Learning to Compile Programs to Neural Networks

Logan Weber [1] **Jesse Michel** [1] **Alex Renda** [1] **Michael Carbin** [1]

Abstract

A *neural surrogate of a program* is a neural network that mimics the behavior of a program. Researchers have used these neural surrogates to automatically tune program inputs, adapt programs to new settings, and accelerate computations. Researchers traditionally develop neural surrogates by training on input-output examples from a single program. Alternatively, language models trained on a large dataset including many programs can consume program text, to act as a neural surrogate. Using a language model to both generate a surrogate and act as a surrogate, however, leading to a trade-off between resource consumption and accuracy. We present *neural surrogate compilation*, a technique for producing neural surrogates directly from program text without coupling neural surrogate generation and execution. We implement neural surrogate compilers using hypernetworks trained on a dataset of C programs and find that they produce neural surrogates that are 1.9-$9.5\times$ as data-efficient, produce visual results that are 1.0-$1.3\times$ more similar to ground truth, and train in 4.3-$7.3\times$ fewer epochs than neural surrogates trained from scratch.

1. Introduction

A *neural surrogate* is a neural network that models a subset of the observable behavior of a program (Renda et al., 2021). Neural surrogates have been used to automatically configure image signal processing units and CPU simulators (Tseng et al., 2019; Renda et al., 2020), improve the accuracy of manufacturing and physics simulations (Tercan et al., 2018; Kustowski et al., 2020), accelerate the computer architecture design process (İpek et al., 2006), and accelerate computations in signal processing, robotics, 3D games, compression, machine learning, and image processing (Esmaeilzadeh et al., 2012a).

[1]MIT CSAIL, Cambridge, MA. Correspondence to: Logan Weber <loganweb@mit.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

Neural Surrogate Training. The research community has developed a variety of techniques to train neural surrogates. The traditional approach is to train a neural surrogate of a single program by collecting and curating a dataset of input-output pairs and then training a neural network to predict the program's output given an input (Renda et al., 2021).

Another point in the spectrum is to amortize the cost of training neural surrogates by training a *universal neural surrogate*: a neural network that directly consumes the text of a program and predicts the program's output for a given input (Zaremba & Sutskever, 2015; Nye et al., 2021; Gu et al., 2024). A key benefit of universal neural surrogates is that one only needs to create a dataset once. Once trained, a universal neural surrogate can act as the neural surrogate of a given program without the need to curate a dataset of program-specific, input-output pairs.

However, universal neural surrogates necessarily use the same model to process the program text as is used to predict the program output, and accurate prediction may require multiple forward passes (Nye et al., 2021; Wei et al., 2022). These limitations pose challenges for deploying such a model as a neural surrogate because small models may not be able to emulate complex programs (Zaremba & Sutskever, 2015) and large models (OpenAI et al., 2023) may not be able to execute in the resource-constrained environments where neural surrogates have been used (Esmaeilzadeh et al., 2012a; Mendis, 2020; Munk et al., 2022).

Our Approach: Neural Surrogate Compilation. To maintain the benefits of universal neural surrogates while bypassing the above limitations, we propose to use a *neural surrogate compiler*. A neural surrogate compiler is a system that accepts a program's text as input and produce an initial neural surrogate of the program, which can vary in behavioral quality. Similarly to a traditional compiler, a neural surrogate compiler requires a significant upfront cost that is amortized over the generation of initializations for many neural surrogates. We demonstrate in this work that when compared to the traditional approach of training a neural surrogate from a random initialization, neural surrogates produced by neural surrogate compilers can be finetuned to closely mimic the behavior of the program at a lower cost, as measured in data efficiency and training time.

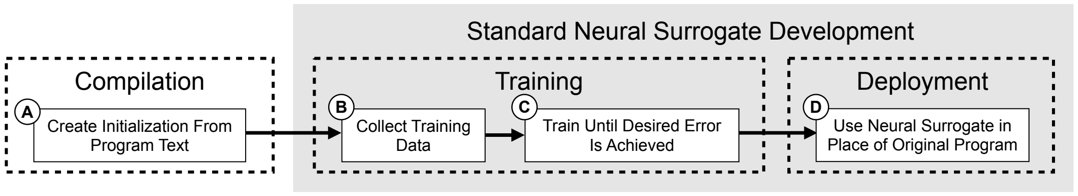

Standard Neural Surrogate Development
Compilation
A
Create Initialization From Program Text
Training
B
Collect Training Data
C
Train Until Desired Error Is Achieved
Deployment
D
Use Neural Surrogate in Place of Original Program

Figure 1: Neural surrogate development with neural surrogate compilation

Contributions. To implement a neural surrogate compiler, we adapt the BERT architecture (Turc et al., 2019) into a *hypernetwork*—a hypernetwork is a neural network that produces the parameters of another neural network (Ha et al., 2017). We name the resulting architecture COMPNET.

To train neural surrogate compilers, we develop EXESTACK, a dataset of 69,083 executable C programs collected from The Stack (Kocetkov et al., 2022), a large corpus of source code. To train COMPNETs, we refine EXESTACK into EXESTACKCPN, a dataset of 37,772 programs that is compatible with our chosen hypernetwork architecture. We then evaluate neural surrogates initialized via COMPNET on EXESTACKCPN and PARROTBENCHCPN, the latter being a set of benchmarks from prior work in approximate computing (Esmaeilzadeh et al., 2012a).

Surrogates trained from COMPNET initializations achieve 1.9-9.5$\times$ lower error than neural surrogates trained from scratch, with the same amount of data; on a color quantization task, they produce images that are 1.0-1.3$\times$ more similar to images produced by an exact implementation than images produced by surrogates trained from random initialization; and they achieve a target error with 4.3-7.3$\times$ fewer epochs than neural surrogates trained from scratch.

2. Neural Surrogate Compilation

A *neural surrogate compiler* is a system that is specialized to a family of neural surrogate architectures to accept a program's text as input and produce an initial neural surrogate of the program. Figure 1 presents the neural surrogate compilation workflow alongside the traditional workflow for developing a neural surrogate. In a traditional neural surrogate development workflow, one collects training data (Ⓑ), trains the neural surrogate until its error meets the desired threshold (Ⓒ), and then uses it in place of the original program (Ⓓ). Neural surrogate compilation (Ⓐ) introduces a new, initial step in the neural surrogate compilation workflow in which a neural surrogate compiler maps the program text to a neural network initialization for use in the training of the neural surrogate. The typical strategy to train a neural surrogate is through supervised learning of a neural network with a curated dataset of input-output pairs from the program (Renda et al., 2021).

In this section, we formalize the problem of efficiently training a neural surrogate and introduce a new approach to solving this problem using a neural surrogate compiler.

2.1. The Efficient Surrogate Training Problem

We first formalize the problem of training a neural surrogate. We assume we are given a program text $p : \mathcal{P}$ that denotes a function $[\![p]\!] : \mathcal{I}_p \to \mathcal{O}_p$,[1] where $\mathcal{P}$ is the space of programs under consideration, $\mathcal{I}_p$ is the type of values p accepts as input and $\mathcal{O}_p$ is the type of values p produces as output. We also assume a target neural surrogate architecture $a : \mathbb{R}^d \to \mathcal{I}_p \to \mathcal{O}_p$, which takes a set of parameters $\theta : \mathbb{R}^d$ and produces a surrogate function from $\mathcal{I}_p$ to $\mathcal{O}_p$. The goal is to find a set of parameters $\theta : \mathbb{R}^d$ such that the neural surrogate $a(\theta) : \mathcal{I}_p \to \mathcal{O}_p$ has low approximation error:

$$\forall i : \mathcal{I}_p.\, a(\theta)(i) \approx [\![p]\!](i)$$

To measure the quality of surrogate outputs, we use a loss function $\ell : \mathcal{O}_p \times \mathcal{O}_p \to \mathbb{R}_{\geq 0}$ that measures the difference between the output of the program and the output of the surrogate. To measure overall surrogate quality, we use the expected loss over a distribution of inputs:

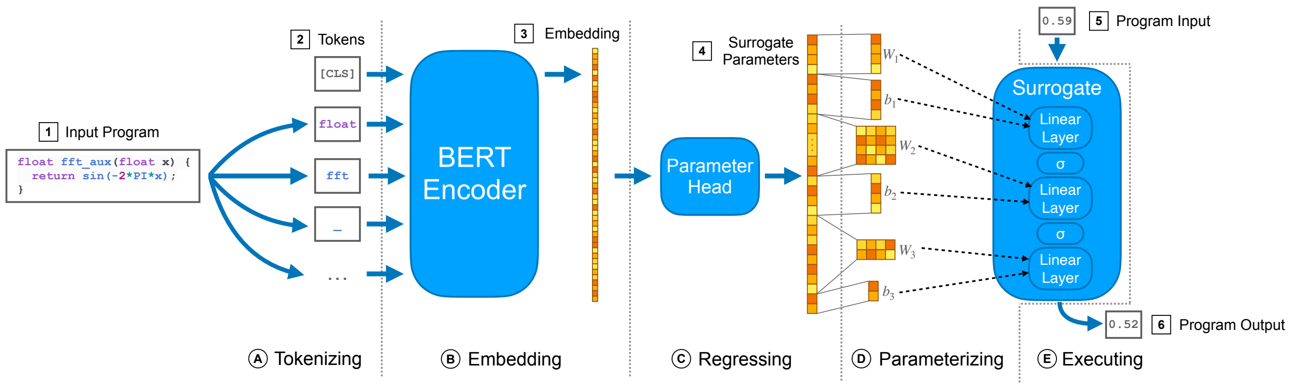
$$\mathcal{L}(a(\theta), p) = \mathbb{E}_{i \sim \mathcal{I}_p}[\ell(a(\theta)(i), [\![p]\!](i))] \tag{1}$$

As with most learning problems, a challenge in training neural surrogates is that the error of a surrogate depends on the budget dedicated to collecting training data (input-output pairs of the program) and the number of epochs used to train the surrogate. We formalize these costs by defining a *training procedure* $t_a : \mathcal{P} \times \mathbb{R}_{\geq 0} \times \mathbb{N} \to \mathbb{R}^d$ for a given surrogate architecture a as a random function that takes program text p, a training data budget $b : \mathbb{R}_{\geq 0}$, and training time budget $n : \mathbb{R}_{\geq 0}$ and produces a set of parameters $\theta : \mathbb{R}^d$ for the surrogate.

We then define the *efficient surrogate training problem* as finding a training procedure t_a for a given program p, architecture a, sample budget b, training time budget n, and loss function ℓ that minimizes the expected loss of the resulting surrogate:

$$\arg\min_{t_a} \mathbb{E}_{\theta \sim t_a(p,b,n)}[\mathcal{L}(a(\theta), p)],$$

[1] $[\![\cdot]\!] : \mathcal{P} \to (\mathcal{I}_p \to \mathcal{O}_p)$ is notation used in programming language theory to refer to the function a program implements.



```
float fft_aux(float x) {
  return sin(-2*PI*x);
}
```

Figure 2: System diagram describing the CompNet architecture, comprising five phases: (A) tokenizing an input program, (B) embedding the program using a BERT encoder, (C) regressing the embeddings to a parameter vector using a parameter head, (D) parameterizing a neural network using the parameter vector, and (E) executing the neural network surrogate.

The standard approach to training a neural surrogate is to randomly initialize the parameters of the surrogate and then use a gradient-based optimization algorithm to minimize the loss against a dataset of input-output pairs from the program (Renda et al., 2021).

2.2. Neural Surrogate Compilation

A neural surrogate compiler is a system $\phi : (p : \mathcal{P}) \to \mathbb{R}^{d_p}$ that accepts program text p and produces parameters $\theta \in \mathbb{R}^{d_p}$ for a neural surrogate architecture a_p depending on the program p. We use a neural surrogate compiler to solve the efficient surrogate training problem.

We formalize the development of a neural surrogate compiler as an optimization problem. The goal is to develop a system ϕ such that for every program p, the surrogate $f = a_p(\phi(p))$ can be trained efficiently. Optimizing for a system that generates surrogates that can be trained efficiently is challenging. As a simple proxy, we optimize for a system that generates surrogates that achieve low loss:

$$\underset{\phi \in \mathcal{P} \to \mathbb{R}^d}{\arg\min}\ \mathbb{E}_{p \sim \mathcal{P}}[\mathcal{L}(a_p(\phi(p)), p)].$$

3. CompNet

The CompNet architecture is an implementation of a neural surrogate compiler using hypernetworks. We explain the architecture, how to train it, then how to extract neural surrogates from its outputs.

3.1. Architecture

Figure 2 presents the design of a CompNet. A CompNet accepts program text $p : \mathcal{P}$ as input and produces parameters $\theta \in \mathbb{R}^d$ for a neural surrogate architecture $a : \mathbb{R}^d \to \mathcal{I} \to \mathcal{O}$ with as many inputs as the largest architecture one wishes to compile to and a single output. We call this architecture a *covering architecture*.

Ⓐ First, CompNet tokenizes an input program ($\boxed{1}$), resulting in a sequence of tokens ($\boxed{2}$) including the distinguished BERT *classification token* `[CLS]`.

Ⓑ CompNet then uses a BERT encoder (Devlin et al., 2019) to embed the sequence of tokens, resulting in an embedding per token. The output of this step is the embedding of the classification token ($\boxed{3}$); CompNet discards the embeddings of the other tokens.

Ⓒ Next, CompNet uses a *parameter head*, implemented as a single linear layer, to map the classification token embedding to a neural surrogate parameter vector ($\boxed{4}$).

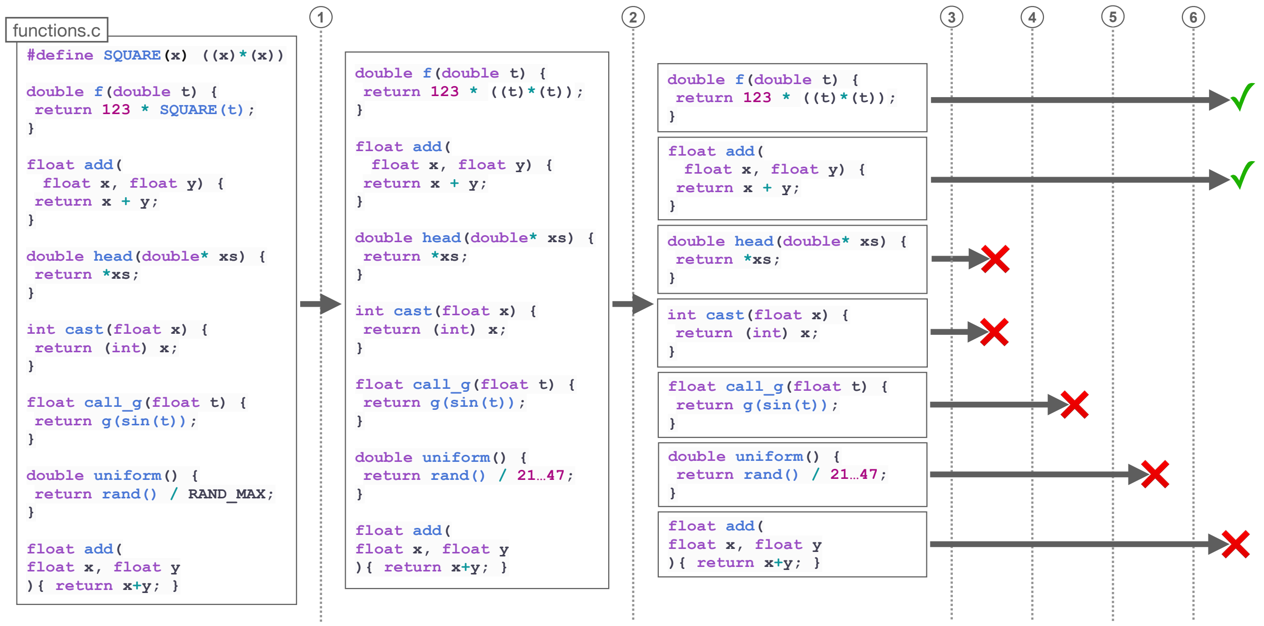
Ⓓ Then, CompNet interprets the vector of parameters as the weights and biases of the covering architecture. The output of this step is a neural surrogate of the input program.

Ⓔ Finally, CompNet executes the neural surrogate with the interpreted parameters on a program input ($\boxed{5}$) to produce a prediction of the program output ($\boxed{6}$).

3.2. Training

Training a CompNet requires a dataset of programs and input-output pairs for each program. Note that this dataset is not considered as part of the budget in the efficient surrogate training problem, since it is amortized over all programs the CompNet is used to compile.

Each step of training proceeds by selecting a batch of programs and input-output pairs for those programs, generating neural surrogate parameters for each program, interpreting the neural surrogate parameters as parameters for the covering architecture, executing each neural surrogate with the batch of inputs, then calculating the loss between the



```c
#define SQUARE(x) ((x)*(x))

double f(double t) {
 return 123 * SQUARE(t);
}

float add(
  float x, float y) {
 return x + y;
}

double head(double* xs) {
 return *xs;
}

int cast(float x) {
 return (int) x;
}

float call_g(float t) {
 return g(sin(t));
}

double uniform() {
 return rand() / RAND_MAX;
}

float add(
float x, float y
){ return x+y; }
```

```c
double f(double t) {
 return 123 * ((t)*(t));
}

float add(
  float x, float y) {
 return x + y;
}

double head(double* xs) {
 return *xs;
}

int cast(float x) {
 return (int) x;
}

float call_g(float t) {
 return g(sin(t));
}

double uniform() {
 return rand() / 21…47;
}

float add(
float x, float y
){ return x+y; }
```

```c
double f(double t) {
 return 123 * ((t)*(t));
}
```
✓

```c
float add(
  float x, float y) {
 return x + y;
}
```
✓

```c
double head(double* xs) {
 return *xs;
}
```
✗

```c
int cast(float x) {
 return (int) x;
}
```
✗

```c
float call_g(float t) {
 return g(sin(t));
}
```
✗

```c
double uniform() {
 return rand() / 21…47;
}
```
✗

```c
float add(
float x, float y
){ return x+y; }
```
✗

Figure 3: The EXESTACK generation pipeline. Starting with C source files from The Stack, we apply a sequence of maps followed by a sequence of filters. The steps are (1) run the C preprocessor, (2) extract functions from the source file, (3) remove functions with pointers in their type signature and nonnumeric functions, (4) remove nonexecutable functions and collect input-output pairs, (5) remove nondeterministic functions, and (6) remove any duplicate programs. Red "X"s denote that a function does not pass a filter and green checkmarks denote that a function passes all filters.

neural surrogates' predicted outputs and the true outputs. To match the signature of the covering architecture, the batch of inputs is padded out to match the number of inputs for the covering architecture (e.g., if a covering architecture has 9 inputs and a program has 3 inputs, the compiled architecture for that program is fed 9 inputs). For padding, we use inputs drawn from the same distribution as the program inputs (see Appendix N for details).

Backpropagation proceeds as usual, except that one does not update the parameters of the neural surrogates, since each generated neural surrogate is ephemeral. Instead, backpropagation only updates the parameters of the COMPNET. Appendix E contains additional training details.

3.3. Surrogate Extraction

The output of a COMPNET is parameters for the covering architecture, which might not match the number of inputs and outputs of the program being compiled. To adapt the covering architecture to the target number of inputs, one finetunes the resulting architecture on data where the excess inputs are set to zero, allowing one to then remove the weights in the input layer corresponding to the excess inputs (see Appendix N for details on this choice). To adapt the covering architecture to the target number of outputs, one clones the weights for the single output in the output layer for each new output that is needed (see Appendix O for details on this choice). To align the program text with the training distribution (i.e., single-output programs), one also modifies the input program to produce a single output (e.g,. the first output of the original program). When neither the number of inputs nor the number of outputs matches the covering architecture, all of the above modifications are applied in the same finetuning run.

4. EXESTACK

The strategy we presented in Section 3 for learning a neural surrogate compiler requires a dataset of programs and input-output examples describing the behavior of each program. To meet this requirement, we developed EXESTACK, a dataset of 69,083 pointer-free, numerical, executable, deterministic C programs and corresponding input-output examples. EXESTACK is based on The Stack (Kocetkov et al., 2022), a dataset of 3 TB of permissively licensed source code written in various programming languages scraped from GitHub.

Figure 3 summarizes the process of generating EXESTACK (see Appendix B for details). The restriction to pointer-free functions simplifies the EXESTACK generation methodology, and yet, a model trained on EXESTACK could still handle programs using statically-sized data structures containing numeric data (e.g., arrays), as they can be transformed into functions with a fixed number of arguments.

Benchmark	Description	Train Inputs	Test Inputs	#Inputs	#Outputs
`fft`	Radix-2 Cooley-Tukey fast Fourier transform	32,768 random floating point numbers	2,048 random floating point numbers	1	2
`invk2j`	Inverse kinematics for 2-joint arm	10,000 random (x, y) coordinates	10,000 random (x, y) coordinates	2	2
`kmeans`	*k*-means clustering	50,000 random (r, g, b) values	220x200 color image	6	1
`sobel`	Sobel edge detector	One 512x512 color image	220x200 color image	9	1

Table 1: The programs from PARROTBENCH we include in PARROTBENCHCPN (Esmaeilzadeh et al., 2012a).

5. Evaluation

To evaluate the claim that neural surrogate compilation lowers the development cost of neural surrogates, we answer the following research questions.

RQ 1: Does a neural surrogate initialized by a COMPNET converge to a lower test loss than a neural surrogate initialized randomly, for a fixed training set size?

RQ 2: Does a neural surrogate initialized by a COMPNET produce better results in an application than a neural surrogate initialized randomly, for a fixed training set size?

RQ 3: Does a neural surrogate initialized by a COMPNET converge to a target test loss in fewer epochs than a neural surrogate initialized randomly?

Our results demonstrate that COMPNETs lead to improvements in data efficiency (Section 5.2), perceptual quality (Section 5.3), and training time (Appendix J).

5.1. Methodology

To develop and evaluate COMPNETs, we select a BERT architecture for the neural surrogate compiler and a multilayer perceptron for the covering architecture, we produce datasets that COMPNETs can be trained and evaluated on, we introduce alternative initialization methods to compare against, and we finetune surrogates produced by each of the initialization methods.

5.1.1. COMPNET ARCHITECTURE

We use the BERT-Tiny architecture (Turc et al., 2019) for the BERT encoder in COMPNET, and we adapt a neural surrogate architecture from Esmaeilzadeh et al. (2012a) into a covering architecture for this COMPNET.

The architecture used by Esmaeilzadeh et al. (2012a) is a multilayer perceptron consisting of a single input, a hidden layer of 4 neurons, another hidden layer of 4 neurons, and 2 outputs, and it uses a sigmoid activation function. For their evaluation, the authors introduce a suite of benchmarks, PARROTBENCH, consisting of numerical programs from various domains. The authors apply their techniques to the architecture above on a fast Fourier transform benchmark in PARROTBENCH and achieve a $3.6\times$ speedup. This architecture therefore places a floor on the system speedup that motivates our investigation of Parrot, in that the architectures Esmaeilzadeh et al. (2012a) use for all other programs in PARROTBENCHCPN are at least as computationally expensive as the one we choose. We adapt this architecture to take in 9 inputs and produce 1 output, so it can be used to compile programs with up to 9 inputs, and so it is compatible with EXESTACK.

As the COMPNET loss function, we use mean squared error (MSE) between predicted and true outputs.

5.1.2. DATASETS

We evaluate the effectiveness of COMPNETs on test programs from EXESTACKCPN and programs from PARROTBENCHCPN (see Table 1). These datasets are refinements of EXESTACK and PARROTBENCH that are compatible with the instantiation of the COMPNET architecture described above.

EXESTACKCPN. We produce EXESTACKCPN by applying additional filters to EXESTACK, resulting in 37,772 programs. See Appendix C for details.

From the full set of programs, we create a training, validation, and testing set using an 80/10/10 split. Each program has input-output examples, so we additionally create a training and testing set for these examples using a 50/50 split. In Sections 5.2 and Appendix J, we evaluate performance on EXESTACKCPN using 1,000 programs from the testing set.

PARROTBENCHCPN. PARROTBENCHCPN programs come from a diverse set of application domains, they are all written in C, each consists of a single function, and they are numeric in nature, making them suitable for evaluating COMPNETs. Table 1 shows the programs in PARROTBENCHCPN, including descriptions of the computations and input datasets. In Appendix D, we explain how we chose these programs, we list the program source, and we explain how we generated input datasets.

Downcasting Error. The covering architecture we chose uses a single-precision floating-point data type, but some programs in EXESTACKCPN use double-precision floating-point data types. In Appendix M, we explain why compiling programs with double-precision data types to the single-precision covering architecture incurs negligible error.

5.1.3. ALTERNATIVE INITIALIZATION METHODS

Besides random initialization, we compare COMPNETs to two alternative initialization methods: model-agnostic meta learning (Finn et al., 2017) and pretrained initializations. Neither initialization method conditions on program text, so they both result in constant initializations that one uses for every program. We briefly describe these techniques here and how we train them, and we provide shorthand for referencing each initialization method. In Appendix A, we survey related work in this area in detail.

Model-Agnostic Meta Learning. Model-agnostic meta learning (MAML) is a meta-learning technique for producing neural network initializations that can be quickly finetuned to achieve low error on a given task. One trains MAML initializations by sampling tasks from some space of training tasks, finetuning on them, and backpropagating through the finetuning process into the initialization.

Pretrained Neural Surrogates. A simpler alternative to MAML is to train a single neural surrogate on the union of all input-output examples from programs in a dataset such as EXESTACKCPN. We call initializations trained in this way *pretrained neural surrogates*.

Training. We train 3 instances of each initialization method on EXESTACKCPN training programs using the same covering architecture as COMPNETs. See Appendices F, G, N, and O for details on MAML training, pretrained surrogate training, variable-input support, and variable-output support, respectively.

Initialization Method Shorthand. We use shorthand names for each initialization method in figures. We refer to COMPNETs as “CPN”, MAML as “MAML”, pretrained surrogates as “PTS”, and random initialization as “RND”.

5.1.4. FINETUNING SURROGATES

Here, we collect the finetuning methodology for surrogates in this evaluation, including optimization methods, hyperparameters, random seed behavior, and how we measure the improvements achieved by these surrogates.

For all surrogates produced by the initialization methods we consider, we use the following finetuning methodology. We use the Adam optimizer with no weight decay, a learning rate of 0.01, and MSE as the loss function. The only difference between our methodology and the methodology of Esmaeilzadeh et al. (2012a) is that we use the Adam optimizer instead of stochastic gradient descent, and we use the He initialization method (He et al., 2015)—they do not specify how they initialize their neural surrogates.

We use 9 trials with different random seeds for every configuration in the experiments of Section 5.2 and Appendix J. Note that, for COMPNET, MAML, and pretrained surrogate initializations, changing random seeds only changes the training data order, since the initialization is deterministic.

For data efficiency and training time, we quantify results using geometric mean improvements over random initialization. These are only relative measures, so in Appendix L, we demonstrate the neural surrogates we train achieve sufficiently low absolute error for downstream applications.

5.2. Data Efficiency Improvements

To assess whether COMPNETs improve data efficiency, we use COMPNETs to initialize neural surrogates, finetune on subsets of training data of various sizes, and then compare the results to those of other initialization methods. We detail the methodology of this experiment then present results.

5.2.1. METHODOLOGY

We now describe the configurations we sweep over and the methodology we use to finetune surrogates.

Experiment Configurations. In this experiment, we sweep over configurations consisting of a program, a dataset size, and an initialization method (e.g., a COMPNET). Each dataset size specifies the percentage of the training data to train neural surrogates on. We sweep over the following percentages: $\{0\%, 0.1\%, 1\%, 10\%, 100\%\}$.

Dataset Selection. Given a configuration consisting of a program, a dataset size percentage $c \in [0, 1]$, and an initialization method, we select a random subset $\mathcal{D}_{\text{sub}}$ of the training data $\mathcal{D}_{\text{train}}$ of size $c|\mathcal{D}_{\text{train}}|$. We use an $80/20$ split to divide $\mathcal{D}_{\text{sub}}$ into train and validation sets $\mathcal{D}_{\text{sub train}}$ and $\mathcal{D}_{\text{sub val}}$. We sample 9 different subsets of this size and use a different training seed for each subset, yielding 9 trials total.

