate is that of the domain [23]. 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 [87].

In the second step of surrogate adaptation, the final surrogate may also be constrained to be close to the original surrogate, another instantiation of surrogate error (treating the original surrogate as a source program) [45, 49].

Table 1. Optimization problem specifications of surrogate compilation from the literature.

Citation and description	Optimization problem specification
Esmaeilzadeh et al. [23]: Training neural surrogates of small numerical kernels to decrease their execution latency by executing them on a neural network accelerator.	$s^* = \arg\min_s o\binom{d(s,p),}{e(s)}$ subj. to $c\binom{d(s,p),}{e(s)}$ • $o(d(s,p))$: The mean squared error between the outputs of the surrogate and the original kernel is minimized [23, Section 4]. • $o(e(s))$: The size of the surrogate (measured by the number of hidden units) is minimized to reduce execution time [23, Section 4]. • $c(d(s,p))$: The end-to-end error of the application that uses the surrogate is constrained to be less than 10% [23, Section 7.1]. • $c(e(s))$: The surrogate is constrained to have lower execution latency than the original kernel [23, Sections 7, 8].
Mendis [57, Chapter 4]: Training neural surrogates of compiler auto-vectorizers, to replace the original exponential-time auto-vectorizer with a linear time surrogate.	$s^* = \arg\min_s o(d(s,p))$ subj. to $c(e(s))$ • $o(d(s,p))$: The cross entropy error between the outputs of the surrogate and the auto-vectorizer is minimized [57, Chapter 4.4]. • $c(e(s))$: The surrogate has predictable (and not data-dependent) linear running time [57, Chapters 1.3.4, 4.8].
Munk et al. [60]: Training neural surrogates of stochastic simulators to accelerate simulation and inference using the simulator.	$s^* = \arg\min_s o\binom{d(s,p),}{e(s)}$ subj. to $c(e(s))$ • $o(d(s,p))$: The KL divergence between the outputs of the surrogate and the original stochastic simulator is minimized [60, Section 3.1]. • $o(e(s))$: The surrogate is as fast as possible to maximize the execution throughput speedup over the original simulator [60, Section 3.2]. • $c(e(s))$: The surrogate is constrained to have higher execution throughput than the original simulator [60, Section 3.2].
Pestourie et al. [67]: Training neural surrogates of partial differential equation (PDE) solvers to aid designing material composites, using active learning to minimize the training cost of the surrogate.	$s^* = \arg\min_s o\binom{d(s,p)}{e(s)}$ subj. to $c(e(s))$ • $o(d(s,p))$: The MAPE between the outputs of the surrogate and the original PDE solver is minimized [67, Figure 5]. • $o(e(s))$: The surrogate is as fast as possible to maximize execution latency speedup over the original solver [67, "Introduction"]. • $c(e(s))$: The surrogate must have higher execution throughput than the original PDE solver [67, "Introduction"].

Downstream error. For surrogate adaptation and surrogate optimization, the second optimization problems use an error metric beyond that of mimicking the original program. This downstream error metric may be that of the downstream task that the original program targets [71, 84]. The downstream error metric may also be unrelated to the domain of the original program: for instance, Kwon and Carloni [49] use an error metric for surrogate adaptation that adapts the surrogate to inputs and outputs of a different domain. She et al. [77] use an error metric for surrogate optimization that measures the extent to which the discovered inputs trigger unseen control flow paths in the program.

Execution Cost. Regardless of the intended use case, a surrogate must be efficient, not exceeding resource budgets to deploy. The execution cost of a surrogate measures the resources required to execute the surrogate in its execution environment. The ideal is a surrogate that is efficient to execute, with low execution latency [23], high throughput [57], low storage cost [34], and minimal energy cost [23].

6.3 Key Benefits

We now demonstrate the key benefits of each design pattern, detailing examples beyond those of the case study.

6.3.1 Surrogate Compilation. Surrogate compilation allows for the ability to execute the surrogate on different hardware and the ability to bound or to accelerate the execution time of the surrogate [23, 57].

Table 2. Optimization problem specifications of surrogate adaptation from the literature.

Citation and description	Optimization problem specification
Tercan et al. [84]: Training neural surrogates of computer simulations of plastic injection molding, then adapting the surrogates on real-world experiments of injection molding to close the gap between simulated and real results.	$s_1^* = \arg\min_s o_1\big(d(s,p)\big)$ $\quad s^* = \begin{cases} \arg\min_s o_2\begin{pmatrix}\ell(s_1^*,x),\\ e(s)\end{pmatrix} \\ \text{subj. to } c_2\Big(\ell\big(s_1^*,x\big)\Big)\end{cases}$ • $o_1(d(s,p))$: The Pearson correlation coefficient between the outputs of the surrogate and the original simulation is maximized [84, Section 5.2]. • $o_2\big(\ell\big(s_1^*,x\big)\big)$: The Pearson correlation coefficient between the trained surrogate and the results of the real-world experiments is maximized [84, Section 5.2]. • $o_2(e(s))$: The surrogate is cheaper to execute than physical experiments [84, Section 1]. • $c_2\big(\ell\big(s_1^*,x\big)\big)$: The L1 loss of the surrogate is constrained to be less than 0.01 [84, Section 4].
Kustowski et al. [48]: Training neural surrogates of computer simulations of inertial confinement fusion, then adapting on a small number of results from real-world experiments to close the gap between simulated and real results.	$s_1^* = \arg\min_s o_1\big(d(s,p)\big)$ $\quad s^* = \arg\min_s o_2\begin{pmatrix}\ell(s_1^*,x),\\ d(s,s_1^*),\\ e(s)\end{pmatrix}$ • $o_1(d(s,p))$: The Pearson correlation coefficient between the outputs of the surrogate and the original simulation is maximized [48, Section II]. • $o_2\big(\ell\big(s_1^*,x\big)\big)$: The Pearson correlation coefficient between the trained surrogate and the results of the real-world experiments is maximized [48, Section II]. • $o_2\big(d\big(s,s_1^*\big)\big)$: s^* is biased to be close to s_1^* by freezing the weights in most layers in the neural network to be equal to their values in s_1^* [48, Section III.B]. • $o_2(e(s))$: The surrogate is cheaper to run than real-world experiments [48, Section I].
Kwon and Carloni [49]: Training neural surrogates of computer architecture simulations of programs for design space exploration of the architecture, then adapting the surrogates for accurate design space exploration when simulating other programs.	$s_1^* = \arg\min_s o_1\big(d(s,p)\big)$ $\quad s^* = \arg\min_s o_2\begin{pmatrix}\ell(s_1^*,x),\\ d(s,s_1^*),\\ e(s)\end{pmatrix}$ • $o_1(d(s,p))$: The mean squared error between the outputs of the surrogate and the simulated running time for the training programs is minimized [49, Section 1]. • $o_2\big(\ell\big(s_1^*,x\big)\big)$: The mean squared error of the surrogate on new programs not in the surrogate's original training set is minimized [49, Section 2]. • $o_2\big(d\big(s,s_1^*\big)\big)$: s^* is biased to be close to s_1^* by using the weights from s_1^* as a warm starting point for the optimization problem [49, Section 3]. • $o_2(e(s))$: The surrogate is cheaper to run than simulation [49, Section 1].
Kaya and Hajimirza [45]: 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.	$s_1^* = \arg\min_s o_1\big(d(s,p)\big)$ $\quad s^* = \begin{cases} \arg\min_s o_2\begin{pmatrix}\ell(s_1^*,x),\\ d(s,s_1^*),\\ e(s)\end{pmatrix} \\ \text{subj. to } c_2\big(d(s,p)\big)\end{cases}$ • $o_1(d(s,p))$: The mean squared error between the outputs of the surrogate and simulation on the base material is minimized [45, "Results and Discussion – Base Case"]. • $o_2\big(\ell\big(s_1^*,x\big)\big)$: The error of the outputs of the trained surrogate on the new material is minimized [45, "Results and Discussion – Transfer Cases"]. • $o_2\big(d\big(s,s_1^*\big)\big)$: s^* is biased to be close to s_1^* by using the weights from s_1^* as a warm starting point for the optimization problem [45, "Introduction"]. • $o_2(e(s))$: The surrogate is cheaper to run than simulation [45, "Introduction"]. • $c_2(d(s,p))$: If s^* is less accurate than simulation, then the transfer learning results in low accuracy and is rejected [45, "Results and Discussion – Transfer Cases"].