Finetuning. For each trial, we initialize a neural surrogate according to the initialization method. We then train on $\mathcal{D}_{\text{sub train}}$ for $5{,}000$ epochs. The final test loss we report for a trial is the test loss at the epoch closest to the epoch with the lowest validation error.[2] When the dataset size is 0%, we use the test loss at the final epoch.

[2]We only compute test loss before training, after every 3 epochs of training, and after training.

Statistic	CPN	MAML	PTS
0th	$6.36 \cdot 10^{-8}\times$	$4.68 \cdot 10^{-6}\times$	$\mathbf{1.35 \cdot 10^{-4}}\times$
25th	$\mathbf{1.23}\times$	$0.87\times$	$0.76\times$
50th	$\mathbf{5.84}\times$	$1.17\times$	$1.28\times$
75th	$\mathbf{54.36}\times$	$1.71\times$	$2.66\times$
100th	$\mathbf{4.43 \cdot 10^{7}}\times$	$8.52 \cdot 10^{3}\times$	$7.14 \cdot 10^{4}\times$
MPI	**21**st	35th	37th
GM	$\mathbf{9.50}\times$	$1.09\times$	$1.08\times$

Dataset Size	CPN	MAML	PTS
0%	$\mathbf{84.40}\times$	$1.42\times$	$2.63\times$
0.1%	$\mathbf{10.43}\times$	$0.91\times$	$0.87\times$
1%	$\mathbf{2.90}\times$	$0.51\times$	$0.90\times$
10%	$\mathbf{4.12}\times$	$1.54\times$	$0.88\times$
100%	$\mathbf{6.67}\times$	$1.53\times$	$0.76\times$

Figure 4: Geometric mean test loss improvement over random initialization on 1,000 ExeStackCPN test programs, taken over all programs and dataset sizes (left) and grouped by dataset sizes (right). The table on the left reports improvements at a sample of percentiles from 0th (performance that is the worst compared to random initialization) to 100th (performance that is the best compared to random initialization), reports the minimum percentile at which an initialization method improves over random initialization (MPI), and reports overall geometric mean improvements (GM).

Stat.	CPN	MAML	PTS
0th	$0.22\times$	$\mathbf{0.28}\times$	$0.23\times$
25th	$\mathbf{0.88}\times$	$0.82\times$	$0.75\times$
50th	$\mathbf{1.23}\times$	$0.97\times$	$0.97\times$
75th	$\mathbf{2.96}\times$	$1.14\times$	$1.26\times$
100th	$\mathbf{106.91}\times$	$1.99\times$	$38.18\times$
MPI	**36**th	54th	54th
GM	$\mathbf{1.91}\times$	$0.93\times$	$1.05\times$

Program	CPN	MAML	PTS
`fft`	$\mathbf{1.47}\times$	$0.98\times$	$0.61\times$
`invk2j`	$1.01\times$	$\mathbf{1.07}\times$	$1.05\times$
`kmeans`	$\mathbf{7.85}\times$	$0.68\times$	$2.24\times$
`sobel`	$\mathbf{1.14}\times$	$1.06\times$	$0.85\times$

% Data	CPN	MAML	PTS
0%	$\mathbf{1.81}\times$	$0.90\times$	$1.56\times$
0.1%	$\mathbf{1.98}\times$	$0.94\times$	$0.98\times$
1%	$\mathbf{1.77}\times$	$0.93\times$	$0.79\times$
10%	$\mathbf{2.38}\times$	$1.11\times$	$1.23\times$
100%	$\mathbf{1.68}\times$	$0.81\times$	$0.86\times$

Figure 5: Geometric mean test loss improvement over random initialization on ParrotBenchCPN, taken over all programs and dataset sizes (left), grouped by programs (middle), and grouped by dataset sizes (right). The top table reports improvements at a sample of percentiles from 0th to 100th, reports the minimum percentile at which an initialization method improves over random initialization (MPI), and reports overall geometric mean improvements (GM).

Quantifying Improvements. We define the improvement for a given configuration (consisting of an initialization method, program, and dataset size) as the ratio of the test loss achieved by random initialization on that configuration and the test loss achieved by that configuration. We average all test losses over trials and instances of an initialization methods (using arithmetic mean) prior to computing ratios. For example, we train 3 instances of CompNets, and for each instance, we perform 9 surrogate finetuning trials, so we compute an average over 27 items. For each initialization method, we report the geometric mean of the improvements grouped by program, grouped by dataset size, and overall. For some programs and initialization methods, the resulting surrogates achieve losses of 0. We discard these results before computing the geometric mean[3].

In some figures, we present the improvement at various percentiles—from 0th to 100th—as well as the minimum percentile of improvement (MPI). The percentiles from 0th to 100th show the performance that is the worst compared to random initialization up to performance that is the best compared to random initialization. The MPI is the minimum percentile at which an initialization method improves over random initialization.

[3] We discard 2.4% of entries total for ExeStackCPN programs and 0% of entries total for ParrotBenchCPN programs. For ExeStackCPN programs, we discard 3.5% of CompNet entries, 0% of MAML entries, 4.5% of pretrained surrogate entries, and 0% of randomly initialized surrogate entries.

5.2.2. Results

Figures 4 and 5 show finetuning results for a sample of 1,000 ExeStackCPN test programs and ParrotBenchCPN, respectively. See Appendix H for the test losses used to compute improvements.

ExeStackCPN Test Programs. CompNets achieve the best results on average, with a $9.50\times$ improvement over random initialization, whereas MAML and pretrained surrogates achieve only a $1.09\times$ and $1.08\times$ improvement on average. CompNets improve over random initialization in as low as the 21st percentile of configurations, whereas MAML and pretrained surrogates improve over random initialization after the 35th and 37th percentiles, respectively.

CompNets improve on ExeStackCPN test programs most prominently in the zero-shot regime, where the improvement is $84.40\times$ over random initialization, whereas MAML and pretrained surrogates achieve improvements of $1.42\times$ and $2.63\times$, respectively. The zero-shot regime

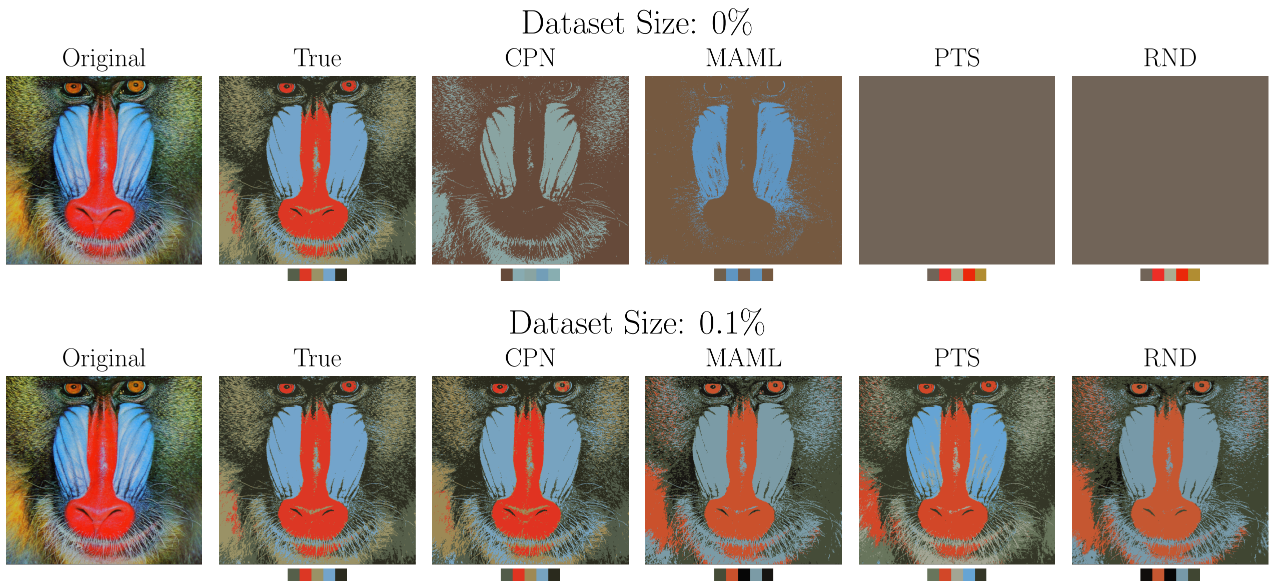


Figure 6: Color quantization results for a ground-truth NumPy implementation ("True") vs. approximate implementations. The original image of a baboon is on the left, followed by images transformed to adhere to a palette of 5 colors.

is also the only regime where pretrained surrogates show an improvement. The worst performance for both CompNets and MAML is in the middle of the dataset sizes we evaluated, at a dataset size of 1%, where they achieved 2.90× and 0.51×, respectively. The worst performance for pretrained surrogates, however, is at a dataset size of 100%, where they achieve a 0.76× improvement.

ParrotBenchCPN Programs. CompNets achieve the best results on average, achieving a 1.91× improvement over random initialization, whereas MAML worsened performance (0.93×) and pretrained surrogates slightly improved performance (1.05×). CompNets improve over random initialization in as low as the 36th percentile of configurations, whereas MAML and pretrained surrogates both improve over random initialization after the 54th percentile.

CompNets improve or do not worsen data efficiency on each ParrotBenchCPN program, with the smallest improvement on `invk2j` (1.01×) and the largest improvement on `kmeans` (7.85×). MAML shows the largest improvement on `invk2j` (1.07×) but worsens performance on `fft` and `kmeans`, achieving 0.98× and 0.68×, respectively. Pretrained surrogates show the largest improvement on `kmeans` (2.24×), but they worsen performance on `fft` and `sobel`, achieving 0.61× and 0.85×, respectively.

Unlike the results for ExeStackCPN, the improvement due to CompNets is greatest near the middle of the dataset sizes we evaluated over. The greatest improvement of 2.38× occurs at 10%, and the smallest improvement of 1.68× occurs at 100%. MAML worsens performance at most dataset sizes, except at 10%, where it achieves a 1.11× improvement over random initialization. Pretrained surrogates worsen performance at most dataset sizes except 0% and 10%, where they achieve 1.56× and 1.23×, respectively.

Since CompNets improve data efficiency over random initialization on both ExeStackCPN and ParrotBenchCPN, we answer yes to RQ 1.

5.3. Neural Surrogates for Color Quantization

To assess whether a CompNet can improve the quality of results in an end-to-end application, we use a trained CompNet to initialize a neural surrogate used for *color quantization* and compare it to other initialization methods (Kanungo et al., 2002). Color quantization is the process of reducing the number of distinct colors in an image. For example, Figure 6 depicts an image of a baboon color quantizated to five colors.

5.3.1. Methodology

We follow the methodology for color quantization from Kanungo et al. (2002) who apply k-means clustering to the (R, G, B) vectors representing the colors of pixels of an image and select the cluster centroids as the colors in the palette. We run k-means clustering for 40 iterations or until the distance between the old centroids and new centroids is less than $1 \cdot 10^{-5}$. Each pixel color is then remapped to the closest color in the palette.

We use the Euclidean distance function to compute the distance between two RGB vectors. We consider both

Dataset Size	CPN	MAML	PTS	RND
0%	$\mathbf{2.67 \cdot 10^3} \pm 541.$	$2.79 \cdot 10^3 \pm 347.$	$3.04 \cdot 10^3 \pm 63.0$	$3.05 \cdot 10^3 \pm 0.00$
0.1%	$\mathbf{984.} \pm 733.$	$1.79 \cdot 10^3 \pm 554.$	$1.73 \cdot 10^3 \pm 725.$	$1.43 \cdot 10^3 \pm 544.$
1%	$\mathbf{528.} \pm 219.$	$782. \pm 300.$	$760. \pm 256.$	$619. \pm 249.$
10%	$\mathbf{452.} \pm 222.$	$717. \pm 212.$	$690. \pm 195.$	$782. \pm 307.$
100%	$\mathbf{504.} \pm 220.$	$766. \pm 189.$	$699. \pm 171.$	$655. \pm 121.$

Dataset Size	CPN	MAML	PTS	RND
0%	$\mathbf{0.33} \pm 0.11$	0.26 ± 0.03	0.25 ± 0.02	0.25 ± 0.0
0.1%	$\mathbf{0.61} \pm 0.15$	0.45 ± 0.12	0.47 ± 0.16	0.53 ± 0.09
1%	$\mathbf{0.72} \pm 0.12$	0.64 ± 0.11	0.65 ± 0.10	0.70 ± 0.11
10%	$\mathbf{0.76} \pm 0.12$	0.64 ± 0.09	0.64 ± 0.08	0.63 ± 0.08
100%	$\mathbf{0.73} \pm 0.13$	0.62 ± 0.08	0.64 ± 0.05	0.65 ± 0.06

Figure 7: Quantitative comparison of end-to-end results produced by various initialization methods on color quantization for a 5-color palette. **(Top)** The average MSE of the image produced by each initialization method compared to the image produced by a ground-truth implementation of the `kmeans` kernel (lower is better). **(Bottom)** The average SSIM of the image produced by each initialization method compared to the image produced by a ground-truth implementation of the `kmeans` kernel (higher is better).

a reference NumPy implementation and approximate implementations given by neural surrogates of the `kmeans` kernel in PARROTBENCHCPN (Harris et al., 2020).

We use surrogates from the data efficiency evaluation of Section 5.2. For visual comparisons, we choose a single surrogate for each dataset size and initialization method. Since here we evaluate on a distinct image from the testing set of the `kmeans` kernel, using the testing set as a validation set does not constitute data leakage, so we choose the surrogates with the lowest test losses. For quantitative comparisons, we aggregate over all surrogates and no selection criterion is necessary.

We quantify the similarity between NumPy-quantized images and surrogate-quantized images using both MSE and the structural similarity index measure (SSIM), the latter of which provides a quantitative model for the percieved similarity of images (Wang et al., 2004).

5.3.2. RESULTS

Figure 6 depicts the result of applying 5-color quantization to an image of a baboon using surrogates trained on dataset sizes of 0% and 0.1% of the training set. Figure 7 shows quantitative results comparing initialization methods on 5-color quantization at various dataset sizes. Each entry shows the average and standard deviation of a metric over all trials and instances of an initialization method. See Appendix I for more dataset sizes and color palette sizes.

Visual Results. At a dataset size of 0%, COMPNET- and MAML-initialized surrogates are the only surrogates that produce images with detail. The image produced by a COMPNET surrogate shows more detail than the image produced by a MAML surrogate, which primarily captures details on the nose. At a dataset size of 0.1%, all initialization methods produce images that resemble the original image. Images produced by COMPNET-initialized and pretrained surrogates have a higher contrast than images produced by MAML-initialized and randomly initialized surrogates.

Quantitative Results. At all dataset sizes, COMPNET-initialized surrogates have the lowest MSE and the highest SSIM on average. Among the other initialization methods, there is no consistent winner across dataset sizes.

The variance for the MSE results is comparable across all initialization methods and is high enough that there is overlap among methods. For example, at a dataset size of 0%, an MSE result that is one standard deviation below the mean for MAML is lower than the mean for COMPNETS. However, for all other dataset sizes the mean MSE for COMPNETS is lower than the mean MSE for MAML, even after subtracting a single standard deviation.

For the SSIM results, at smaller dataset sizes, the variance is high enough that there is overlap among methods. At larger dataset sizes though, the results are more clearly separated, with COMPNETS having the highest mean SSIM, even when one adds a single standard deviation to the mean SSIM for each of the other initialization methods.

6. Conclusion

In this paper, we presented the concept of a neural surrogate compiler and demonstrated how a neural surrogate compiler can be implemented with COMPNETS. We provided a dataset, EXESTACK, that one can use to learn neural surrogate compilers. We demonstrated the effectiveness of COMPNETs on EXESTACKCPN programs and PARROTBENCHCPN, a suite of numerical benchmarks. Specifically, we showed COMPNET-initialized surrogates achieve losses that are 1.9-$9.5\times$ lower than randomly initialized surrogates, they produce color-quantized images that are 1.0-$1.3\times$ more similar to images produced by an exact implementation than images produced by randomly initialized surrogates, and they train in 4.3-$7.3\times$ fewer epochs than randomly initialized surrogates.

The key insight of our work is that a programming language can condition the space of neural network initializations. In the limit, a neural surrogate compiler could produce initializations requiring no training to achieve low error. More broadly, neural surrogate compilers could be used to encode programmatically specified behaviors in neural networks, potentially accelerating training for more general tasks.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

Acknowledgements

We would like to thank Ellie Cheng, Charles Yuan, 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-2217064) and an Intel Research Fellowship.

References

An, S., Fowler, C., Zheng, B., Shalaginov, M. Y., Tang, H., Li, H., Zhou, L., Ding, J., Agarwal, A. M., Rivero-Baleine, C., Richardson, K. A., Gu, T., Hu, J., and Zhang, H. A deep learning approach for objective-driven all-dielectric metasurface design. *ACS Photonics*, 2019.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models. *arXiv preprint 2108.07732*, 2021.

Baek, W. and Chilimbi, T. M. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Programming Language Design and Implementation*, 2010.

Bieber, D., Sutton, C., Larochelle, H., and Tarlow, D. Learning to execute programs with instruction pointer attention graph neural networks. In *International Conference on Neural Information Processing Systems*, 2020.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. T., and Zhang, Y. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint 2303.12712*, 2023.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019.

Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Neural acceleration for general-purpose approximate programs. In *International Symposium on Microarchitecture*, 2012a.

Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Architecture support for disciplined approximate programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012b.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, 2017.

Giannou, A., Rajput, S., Sohn, J.-Y., Lee, K., Lee, J. D., and Papailiopoulos, D. Looped transformers as programmable computers. In *International Conference on Machine Learning*, 2023.

Gu, A., Rozière, B., Leather, H., Solar-Lezama, A., Synnaeve, G., and Wang, S. I. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint 2401.03065*, 2024.

Ha, D., Dai, A. M., and Le, Q. V. Hypernetworks. In *International Conference on Learning Representations*, 2017.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, 2020.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *IEEE International Conference on Computer Vision*, 2015.

Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. Meta-learning in neural networks: A survey. 44(09), 2022.

İpek, E., McKee, S. A., Caruana, R., de Supinski, B. R., and Schulz, M. Efficiently exploring architectural design spaces via predictive modeling. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

Jin, T., Liu, Z., Yan, S., Eichenberger, A., and Morency, L.-P. Language to network: Conditional parameter adaptation with natural language descriptions. In *Annual Meeting of the Association for Computational Linguistics*, 2020.

Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., and Wu, A. Y. An efficient k-means clustering algorithm: Analysis and implementation. 24, 2002.

Kaya, M. and Hajimirza, S. Using a novel transfer learning method for designing thin film solar cells with enhanced quantum efficiencies. *Scientific Reports*, 2019.

Kocetkov, D., Li, R., Ben Allal, L., Li, J., Mou, C., Muñoz Ferrandis, C., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., Bahdanau, D., von Werra, L., and de Vries, H. The stack: 3 tb of permissively licensed source code. *arXiv preprint 2211.15533*, 2022.

Kustowski, B., Gaffney, J. A., Spears, B. K., Anderson, G. J., Thiagarajan, J. J., and Anirudh, R. Transfer learning as a tool for reducing simulation bias: Application to inertial confinement fusion. *Transactions on Plasma Science*, 2020.

Kwon, J. and Carloni, L. P. Transfer learning for design-space exploration with high-level synthesis. In *Workshop on Machine Learning for CAD*, 2020.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2022.

Lindner, D., Kramar, J., Farquhar, S., Rahtz, M., McGrath, T., and Mikulik, V. Tracr: Compiled transformers as a laboratory for interpretability. In *Neural Information Processing Systems*, 2023.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.-D., Risdal, M., Li, J., Zhu, J., Zhuo, T. Y., Zheltonozhskii, E., Dade, N. O. O., Yu, W., Krauß, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H., Scholak, T., Paquet, S., Robinson, J., Anderson, C. J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y., Ferrandis, C. M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder 2 and the stack v2: The next generation. *arXiv preprint 2402.19173*, 2024.

Mendis, C. *Towards Automated Construction of Compiler Optimizations*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, 2020.

Misailovic, S., Sidiroglou, S., Hoffmann, H., and Rinard, M. Quality of service profiling. In *International Conference on Software Engineering*, 2010.

Munk, A., Zwartsenberg, B., Scibior, A., Baydin, A. G., Stewart, A. L., Fernlund, G., Poursartip, A., and Wood, F. Probabilistic surrogate networks for simulators with unbounded randomness. In *Uncertainty in Artificial Intelligence*, 2022.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., and Odena, A. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint 2112.00114*, 2021.

OpenAI, :, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., Berdine, J., Bernadett-Shapiro, G., Berner, C., Bogdonoff, L., Boiko, O., Boyd, M., Brakman, A.-L., Brockman, G., Brooks, T., Brundage, M., Button, K., Cai, T., Campbell, R., Cann, A., Carey, B., Carlson, C., Carmichael, R., Chan, B., Chang, C., Chantzis, F., Chen, D., Chen, S., Chen, R., Chen, J., Chen, M., Chess, B., Cho, C., Chu, C., Chung, H. W., Cummings, D., Currier, J., Dai, Y., Decareaux, C., Degry, T., Deutsch, N., Deville, D., Dhar, A., Dohan, D., Dowling, S., Dunning, S., Ecoffet, A., Eleti, A., Eloundou, T., Farhi, D., Fedus, L., Felix, N., Fishman, S. P., Forte, J., Fulford, I., Gao, L., Georges, E., Gibson, C., Goel, V., Gogineni, T., Goh, G., Gontijo-Lopes, R., Gordon, J., Grafstein, M., Gray, S., Greene, R., Gross, J., Gu, S. S., Guo, Y., Hallacy, C., Han, J., Harris, J., He, Y., Heaton, M., Heidecke, J., Hesse, C., Hickey, A., Hickey, W., Hoeschele, P., Houghton, B., Hsu, K., Hu, S., Hu, X., Huizinga, J., Jain, S., Jain, S., Jang, J., Jiang, A., Jiang, R., Jin, H., Jin, D., Jomoto, S., Jonn, B., Jun, H., Kaftan, T., Łukasz Kaiser, Kamali, A., Kanitscheider, I., Keskar, N. S., Khan, T., Kilpatrick, L., Kim, J. W., Kim, C., Kim, Y., Kirchner, H., Kiros, J., Knight, M., Kokotajlo, D., Łukasz Kondraciuk, Kondrich, A., Konstantinidis, A., Kosic, K., Krueger, G., Kuo, V., Lampe, M., Lan, I., Lee, T., Leike, J., Leung, J., Levy, D., Li, C. M., Lim, R., Lin, M., Lin, S., Litwin, M., Lopez, T., Lowe, R., Lue, P., Makanju, A., Malfacini, K., Manning, S., Markov, T., Markovski, Y., Martin, B., Mayer, K., Mayne, A., McGrew, B., McKinney, S. M., McLeavey, C., McMillan, P., McNeil, J., Medina, D., Mehta, A., Menick, J., Metz, L., Mishchenko, A., Mishkin, P., Monaco, V., Morikawa, E., Mossing, D., Mu, T., Murati, M., Murk, O., Mély, D., Nair, A., Nakano, R., Nayak, R., Neelakantan, A., Ngo, R., Noh, H., Ouyang, L., O'Keefe, C., Pachocki, J., Paino, A., Palermo, J., Pantuliano, A., Parascandolo, G., Parish, J., Parparita, E., Passos, A., Pavlov, M., Peng, A., Perelman, A., de Avila Belbute Peres, F., Petrov, M., de Oliveira Pinto, H. P., Michael, Pokorny, Pokrass, M., Pong, V., Powell, T., Power, A., Power, B., Proehl, E., Puri, R., Radford, A., Rae, J., Ramesh, A., Raymond, C., Real, F., Rimbach, K., Ross, C., Rotsted, B., Roussez, H., Ryder, N., Saltarelli, M., Sanders, T., Santurkar, S., Sastry, G., Schmidt, H., Schnurr, D., Schulman, J., Selsam, D., Sheppard, K., Sherbakov, T., Shieh, J., Shoker, S., Shyam, P., Sidor, S., Sigler, E., Simens, M., Sitkin, J., Slama, K., Sohl, I., Sokolowsky, B., Song, Y., Staudacher, N., Such, F. P., Summers, N., Sutskever, I., Tang, J., Tezak, N., Thompson, M., Tillet, P., Tootoonchian, A., Tseng, E., Tuggle, P., Turley, N., Tworek, J., Uribe, J. F. C., Vallone, A., Vijayvergiya, A., Voss, C., Wainwright, C., Wang, J. J., Wang, A., Wang, B., Ward, J., Wei, J., Weinmann, C., Welihinda, A., Welinder, P., Weng, J., Weng, L., Wiethoff, M., Willner, D., Winter, C., Wolrich, S., Wong, H., Workman, L., Wu, S., Wu, J., Wu, M., Xiao, K., Xu, T., Yoo, S., Yu, K., Yuan, Q., Zaremba, W., Zellers, R., Zhang, C., Zhang, M., Zhao, S., Zheng, T., Zhuang, J., Zhuk, W., and Zoph, B. Gpt-4 technical report. 2023.

Park, J., Amaro, E., Mahajan, D., Thwaites, B., and Esmaeilzadeh, H. Axgames: Towards crowdsourcing quality target determination in approximate computing. 2016.

Pestourie, R., Mroueh, Y., Nguyen, T. V., Das, P., and Johnson, S. G. Active learning of deep surrogates for pdes: application to metasurface design. *npj Computational Materials*, 2020.

Renda, A., Chen, Y., Mendis, C., and Carbin, M. Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates. In *International Symposium on Microarchitecture*, 2020.

Renda, A., Ding, Y., and Carbin, M. Programming with neural surrogates of programs. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2021.

Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. Enerj: approximate data types for safe and general low-power computation. In *Programming Language Design and Implementation*, 2011.

Shirobokov, S., Belavin, V., Kagan, M., Ustyuzhanin, A., and Baydin, A. G. Black-box optimization with local generative surrogates. In *Advances in Neural Information Processing Systems*, 2020.

Tercan, H., Guajardo, A., Heinisch, J., Thiele, T., Hopmann, C., and Meisen, T. Transfer-learning: Bridging the gap between real and simulation data for machine learning in injection molding. *Procedia CIRP*, 2018.

Tseng, E., Yu, F., Yang, Y., Mannan, F., Arnaud, K. S., Nowrouzezahrai, D., Lalonde, J.-F., and Heide, F. Hyperparameter optimization in black-box image processing using differentiable proxies. *Transactions on Graphics*, 2019.

Turc, I., Chang, M., Lee, K., and Toutanova, K. Well-read students learn better: The impact of student initialization on knowledge distillation. *arXiv preprint 1908.08962*, 2019.

Wang, Z., Bovik, A., Sheikh, H., and Simoncelli, E. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 2004.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., brian ichter, Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

Weiss, G., Goldberg, Y., and Yahav, E. Thinking like transformers. *ArXiv*, abs/2106.06981, 2021.

Zaremba, W. and Sutskever, I. Learning to execute. *arXiv preprint 1410.4615*, 2015.

Zhmoginov, A., Sandler, M., and Vladymyrov, M. Hypertransformer: Model generation for supervised and semi-supervised few-shot learning. In *International Conference on Machine Learning*, 2022.

A. Related Work

Neural surrogate compilation is inspired by literature on neural surrogates of programs and meta-learning. In the following sections, we survey these fields and describe other efforts to compile programs to neural networks.

A.1. Neural Surrogates of Programs

A common approach to developing neural surrogates of programs is to train a program-specific neural surrogate[4] on a dataset of input-output examples (Renda et al., 2021), or more recently, to train a universal neural surrogate on a dataset that includes many programs (Zaremba & Sutskever, 2015; Nye et al., 2021). Our work presents an alternative method for training neural surrogates of numerical programs that maintains the speed of program-specific neural surrogates but incorporates the data efficiency benefits of universal neural surrogates.

Program-Specific Neural Surrogates. Researchers across scientific disciplines have used neural surrogates of numerical programs to accelerate computations, adapt to new settings, and enable gradient-based optimization. Esmaeilzadeh et al. (2012a) demonstrate that neural surrogates of numerical programs can improve performance for computations in signal processing, robotics, 3D games, compression, machine learning, and image processing. To accelerate optical metasurface design, An et al. (2019) use neural surrogates of numerical simulators and Pestourie et al. (2020) use neural surrogates of partial differential equations. Tercan et al. (2018) and Kustowski et al. (2020) use neural surrogates of numerical simulators for plastic injection molding and inertial confinement fusion, respectively, to facilitate data-efficient finetuning on real physical data. Kaya & Hajimirza (2019) accelerate numerical simulations for solar cells using neural surrogates, and they use transfer learning to quickly adapt neural surrogates when simulator configurations change. Shirobokov et al. (2020) use neural surrogates of non-differentiable, numerical physical simulators, to enable gradient-based optimization of simulator parameters.