Table 3. Optimization problem specifications of surrogate optimization from the literature.

Citation and description	Optimization problem specification
Renda et al. [71]: Training neural surrogates of CPU simulators that predict execution time of code, then optimizing parameters of the CPU simulator to more closely match ground-truth execution times measured on real hardware.	$s_1^* = \arg\min_s o(d(s,p))$ $\quad x^* = \arg\min_x o\left(\ell\left(s_1^*, x\right)\right)$ • $o(d(s,p))$: The MAPE between the outputs of the surrogate and the CPU simulator on a given input code snippet is minimized [71, Section III]. • $\ell\left(s_1^*, x\right)$: The MAPE of the output of the trained surrogate induced by the set of simulation parameters is minimized against the ground-truth data [71, Section III].
She et al. [77]: Training neural surrogates of the branching behavior of programs to find inputs that trigger branches that cause bugs in the program.	$s_1^* = \arg\min_s o(d(s,p))$ $\quad x^* = \arg\min_x o\left(\ell\left(s_1^*, x\right)\right)$ • $o(d(s,p))$: The binary cross-entropy error between the output of the surrogate and the actual branching behavior of the program is minimized [77, Section IV.B]. • $\ell\left(s_1^*, x\right)$: Gradient descent tries to find an input that lead to an unseen set of branches taken in the program [77, Section IV.C].
Tseng et al. [87]: Training neural surrogates of camera pipelines, to find parameters for the pipelines that lead to the cameras producing the most photorealistic images.	$s_1^* = \arg\min_s o(d(s,p))$ $\quad x^* = \arg\min_x o\left(\ell\left(s_1^*, x\right)\right)$ • $o(d(s,p))$: The L2 error between the predicted image from the surrogate and the image resulting from the pipeline is minimized [87, Section 4.2]. • $\ell\left(s_1^*, x\right)$: Gradient descent tries to find parameters that lead to images being as similar as possible in L2 distance to the ground-truth [87, Section 4.2].
Shirobokov et al. [78]: Training neural surrogates of physics simulators to find inputs that lead to local optima.	$s_1^* = \arg\min_s o(d(s,p))$ $\quad x^* = \arg\min_x o\left(\ell\left(s_1^*, x\right)\right)$ • $o(d(s,p))$: The error (as measured by a domain-specific loss function per-task) between the outputs of the surrogate and the simulation is minimized [78, Section 2.2]. • $\ell\left(s_1^*, x\right)$: Gradient descent tries to find parameters that lead to local optima in the problem space against the same domain-specific loss function [78, Section 2.2].

Compiling to different hardware. Esmaeilzadeh et al. [23] develop surrogates of small computational kernels, then deploy the surrogates on a hardware accelerator that reduces the latency and energy cost of executing the surrogate. More generally, surrogates can be deployed on any hardware that supports the surrogate architecture, resulting in different trade-offs compared to the CPU architectures that many conventional programs execute on.

Different algorithmic complexity. Algorithmic complexity can differ between a program and its surrogate: for example, while an algorithm may require an exponential number of operations in the size of the input, a surrogate of that algorithm may only require a linear number of operations to approximate the algorithm to satisfactory accuracy [44, 59].

6.4 Surrogate Adaptation

Surrogate adaptation makes it possible to alter the semantics of the program to perform a different task of interest. Surrogate adaptation may be more data-efficient or result in higher accuracy than training a model from scratch [48, 84].