Researchers have used nonnumerical surrogates to optimize and explore discrete configuration spaces. Tseng et al. (2019) and Renda et al. (2020) develop neural surrogates of a black-box image signal processing unit and a cycle-accurate CPU simulator, respectively; both techniques enable gradient-based optimization of program inputs, to match some desired input-output behavior. Kwon & Carloni (2020) develop a neural surrogate of a high-level synthesis pipeline for hardware. Using this surrogate, they lower the cost of predicting the performance and cost of hardware configurations, and they use transfer learning to lower the cost of developing neural surrogates for new configuration spaces.

[4] In this section, we emphasize when neural surrogates are program-specific, to contrast with universal neural surrogates.

Universal Neural Surrogates. Researchers have developed universal neural surrogates using a variety of architectures. Early work in this area uses long short-term memory networks to predict the results of executing simple, synthetic Python programs (Zaremba & Sutskever, 2015). Later work uses graph neural networks that model program structure in a similar evaluation setup (Bieber et al., 2020). More recently, researchers have trained Transformer-based models on synthetic datasets of programs or large datasets that include programs (Austin et al., 2021; Nye et al., 2021; OpenAI et al., 2023; Bubeck et al., 2023; Gu et al., 2024).

A.2. Meta-Learning

Meta-learning can improve data efficiency and transfer learning when there is task-agnostic knowledge that can be extracted from a family of tasks (Hospedales et al., 2022). For example, in the setting we consider, the knowledge of how to execute programs is not specific to any one program but is useful for compiling each program. We describe the technique we employ, hypernetworks (Ha et al., 2017), as well as another meta-learning technique, MAML (model-agnostic meta-learning) (Finn et al., 2017). The most noteworthy difference between the two is that, in the former, the parameter space of the meta-learner and the learners differ, whereas, in the latter, these spaces are the same.

Hypernetworks. Hypernetworks were first proposed by Ha et al. and achieve state-of-the-art results on sequence modeling tasks (Ha et al., 2017). More recent work by Jin et al. proposes a system, N^3, that adapts Transformers to function as hypernetworks that condition on text for few-shot learning on image classification tasks (2020).

Model-Agnostic Meta-Learning. MAML is a framework for developing neural network initializations that can be finetuned to new tasks with a small amount of data and a few iterations of SGD (Finn et al., 2017). Some authors have noted, however, that MAML couples the task space complexity to the complexity of the individual tasks (Zhmoginov et al., 2022), making the parameter space a bottleneck as the task space grows. Our technique does not suffer from this issue because the hypernetwork can be larger than the generated neural surrogate.

A.3. Compiling Programs to Neural Networks

There exists prior work on compiling programs to neural networks, though usually as a means of understanding neural network architectures, rather than producing neural surrogates of programs.

Lindner et al. present a compiler, *Tracr*, from the *RASP* *programming language* to Transformer weights. The Restricted Access Sequence Processing (RASP) Language is a language with operations developed in analogy to the attention and feedforward operations in a Transformer; notably, RASP is not Turing-complete. Tracr was designed for the purpose of conducting interpretability experiments and evaluating interpretability methods (Lindner et al., 2023; Weiss et al., 2021). Since Tracr was not designed with model efficiency in mind, the resulting models are much larger than a roughly equivalent model trained from gradient descent would be, as evidenced by their evaluation. The COMPNET architecture, however, can be trained to target any size of architecture.

Giannou et al. present the *looped Transformer*, a Transformer-based architecture that functions as a programmable computer (Giannou et al., 2023). To execute a program, one expresses the program as commands in their instruction set, encodes this sequence of commands as the Transformer input, then executes the Transformer in a loop until it reaches a halt command. Their instruction set is Turing-complete, and they use it to implement a calculator, linear algebra library, and in-context learning algorithms. This architecture can be thought of as a universal neural surrogate, in contrast to a neural surrogate compiler.

B. EXESTACK Generation (Extended)

Here we provide a detailed explanation of each step in generating EXESTACK, following the flow of Figure 3.

① **Preprocessing.** We pull the functions in EXESTACK from files that may contain preprocessor directives, which may affect the ability for these functions to be executed in isolation, if left unexpanded. We run the C preprocessor on source files until no more lines begin with "#", or we have run it twice, or an invocation fails.

② **Extracting Functions.** Recognize and collect all functions from each source file.

③ **Filtering for Pointer-Free Numeric Functions.** To filter for numeric functions in C programs, we only include C functions that use exclusively `float` and `double` data types in the function signature. Due to the possibility of dynamically sized inputs in the presence of pointers and the ambiguity of whether a pointer represents an input or output, we do not allow pointer types. Consequently, we also do not allow `void` as an output type. If checking a file for the above conditions takes longer than 8 seconds, we discard it. Note that these filters still allow integral and pointer data types to be used within the function.

④ **Filtering for Executable Functions and Collecting Outputs.** To simultaneously check for executability and collect outputs from a function, we first generate $2{,}048$ sets of inputs by sampling from the uniform distribution $\mathcal{U}([-1,1]^n)$, where n is the maximum number of desired inputs, and we use the same sets of inputs for all programs. We embed these inputs in a C program that includes the function source, as well as an execution harness for collecting outputs. When a program has fewer inputs than the maximum of n, we truncate the inputs we embed to the number of inputs the program has. When a program has more inputs than the maximum of n, we discard it. We compile the harness with the C standard math library included, since many numerical functions in C make use of this library. If there are any errors during compilation or execution of a function, we discard the function. Figure 8 shows an example of the execution harness instantiated for a function.

⑤ **Filtering for Deterministic Functions.** Since a neural surrogate is often a deterministic function of its inputs and weights (e.g., multilayer perceptrons), we filter nondeterministic functions from our dataset. We check for determinism by running a function 5 times on the same inputs, all sampled from $\mathcal{U}(-1,1)$, and observing whether the output differs on any execution. For neural surrogate architectures that are not deterministic, this step can be omitted.

⑥ **Deduplication.** We use a whitespace-invariant tokenizer to remove duplicate tokenized programs.

C. EXESTACKCPN Generation

To produce EXESTACKCPN, we apply the following additional filters to EXESTACK:

- **Filtering Long Programs.** Since BERT-Tiny has a maximum context length of 512 tokens, we remove functions with more than 512 tokens. We first strip comments from all programs to allow more programs to fit within the context.
- **Filtering Large Outputs.** Large or NaN outputs can lead to training instability for neural networks, so we additionally remove functions with any outputs with an absolute magnitude of 10 or larger or a NaN value.
- **Decontaminating Against PARROTBENCHCPN.** It is possible that EXESTACK contains similar programs to those in PARROTBENCHCPN. If we trained a COMPNET on these programs, improvements over random initialization could be due to memorization. To address this problem, we remove any programs from EXESTACK that are syntactically similar to programs in PARROTBENCHCPN.

For the evaluation in Section 5, we allow programs with a maximum of 9 inputs in the execution filter of

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

float inputs[1024][1] = {
  {0.10740153873327762},
  ...
};

float fftSin_Output0(float x) {
    return sin(-2 * 3.1415 * x);
}

int main() {
    for (int i = 0; i < 1024; i++) {
        float arg0 = inputs[i][0];
        float out = fftSin(arg0);
        printf("%f,", out);
        printf("\n");
    }
    return 0;
}
```

Figure 8: Source code template used for checking executability and collecting outputs, instantiated with the source of the `fft` kernel in PARROTBENCHCPN.

EXESTACK, since this is the number of inputs in the covering architecture we choose.

Figure 9 depicts the entire pipeline for generating EXESTACKCPN, Figure 10 shows a summary of the characteristics of EXESTACKCPN, and Figure 11 contains a histogram showing the distribution of arity among EXESTACKCPN programs. For the remainder of this section, we detail the decontamination step.

C.1. EXESTACKCPN Decontamination

To ensure the improvements observed in Section 5 are not due to memorization, the final step of EXESTACKCPN generation is decontamination against PARROTBENCHCPN programs. A prevailing decontamination methodology is to remove any syntactic matches up to whitespace (Li et al., 2022; Lozhkov et al., 2024). Though EXESTACK is not contaminated with PARROTBENCHCPN programs according to this methodology, we strengthen our methodology to additionally remove syntactically similar programs. This decontamination consists of bespoke syntactic analyses—one for each PARROTBENCHCPN program. For the remainder of this section, we present each of these syntactic analyses and a sample of the near-duplicate programs they detect. In total, decontamination removes 375 functions.

C.1.1. FFT (OUTPUT 0)

Recall, the source for the `fft (0)` kernel in PARROTBENCHCPN is

```
float fftSin_Output0(float x) {
    return sin(-2 * 3.1415 * x);
}
```

To decontaminate EXESTACKCPN against this program, we search for programs satisfying all conditions below:

- Contains “sin”
- Contains either “3.14” or “M_PI”
- Is at most 5 (non-empty) lines long
- Has one input

This methodology surfaces 29 matches. Below, we include a sample of 5 of these matches:

```
float seno(float x) {
  return sin(x * M_PI / 180);
}

float exponential(float value) {
  return sin(value * 3.14f / 2);
}

float easeOutSine(float time) {
  return sin(time * M_PI / 2);
}

double sine(double t) {
  return sin(2 * M_PI * t);
}

double cosine(double t) {
  return cos(2 * M_PI * t);
}
```

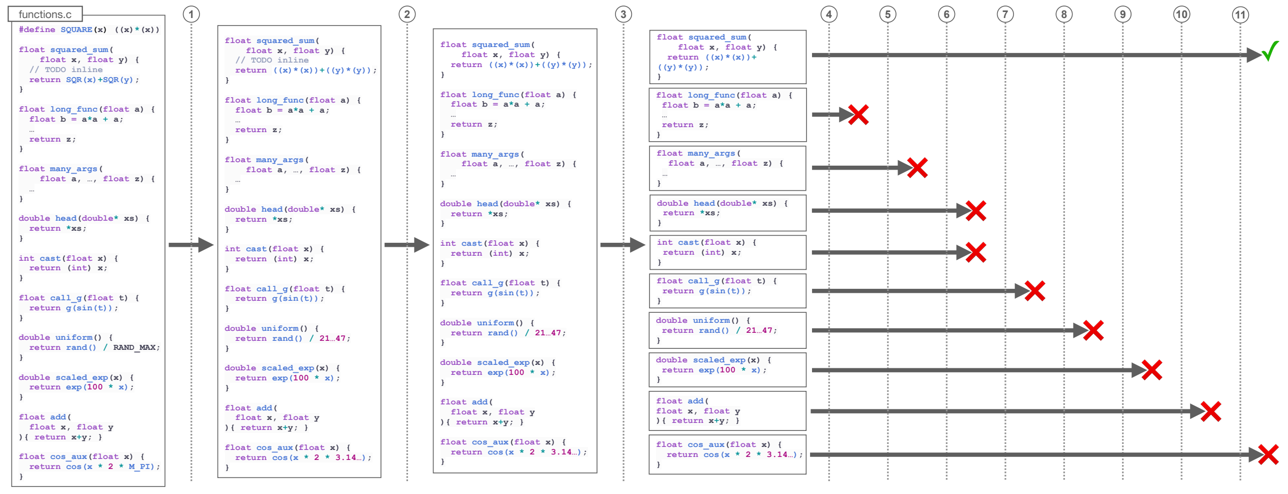
C.1.2. FFT (OUTPUT 1)

Recall, the source for the `fft (1)` kernel in PARROTBENCHCPN is

```
float fftSin_Output1(float x) {
    return cos(-2 * 3.1415 * x);
}
```

To decontaminate EXESTACKCPN against this program, we search for programs satisfying all conditions below:

- Contains “cos”



```c
#define SQUARE(x) ((x)*(x))

float squared_sum(
    float x, float y) {
  // TODO inline
  return SQR(x)+SQR(y);
}

float long_func(float a) {
  float b = a*a + a;
  …
  return z;
}

float many_args(
    float a, …, float z) {
  …
}

double head(double* xs) {
  return *xs;
}

int cast(float x) {
  return (int) x;
}

float call_g(float t) {
  return g(sin(t));
}

double uniform() {
  return rand() / RAND_MAX;
}

double scaled_exp(x) {
  return exp(100 * x);
}

float add(
  float x, float y
){ return x+y; }

float cos_aux(float x) {
  return cos(x * 2 * M_PI);
}
```

Figure 9: The EXESTACKCPN generation pipeline (i.e., EXESTACK tailored to COMPNETS). Starting with C source files from The Stack, we apply a sequence of maps followed by a sequence of filters. The steps are (1) run the C preprocessor, (2) remove comments, (3) extract functions from the source file, (4) remove functions with more tokens than a user-specified threshold (e.g., the maximum context length), (5) remove functions with more inputs than the target topology, (6) remove functions with pointers in their type signature and nonnumeric functions, (7) remove nonexecutable functions and collect input-output pairs, (8) remove nondeterministic functions, (9) remove functions with any outputs larger than a user-specified threshold, when run on the set of input-output pairs, (10) remove any duplicate programs, and (11) remove any programs syntactically similar to programs in PARROTBENCHCPN. Red "X"s denote that a function does not pass a filter and green checkmarks denote that a function passes all filters.

Characteristic	Value
Max Program Length (In Tokens)	512
Tokenizer Vocab Size	30,522
# Programs in Dataset	37,772
# Tokens in Dataset	1,728,304
# I/O Pairs Per Program	2,048

Figure 10: Summary of EXESTACKCPN characteristics.

- Contains either "3.14" or "M_PI"
- Is at most 5 (non-empty) lines long
- Has one input

This methodology surfaces 20 matches. Below, we include a sample of 5 of these matches:

```c
float coss(float x) {
  return cos(x * M_PI / 180);
}

double cosine(double t) {
  return cos(2 * M_PI * t);
}
```

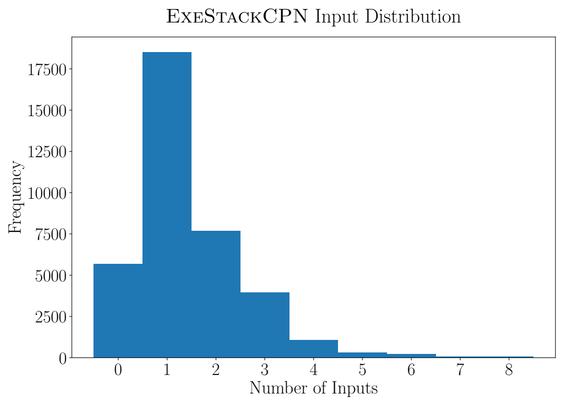


Figure 11: Distribution of the number of program inputs for programs in EXESTACKCPN.

```c
float hamming(float x) {
  return 0.54-0.46*cos(2*M_PI*x);
}

float easeInSine(float time) {
  return 1 - cos(time * M_PI / 2);
}

float easeInOutSine(float time) {
  return 0.5 * (1 - cos(M_PI * time));
}
```

C.2. InverseK2J (Output 0)

Recall, the source for the `invk2j (0)` kernel in ParrotBenchCPN is

```
float inversek2j_Output0(
    float x, float y) {
  float l1 = 0.5 ;
  float l2 = 0.5 ;
  float theta2 = (float) acos(
    ((x * x) + (y * y) -
      (l1 * l1) -
      (l2 * l2)) /
    (2 * l1 * l2)
  );
  return (float) asin(
    (y * (l1 + l2 * cos(theta2)) -
      x * l2 * sin(theta2)) /
    (x * x + y * y)
  );
}
```

To decontaminate ExeStackCPN against this program, we search for programs satisfying all conditions below:

- Contains "asin", "acos", "sin", and "cos"
- Contains either ".5" or ("/" and "2")
- Is at most 7 (non-empty) lines long
- Has two inputs

This methodology surfaces 0 matches.

C.3. InvK2J (Output 1)

Recall, the source for the `invk2j (1)` kernel in ParrotBenchCPN is

```
float inversek2j_Output1(
    float x, float y) {
  float l1 = 0.5 ;
  float l2 = 0.5 ;
  return (float) acos(
    ((x * x) + (y * y) -
      (l1 * l1) - (l2 * l2)) /
    (2 * l1 * l2)
  );
}
```

To decontaminate ExeStackCPN against this program, we search for programs satisfying all conditions below:

- Contains "acos"
- Contains either ".5" or ("/" and "2")
- Is at most 6 (non-empty) lines long
- Has two inputs

This methodology surfaces 0 matches.

C.3.1. KMeans

Recall, the source for the `kmeans` kernel in ParrotBenchCPN is

```
float euclideanDistance(
    float p_0, float p_1, float p_2,
    float c1_0, float c1_1, float c1_2) {
  float r;

  r = 0;
  r += (p_0 - c1_0) * (p_0 - c1_0);
  r += (p_1 - c1_1) * (p_1 - c1_1);
  r += (p_2 - c1_2) * (p_2 - c1_2);

  return sqrt(r);
}
```

To decontaminate ExeStackCPN against this program, we search for programs satisfying all conditions below:

- Contains "sqrt", "*", "+", and "-"
- Has 6 inputs

This methodology surfaces 10 matches. Below, we include a sample of 5 of these matches:

```
float len(
  float x0, float y0, float z0,
  float x1, float y1, float z1 ){
    return sqrt(
      (x1-x0)*(x1-x0) +
      (y1-y0)*(y1-y0) +
      (z1-z0)*(z1-z0)
    );
}

float dist(
    float x1, float y1,float z1,
    float x2,float y2,float z2) {
  return sqrt(
    (x1-x2)*(x1-x2) +
    (y1-y2)*(y1-y2) +
    (z1-z2)*(z1-z2)
  );
}

float calc_dist(
    float x0, float y0, float z0,
    float x1, float y1, float z1) {
  float dx   = (x1 - x0);
  float dy   = (y1 - y0);
  float dz   = (z1 - z0);
  float dist = sqrtf(
    (dx * dx) +
    (dy * dy) +
    (dz * dz)
  );
  return dist;
}

double dist(
    double x0, double y0, double z0,
    double x1, double y1, double z1) {
  return sqrt(
    (x1 - x0) * (x1 - x0) +
    (y1 - y0) * (y1 - y0) +
    (z1 - z0) * (z1 - z0)
  );
}

double dist(
    double ax, double ay, double az,
    double bx, double by, double bz) {
  return sqrt(
    (ax - bx)*(ax - bx) +
    (ay - by)*(ay - by) +
    (az - bz)*(az - bz)
  );
}
```

C.3.2. SOBEL

Recall, the source for the `sobel` kernel is

```
float sobel(
    float w00, float w01, float w02,
    float w10, float w11, float w12,
    float w20, float w21, float w22) {
  float sx = 0.0;
  sx += w00 * -1;
  sx += w10 * 0;
  sx += w20 * 1;
  sx += w01 * -2;
  sx += w11 * 0;
  sx += w21 * 2;
  sx += w02 * -1;
  sx += w12 * 0;
  sx += w22 * 1;

  float sy = 0.0;
  sy += w00 * -1;
  sy += w10 * -2;
  sy += w20 * -1;
  sy += w01 * 0;
  sy += w11 * 0;
  sy += w21 * 0;
  sy += w02 * 1;
  sy += w12 * 2;
  sy += w22 * 1;

  float s = sqrt(
    sx * sx + sy * sy);
  if (s >= (256 / sqrt(
      256 * 256 + 256 * 256)))
    s = 255 / sqrt(
      256 * 256 + 256 * 256);
  return s;
}
```

To decontaminate EXESTACKCPN against this program, we search for programs satisfying all conditions below:

- Contains “sqrt”, “+”, “*”, and “/”
- Has 9 inputs

This methodology surfaces 0 matches.

Benchmark	Train Inputs	Test Inputs
`fft` (0)	32768	2048
`fft` (1)	32768	2048
`invk2j` (0)	10000	10000
`invk2j` (1)	10000	10000
`kmeans`	50000	48400
`sobel`	18725	17976

Figure 12: Number of training and testing inputs for each benchmark in PARROTBENCHCPN.

D. PARROTBENCHCPN Generation

Here, we present the PARROTBENCH programs, explain the modifications we made to PARROTBENCH to produce PARROTBENCHCPN, list the resulting source code, and describe how we generate inputs for these programs.

D.1. PARROTBENCH Source

The kernels in PARROTBENCH are `fft` (Figure 13), `inversek2j` (Figure 14), `jmeint` (Figure 15), `jpeg` (Figure 16), `kmeans` (Figure 17), and `sobel` (Figure 18).

```
void fftSinCos(float x, float* s, float* c) {
    *s = sin(-2 * PI * x);
    *c = cos(-2 * PI * x);
}
```

Figure 13: Code for the `fft` benchmark in PARROTBENCH.

```
float l1 = 0.5 ;
float l2 = 0.5 ;

void inversek2j(float x, float y, float* theta1, float* theta2) {
        *theta2 = (float) acos(
    ((x * x) + (y * y) - (l1 * l1) - (l2 * l2)) /
    (2 * l1 * l2)) ;
        *theta1 = (float) asin(
    (y * (l1 + l2 * cos(*theta2)) - x * l2 * sin(*theta2)) /
    (x * x + y * y)) ;
}
```

Figure 14: Code for the `invk2j` benchmark in PARROTBENCH.

We obtained these kernels from the AxBench repository. For brevity, we have referred to the `inversek2j` kernel as `invk2j` throughout this paper.

D.2. PARROTBENCH Modifications

Due to methodological choices in EXESTACK and architectural choices for COMPNETS, we omit some PARROTBENCH benchmarks from PARROTBENCHCPN and modify others. We omit the `jmeint` and `jpeg` benchmarks in PARROTBENCH because they are significantly longer than the 512-token context length of a BERT-Tiny (1,192 and 1,250 tokens, respectively). We modify the `fft` and `invk2j` benchmarks because they both use pointer arguments to store outputs, and our COMPNETS were not trained to support pointer arguments. To make each function pointer-free, we split it into two functions, each function computing one of the outputs (Figures 19 and 20). Additionally, the `sobel` benchmark uses pointer inputs, so we rewrite it to only use scalar inputs (Figure 22). Finally, the `kmeans` benchmark uses custom structs to pass arguments, so we rewrite the benchmark to desugar these structs into their scalar components (Figure 21).

D.3. PARROTBENCHCPN Input Generation

We attempt to exactly replicate the dataset used by Esmaeilzadeh et al. (2012a) for the subset of benchmarks we consider from PARROTBENCH. To replicate their dataset, we analyze the source code in the AxBench repository, which contains all kernels in PARROTBENCH. Figure 12 shows the size of the dataset produced by the methodology in the following sections.

D.3.1. FFT

To generate train inputs for `fft`, we generate 32,768 inputs uniformly at random from $[0, 1/2]$. To generate test inputs for `fft`, we generate 2,048 inputs uniformly at random from $[0, 1/2]$, resampling as necessary whenever an input is generated that exists in the training set.

D.3.2. INVERSEK2J

To generate train inputs for `invk2j`, we generate 10,000 inputs uniformly at random from $[-1/2, 1] \times [0, 1]$. To generate test inputs for `invk2j`, we again generate 10,000 inputs uniformly at random from $[-1/2, 1] \times [0, 1]$, but we resample whenever an input exists in the training set.

D.3.3. KMEANS

To generate train inputs for `kmeans`, we generate 50,000 inputs uniformly at random from $[0, 1]^6$.

To generate test inputs for `kmeans`, we use an image of peppers for RGB inputs (see Figure 23) and we generate 6 centroids with uniformly random coordinates in $[0, 1]^3$, the number of centroids used by Esmaeilzadeh et al. (source). For each RGB input, we choose a random centroid to compute the `kmeans` kernel on, and we add the resulting I/O sample to the testing set. This procedure results in a testing set containing 48,400 inputs.

D.3.4. SOBEL

To generate training and testing inputs for `sobel`, we read from files on the official repo of Esmaeilzadeh et al. (2012a) (here and here, respectively). These files contain 18,725 and 17,976 input-output pairs, respectively.

```c
int tri_tri_intersect(float V0[3],float V1[3],float V2[3],
                      float U0[3],float U1[3],float U2[3])
{
  float E1[3],E2[3];
  float N1[3],N2[3],d1,d2;
  float du0,du1,du2,dv0,dv1,dv2;
  float D[3];
  float isect1[2], isect2[2];
  float du0du1,du0du2,dv0dv1,dv0dv2;
  short index;
  float vp0,vp1,vp2;
  float up0,up1,up2;
  float b,c,max;
  //int r;

  /* compute plane equation of triangle(V0,V1,V2) */
  SUB(E1,V1,V0);
  SUB(E2,V2,V0);
  CROSS(N1,E1,E2);
  d1=-DOT(N1,V0);
  /* plane equation 1: N1.X+d1=0 */

  /* put U0,U1,U2 into plane equation 1 to compute
  signed distances to the plane*/
  du0=DOT(N1,U0)+d1;
  du1=DOT(N1,U1)+d1;
  du2=DOT(N1,U2)+d1;

  /* coplanarity robustness check */
#if USE_EPSILON_TEST==true
  //printf("HERE\n");
  if(fabs(du0)<EPSILON) du0=0.0;
  if(fabs(du1)<EPSILON) du1=0.0;
  if(fabs(du2)<EPSILON) du2=0.0;
#endif
  du0du1=du0*du1;
  du0du2=du0*du2;


  if(du0du1>0.0f && du0du2>0.0f)
  { /* same sign on all of them + not equal 0 ? */
    //*output = 0 ;
    return 0;  /* no intersection occurs */
  }

  /* compute plane of triangle (U0,U1,U2) */
  SUB(E1,U1,U0);
  SUB(E2,U2,U0);
  CROSS(N2,E1,E2);
  d2=-DOT(N2,U0);
  /* plane equation 2: N2.X+d2=0 */

  /* put V0,V1,V2 into plane equation 2 */
  dv0=DOT(N2,V0)+d2;
  dv1=DOT(N2,V1)+d2;
  dv2=DOT(N2,V2)+d2;

#if USE_EPSILON_TEST==true
  //printf("THERE\n");
  if(fabs(dv0)<EPSILON) dv0=0.0;
  if(fabs(dv1)<EPSILON) dv1=0.0;
  if(fabs(dv2)<EPSILON) dv2=0.0;
#endif

  dv0dv1=dv0*dv1;
  dv0dv2=dv0*dv2;

  if(dv0dv1>0.0f && dv0dv2>0.0f)
  { /* same sign on all of them + not equal 0 ? */
    //*output = 1 ;
    return 0;  /* no intersection occurs */
  }
  /* compute direction of intersection line */
  CROSS(D,N1,N2);

  /* compute and index to the largest component of D */
  max=fabs(D[0]);
  index=0;
  b=fabs(D[1]);
  c=fabs(D[2]);
  if(b>max) max=b,index=1;
  if(c>max) max=c,index=2;

  /* this is the simplified projection onto L*/
  vp0=V0[index];
  vp1=V1[index];
  vp2=V2[index];

  up0=U0[index];
  up1=U1[index];
  up2=U2[index];

  /* compute interval for triangle 1 */
  COMPUTE_INTERVALS(
    vp0,vp1,vp2,
    dv0,dv1,dv2,
    dv0dv1,dv0dv2,
    isect1[0],isect1[1]);

  /* compute interval for triangle 2 */
  COMPUTE_INTERVALS(
    up0,up1,up2,
    du0,du1,du2,
    du0du1,du0du2,
    isect2[0],isect2[1]);

  SORT(isect1[0],isect1[1]);
  SORT(isect2[0],isect2[1]);

  if(isect1[1]<isect2[0] || isect2[1]<isect1[0])
  {
    //*output = 2 ;
    return 0;
  }
  //*output = 3 ;
  return 1;
}
```

Figure 15: Code for the `jmeint` benchmark in PARROTBENCH.