Data efficiency. Tercan et al. [84] develop models that accurately simulate a plastic injection molding process. Tercan et al. train surrogates of computer simulations of injection molding, then adapt the surrogates on real-world experiments of the injection molding process to close the gap between simulation and ground-truth. Tercan et al. show that the surrogate resulting from surrogate adaptation requires less training data than a neural network trained from scratch.

Accuracy. Kustowski et al. [48] learn a model of a physical process involved in nuclear fusion, inertial confinement fusion (ICF). Physical simulation is critical for this area of research, but it is not accurate in part due to unknown biases and inaccuracies in the models of ICF. Kustowski et al. use surrogate adaptation to increase the accuracy of simulators by training a surrogate of simulation then adapting the surrogate on data from physical experiments.

6.4.1 Surrogate Optimization.

Surrogate optimization optimizes inputs faster than optimizing inputs directly against the program, due to the potential for faster execution speed of the surrogate and the potential for the surrogate to be differentiable even when the original program is not [71, 77, 87].

Faster execution time. İpek et al. [42] perform design space exploration on a simulated computer architecture, finding the physical parameters (e.g., cache size, cache associativity, etc.) that lead to the best performance. İpek et al. use surrogate optimization to optimize these parameters, exploiting the significantly faster execution of the surrogate compared to the execution of the original simulation.

Differentiable output domain of programs. She et al. [77] construct neural surrogates of programs for *fuzzing*, generating inputs that cause bugs in the program. For a given input, a classical program has an *execution trace*, the set of edges taken in the control flow graph, which can be represented as a bitvector where 1 denotes that a given edge is taken, and 0 denotes that it is not. She et al. construct a neural surrogate that, for a given input, predicts an approximation of the execution trace of the program with each element between 0 and 1 (rather than strictly set to 0 or 1). This allows for a smooth output of the surrogate, which then allows She et al. to use gradient descent to find inputs that induce a specific execution trace on the original program.

Relaxing the input domain of programs. Grathwohl et al. [30] use neural surrogates to approximate the gradient of non-differentiable functions, in order to reduce the variance of gradient estimators of random variables. Though the input variables are discrete, Grathwohl et al.'s surrogates take continuous values as input, allowing for optimizing these inputs with gradient descent.

7 Design

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

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

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

Domain-agnostic architectures. One design methodology is to use a domain-agnostic architecture for the surrogate, a neural network architecture designed independently of the behavior and domain of application of the program under consideration. A common choice of domain-agnostic architectures for neural surrogates with fixed-size inputs are multilayer perceptrons (MLPs) [42, 77]. In Sections 3 and 4 we use a BERT encoder [20], a type of Transformer [90], which is a common architecture for sequence processing tasks. While simple to design, such domain-agnostic architectures may have high training costs or low accuracy [64, 89].

Domain-specific architectures. An alternative is to design the architecture based on the program and domain under study [71, 87]. However designing such architectures requires manual effort and expertise, both in the original program and in its domain. For instance, our surrogate optimization case study [71] uses a derivative of the architecture proposed by our prior work [58], a model with high accuracy on basic block throughput prediction. This architecture also exploits input sparsity in the simulation: rather than using the entire set of CPU parameters, we only input parameters that influence simulation of instructions in the basic block.

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

Determining the capacity of the neural surrogate trades off between accuracy and execution cost, core tasks in any approximate programming task [80]. Possible approaches include manually selecting the architecture based on reasoning about the complexity of the program [71] and automatically searching for the capacity that leads to the optimal trade-offs among the components of the surrogate's specification [23].

8 Training

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

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

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

Instrumenting the program. One approach is to instrument the execution of the original program and record observed inputs [14, 23]. 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 [71, 84, 87].

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 [79].

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

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

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

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

There are two primary approaches in the literature for determining an appropriate training time of the surrogate. One approach is training for a fixed training time, typically determined via experiments on a validation set [23, 71]. Another approach is training until an acceptable accuracy is reached, whether via a plateau of the training loss [87] or via reaching a minimum acceptable accuracy [84]. Such variable-length training time approaches are discussed in more depth by Goodfellow et al. [27, Chapter 7.8].