```
/* DCT for One block(8x8) */
void dct (INT16 *data)
{
  UINT16 i;
  INT32 x0, x1, x2, x3, x4, x5, x6, x7, x8;

  /*All values are shifted left by 10
  and rounded off to nearest integer */

  /* cos PI/16 * root(2)  */
  static const UINT16 c1=1420;
  /* cos PI/8 * root(2)  */
  static const UINT16 c2=1338;
  /* cos 3PI/16 * root(2)  */
  static const UINT16 c3=1204;
  /* cos 5PI/16 * root(2)  */
  static const UINT16 c5=805;
  /* cos 3PI/8 * root(2)  */
  static const UINT16 c6=554;
  /* cos 7PI/16 * root(2)  */
  static const UINT16 c7=283;

  static const UINT16 s1=3;
  static const UINT16 s2=10;
  static const UINT16 s3=13;

  for (i=8; i>0; i--)
  {
    x8 = data [0] + data [7];
    x0 = data [0] - data [7];

    x7 = data [1] + data [6];
    x1 = data [1] - data [6];

    x6 = data [2] + data [5];
    x2 = data [2] - data [5];

    x5 = data [3] + data [4];
    x3 = data [3] - data [4];

    x4 = x8 + x5;
    x8 -= x5;

    x5 = x7 + x6;
    x7 -= x6;

    data [0] = (INT16) (x4 + x5);
    data [4] = (INT16) (x4 - x5);

    data [2] = (INT16) ((x8*c2 + x7*c6) >> s2);
    data [6] = (INT16) ((x8*c6 - x7*c2) >> s2);

    data [7] = (INT16) (
      (x0*c7 - x1*c5 + x2*c3 - x3*c1) >> s2);
    data [5] = (INT16) (
      (x0*c5 - x1*c1 + x2*c7 + x3*c3) >> s2);
    data [3] = (INT16) (
      (x0*c3 - x1*c7 - x2*c1 - x3*c5) >> s2);
    data [1] = (INT16) (
      (x0*c1 + x1*c3 + x2*c5 + x3*c7) >> s2);

    data += 8;
  }

  data -= 64;

  for (i=8; i>0; i--)
  {
    x8 = data [0] + data [56];
    x0 = data [0] - data [56];

    x7 = data [8] + data [48];
    x1 = data [8] - data [48];

    x6 = data [16] + data [40];
    x2 = data [16] - data [40];

    x5 = data [24] + data [32];
    x3 = data [24] - data [32];

    x4 = x8 + x5;
    x8 -= x5;

    x5 = x7 + x6;
    x7 -= x6;

    data [0] = (INT16) ((x4 + x5) >> s1);
    data [32] = (INT16) ((x4 - x5) >> s1);

    data [16] = (INT16) ((x8*c2 + x7*c6) >> s3);
    data [48] = (INT16) ((x8*c6 - x7*c2) >> s3);

    data [56] = (INT16) ((x0*c7 - x1*c5 + x2*c3 - x3*c1) >> s3);
    data [40] = (INT16) ((x0*c5 - x1*c1 + x2*c7 + x3*c3) >> s3);
    data [24] = (INT16) ((x0*c3 - x1*c7 - x2*c1 - x3*c5) >> s3);
    data [8] = (INT16) ((x0*c1 + x1*c3 + x2*c5 + x3*c7) >> s3);

    data++;
  }
}

/* Multiply DCT Coefficients with Quantization table
and store in ZigZag location */
void quantization(
    INT16* const data, UINT16* const quant_table_ptr) {
  INT16 i;
  INT32 value;

  for (i = 63; i >= 0; i--) {
    value = data[i] * quant_table_ptr[i];
    value = (value + 0x4000) >> 15;

    Temp[zigzagTable[i]] = (INT16) value;
  }
}

// Kernel is:
//   dct(Y1);
//   quantization(Y1, ILqt);
```

Figure 16: Code for the `jpeg` benchmark in PARROTBENCH.

```
float euclideanDistance(RgbPixel* p, Centroid* c1) {
        float r;

        r = 0;
        r += (p->r - c1->r) * (p->r - c1->r);
        r += (p->g - c1->g) * (p->g - c1->g);
        r += (p->b - c1->b) * (p->b - c1->b);

        return sqrt(r);
}
```

Figure 17: Code for the `kmeans` benchmark in PARROTBENCH.

```
static float kx[][3] =
                {
                        { -1, -2, -1 },
                        {  0,  0,  0 },
                        {  1,  2,  1 }
                } ;

static float ky[][3] =
                {
                        { -1, 0, 1 },
                        { -2, 0, 2 },
                        { -1, 0, 1 }
                } ;

float convolve(float w[][3], float k[][3])
{
        float r ;
        r = 0.0 ;
        for( int j = 0 ; j < 3 ; j++ )
                for ( int i = 0 ; i < 3 ; i++ )
                {
                        r += w[i][j] * k[j][i] ;
                }
        return r ;
}

float sobel(float w[][3])
{
        float sx ;
        float sy ;
        float s  ;

        sx = convolve(w, ky) ;
        sy = convolve(w, kx) ;
        s = sqrt(sx * sx + sy * sy) ;
        if (s >= (256 / sqrt(256 * 256 + 256 * 256)))
                s = 255 / sqrt(256 * 256 + 256 * 256);
        return s ;
}
```

Figure 18: Code for the sobel benchmark in PARROTBENCH.

```
float fftSin_Output0(float x) {
    return sin(-2 * 3.1415 * x);
}

float fftSin_Output1(float x) {
    return cos(-2 * 3.1415 * x);
}
```

Figure 19: Code for the fft benchmark in PARROTBENCHCPN.

```
float invk2j_Output0(float x, float y) {
  float l1 = 0.5 ;
  float l2 = 0.5 ;
  float theta2 = (float)acos(
    ((x * x) + (y * y) - (l1 * l1) - (l2 * l2)) /
    (2 * l1 * l2)) ;
  return (float)asin(
    (y * (l1 + l2 * cos(theta2)) - x * l2 * sin(theta2)) /
    (x * x + y * y)) ;
}

float invk2j_Output1(float x, float y) {
  float l1 = 0.5 ;
  float l2 = 0.5 ;
  return (float)acos(
    ((x * x) + (y * y) - (l1 * l1) - (l2 * l2)) /
    (2 * l1 * l2)) ;
}
```

Figure 20: Code for the `invk2j` benchmark in PARROTBENCHCPN.

```
float euclideanDistance(
  float p_0, float p_1, float p_2,
  float c1_0, float c1_1, float c1_2) {
  float r;

  r = 0;
  r += (p_0 - c1_0) * (p_0 - c1_0);
  r += (p_1 - c1_1) * (p_1 - c1_1);
  r += (p_2 - c1_2) * (p_2 - c1_2);

  return sqrt(r);
}
```

Figure 21: Code for the `kmeans` benchmark in PARROTBENCHCPN.

```
float sobel(
  float w00, float w01, float w02,
  float w10, float w11, float w12,
  float w20, float w21, float w22)
{
  float sx = 0.0;
  sx += w00 * -1;
  sx += w10 * 0;
  sx += w20 * 1;
  sx += w01 * -2;
  sx += w11 * 0;
  sx += w21 * 2;
  sx += w02 * -1;
  sx += w12 * 0;
  sx += w22 * 1;

  float sy = 0.0;
  sy += w00 * -1;
  sy += w10 * -2;
  sy += w20 * -1;
  sy += w01 * 0;
  sy += w11 * 0;
  sy += w21 * 0;
  sy += w02 * 1;
  sy += w12 * 2;
  sy += w22 * 1;

  float s = sqrt(sx * sx + sy * sy) ;
  if (s >= (256 / sqrt(256 * 256 + 256 * 256)))
    s = 255 / sqrt(256 * 256 + 256 * 256);
  return s ;
}
```

Figure 22: Code for the `sobel` benchmark in PARROTBENCHCPN.

Figure 23: Image used to generate testing data for `kmeans`.

E. CompNet Training Details

CompNets are controlled by the following hyperparameters: program batch size, input batch size, learning rate, number of training epochs, dataset program split, dataset input split, and the surrogate topology. We swept over learning rates and chose fixed values for all other hyperparameters. We selected the learning rate that achieved the best final loss on validation programs, averaged over trials, and we used all trials of the winning configuration as initialization methods.

We summarize the training configuration for CompNets in Figure 24. Figures 25 and 26 show loss curves for CompNets trained on ExeStackCPN, using both padding modes described in Appendix N.

F. MAML Training Details

MAML is controlled by the following hyperparameters: the meta batch size (number of tasks per batch), the input batch size (number of inputs per task), the number of epochs, the inner gradient update step size (α), the outer gradient update step size (β), and the number of inner gradient update steps. We modify the official MAML implementation[5] to support training on ExeStackCPN.

We choose the meta batch size and the input batch size to align with how we train CompNets (Appendix E). We decided to use the maximum number of epochs Finn et al. (2017) use in their applications ($70,000$), and we observed that in all applications, β is fixed at 0.001. For the remaining parameters, α and the number of inner update steps, we perform a hyperparameter sweep, backing each configuration with 3 trials. We choose the extents of each hyperparameter in the sweep as the minimum and maximum of hyperparameter settings observed in applications, and we add some points between these extents. However, we limit the hyperparameter settings for α to a maximum of 0.2, as previous experiments (not reported in this paper) showed training instability at higher values.

After each configuration finishes training, we finetune it for 20 epochs for each of a sample of 5 programs from the ExeStackCPN validation set. We choose the hyperparameters with the lowest loss on the validation inputs at the end of finetuning, averaged over the sample of programs and trials. We use all 3 trials of the winning configuration as initialization methods.

We summarize the training configuration for MAML in Figure 27. Figures 28 and 29 show loss curves for MAML initializations trained on ExeStackCPN, using both padding modes described in Appendix N. The curves include *prelosses* and *postlosses* for training programs. In MAML training, the preloss is the loss of the current initialization when evaluated on a task, and the postloss is the loss of the initialization after finetuning for the number of inner gradient update steps.

G. Neural Surrogate Pretraining Details

Similarly to CompNets, pretrained surrogates are controlled by the following hyperparameters: program batch size, input batch size, learning rate, number of training epochs, dataset program split, dataset input split, and the surrogate topology. We sweep over the same set of learning rates as for CompNets, and we use the same values for other hyperparameters that we use for CompNets. We select the learning rate that achieves the best final loss on test programs, averaged over trials, and we use all trials of the winning configuration as initialization methods.

We summarize the training configuration for pretrained surrogates in Figure 30. Figures 31 and 32 show loss curves for surrogates pretrained on ExeStackCPN, using both padding modes described in Appendix N.

H. Data Efficiency Improvements (Extended)

We compute improvements in the data efficiency evaluation as a ratio of the test loss achieved by random initialization over the test loss achieved by an initialization method. Here, we present the test losses we use to compute these ratios, as well as test loss improvements grouped at a finer granularity.

Figure 33 shows test loss as a function of the dataset size. For all initialization methods, performance improves the most on `fft` as more training data becomes available. The difference between the average test loss at 0% and 100% is ≈ 6 orders of magnitude for every initialization method, whereas it is only $\approx$ 2-3 on other benchmarks. The `kmeans` benchmark exhibits the least variation, with all initialization methods dropping by ≤ 2 orders of magnitude on average, from 0% to 100% of training data. The only benchmark where CompNets dominate at all dataset sizes is `kmeans`. For other benchmarks, CompNets perform, on par, slightly better, or slightly worse, varying across dataset sizes.

[5]https://github.com/cbfinn/maml

Setting	Value
Architecture	BERT-Tiny
Program Batch Size	32
Input Batch Size	1024
Learning Rate	$\in \{1 \cdot 10^{-5}, 2 \cdot 10^{-5}, \mathbf{5 \cdot 10^{-5}}, 5 \cdot 10^{-4}, 8 \cdot 10^{-4}\}$
# Epochs	$1,500$
Dataset Program Split	$80/10/10$
Dataset Input Split	$50/0/50$
Surrogate Topology	$9 \to 4 \to 4 \to 1$
GPU	NVIDIA Tesla T4 16GB
# Trials	3

Figure 24: Training configuration for COMPNETS. We represent any values we sweep over as a set, and we bold the values that obtain the best final loss on test programs.

Figure 34 contains histograms showing, for a sample of $1{,}000$ EXESTACKCPN test programs, the test loss each initialization method achieves at the epoch with the lowest validation loss. At a dataset size of 0%, the distribution of COMPNET losses is significantly skewed to smaller losses, relative to other initialization methods. At a dataset size of 0.1%, every initialization method has roughly a bimodal loss distribution. In the smaller-loss mode, each initialization method has comparable performance, but on the larger-loss mode, COMPNETS still skew towards smaller losses. As the dataset sizes increase, these modes begin to merge together, with COMPNETS continuing to retain more mass in lower losses than other initialization methods.

Figure 35 contains tables showing the test loss each initialization method achieves at the epoch with the lowest validation loss for PARROTBENCHCPN programs. At a dataset size of 0%, COMPNETS have the worst loss on `fft`, but the best or among the best loss for all other benchmarks. At a dataset size of 0.1%, COMPNETS have the best or among the best loss for all benchmarks except for `sobel`, where MAML achieves the best loss by a significant margin. At higher dataset sizes, none of the initialization methods consistently win for each benchmark.

Figure 36 shows test loss improvements grouped by both programs and dataset sizes. At a dataset size of 0% for `fft`, COMPNETS have among the worst test loss improvement, but for higher dataset sizes, COMPNETS significantly outperform the other initialization methods, except at 100%, where MAML achieves a slightly better test loss improvement ($0.70\times$ vs. $0.75\times$). At a dataset size of 0% for `invk2j`, COMPNETS have the best test loss improvement, but for higher dataset sizes, COMPNETS achieve lower test loss improvements than other initialization methods, except at 100%, where COMPNETS barely achieve the best test loss improvement ($0.93\times$ vs. $0.92\times$ for MAML and $0.90\times$ for pretrained surrogates). At a dataset size of 0% for `kmeans`, both COMPNETS and pretrained surrogates achieve significant test loss improvements of greater than $3.5\times$, but at higher dataset sizes, COMPNETS achieve better and better test loss improvements (maximum of $15.74\times$ at 100% dataset size), whereas pretrained surrogates remain the same or worse. At a dataset size of 0% for `sobel`, all initialization methods achieve a test loss improvement of more than $1.6\times$, with COMPNETS achieving the highest at $2.84\times$. However, at higher dataset sizes, COMPNETS and MAML alternate between improving, matching, or worsening test loss, and pretrained surrogates only worsen test loss, achieving a maximum of $0.77\times$ test loss improvement.

I. Neural Surrogates for Color Quantization (Extended)

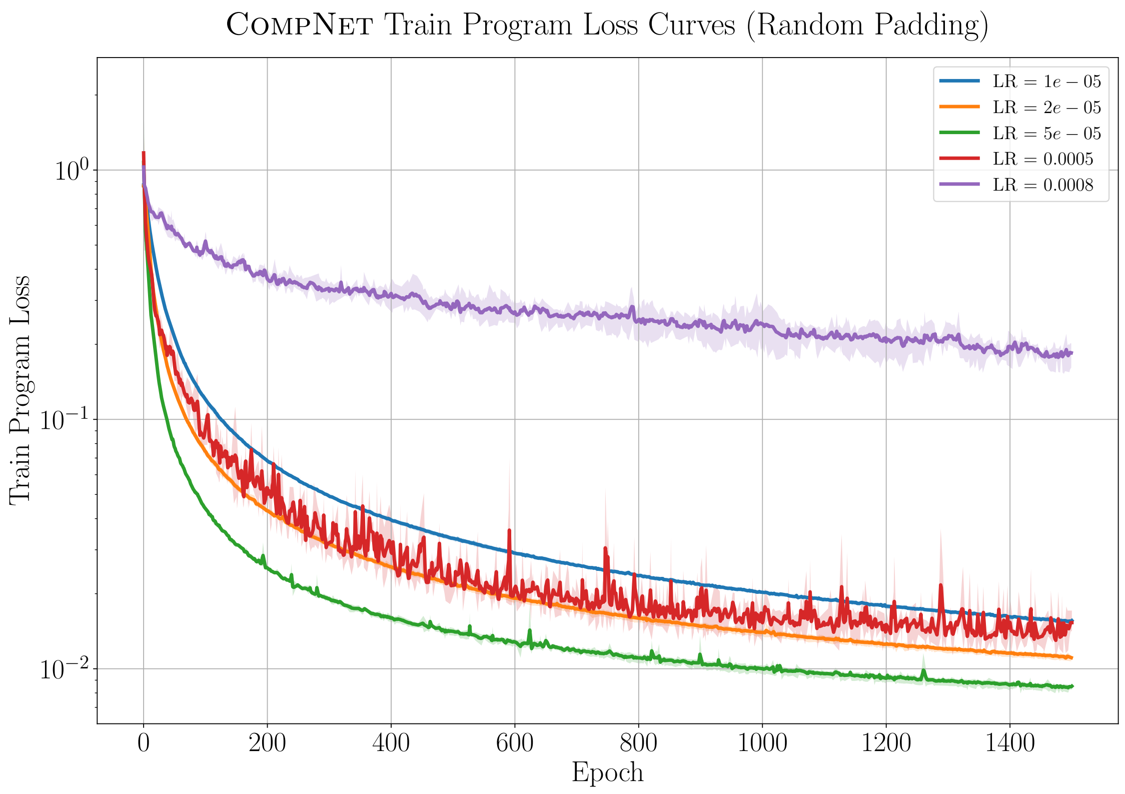
In this section, we present visual results for all dataset sizes in the data efficiency evaluation, as well as quantitative results for 10- and 15-color palettes.

I.1. Visual Results

Figure 37 contains visual results for surrogates trained on 0%, 0.1%, 1%, 10%, and 100% of the `kmeans` training set. COMPNETS and MAML initializations are the only initialization methods that produce an image with detail at a dataset size of 0%. Of the two, the COMPNET result has more definition. At a dataset size of 0.1%, all initialization methods produce images with detail. MAML- and randomly initialized surrogates produce images with duller colors than COMPNET-initialized and pretrained surrogates. At larger dataset sizes, all initialization methods converge to images that look similar to the reference image.

I.2. Quantitative Results

Figures 38 and 39 present quantitative results for 10- and 15-color palettes, respectively. At all color palette sizes



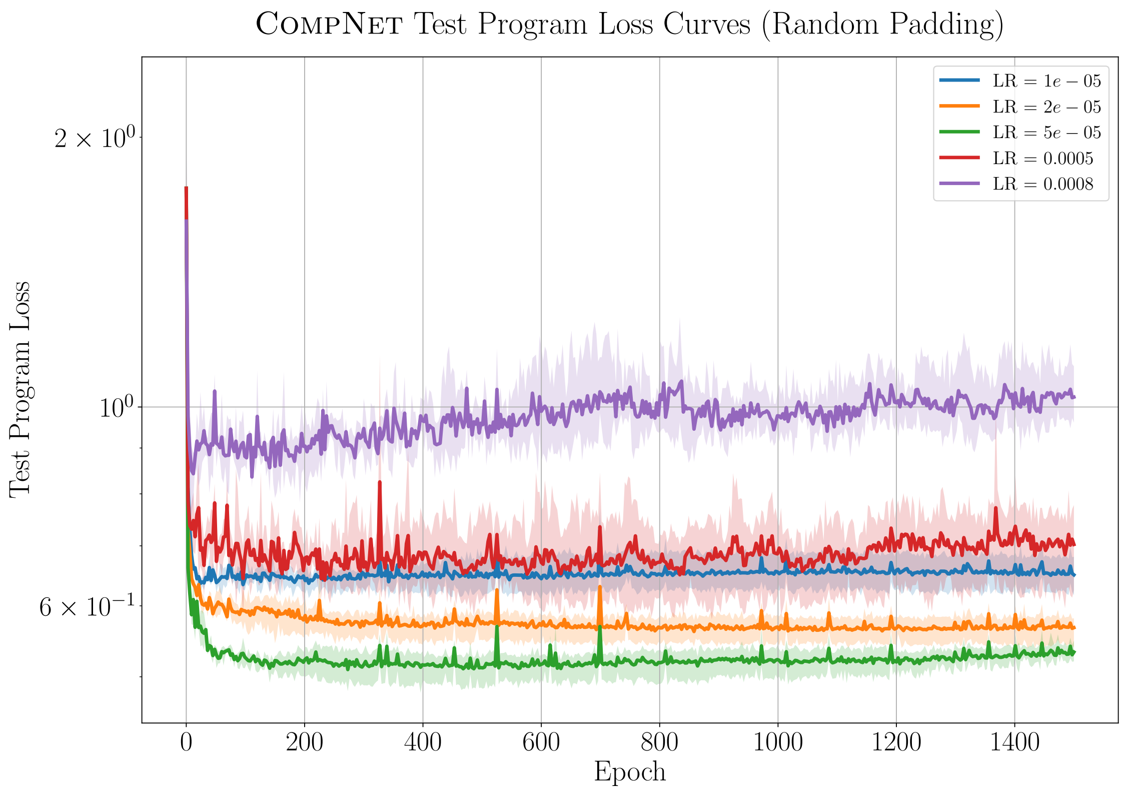
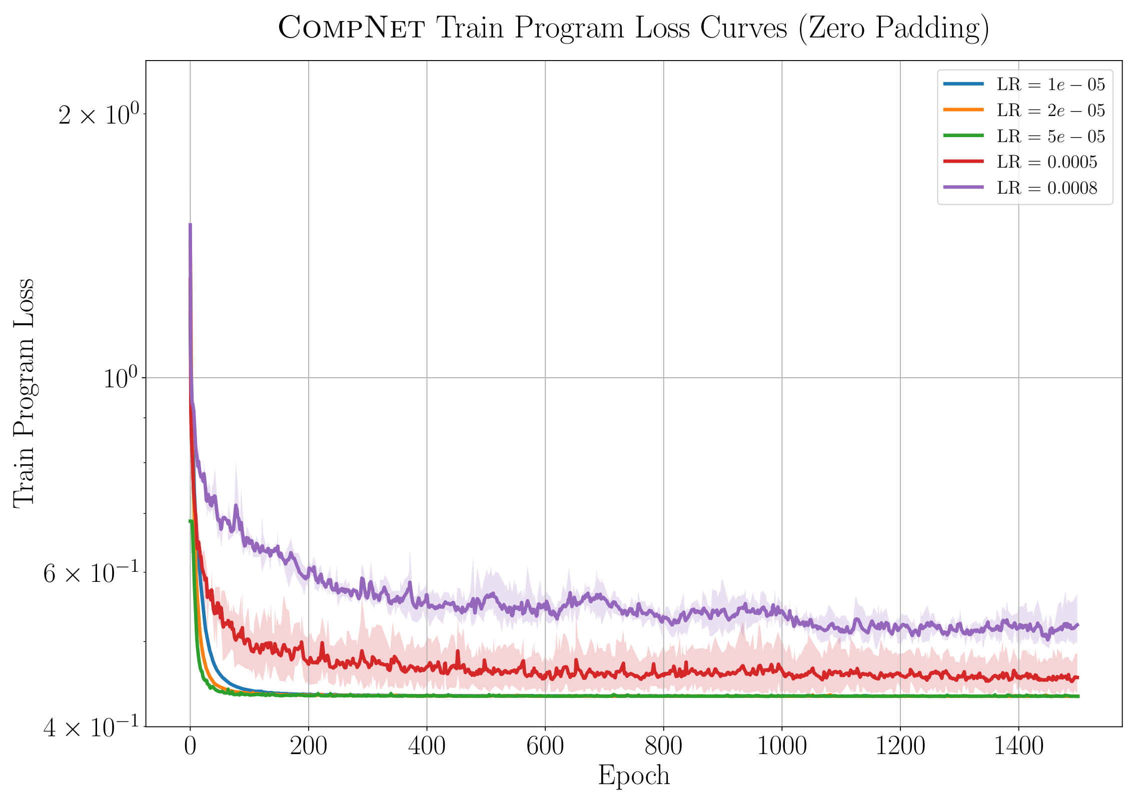


Figure 25: Loss curves for CompNet training hyperparameter sweep, using ExeStackCPN with random-padded inputs (see Appendix N). Each curve represents a training trial. The test program loss includes validation programs as well.



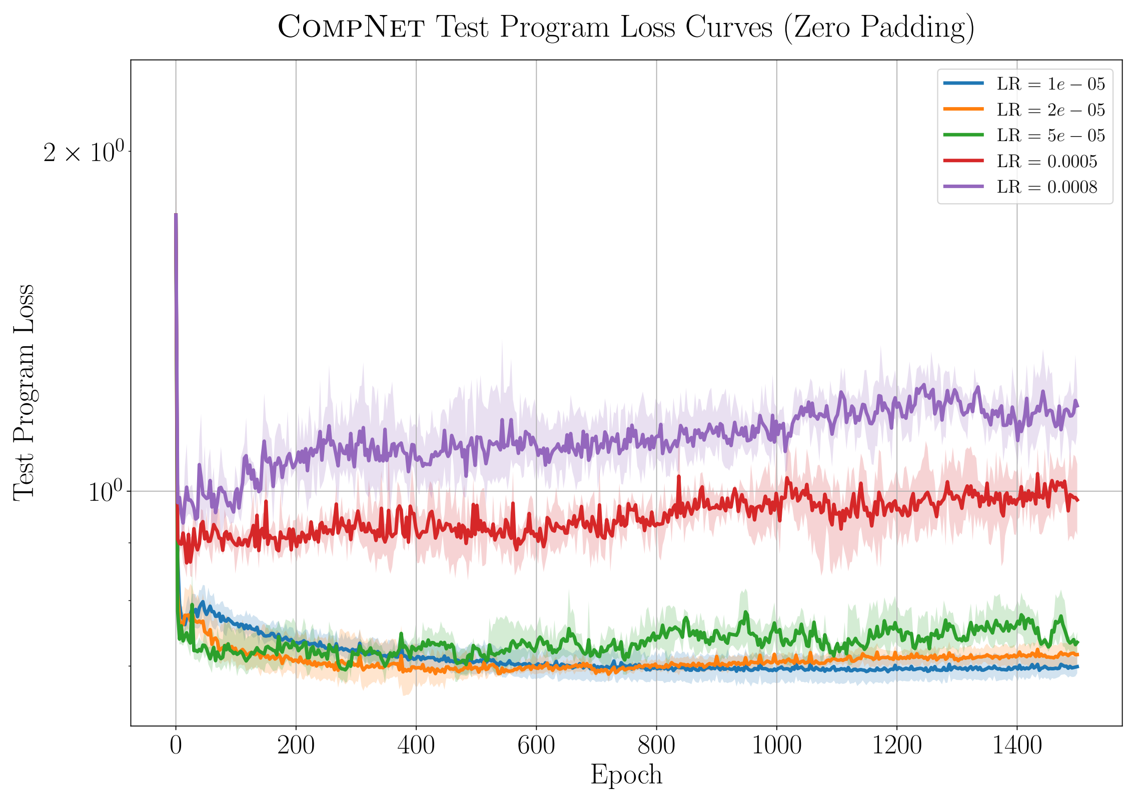


Figure 26: Loss curves for CompNet training hyperparameter sweep, using ExeStackCPN with zero-padded inputs (see Appendix N). Each curve represents a training trial. The test program loss includes validation programs as well.

Setting	Value
Meta Batch Size	32
Input Batch Size	1024
# Epochs	$70,000$
α	$\in \{0.01, 0.05, 0.1, \mathbf{0.2}\}$
β	0.001
# Inner Update Steps	$\in \{1, 2, \mathbf{3}, 4, 5\}$
# Finetuning Epochs	20
# Trials	3
Dataset Program Split	80/10/10
Dataset Input Split	50/20/30
Surrogate Topology	$9 \rightarrow 4 \rightarrow 4 \rightarrow 1$
GPU	NVIDIA Tesla T4 16GB

Figure 27: Training configuration for MAML. We represent any values we sweep over as a set, and we bold the values that obtain the best finetuning loss on validation programs.

and dataset sizes, COMPNET-initialized surrogates produce better results, in terms of both MSE and SSIM.

J. Training Time Improvements

To assess whether COMPNETs improve training time of neural surrogates, we use COMPNETs to initialize neural surrogates, finetune on training data until they reach a target test loss, then compare the results to those of other initialization methods. We first detail the methodology of this experiment, then present results.

J.0.1. METHODOLOGY

We now describe the methodology for setting a target test loss to use as a stopping condition, the configuration space we sweep over, how we quantify improvements, and how we visualize results.

Setting a Target Test Loss. We set a target test loss for each program by training 9 randomly initialized surrogates for 5,000 epochs. The average final test loss is the target test loss for all initialization methods.

Experiment Configurations. In this experiment, we sweep over configurations consisting of a program and an initialization method. Given a program and initialization method, we produce a neural surrogate initialization. We then train the initialized neural surrogate on the training input set until it reaches the target test loss or until it reaches 15,000 epochs. We call whichever epoch comes first the *finish epoch* for the trial.