Determining the training length for surrogate adaptation is especially important due to the challenges imposed by *catastrophic forgetting* [56, 70], 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 [15, 46, 76, 99]; 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.

9 Deployment

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

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

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

GPUs. Modern large-scale deep neural networks can be executed on GPUs [16], 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 [32, 35, 53].

CPUs. Other applications use a CPU to deploy the surrogate [42]. 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 [32, 36, 52, 53] (though recent work challenges some of these assumptions [18]). CPUs are also more widely available than GPUs, including on edge devices [97].

Machine learning accelerators. Esmaeilzadeh et al. [23] design and deploy a custom neural processing unit (NPU) to accelerate neural surrogates with low latency and energy cost. Other machine learning accelerators offer different trade-offs, such as TPUs increasing throughput even further [43], or the Efficient Inference Engine decreasing energy costs while approximating the surrogate [33].

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

Neural networks require specialized software runtime environments. Choosing the runtime environment requires navigating concerns about both the implementation of the program that uses the surrogate and the deployment of the surrogate across varying devices. Software execution environments include custom frameworks and runtimes which provide bespoke trade-offs for specific applications [23].

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

Normalization. *Data normalization*, which involves pre- and post-processing the inputs and outputs to be suitable for neural networks [51], 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 [75]. Esmaeilzadeh et al. [23] address these issues by integrating the normalization and denormalization steps into the custom hardware (the NPU), eliminating the opportunity for software bugs.

Batching. *Batching*, determining the number of inputs to process at a time, induces a trade-off between latency and throughput for the surrogate. Parrot [23], a surrogate compilation approach that deploys the surrogate to end-users, focuses entirely on latency and uses a single data item in each batch, sacrificing throughput for decreased latency. DiffTune [71], a surrogate optimization approach, has no explicit latency requirements and focuses entirely on throughput, increasing throughput by batching large numbers of training examples into single invocations of the surrogate.

10 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 design patterns in Section 6 are general to all types of surrogate models, the neural surrogate programming methodology in Sections 7 to 9 is specific to when using neural networks as surrogate models. However, other surrogate models are popular in the literature, including surrogates based on Gaussian processes [3, 69], linear models [22, 25], and random forests [38, 62]. Future work in this direction can extend the programming methodology presented in this paper to other classes of surrogate models beyond just neural networks.

More mechanization and systematization. We have presented a programming methodology for developing neural surrogates. However, our methodology is not mechanized: programmers still must manually navigate the trade-off space between desiderata. Future work in this domain should mechanize the various aspects of surrogate construction, from automating the surrogate's design based on the semantics of the original program, to automatically training the surrogate based on specifications and objectives over a data distribution, to automatically integrating the surrogate into the original program's execution context. While prior work has addressed some of these concerns [23], fully mechanizing this process 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 these surrogate-based design patterns are not suitable for. Future work in this domain can more precisely characterize what aspects of a given task admit or preclude surrogates as a candidate solution.

Generalization and robustness. Large-scale neural networks struggle to generalize outside of their training dataset [5, 41, 98]. 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 [68]. To address situations where the neural surrogate is expected to extrapolate outside of its training data, neural surrogate programmers must develop new approaches to recognizing and addressing generalization issues. This may be easier for surrogates of programs than for neural networks in general, because programmers still have access to the original program when developing a surrogate of that program.

Interpretability. Neural networks do not generate explanations for predictions [26], leading to difficulties when reasoning about neural surrogates' predictions. Future work can address these issues by better characterizing what interpretability means for different domains, developing interpretability tools for neural surrogates specifically (again aided by access to the original program), and characterizing when interpretability is and is not a relevant concern for neural surrogates. For example, surrogate optimization uses surrogates as an intermediate artifact to aid another optimization process, where interpretability is less of a concern.

11 Related Work Addressing Similar Tasks

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

Function approximation. Surrogate construction is an instance of function approximation, which encompasses a broad set of techniques ranging from polynomial approximations like the Taylor series to machine learning approaches like Gaussian processes and neural networks [69, 86]. The conventional wisdom is that compared to other approaches, neural networks excel at *feature extraction* [40], 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 [47]. Other function approximation approaches have different trade-offs relative to neural networks, and may be appropriate in circumstances with limited execution cost or data, or when requiring specific bounds on the behavior of the function approximation.