Quantifying Improvements. We define the improvement for a given configuration (consisting of a program and initialization method) as the ratio of the finish epoch for random initialization and the finish epoch by the configuration's initialization method. All finish epochs are averaged over trials (using arithmetic mean) prior to computing ratios. For each initialization method, we report the geometric mean of the improvements grouped by program, grouped by dataset size, and overall. For some programs and initialization methods, the resulting surrogates achieve losses of 0. We discard these results before computing the geometric mean.

There are a few subtleties in this methodology. First, note that random initialization does not always have a finish epoch of 5,000, because the target error set after 5,000 epochs of training may have already been achieved earlier in training. Also, since the timeout epoch (15,000) is $3\times$ the baseline finish epoch (5,000), the worst case slowdown for each initialization method is $\frac{1}{3}\times$.

Visualizing Results. Since we evaluate on many programs in EXESTACKCPN, we plot the number of finished programs as a function of the number of epochs for each initialization method. For each program and initialization method, we calculate the finish epoch for that program as the average finish epoch over all instances of the initialization method and all trials for that instance.

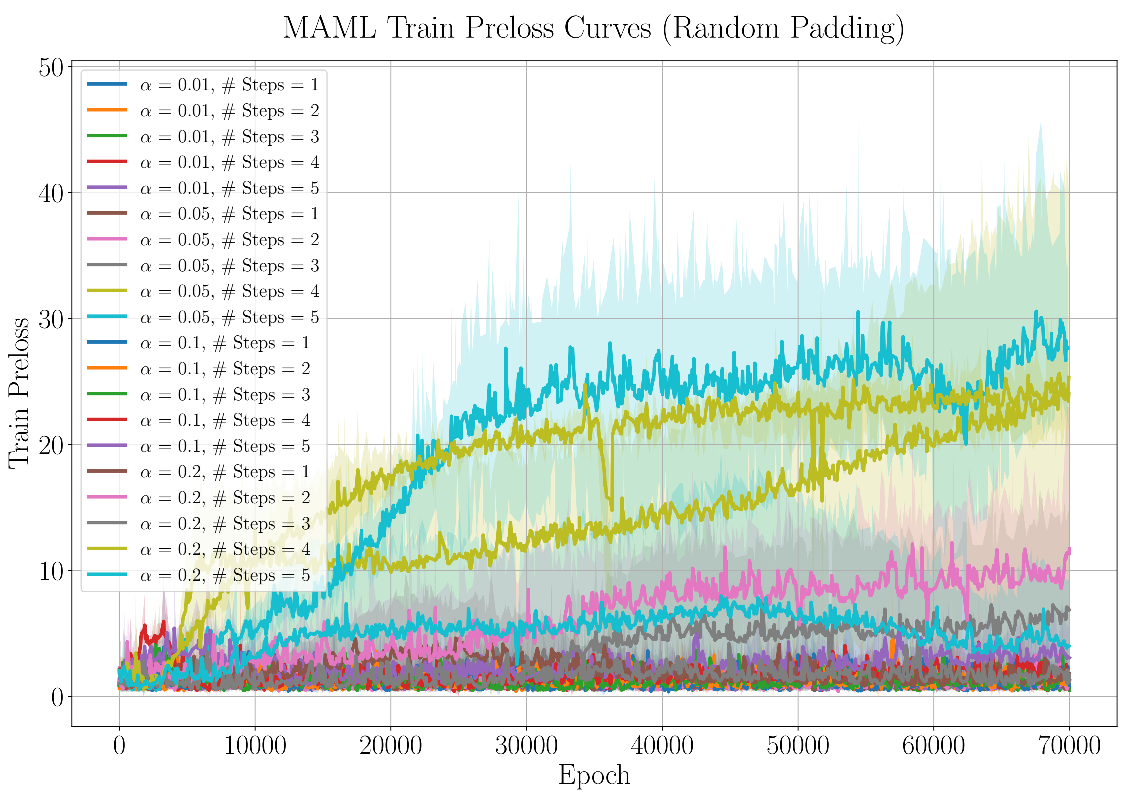
J.0.2. RESULTS

The results are summarized in Figures 40 and 41 for the sample of EXESTACKCPN test programs and Figures 42 and 43 for ParrotBenchCPN.

EXESTACKCPN Test Programs. COMPNETs achieve the best results on average, with a $7.28\times$ improvement over random initialization, whereas MAML and pretrained surrogates achieve $1.16\times$ and $0.93\times$ improvements, respectively. COMPNETs improve over random initialization in as low as the 18th perecentile, whereas MAML and pretrained surrogates improve over random initialization after the 36th and 48th percentile, respectively.

Until the $\approx$ 5,000th epoch, COMPNETs finish training on strictly more programs than all other initialization methods. At the 5,000th epoch, COMPNETs finish training for $\approx 90\%$ of programs. For the remaining 10% of programs, random initialization and MAML begin to overtake COMPNETs, at epochs $\approx$ 6,250 and $\approx$ 9,000, respectively.

PARROTBENCHCPN Programs. COMPNETs achieve the best results on average, with a $4.31\times$ improvement over random initialization, whereas MAML and pretrained surrogates achieve $1.07\times$ and $2.35\times$ improvements, respectively. COMPNETs improve over random initialization after the 50th perecentile, MAML improves over random initialization after the 54th percentile, and pretrained surrogates



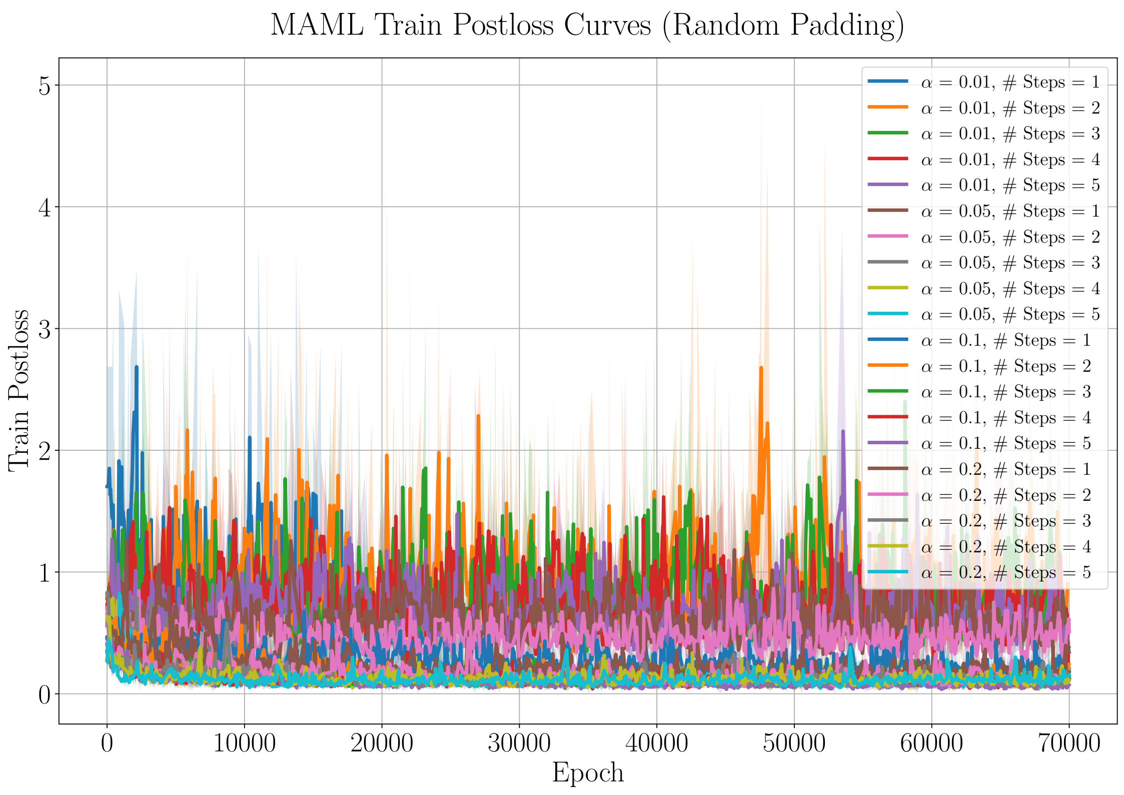
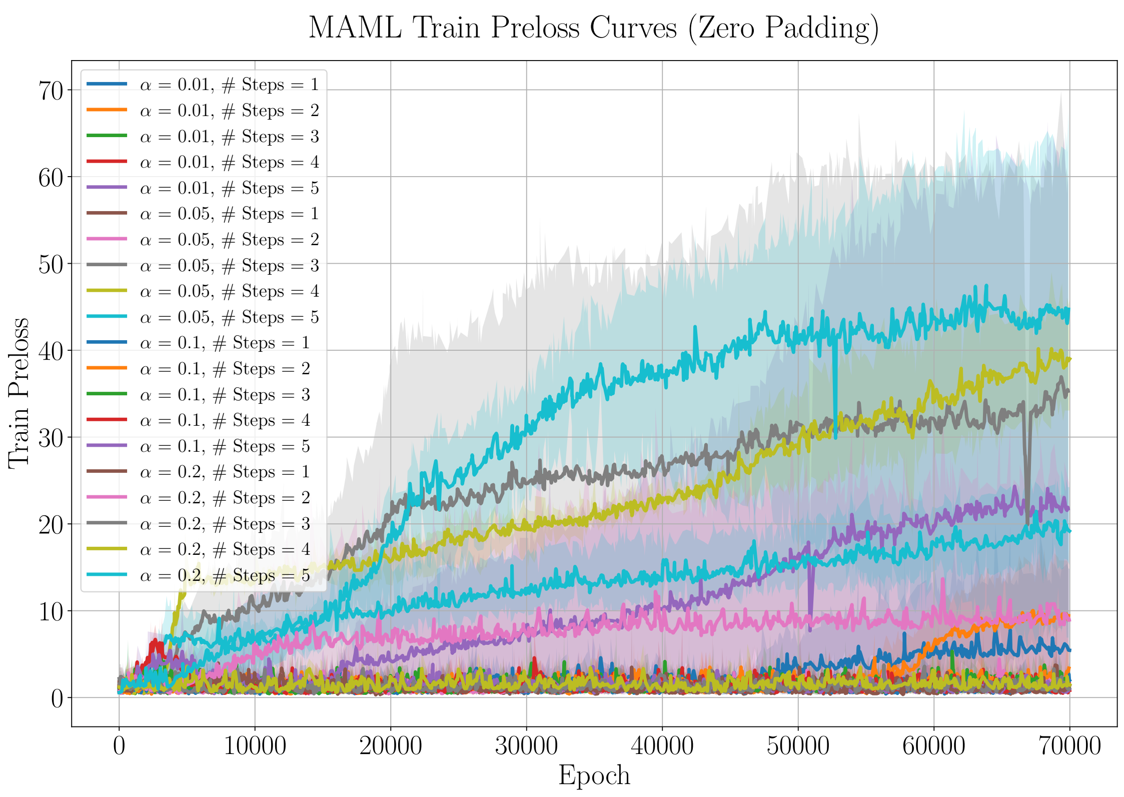


Figure 28: Loss curves for MAML training hyperparameter sweep, using ExeStackCPN with random-padded inputs (see Appendix N). Each curve represents a training trial.



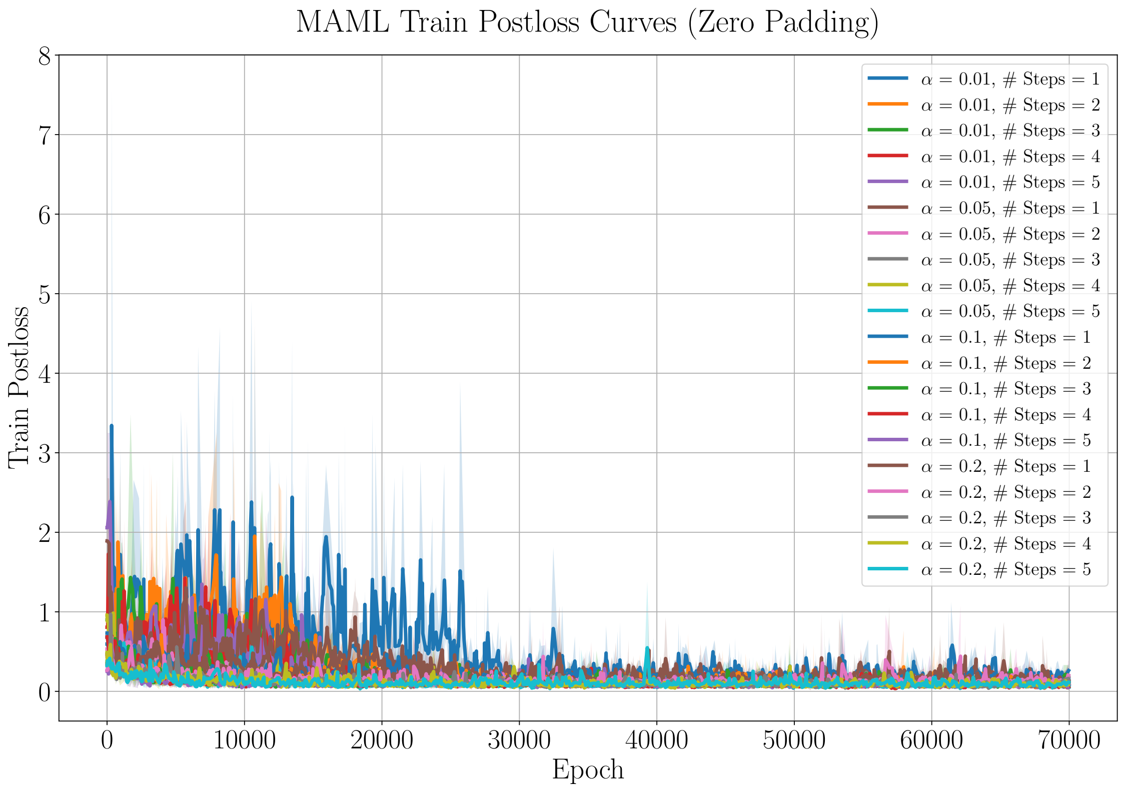


Figure 29: Loss curves for MAML training hyperparameter sweep, using EXESTACKCPN with zero-padded inputs (see Appendix N). Each curve represents a training trial.

Setting	Value
Program Batch Size	32
Input Batch Size	1024
Learning Rate	$\in \{\mathbf{1 \cdot 10^{-5}}, 2 \cdot 10^{-5}, 5 \cdot 10^{-5}, 5 \cdot 10^{-4}, 8 \cdot 10^{-4}\}$
# Epochs	$1,500$
Dataset Program Split	$80/0/20$
Dataset Input Split	$50/0/50$
Surrogate Topology	$9 \to 4 \to 4 \to 1$
GPU	NVIDIA Tesla T4 16GB
# Trials	3

Figure 30: Training configuration for pretrained surrogates. We represent any values we sweep over as a set, and we bold the values that obtain the best final loss on test programs.

improve over random initialization after the 48th percentile.

COMPNETs range between improvements of $0.49\times$ on `invk2j` to $674\times$ on `kmeans`. The variance between other techniques is smaller, with MAML varying between $0.65\times$ on `invk2j` and $2.15\times$ on `kmeans`, and pretrained surrogates varying between $0.56\times$ on `invk2j` and $87\times$ on `kmeans`. All initialization methods present slowdowns on `invk2j` and speedups on `kmeans`, so it is possible `ExeStackCPN` does not include similar computations to `invk2j` but does include similar computations to `kmeans`. However, we use an extensive decontamination methodology (see Appendix C.1), so we conclude these similarities are abstract in nature.

Since COMPNETs improve training time over random initialization for programs in both EXESTACKCPN and PARROTBENCHCPN, we answer yes to RQ 2.

Additional Data. We present the initial train losses, initial test lossses, and target test losses for both EXESTACK-CPN (Figure 44) and PARROTBENCHCPN (Figures 45, 46, and 47). We also present the average finish epoch (Figure 48) for each initialization method and the number of timeouts (Figure 49) on PARROTBENCHCPN programs.

K. PARROTBENCHCPN True vs. Predicted Functions

In this appendix, we present graphs showing the function each PARROTBENCHCPN program implements, as well as the approximations of the function each initialization method produces in the data efficiency evaluation of Section 5.2. To visualize the behavior of multivariate functions, we generate a graph for each argument, where we vary that argument and fix all other arguments to zero. We include graphs for the training set at each of the dataset sizes we evaluated on in Section 5.2 (i.e., $\{0\%, 0.1\%, 1\%, 10\%, 100\%\}$). Each line is an average over compilers of that type (e.g., all the COMPNETs we trained for the CPN compiler type) and all trials for those compilers. The fill-between for each compiler type shows the minimum and maximum predictions across each instance of that compiler type and each surrogate initialized by that instance.

Figure 50 shows results for `fft`, Figures 51 and 52 show results for both inputs of `invk2j`, Figures 53, 54, 55, 56, 57, and 58 show results for each input of `kmeans`, and Figures 59, 60, 61, 62, 63, 64, 65, 66, and 67 show results for each input of `sobel`.

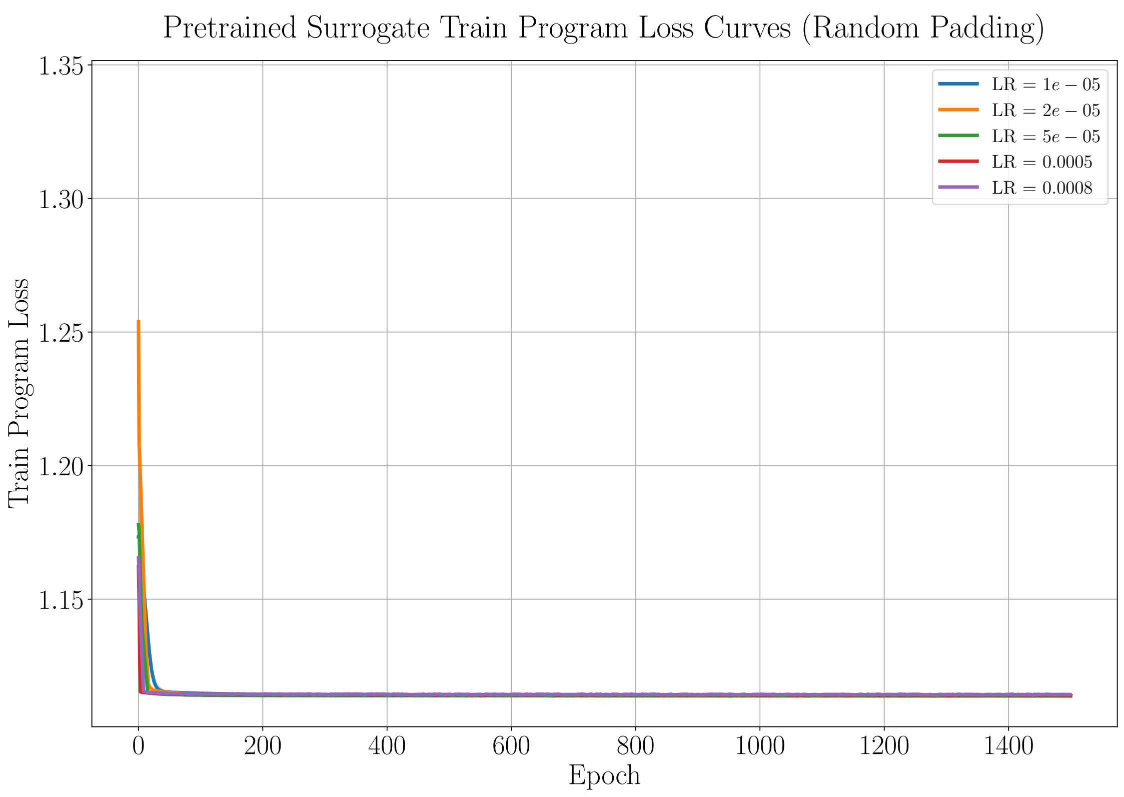
L. Neural Surrogates Achieve Acceptable Error

In this section, we show that, in the context of our evaluation, the error incurred from using neural surrogates is satisfactory for downstream applications. We first show that the surrogates of Esmaeilzadeh et al. (2012a) achieve acceptable end-to-end error on PARROTBENCHCPN programs, then we show that our surrogates achieve commensurate or lower error than their surrogates.

L.1. End-to-End Error

Esmaeilzadeh et al. calculate end-to-end error for the benchmarks we consider from PARROTBENCH as follows:

- **`fft`.** Apply the fast Fourier transform to a sequence of 2,048 values, where the value at the ith index is i, and measure the average relative error between the output of the original `fft` implementation and the approximate `fft` implementation.
- **`invk2j`.** Generate 1,000 pairs of joint angles (θ_1, θ_2), with both angles sampled uniformly at random from $[0, \pi/2]$. Run forward kinematics on these angles, to obtain (x, y) coordinates for the tip of the joint arm. Run inverse kinematics on these (x, y) coordinates, to obtain joint angles $(\tilde{\theta_1}, \tilde{\theta_2})$ that place the tip of



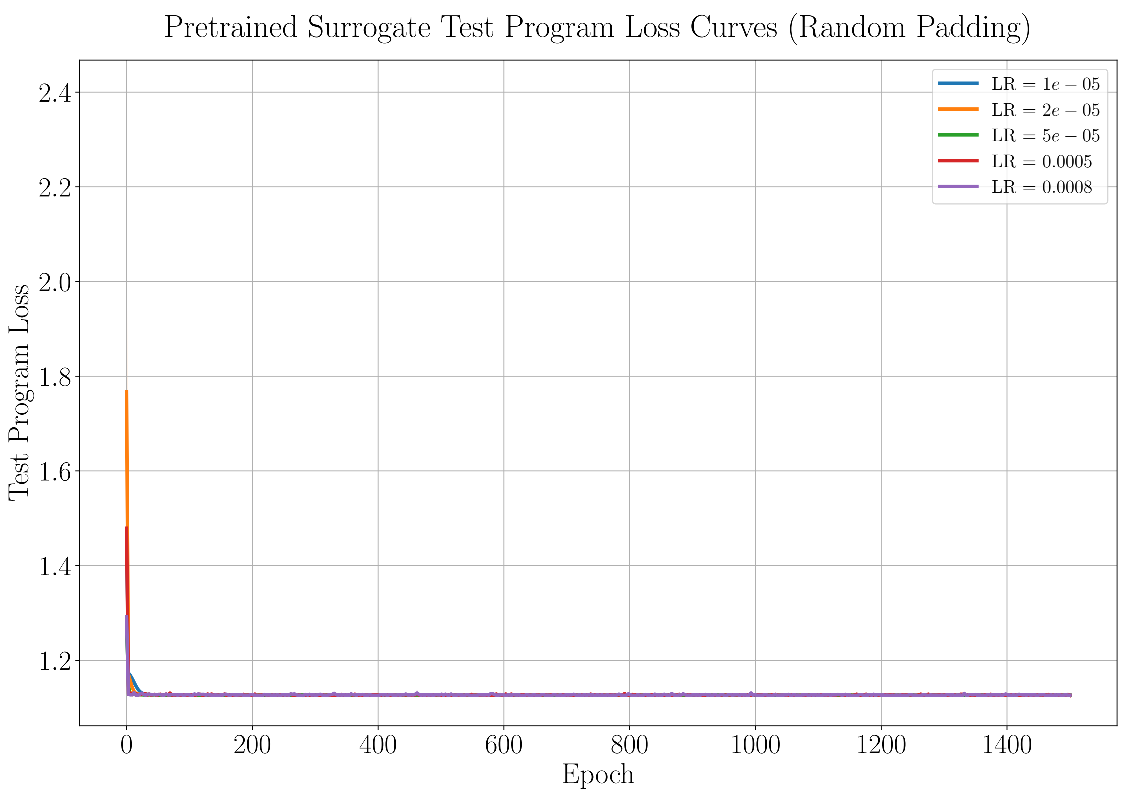
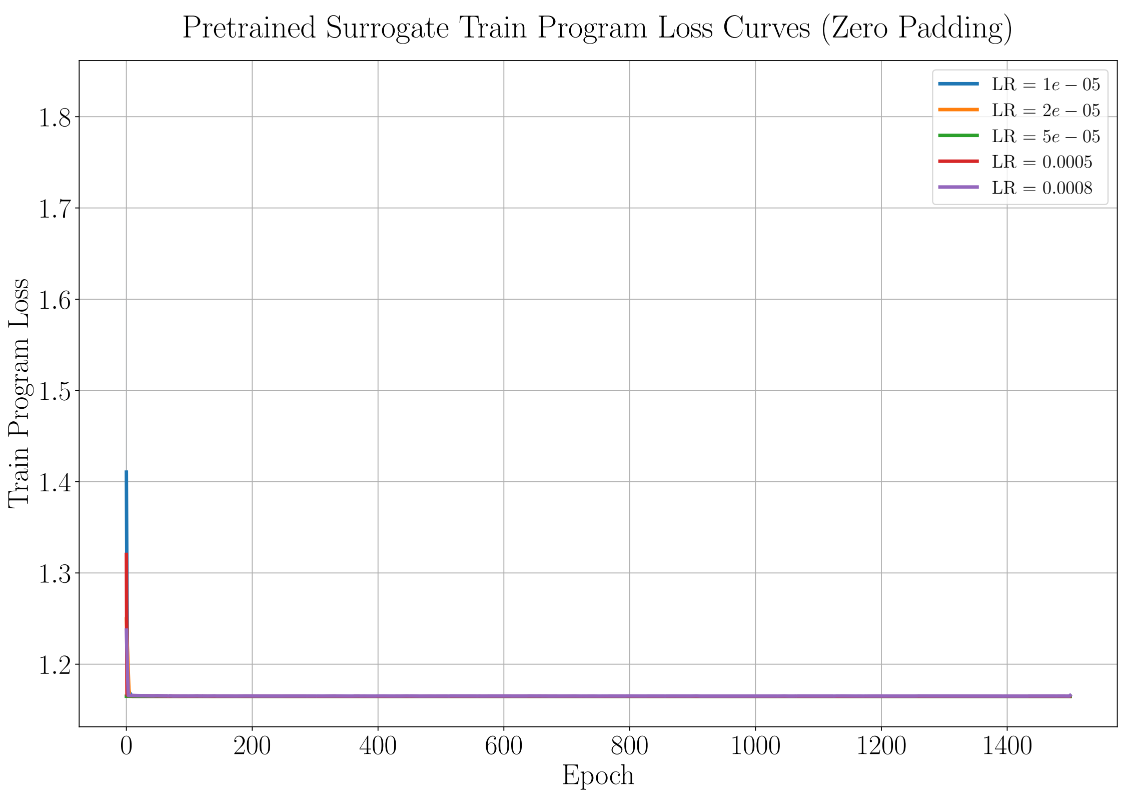


Figure 31: Loss curves for pretrained surrogate hyperparameter sweep, using EXESTACKCPN with random-padded inputs (see Appendix N). Each curve represents a training trial. The test program loss includes validation programs as well.



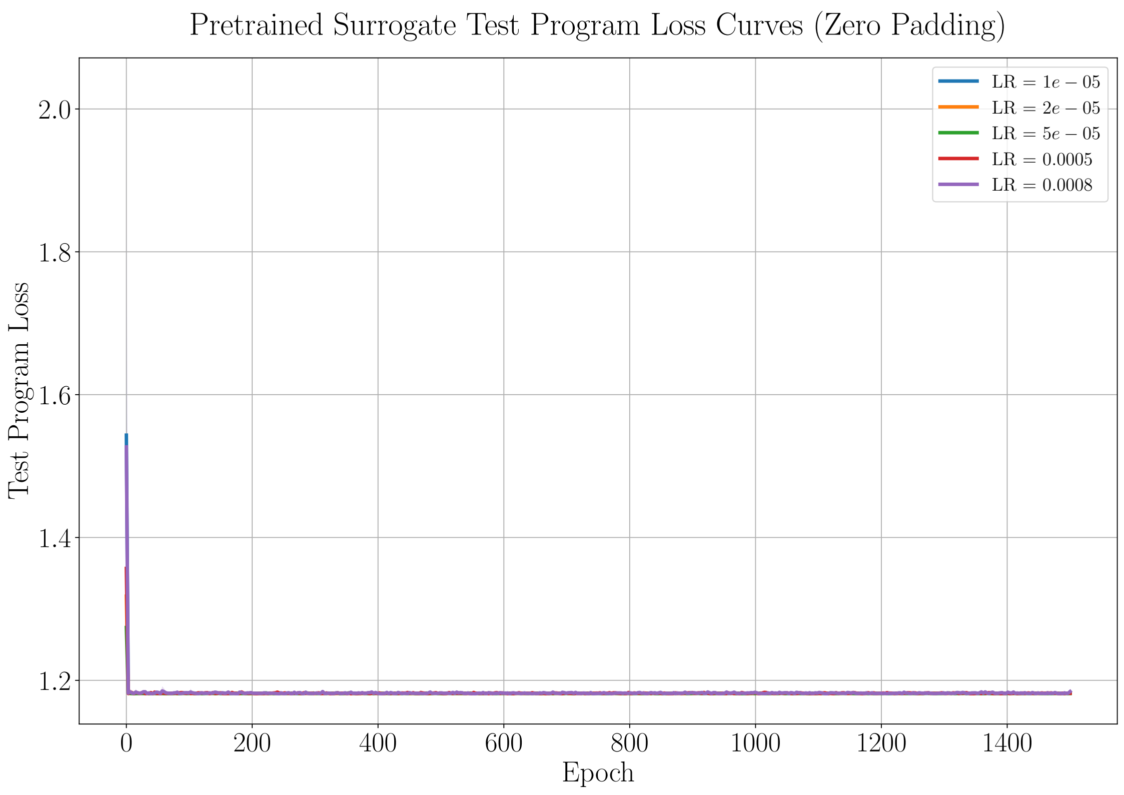


Figure 32: Loss curves for pretrained surrogate hyperparameter sweep, using EXESTACKCPN with zero-padded inputs (see Appendix N). Each curve represents a training trial. The test program loss includes validation programs as well.

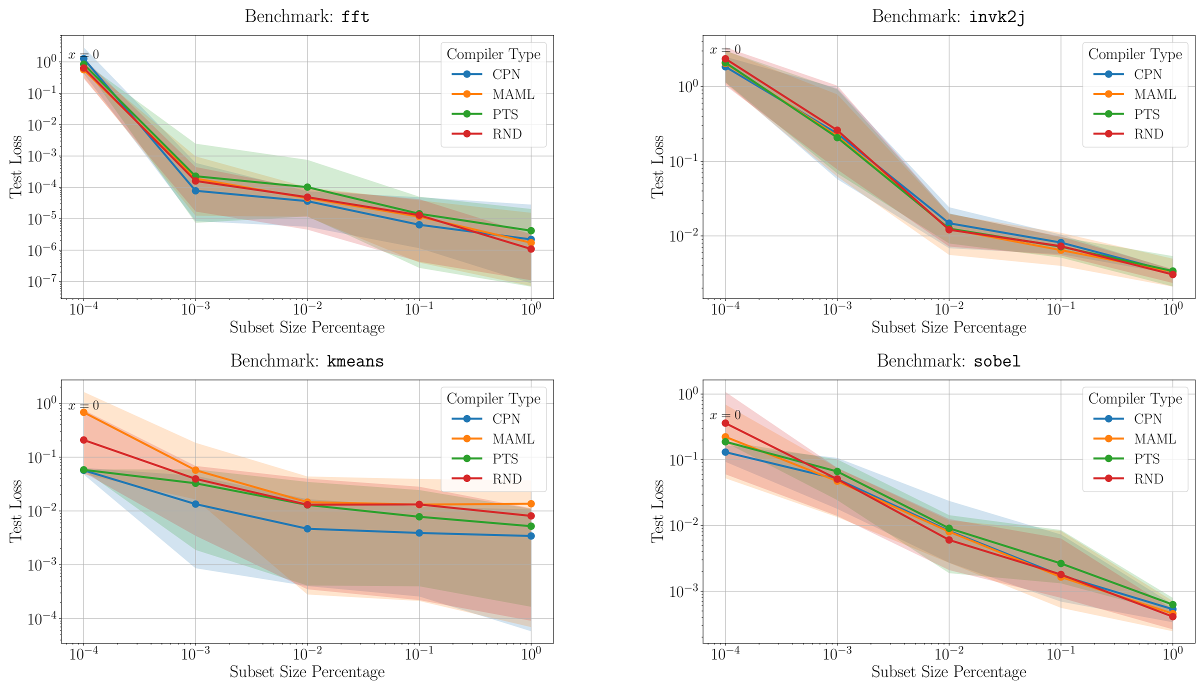


Figure 33: Log-log plot showing dataset size vs. test loss for each initialization method on each benchmark in ParrotBenchCPN. We present the test loss at the epoch with the lowest validation loss. When the validation set is empty, we use the test loss at the final epoch. Since zero is not a valid point on a logarithmic scale, we include the loss values for the empty dataset at a nonzero point on the x axis that is also smaller than the smallest nonzero dataset size, and we label it with "$x = 0$".

the joint arm at (x, y). Measure the average relative error between the joint angles recovered by the original `invk2j` implementation and the approximate `invk2j` implementation.

- **`kmeans`.** Apply one iteration of k-means clustering to each pixel of the image in Figure 23, then set each pixel's color to the color of the closest centroid. Measure the average root mean squared error between the image produced by the original `kmeans` implementation and the approximate `kmeans` implementation.

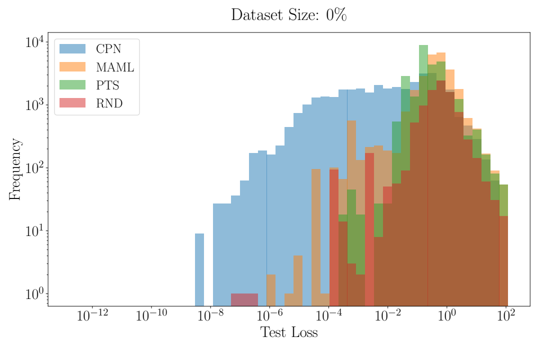
- **`sobel`.** Convert the image in Figure 23 to grayscale using a weighted average of 30% red, 59% green, and 11% blue. Apply the `sobel` filter to the first row of the image, the first column, and the last row. Measure the average root mean squared error between the image produced by the original `sobel` implementation and the approximate `sobel` implementation.

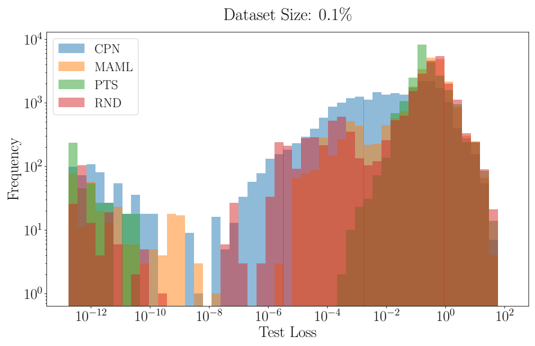
Figure 68 shows the end-to-end error of neural surrogates of ParrotBenchCPN programs, which are semantically equivalent to a subset of the benchmarks Esmaeilzadeh et al. evaluated on. The end-to-end error for each benchmark is $\leq 7.5\%$, and an end-to end quality loss of 10% or more is common in the approximate computing literature (Esmaeilzadeh et al., 2012a;b; Sampson et al., 2011; Baek & Chilimbi, 2010; Misailovic et al., 2010). For example, Park et al. develop neural surrogates of programs for image processing, audio processing, and speech processing, and they collect user feedback on the perceptual quality of the approximate programs (Park et al., 2016). Their results show that, on a majority of the benchmarks they consider, a quality loss of $\geq 10\%$ is deemed acceptable by $\geq 80\%$ of users. The neural surrogates we train achieve commensurate and often lower test error than the surrogates of Esmaeilzadeh et al. (2012a). Thus, the neural surrogates we train achieve an acceptable level of approximation.

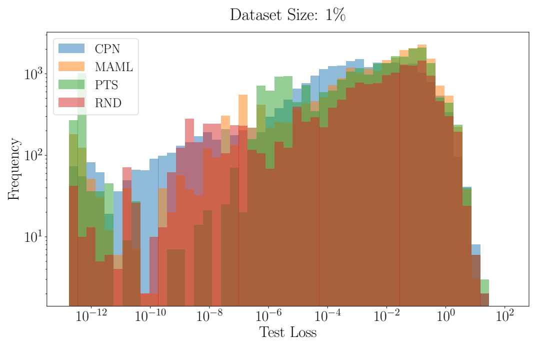
M. Downcasting Incurs Negligible Error

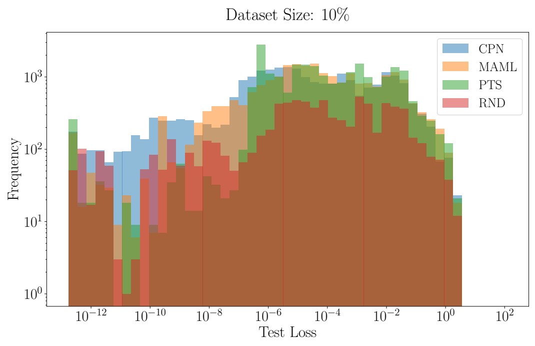
The neural surrogate architectures we target uses a single-precision floating-point data type, but many of the programs we compile in Section 5 use double-precision data types. For example, 59% of ExeStackCPN programs use at least one double-precision datatype and 56% of ExeStackCPN programs use exclusively double-precision datatypes. We now show this implicit downcasting incurs low error relative to overall neural surrogate approximation error.

Methodology. We generate two versions of each ParrotBenchCPN program: one using only the `float` type and one using only the `double` type. This replacement









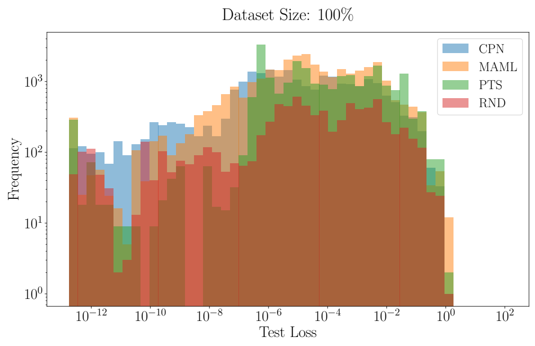


Figure 34: Histograms showing testing input losses of each initialization method at the epoch with the lowest validation loss for EXESTACKCPN testing programs.

Dataset Size 0%				
Program	CPN	MAML	PTS	RND
`fft`	1.3 ± 1.2	0.6 ± 0.2	0.8 ± 0.1	0.6 ± 0.3
`invk2j`	1.8 ± 0.6	2.0 ± 0.6	2.1 ± 0.5	2.4 ± 0.8
`kmeans`	$0.1 \pm 6.7 \cdot 10^{-3}$	0.7 ± 0.4	$0.1 \pm 7.4 \cdot 10^{-4}$	0.2 ± 0.2
`sobel`	$0.1 \pm 3.0 \cdot 10^{-2}$	0.2 ± 0.2	$0.2 \pm 3.5 \cdot 10^{-3}$	0.4 ± 0.3

Dataset Size 0.1%				
Program	CPN	MAML	PTS	RND
`fft`	$7.8 \cdot 10^{-5} \pm 1.1 \cdot 10^{-4}$	$1.9 \cdot 10^{-4} \pm 2.4 \cdot 10^{-4}$	$2.3 \cdot 10^{-4} \pm 4.6 \cdot 10^{-4}$	$1.6 \cdot 10^{-4} \pm 1.1 \cdot 10^{-4}$
`invk2j`	0.2 ± 0.2	0.2 ± 0.2	0.2 ± 0.2	0.3 ± 0.3
`kmeans`	$1.3 \cdot 10^{-2} \pm 1.5 \cdot 10^{-2}$	$0.1 \pm 2.9 \cdot 10^{-2}$	$3.3 \cdot 10^{-2} \pm 1.6 \cdot 10^{-2}$	$3.9 \cdot 10^{-2} \pm 2.1 \cdot 10^{-2}$
`sobel`	$0.1 \pm 2.0 \cdot 10^{-2}$	$4.7 \cdot 10^{-2} \pm 2.1 \cdot 10^{-2}$	$0.1 \pm 2.3 \cdot 10^{-2}$	$0.1 \pm 1.9 \cdot 10^{-2}$

Dataset Size 1%				
Program	CPN	MAML	PTS	RND
`fft`	$3.6 \cdot 10^{-5} \pm 2.7 \cdot 10^{-5}$	$4.6 \cdot 10^{-5} \pm 2.2 \cdot 10^{-5}$	$1.0 \cdot 10^{-4} \pm 1.4 \cdot 10^{-4}$	$4.9 \cdot 10^{-5} \pm 2.8 \cdot 10^{-5}$
`invk2j`	$1.5 \cdot 10^{-2} \pm 4.7 \cdot 10^{-3}$	$1.2 \cdot 10^{-2} \pm 3.7 \cdot 10^{-3}$	$1.3 \cdot 10^{-2} \pm 3.8 \cdot 10^{-3}$	$1.2 \cdot 10^{-2} \pm 3.9 \cdot 10^{-3}$
`kmeans`	$4.7 \cdot 10^{-3} \pm 5.9 \cdot 10^{-3}$	$1.5 \cdot 10^{-2} \pm 1.1 \cdot 10^{-2}$	$1.3 \cdot 10^{-2} \pm 9.3 \cdot 10^{-3}$	$1.3 \cdot 10^{-2} \pm 1.3 \cdot 10^{-2}$
`sobel`	$8.3 \cdot 10^{-3} \pm 4.9 \cdot 10^{-3}$	$8.1 \cdot 10^{-3} \pm 2.8 \cdot 10^{-3}$	$9.1 \cdot 10^{-3} \pm 3.2 \cdot 10^{-3}$	$6.0 \cdot 10^{-3} \pm 3.2 \cdot 10^{-3}$

Dataset Size 10%				
Program	CPN	MAML	PTS	RND
`fft`	$6.4 \cdot 10^{-6} \pm 9.0 \cdot 10^{-6}$	$1.2 \cdot 10^{-5} \pm 1.2 \cdot 10^{-5}$	$1.4 \cdot 10^{-5} \pm 1.5 \cdot 10^{-5}$	$1.3 \cdot 10^{-5} \pm 1.6 \cdot 10^{-5}$
`invk2j`	$8.1 \cdot 10^{-3} \pm 1.4 \cdot 10^{-3}$	$6.5 \cdot 10^{-3} \pm 1.7 \cdot 10^{-3}$	$7.2 \cdot 10^{-3} \pm 1.3 \cdot 10^{-3}$	$7.3 \cdot 10^{-3} \pm 1.4 \cdot 10^{-3}$
`kmeans`	$3.9 \cdot 10^{-3} \pm 5.0 \cdot 10^{-3}$	$1.3 \cdot 10^{-2} \pm 7.9 \cdot 10^{-3}$	$7.8 \cdot 10^{-3} \pm 7.0 \cdot 10^{-3}$	$1.3 \cdot 10^{-2} \pm 1.1 \cdot 10^{-2}$
`sobel`	$1.7 \cdot 10^{-3} \pm 1.7 \cdot 10^{-3}$	$1.6 \cdot 10^{-3} \pm 1.4 \cdot 10^{-3}$	$2.6 \cdot 10^{-3} \pm 1.9 \cdot 10^{-3}$	$1.8 \cdot 10^{-3} \pm 1.6 \cdot 10^{-3}$

Dataset Size 100%				
Program	CPN	MAML	PTS	RND
`fft`	$2.2 \cdot 10^{-6} \pm 5.4 \cdot 10^{-6}$	$1.7 \cdot 10^{-6} \pm 3.1 \cdot 10^{-6}$	$4.2 \cdot 10^{-6} \pm 5.6 \cdot 10^{-6}$	$1.1 \cdot 10^{-6} \pm 1.1 \cdot 10^{-6}$
`invk2j`	$3.3 \cdot 10^{-3} \pm 1.5 \cdot 10^{-4}$	$3.3 \cdot 10^{-3} \pm 6.5 \cdot 10^{-4}$	$3.4 \cdot 10^{-3} \pm 7.2 \cdot 10^{-4}$	$3.0 \cdot 10^{-3} \pm 4.8 \cdot 10^{-4}$
`kmeans`	$3.4 \cdot 10^{-3} \pm 4.7 \cdot 10^{-3}$	$1.4 \cdot 10^{-2} \pm 9.5 \cdot 10^{-3}$	$5.2 \cdot 10^{-3} \pm 4.8 \cdot 10^{-3}$	$8.1 \cdot 10^{-3} \pm 4.3 \cdot 10^{-3}$
`sobel`	$5.3 \cdot 10^{-4} \pm 7.9 \cdot 10^{-5}$	$4.6 \cdot 10^{-4} \pm 1.1 \cdot 10^{-4}$	$6.3 \cdot 10^{-4} \pm 8.5 \cdot 10^{-5}$	$4.1 \cdot 10^{-4} \pm 8.1 \cdot 10^{-5}$

Figure 35: Average test loss achieved by each initialization method on the epoch with the best validation loss for ParrotBenchCPN programs. We include a table for each dataset size we evaluated on.

Benchmark: `fft`			
Dataset Size	CPN	MAML	PTS
0%	0.76×	1.13×	0.76×
0.1%	2.61×	0.84×	0.77×
1%	1.94×	1.08×	0.54×
10%	2.54×	1.17×	0.98×
100%	0.70×	0.75×	0.26×

Benchmark: `invk2j`			
Dataset Size	CPN	MAML	PTS
0%	1.35×	1.15×	1.12×
0.1%	1.14×	1.21×	1.26×
1%	0.83×	0.98×	0.97×
10%	0.90×	1.13×	1.02×
100%	0.93×	0.92×	0.90×

Benchmark: `kmeans`			
Dataset Size	CPN	MAML	PTS
0%	3.68×	0.31×	3.58×
0.1%	5.24×	0.70×	1.23×
1%	8.22×	0.94×	1.08×
10%	11.97×	1.04×	3.30×
100%	15.74×	0.68×	3.59×

Benchmark: `sobel`			
Dataset Size	CPN	MAML	PTS
0%	2.84×	1.63×	1.92×
0.1%	1.00×	1.08×	0.77×
1%	0.75×	0.75×	0.67×
10%	1.17×	1.11×	0.70×
100%	0.77×	0.90×	0.66×

Figure 36: Geometric mean testing loss improvement over randomly initialized surrogates on PARROTBENCHCPN programs, grouped by both programs and dataset sizes.

includes arguments, internal variables, and any casts. We then generate random double-precision inputs according to the methodology in Section D.3 and execute each version of each program. We report the mean squared error (MSE) between the outputs of the single- and double-precision versions of each program in Figure 68. We deem a downcasting error acceptable if it is an order of magnitude smaller than the error incurred by using neural surrogates at all, compared to the original implementation (see Appendix L).

Results. Figure 69 shows the downcasting error for each PARROTBENCHCPN program, which we compare to Figure L from Appendix L. The downcasting error of `fft` is significantly smaller than the surrogate error ($1.12 \cdot 10^{-14}$ vs. $2.0 \cdot 10^{-5}$). The downcasting error of `invk2j` is significantly smaller than the surrogate error ($1.6 \cdot 10^{-11}$ vs. $5.6 \cdot 10^{-3}$). Surprisingly, the downcasting error of `kmeans` is too small to be captured by a floating-point data type, so it registers as 0.0. The downcasting error of `sobel` is significantly smaller than the surrogate error ($6.51 \cdot 10^{-8}$ vs. $2.3 \cdot 10^{-3}$). We conclude that downcasting from double-precision data types does not significantly affect the overall neural surrogate approximation, error, which we have already shown is acceptable in Appendix L.

N. Variable-Input Support for Initialization Methods

COMPNETS, MAML, and pretrained surrogates each produce a fixed-size weight vector for initializing surrogates. However, programs in EXESTACKCPN and PARROTBENCHCPN have various numbers of inputs. To support programs in these datasets, we develop strategies for adapting these initialization methods, and we present a methodology for choosing the best of these strategies.

Variable-Input Strategies. To develop variable-input initialization methods, we chose a vector size with as many parameters as the architecture with the largest number of inputs we wish to support, defined as the *covering architecture* in Section 3, and we developed strategies for supplying data to unused inputs.

There are two types of padding data we considered: randomly distributed and constantly zero. With random padding, any excess inputs are supplied with values from the same distribution as the primary inputs. With zero padding, any excess inputs are supplied with zeroes.

There are three phases in which data is supplied to an initialization method: training the initialization method, finetuning surrogates initialized by the method, and evaluating surrogates initialized by the method. Thus, we categorize the strategies we consider by the type of data the initialization method is trained on, the type of data the initialized surrogates are finetuned on, and the type of data the initialized surrogates are evaluated on. We considered most permutations of random and zero padding for each of these three phases. Notably, however, we did not consider the family of strategies where one finetunes on zero-padded inputs and evaluates on random-padded inputs because it seemed unlikely that adding a new source of noise at inference time would lead to any improvement.

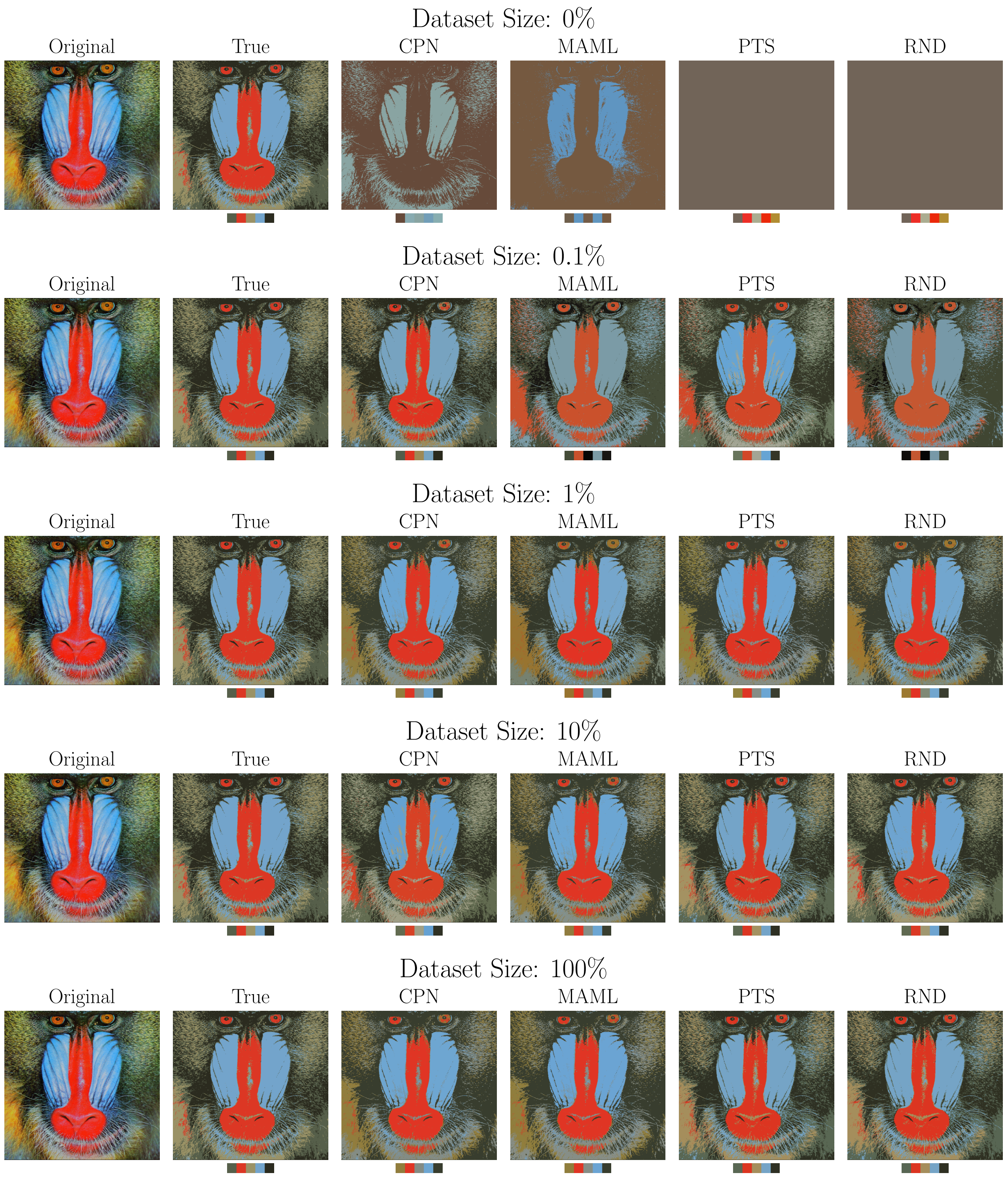


Figure 37: Color quantization results for a ground-truth NumPy implementation ("True") vs. approximate implementations over various dataset sizes, each on a separate row. In each row, the original image of a baboon is on the left, followed by images transformed to adhere to a palette of 5 colors.

Dataset Size	CPN	MAML	PTS	RND
0%	$\mathbf{2.87 \cdot 10^3} \pm 615.$	$3.03 \cdot 10^3 \pm 394.$	$3.28 \cdot 10^3 \pm 129.$	$3.30 \cdot 10^3 \pm 0.0$
0.1%	$\mathbf{979.} \pm 853.$	$1.90 \cdot 10^3 \pm 594.$	$1.72 \cdot 10^3 \pm 793.$	$1.46 \cdot 10^3 \pm 583.$
1%	$\mathbf{410.} \pm 194.$	$677. \pm 267.$	$631. \pm 226.$	$615. \pm 298.$
10%	$\mathbf{401.} \pm 181.$	$639. \pm 169.$	$576. \pm 252.$	$631. \pm 317.$
100%	$\mathbf{395.} \pm 184.$	$627. \pm 237.$	$498. \pm 229.$	$510. \pm 154.$

Dataset Size	CPN	MAML	PTS	RND
0%	$\mathbf{0.28} \pm 0.13$	0.20 ± 0.04	0.19 ± 0.03	0.19 ± 0.00
0.1%	$\mathbf{0.60} \pm 0.16$	0.42 ± 0.12	0.45 ± 0.16	0.49 ± 0.09
1%	$\mathbf{0.73} \pm 0.10$	0.63 ± 0.11	0.64 ± 0.09	0.66 ± 0.12
10%	$\mathbf{0.74} \pm 0.10$	0.62 ± 0.07	0.66 ± 0.12	0.65 ± 0.14
100%	$\mathbf{0.74} \pm 0.10$	0.63 ± 0.11	0.69 ± 0.12	0.68 ± 0.09

Figure 38: Quantitative comparison of end-to-end results produced by various initialization methods on color quantization with a palette size of 10 colors. **(Top)** The average mean squared error (MSE) of the image produced by each initialization method compared to the image produced by a ground-truth implementation of the `kmeans` kernel (lower is better). **(Bottom)** The average structural similarity index measure (SSIM) of the image produced by each initialization method compared to the image produced by a ground-truth implementation of the `kmeans` kernel (higher is better).

Dataset Size	CPN	MAML	PTS	RND
0%	$\mathbf{2.74 \cdot 10^3} \pm 506.$	$3.12 \cdot 10^3 \pm 386.$	$3.39 \cdot 10^3 \pm 81.$	$3.40 \cdot 10^3 \pm 0.0$
0.1%	$\mathbf{906.} \pm 782.$	$1.84 \cdot 10^3 \pm 633.$	$1.74 \cdot 10^3 \pm 801.$	$1.53 \cdot 10^3 \pm 783.$
1%	$\mathbf{417.} \pm 166.$	$647. \pm 250.$	$588. \pm 206.$	$588. \pm 250.$
10%	$\mathbf{404.} \pm 163.$	$578. \pm 155.$	$545. \pm 207.$	$577. \pm 279.$
100%	$\mathbf{392.} \pm 150.$	$588. \pm 204.$	$477. \pm 177.$	$484. \pm 100.$

Dataset Size	CPN	MAML	PTS	RND
0%	$\mathbf{0.31} \pm 0.11$	0.18 ± 0.04	0.16 ± 0.02	0.16 ± 0.00
0.1%	$\mathbf{0.61} \pm 0.15$	0.42 ± 0.12	0.44 ± 0.16	0.48 ± 0.12
1%	$\mathbf{0.71} \pm 0.07$	0.63 ± 0.09	0.65 ± 0.08	0.66 ± 0.08
10%	$\mathbf{0.72} \pm 0.08$	0.64 ± 0.06	0.66 ± 0.08	0.66 ± 0.11
100%	$\mathbf{0.73} \pm 0.07$	0.64 ± 0.08	0.69 ± 0.07	0.68 ± 0.05

Figure 39: Quantitative comparison of end-to-end results produced by various initialization methods on color quantization with a palette size of 15 colors. **(Top)** The average mean squared error (MSE) of the image produced by each initialization method compared to the image produced by a ground-truth implementation of the `kmeans` kernel (lower is better). **(Bottom)** The average structural similarity index measure (SSIM) of the image produced by each initialization method compared to the image produced by a ground-truth implementation of the `kmeans` kernel (higher is better).

Statistic	CPN	MAML	PTS
0th	0.03×	**0.06**×	0.03×
25th	**1.16**×	0.85×	0.61×
50th	**3.43**×	1.19×	1.03×
75th	**23.96**×	1.68×	1.56×
100th	$\mathbf{8.27 \cdot 10^3}\times$	26.54×	49.39×
MPI	**18**th	36th	48th
GM	**7.28**×	1.16×	0.93×

Figure 40: Geometric mean improvements and percentile improvements to training time over randomly initialized surrogates on a sample of 1,000 EXESTACKCPN test programs. MPI is the minimum percentile at which an initialization method improves over random initialization.

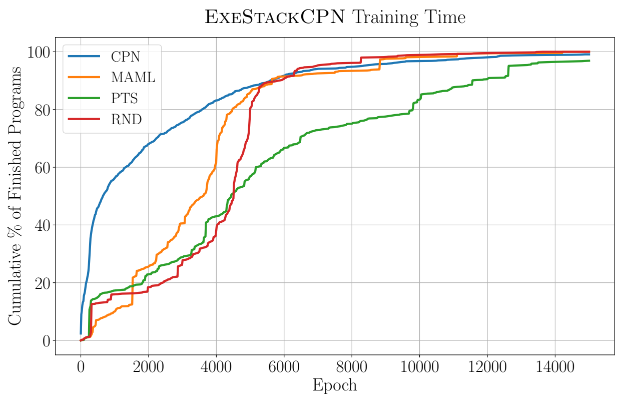


Figure 41: Epoch vs. percentage of EXESTACKCPN programs that each initialization method finished at that epoch.

Statistic	CPN	MAML	PTS
0th	0.39×	0.38×	**0.42**×
25th	0.54×	**0.63**×	0.57×
50th	**1.01**×	0.96×	0.83×
75th	**108.21**×	1.07×	7.91×
100th	**849.78**×	25.66×	278.11×
MPI	**50**th	54th	60th
GM	**4.31**×	1.07×	2.35×

Figure 42: Training time improvements at over random initialization on PARROTBENCHCPN. We include percentiles from 0th to 100th, the minimum percentile of improvement (MPI), and the overall geometric mean improvement (GM).

Program	CPN	MAML	PTS
`fft`	**1.43**×	0.83×	0.80×
`invk2j`	0.49×	**0.65**×	0.56×
`kmeans`	**674.47**×	2.15×	86.87×
`sobel`	0.74×	**1.14**×	0.79×