Program repair. Similar to surrogate adaptation, program repair techniques alter the semantics of a program to meet a downstream objective [54, 66, 94]. These approaches typically make local changes to a program in response to a single identified bug. In contrast, surrogate adaptation can change the entire behavior of the program to achieve good performance on a large dataset of examples.

Probabilistic programming. Probabilistic programming is a broad set of techniques for defining probabilistic models, then fitting parameters for these probabilistic models automatically given observations of real-world data [17, 28]. When fitting parameters of a probabilistic program, such techniques require the program to be explicitly specified as a probabilistic program. The parameters are then optimized using inference techniques like Monte Carlo inference [63] and variational inference [11]. In contrast, when optimizing parameters with surrogate optimization the original program can be specified in any form, while the parameters are optimized with stochastic gradient descent.

Differentiable programming. Differentiable programming is a set of techniques that calculates the derivatives of programs with respect to their input parameters [7]. In contrast with estimating the program's gradient with surrogate optimization, differentiable programming calculates the exact derivative without requiring the design and training processes of developing neural surrogates.

While differentiable programming is an appropriate alternative to surrogate optimization in contexts with smooth and continuous original programs, it struggles in cases where the original program is not smooth or is not continuous. For instance, differentiating through control flow constructs like branches and loops results in a discontinuity. Such control flow constructs can also induce a true derivative of 0 almost everywhere, which poses challenges for gradient-based optimization. Differentiable programming also relies on implementing the program in a language amenable to differentiable programming such as Pytorch or TensorFlow [1, 9, 65].

In contrast, surrogate optimization approximates the program regardless of the provenance of its original implementation. This means that while some points in the original program may be non-smooth, discontinuous, or have derivative 0, those points may be better behaved in the surrogate model (which only approximates the original program) allowing for optimizing the inputs with gradient descent despite challenges posed by the original program [71].

Program smoothing. Chaudhuri and Solar-Lezama [13] present a method to approximate numerical programs by executing the programs probabilistically. This approach lets Chaudhuri and Solar-Lezama apply gradient descent to optimize parameters of arbitrary numerical programs, similar to surrogate optimization. However, the semantics presented by Chaudhuri and Solar-Lezama only apply to a limited set of program constructs and do not easily extend to the set of program constructs exhibited by large-scale programs. In contrast, surrogate optimization estimates the gradients of arbitrary programs regardless of the constructs used in the program's implementation.

Automating construction of surrogates. Munk et al. [60] present an approach for automatic construction of neural surrogates of stochastic simulators for surrogate compilation. Munk et al. propose an LSTM architecture that predicts the sequence of samples output by the original stochastic simulator. This approach is applicable to all stochastic simulators, regardless of the number or order of samples output by the original simulator. Munk et al. show that this surrogate executes faster than the original simulator. Though this approach addresses some questions of our neural surrogate programming methodology (specifically, how to design a surrogate for a given program), it does not address questions about how to train and how to deploy the surrogate.

12 Related Work Addressing Other Tasks

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

Surrogates of non-programs. Surrogates of black-box processes (beyond just programs) are used across a wide variety of domains from computer systems to physical sciences [12, 58, 81]. For example, in our prior work [58] we train a surrogate of the execution behavior of Intel CPUs to predict the execution time of code. This is not an example of a surrogate of a program because this is performed without precise knowledge of the execution behavior of the CPU. This 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 [85, 91, 93]. 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. [91] train neural networks that augment programmatic reinforcement learning policies [82]. 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 [6, 91, 92]. For example, after training a residual model, Verma et al. [91] 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.

13 Conclusion

Our work demonstrates the promise of using surrogates to develop complex systems, especially in contexts where programmers lack a full characterization of the system and its operating environment. By identifying the surrogate-based design patterns and describing the methodology used to develop neural surrogates, our work provides a taxonomy for developing surrogates. Our work builds a foundation on which the programming languages community can build new tools that aid in the development of surrogates of programs.

Acknowledgments

We would like to thank Alana Marzoev, Ben Sherman, Cambridge Yang, Charith Mendis, Charles Yuan, Eric Atkinson, 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 provides more details on 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.

We compile llvm-mca in release mode from version 8.0.1, using the version at https://github.com/ithemal/DiffTune/tree/9992f69/llvm-mca-parametric. We invoke llvm-mca a single time and pass it a random sample of 10,000 basic blocks from BHive over stdin, with the following 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 speed is the time from invocation to exit of the `llvm-mca` command.

The neural surrogate is run 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 surrogate uses a batch size of 1. The surrogate is invoked repeatedly by a Python script, and is passed the same 10,000 basic blocks to predict timing values for. The reported execution speed is the time from the invocation of the Python script to its exit.

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 width of 64 results in the fastest BERT model that achieves less than 10% validation MAPE.

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

B BERT Hyperparameter Selection and Training Telemetry

This appendix describes the hyperparameter selection process, the loss curves over the course of training, and the epochs with minimum validation loss for the BERT models 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, available online at https://doi.org/10.6084/m9.figshare.16622839 due to space limitations, we show the training, validation, and test loss curves of the models, along with the total amount of time taken to train all model and the epochs resulting in minimum validation loss.

B.1 Hyperparameters

We base our BERT model on the BERT-Tiny model described by Turc et al. [88], 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 the 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] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *USENIX Conference on Networked Systems Design and Implementation*.

[4] 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

[5] 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

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

[7] 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).

[8] 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

[9] 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

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

[11] 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

[12] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. 2019. Machine learning and the physical sciences. *Reviews of Modern Physics* 91 (2019). Issue 4. https://doi.org/10.1103/RevModPhys.91.045002

[13] 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

[14] 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

[15] 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

[16] 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

[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] Yi Ding, Ahsan Pervaiz, Michael Carbin, and Henry Hoffmann. 2021. Generalizable and Interpretable Learning for Configuration Extrapolation. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3468264.3468603

[23] 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

[24] 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.

[25] Andrew Gelman and Jennifer Hill. 2006. *Data analysis using regression and multilevel/hierarchical models*. Cambridge University Press. https://doi.org/10.1017/CBO9780511790942

[26] 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

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

[28] 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*.

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

[30] 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*.

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

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

[33] 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

[34] 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*.

[35] 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

[36] 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

[37] 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

[38] Tin Kam Ho. 1995. Random Decision Forests. In *International Conference on Document Analysis and Recognition*. https://doi.org/10.1109/ICDAR.1995.598994

[39] 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

[40] 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

[41] 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*.

[42] 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

[43] 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

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

[45] 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

[46] 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

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

[48] 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

[49] 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

[50] 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

[51] 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

[52] 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*. https://doi.org/10.1145/1815961.1816021

[53] 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*. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.76

[54] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2837614.2837617

[55] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *International Conference on Architectual Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/36177.36194

[56] 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

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

[58] 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*.

[59] 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*.

[60] 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]

[61] 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.

[62] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. 2019. HyperMapper: a Practical Design Space Exploration Framework. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. https://doi.org/10.1109/MASCOTS.2019.00053

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

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

[65] 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*.

[66] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically Patching Errors in Deployed Software. In *ACM SIGOPS Symposium on Operating Systems Principles*. https://doi.org/10.1145/1629575.1629585

[67] 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

[68] 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

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

[70] 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

[71] 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

[72] 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

[73] 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

[74] 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

[75] 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)*.

[76] 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*.

[77] 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

[78] 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*.

[79] 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

[80] 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

[81] 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

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

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

[84] 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

[85] 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

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

[87] 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

[88] 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]

[89] 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*.

[90] 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*.

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

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

[93] 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

[94] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2009.5070536

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

[96] 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

[97] 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

[98] 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*.

[99] 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*.

[100] 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*.