Figure 43: Geometric mean training time improvements over random initialization on PARROTBENCHCPN.

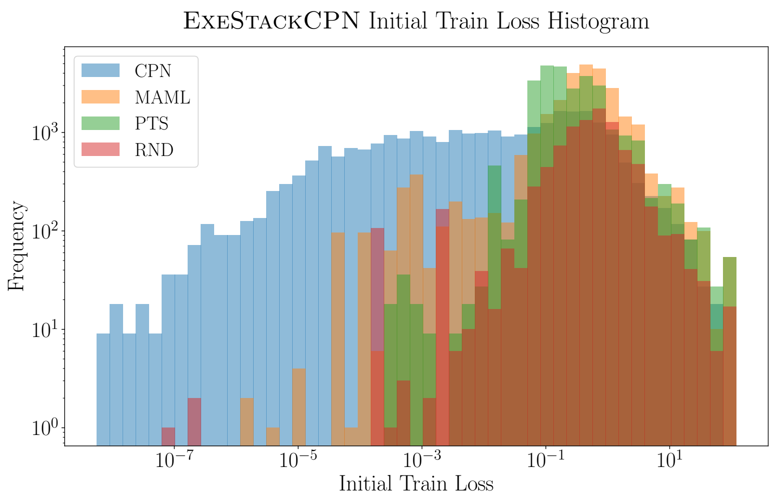
Methodology. To decide which strategy to use for each initialization method, we performed the PARROTBENCHCPN data efficiency evaluation of Section 5.2 with a set of padding strategies applied to each initialization method. For each initialization method, we chose the strategy that achieved the greatest overall test loss improvement over random initialization. Note that these experiments were performed prior to adding variable-output support, so we split the `fft` and `invk2j` benchmarks in PARROTBENCHCPN into multiple programs—one for each output.

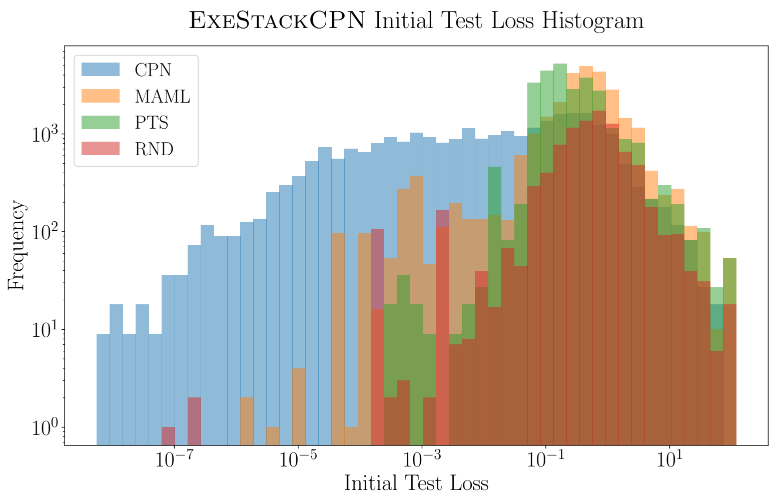
Results. We present the results in separate figures for random initialization (Figure 70), pretrained surrogates trained on random-padded and zero-padded inputs (Figures 71 and 72), MAML initializations trained on random-padded and zero-padded inputs (Figures 73 and 74), and COMPNETS trained on random-padded and zero-padded inputs (Figures 75 and 76).

Random initialization sees performance degradation with every padding strategy. One explanation for this degradation is that the baseline is random initialization with an architecture that has exactly as many inputs as needed, whereas each of the padding strategies operates on the covering architecture. Since the magnitude of weights in the He initialization is inversely proportional to the fan-in and fan-out of a neuron (He et al., 2015), the magnitude of weights in the first layer of the network will be smaller, potentially slowing convergence.

Surrogates that are pretrained on random-padded inputs perform approximately as well as surrogates pretrained on zero-padded inputs for all finetuning and evaluation variants. Among the finetuning and evaluation variants, finetuning and evaluating on zero-padded inputs performs the best. The best configuration by a small margin is pretraining on random-padded inputs and finetuning and evaluating on zero-padded inputs; this configuration achieves a geometric mean test loss improvement of 1.13×.

Across all finetuning and evaluation modes, MAML initializations trained on zero-padded inputs outperform MAML initializations trained on random-padded inputs. When MAML initializations are trained on zero-padded inputs, finetuning and evaluating on zero-padded inputs





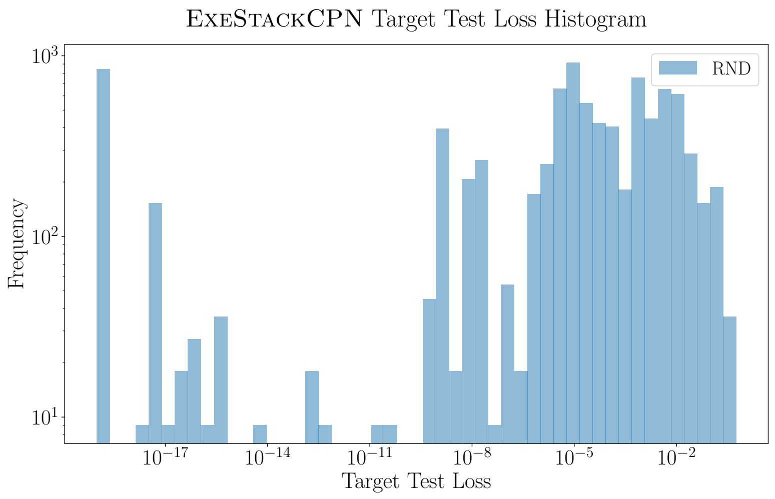


Figure 44: Histogram of initial training losses (top) and initial testing losses (bottom) for surrogates produced by each initialization method in the training time evaluation, as well as a histogram of the target testing losses set by random initialization after training for 5,000 epochs. Losses are not averaged across instances of initialization methods and trials. Note that both the x and y axes are log-scale.

Program	CPN (0)	CPN (1)	CPN (2)	CPN
`fft`	$0.48 \pm 1.81 \cdot 10^{-6}$	$0.43 \pm 8.51 \cdot 10^{-6}$	$2.94 \pm 6.80 \cdot 10^{-5}$	1.28 ± 1.20
`invk2j`	$1.14 \pm 5.67 \cdot 10^{-4}$	$1.90 \pm 8.89 \cdot 10^{-4}$	$2.54 \pm 1.52 \cdot 10^{-3}$	1.86 ± 0.58
`kmeans`	$0.12 \pm 1.44 \cdot 10^{-5}$	$0.09 \pm 1.33 \cdot 10^{-5}$	$0.08 \pm 1.20 \cdot 10^{-5}$	0.10 ± 0.02
`sobel`	$0.09 \pm 3.62 \cdot 10^{-4}$	$0.13 \pm 4.21 \cdot 10^{-4}$	$0.17 \pm 3.72 \cdot 10^{-4}$	0.13 ± 0.03

Program	MAML (0)	MAML (1)	MAML (2)	MAML
`fft`	0.57 ± 0.18	0.59 ± 0.19	0.53 ± 0.14	0.56 ± 0.16
`invk2j`	2.08 ± 0.64	2.07 ± 0.59	2.02 ± 0.56	2.06 ± 0.58
`kmeans`	0.76 ± 0.46	0.64 ± 0.38	0.79 ± 0.52	0.73 ± 0.44
`sobel`	0.24 ± 0.16	0.18 ± 0.12	0.25 ± 0.22	0.22 ± 0.17

Program	PTS (0)	PTS (1)	PTS (2)	PTS
`fft`	0.83 ± 0.07	0.84 ± 0.07	0.84 ± 0.07	0.84 ± 0.07
`invk2j`	2.11 ± 0.57	2.10 ± 0.56	2.12 ± 0.59	2.11 ± 0.55
`kmeans`	$0.10 \pm 1.92 \cdot 10^{-5}$	$0.10 \pm 1.88 \cdot 10^{-5}$	$0.10 \pm 1.87 \cdot 10^{-5}$	$0.10 \pm 1.34 \cdot 10^{-3}$
`sobel`	$0.18 \pm 4.37 \cdot 10^{-4}$	$0.19 \pm 4.40 \cdot 10^{-4}$	$0.19 \pm 4.43 \cdot 10^{-4}$	$0.19 \pm 3.55 \cdot 10^{-3}$

Program	RND
`fft`	0.64 ± 0.35
`invk2j`	2.37 ± 0.80
`kmeans`	0.24 ± 0.25
`sobel`	0.36 ± 0.34

Figure 45: Average initial train loss on PARROTBENCHCPN for surrogates produced by each initialization method. We include a column for each instance of an initialization method (e.g., "CPN (0)" is only one of the COMPNETs we trained) as well as a column that averages over each instance (e.g., "CPN" is an average over all COMPNETs we trained).

Program	CPN (0)	CPN (1)	CPN (2)	CPN
`fft`	0.48 ± 0.00	0.42 ± 0.00	2.92 ± 0.00	1.27 ± 1.18
`invk2j`	1.14 ± 0.00	1.89 ± 0.00	2.51 ± 0.00	1.84 ± 0.57
`kmeans`	0.06 ± 0.00	0.06 ± 0.00	0.05 ± 0.00	0.06 ± 0.01
`sobel`	0.09 ± 0.00	0.13 ± 0.00	0.17 ± 0.00	0.13 ± 0.03

Program	MAML (0)	MAML (1)	MAML (2)	MAML
`fft`	0.57 ± 0.18	0.59 ± 0.19	0.53 ± 0.14	0.56 ± 0.17
`invk2j`	2.07 ± 0.63	2.06 ± 0.59	2.01 ± 0.56	2.05 ± 0.57
`kmeans`	0.71 ± 0.43	0.60 ± 0.37	0.74 ± 0.50	0.68 ± 0.42
`sobel`	0.24 ± 0.16	0.18 ± 0.12	0.25 ± 0.22	0.22 ± 0.17

Program	PTS (0)	PTS (1)	PTS (2)	PTS
`fft`	0.83 ± 0.07	0.84 ± 0.07	0.85 ± 0.07	0.84 ± 0.07
`invk2j`	2.09 ± 0.57	2.09 ± 0.56	2.11 ± 0.58	2.10 ± 0.55
`kmeans`	0.06 ± 0.00	0.06 ± 0.00	0.06 ± 0.00	$0.06 \pm 7.58 \cdot 10^{-4}$
`sobel`	0.18 ± 0.00	0.19 ± 0.00	0.19 ± 0.00	$0.19 \pm 3.52 \cdot 10^{-3}$

Program	RND
`fft`	0.64 ± 0.36
`invk2j`	2.36 ± 0.80
`kmeans`	0.21 ± 0.23
`sobel`	0.36 ± 0.34

Figure 46: Average initial test loss on PARROTBENCHCPN for surrogates produced by each initialization method. We include a column for each instance of an initialization method (e.g., "CPN (0)" is only one of the COMPNETs we trained) as well as a column that averages over each instance (e.g., "CPN" is an average over all COMPNETs we trained).

Program	RND
`fft`	$3.96 \cdot 10^{-6}$
`invk2j`	$2.83 \cdot 10^{-3}$
`kmeans`	0.01
`sobel`	$4.16 \cdot 10^{-4}$

Figure 47: Target test loss for each PARROTBENCHCPN program, set by training randomly initialized surrogates for $5{,}000$ epochs over 9 trials and using the average final test loss.

Program	CPN-R Z/Z (Clone) (0)	CPN-R Z/Z (Clone) (1)	CPN-R Z/Z (Clone) (2)	CPN-R Z/Z (Clone)
`fft`	379.7 ± 31.5	262.0 ± 15.9	1140.7 ± 40.3	594.1 ± 398.0
`invk2j`	15000.0 ± 0.0	9200.7 ± 885.0	12581.3 ± 1956.4	12260.7 ± 2700.6
`kmeans`	12.0 ± 1.5	6.0 ± 0.0	6.0 ± 0.0	8.0 ± 3.0
`sobel`	11316.7 ± 2774.9	10637.0 ± 2332.6	4232.3 ± 2168.6	8728.7 ± 4008.5

Program	MAML-Z Z/Z (Reinit) (0)	MAML-Z Z/Z (Reinit) (1)	MAML-Z Z/Z (Reinit) (2)	MAML-Z Z/Z (Reinit)
`fft`	649.3 ± 469.7	824.0 ± 527.4	1069.0 ± 1047.5	847.4 ± 722.4
`invk2j`	9837.3 ± 5124.5	5327.7 ± 4358.9	13949.0 ± 3153.0	9704.7 ± 5464.3
`kmeans`	198.7 ± 192.1	13338.7 ± 4984.0	5018.0 ± 7486.5	6185.1 ± 7449.2
`sobel`	5695.7 ± 3764.1	6572.7 ± 5221.4	3668.3 ± 2193.8	5312.2 ± 3970.6

Program	PTS (0)	PTS (1)	PTS (2)	PTS
`fft`	459.3 ± 241.2	1384.7 ± 1098.1	1024.3 ± 1087.0	956.1 ± 950.3
`invk2j`	13794.3 ± 3345.0	12470.3 ± 5072.0	6632.7 ± 5282.7	10965.8 ± 5477.0
`kmeans`	188.0 ± 36.3	58.7 ± 42.3	18.3 ± 1.0	88.3 ± 80.0
`sobel`	9964.0 ± 4035.9	5436.7 ± 120.1	7555.7 ± 485.2	7652.1 ± 2939.6

Program	RND
`fft`	693.0 ± 689.0
`invk2j`	5835.7 ± 5133.0
`kmeans`	5098.7 ± 7429.2
`sobel`	5881.0 ± 3900.7

Figure 48: Average epoch at which each initialization method achieves the target testing loss for the training time evaluation on PARROTBENCHCPN. We include a column for each instance of an initialization method (e.g., "CPN (0)" is only one of the COMPNETs we trained), as well as a column that averages over all instances of an initialization method (e.g., "CPN" is an average over all COMPNETs we trained).

Program	CPN (0)	CPN (1)	CPN (2)	CPN
`fft`	0/9	0/9	0/9	0/27
`invk2j`	9/9	0/9	3/9	12/27
`kmeans`	0/9	0/9	0/9	0/27
`sobel`	0/9	0/9	0/9	0/27

Program	MAML (0)	MAML (1)	MAML (2)	MAML
`fft`	0/9	0/9	0/9	0/27
`invk2j`	4/9	1/9	8/9	13/27
`kmeans`	0/9	8/9	3/9	11/27
`sobel`	1/9	2/9	0/9	3/27

Program	PTS (0)	PTS (1)	PTS (2)	PTS
`fft`	0/9	0/9	0/9	0/27
`invk2j`	7/9	7/9	1/9	15/27
`kmeans`	0/9	0/9	0/9	0/27
`sobel`	3/9	0/9	0/9	3/27

Program	RND
`fft`	0/9
`invk2j`	0/9
`kmeans`	3/9
`sobel`	1/9

Figure 49: Number of trials where each initialization method does not achieve the target test loss after training for 15,000 epochs during the training time evaluation on ParrotBenchCPN. We include a column for each instance of an initialization method (e.g., "CPN (0)" is only one of the CompNets we trained) as well as a column that sums over each instance (e.g., "CPN" is a sum over all CompNets we trained).

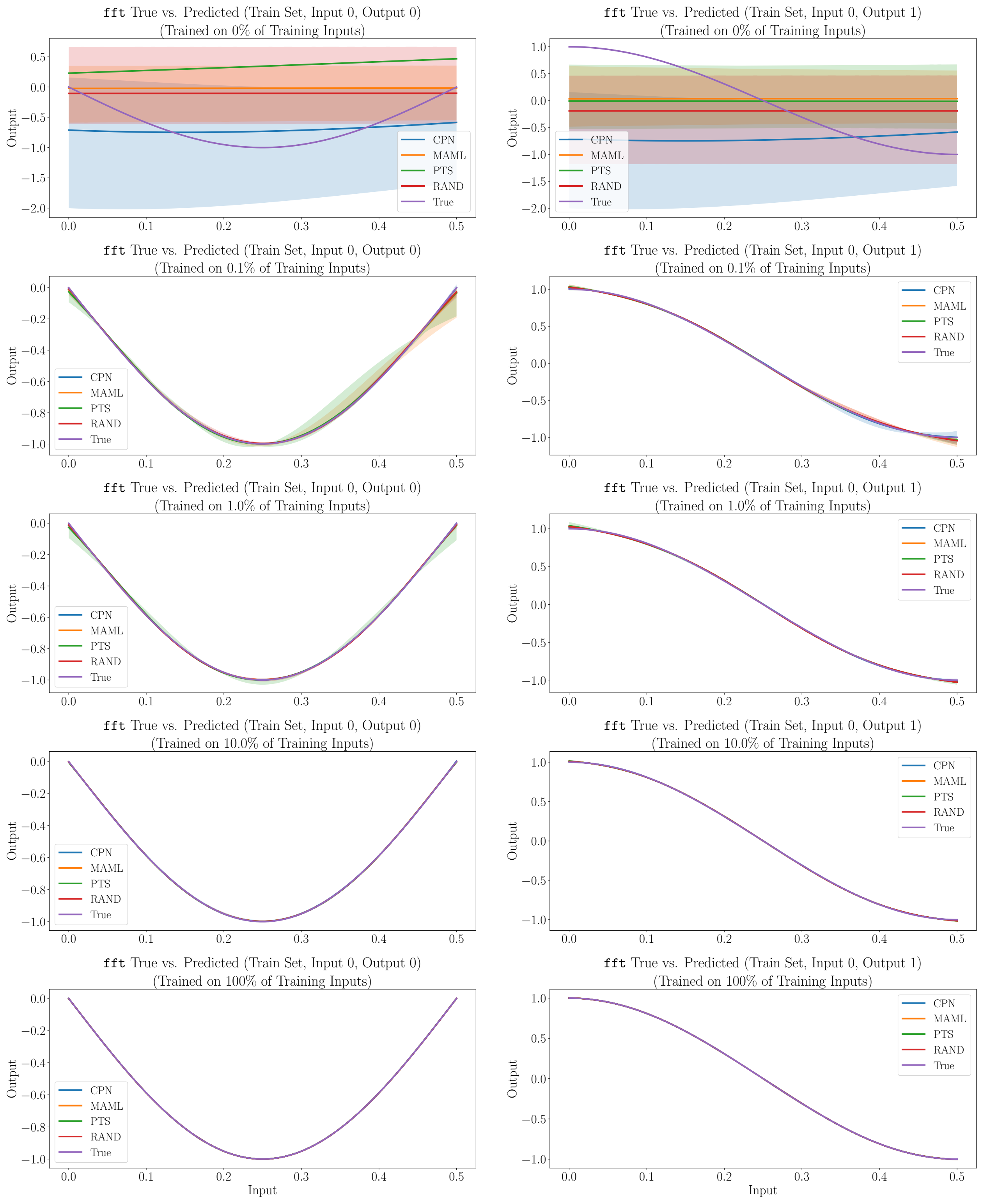


Figure 50: Visual comparisons of the ground-truth `fft` function from ParrotBenchCPN and neural surrogate approximations thereof. We include results for all dataset sizes evaluated in Section 5.2, and we include plots for each output of the kernel when the input is varied.

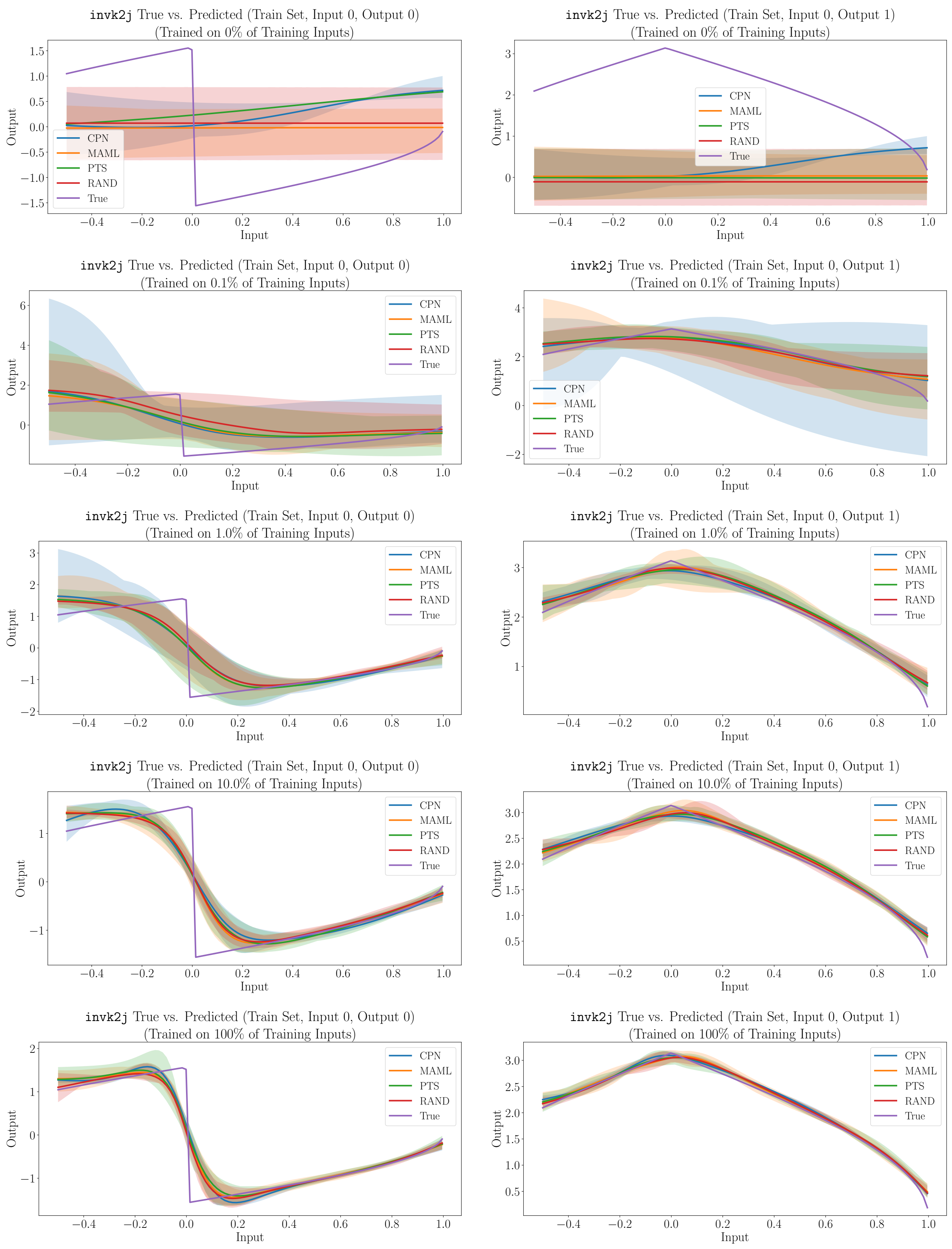


Figure 51: Visual comparisons of the ground-truth `invk2j` function from PARROTBENCHCPN and neural surrogate approximations thereof, when the first input is varied. We include results for all dataset sizes evaluated in Section 5.2, and we plot each output of the kernel when the input is varied.

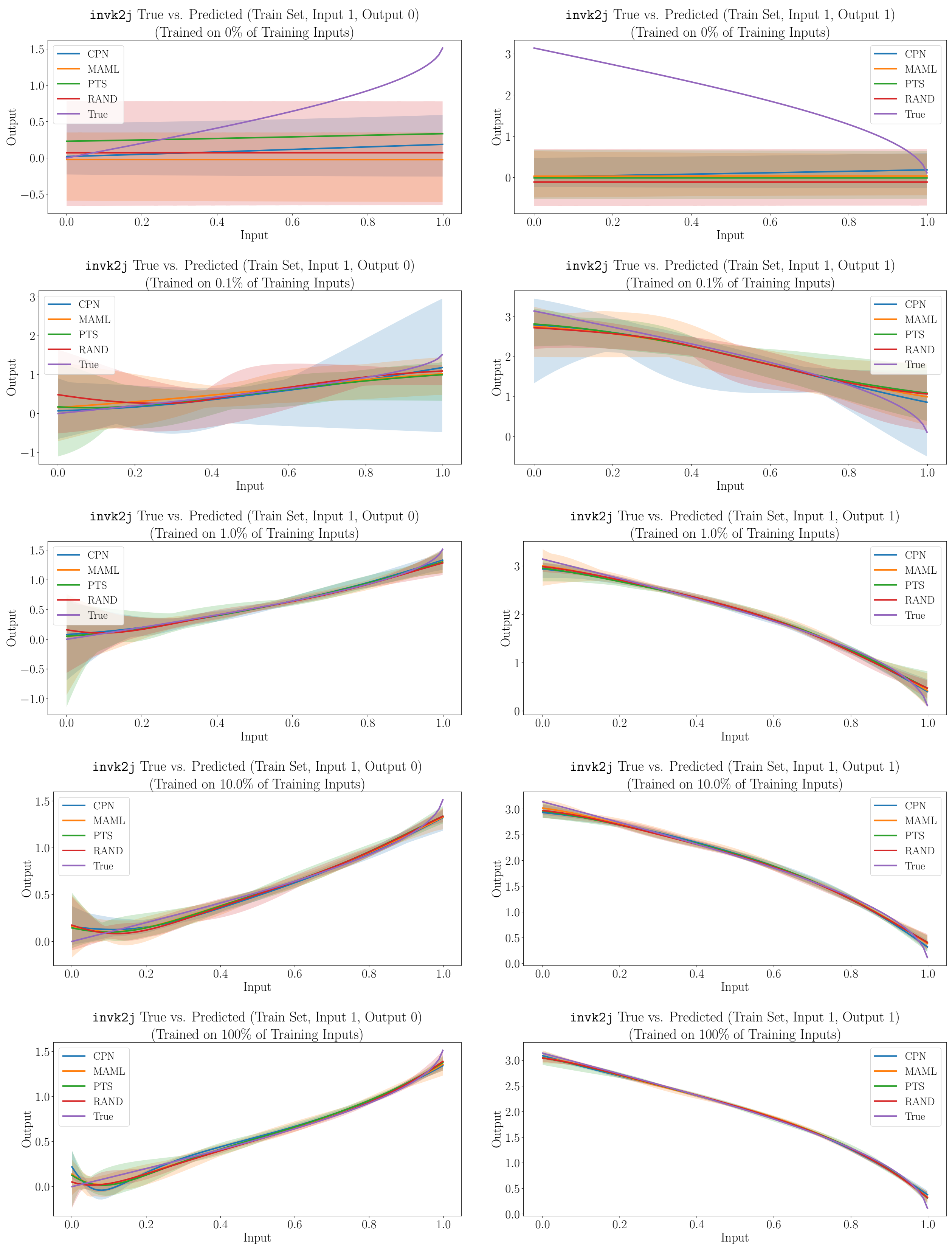


Figure 52: Visual comparisons of the ground-truth `invk2j` function from PARROTBENCHCPN and neural surrogate approximations thereof, when the second input is varied. We include results for all dataset sizes evaluated in Section 5.2, and we plot each output of the kernel when the input is varied.

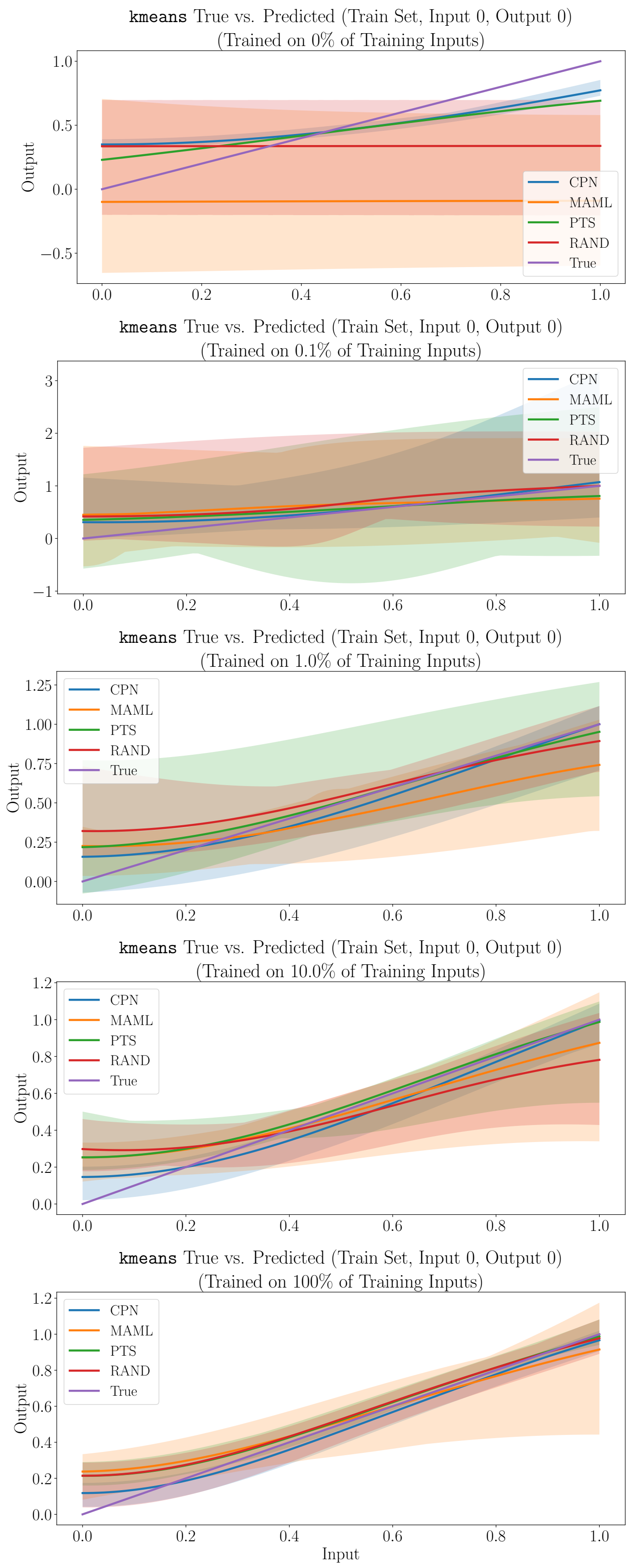


Figure 53: Visual comparisons of the ground-truth `kmeans` function from PARROTBENCHCPN and neural surrogate approximations thereof, when the first input is varied. We include results for all dataset sizes evaluated in Section 5.2.

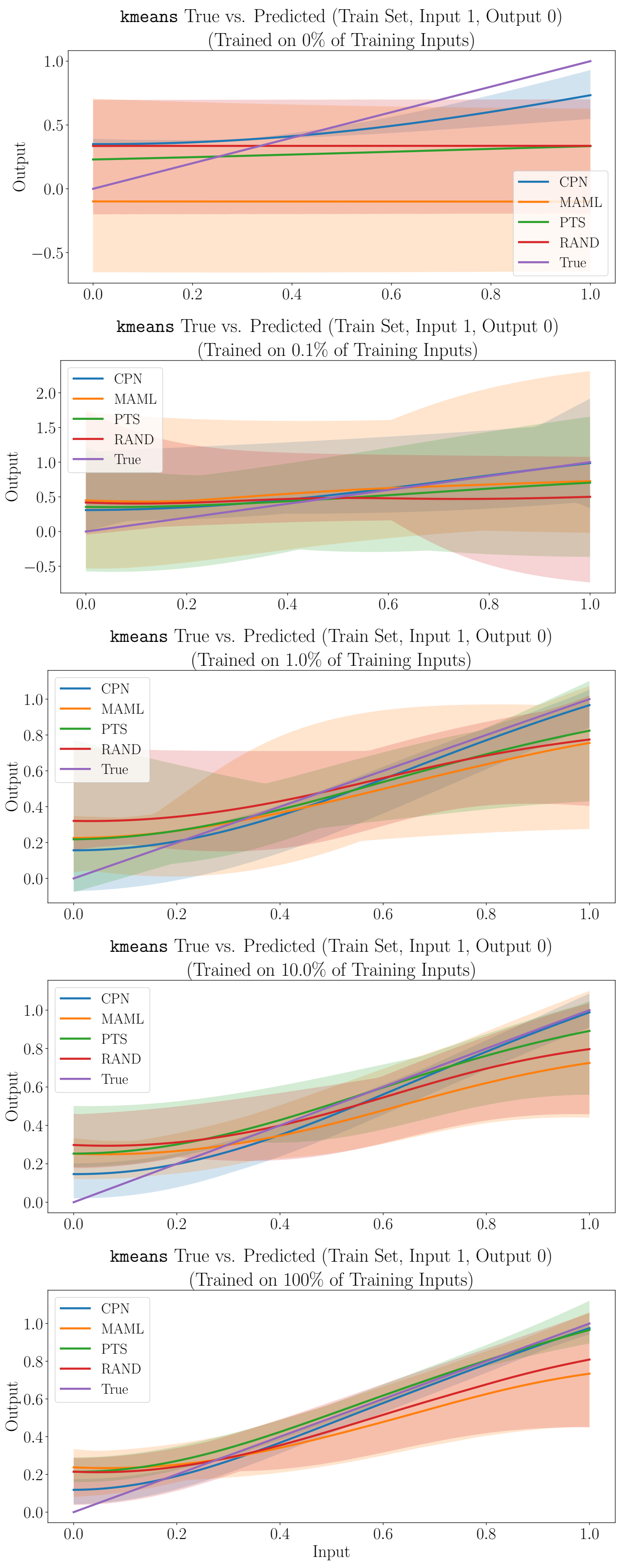


Figure 54: Visual comparisons of the ground-truth `kmeans` function from ParrotBenchCPN and neural surrogate approximations thereof, when the second input is varied. We include results for all dataset sizes evaluated in Section 5.2.

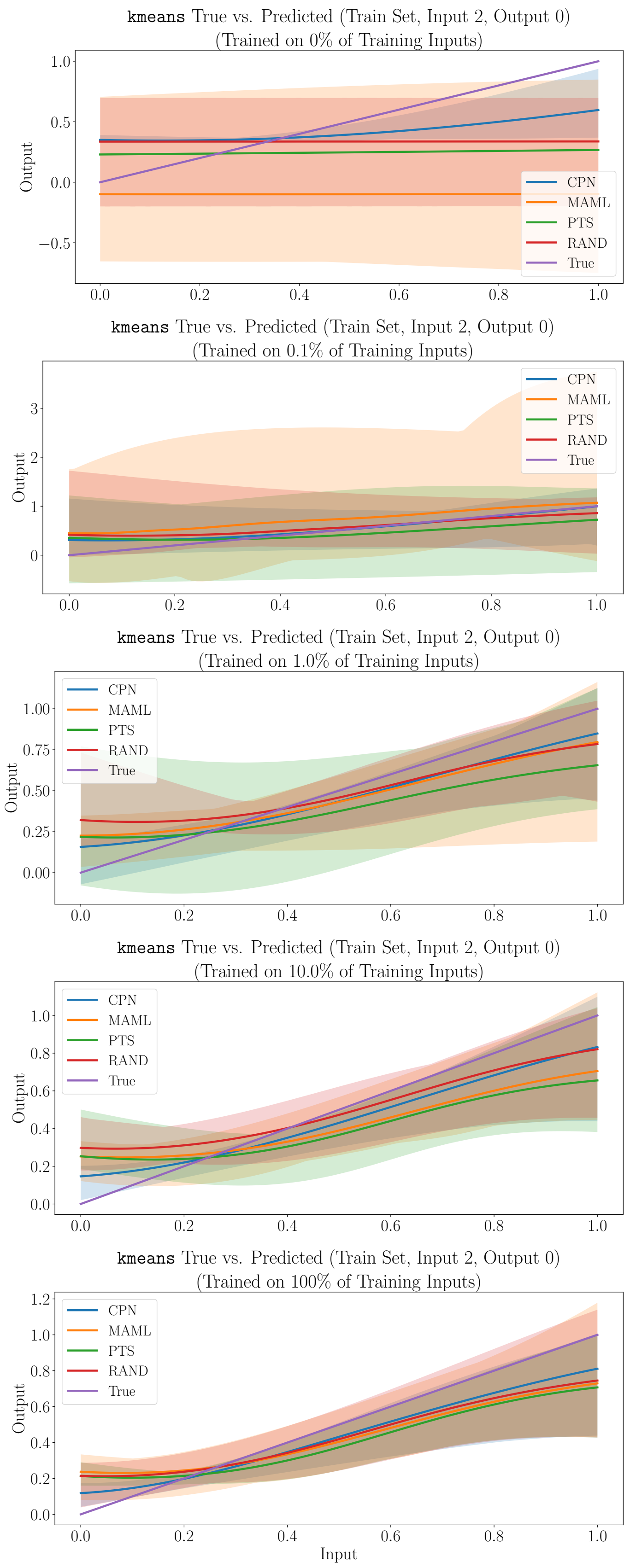


Figure 55: Visual comparisons of the ground-truth `kmeans` function from ParrotBenchCPN and neural surrogate approximations thereof, when the third input is varied. We include results for all dataset sizes evaluated in Section 5.2.

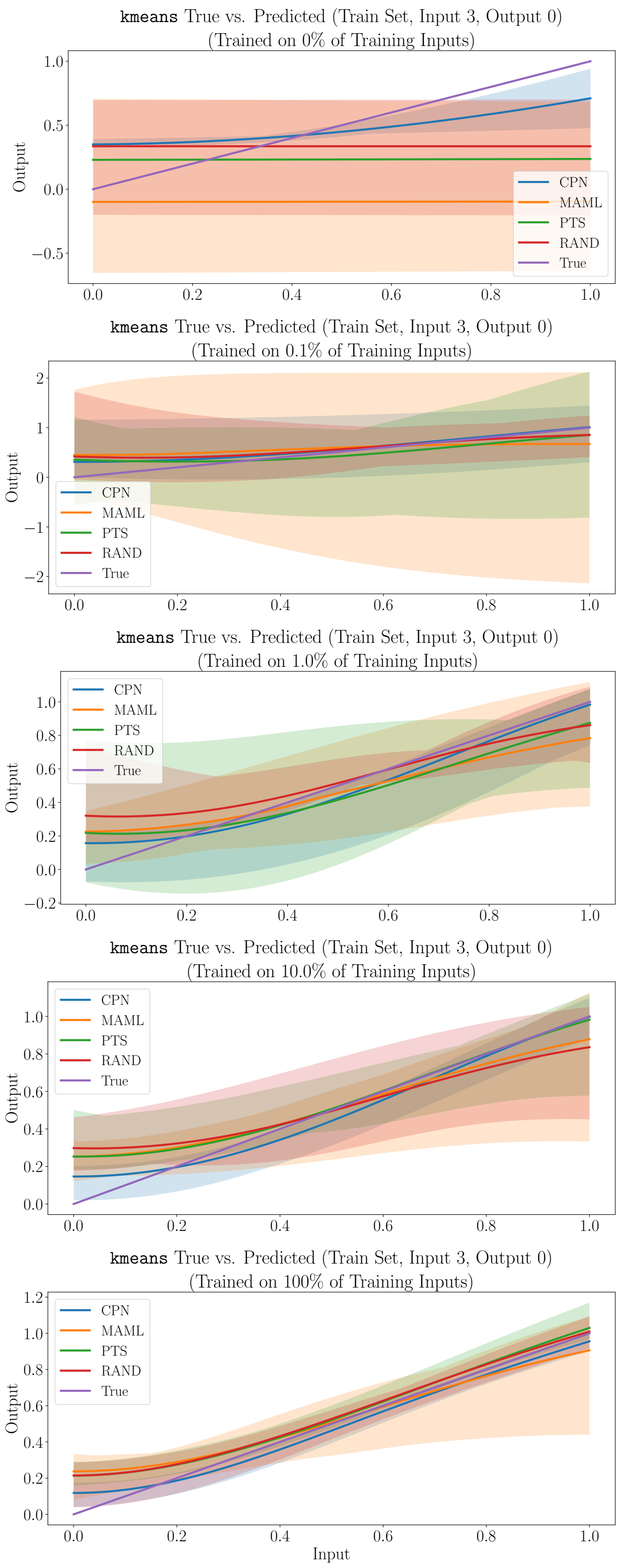


Figure 56: Visual comparisons of the ground-truth `kmeans` function from ParrotBenchCPN and neural surrogate approximations thereof, when the fourth input is varied. We include results for all dataset sizes evaluated in Section 5.2.

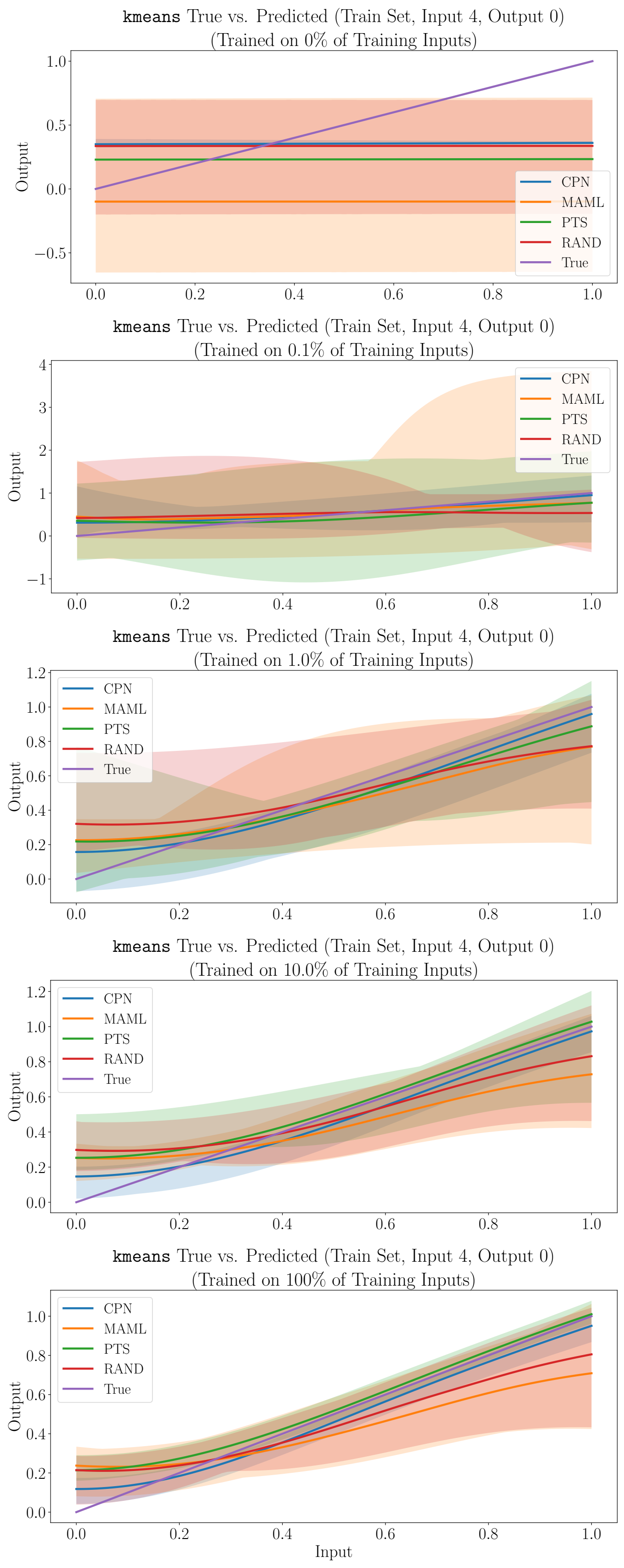


Figure 57: Visual comparisons of the ground-truth `kmeans` function from ParrotBenchCPN and neural surrogate approximations thereof, when the fifth input is varied. We include results for all dataset sizes evaluated in Section 5.2.

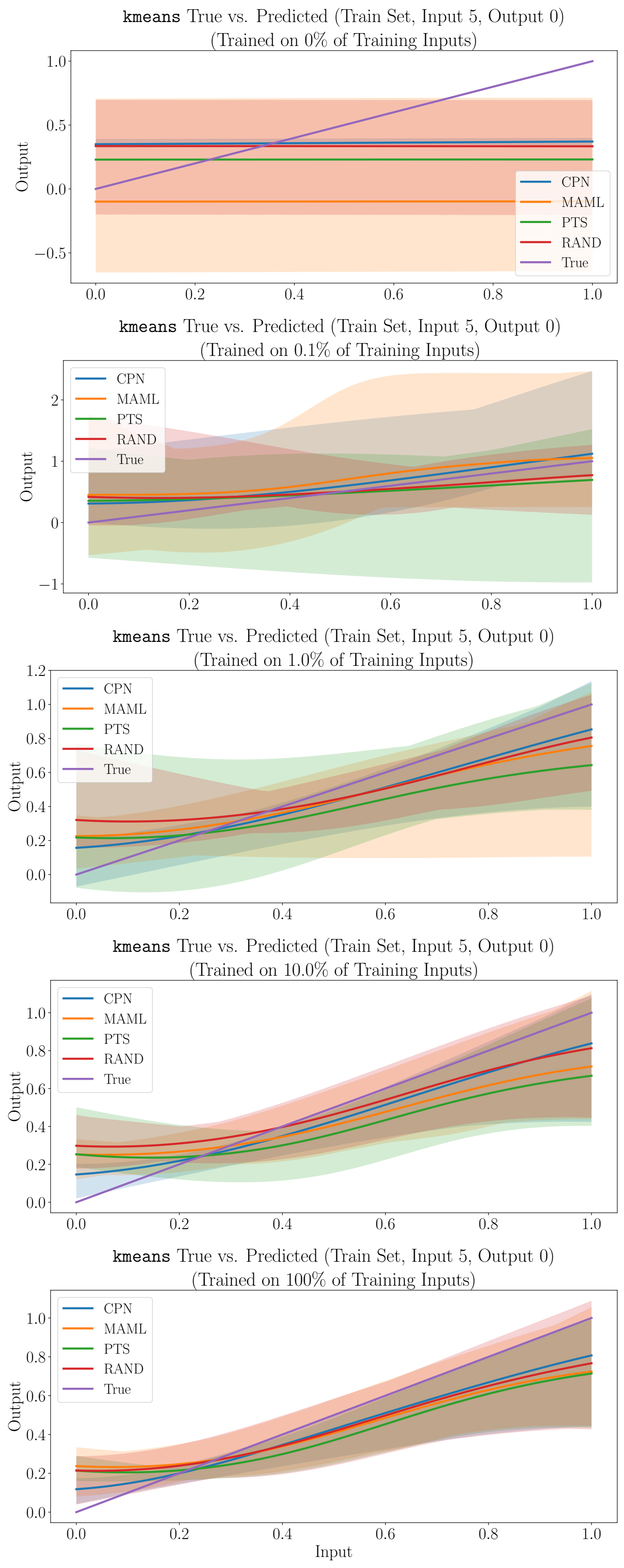


Figure 58: Visual comparisons of the ground-truth `kmeans` function from ParrotBenchCPN and neural surrogate approximations thereof, when the sixth input is varied. We include results for all dataset sizes evaluated in Section 5.2.

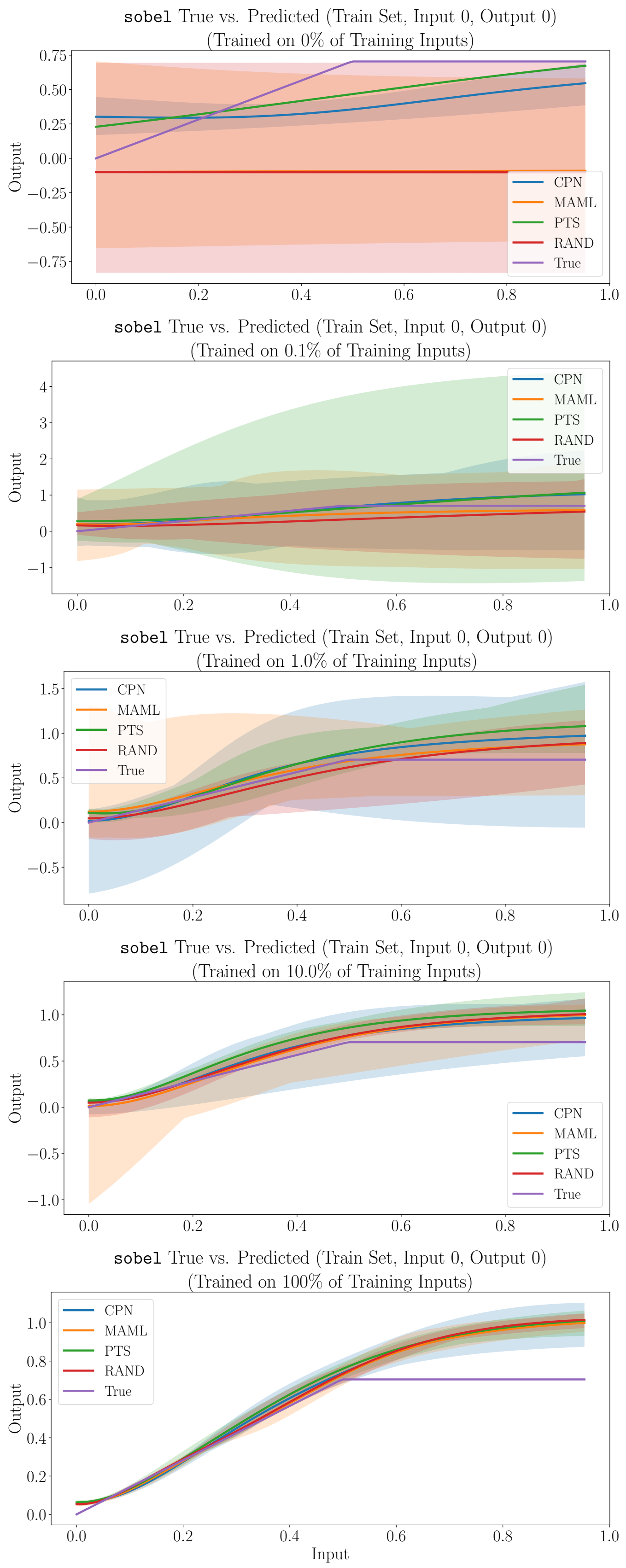


Figure 59: Visual comparisons of the ground-truth `sobel` function from ParrotBenchCPN and neural surrogate approximations thereof, when the first input is varied. We include results for all dataset sizes evaluated in Section 5.2.

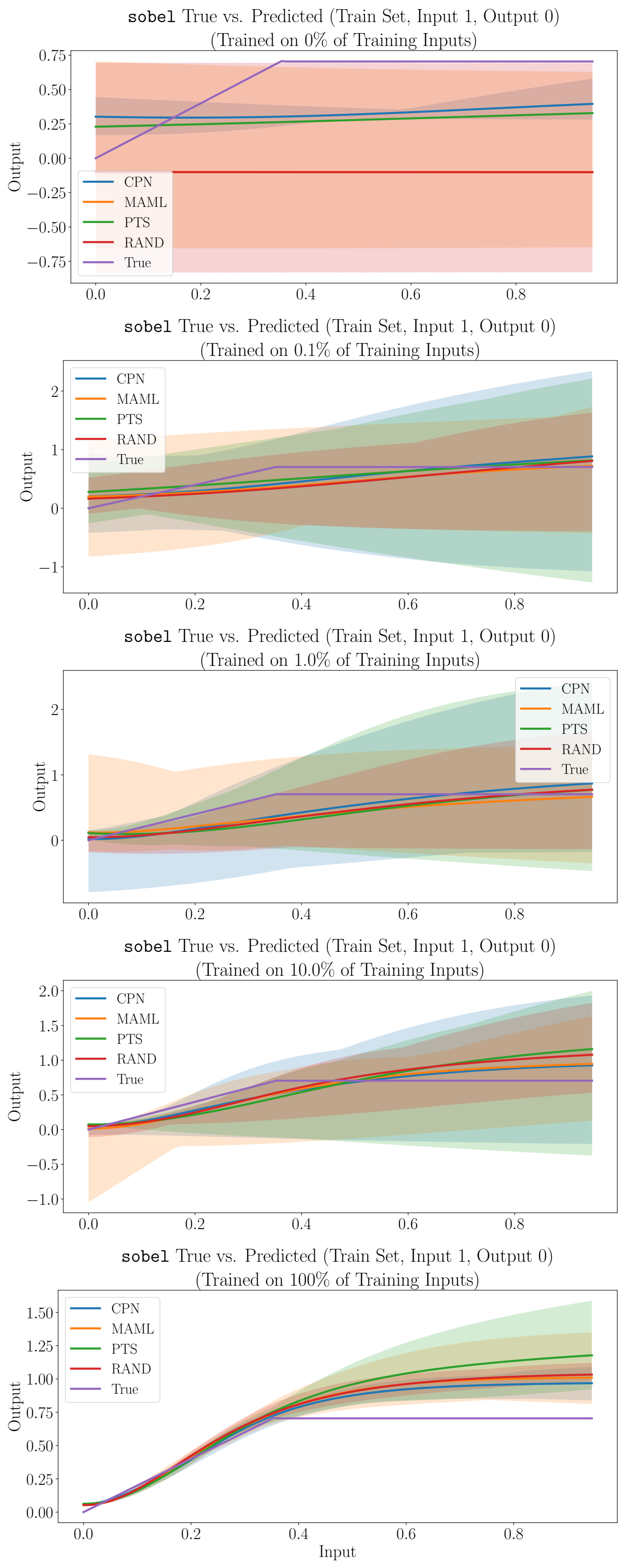


Figure 60: Visual comparisons of the ground-truth `sobel` function from ParrotBenchCPN and neural surrogate approximations thereof, when the second input is varied. We include results for all dataset sizes evaluated in Section 5.2.

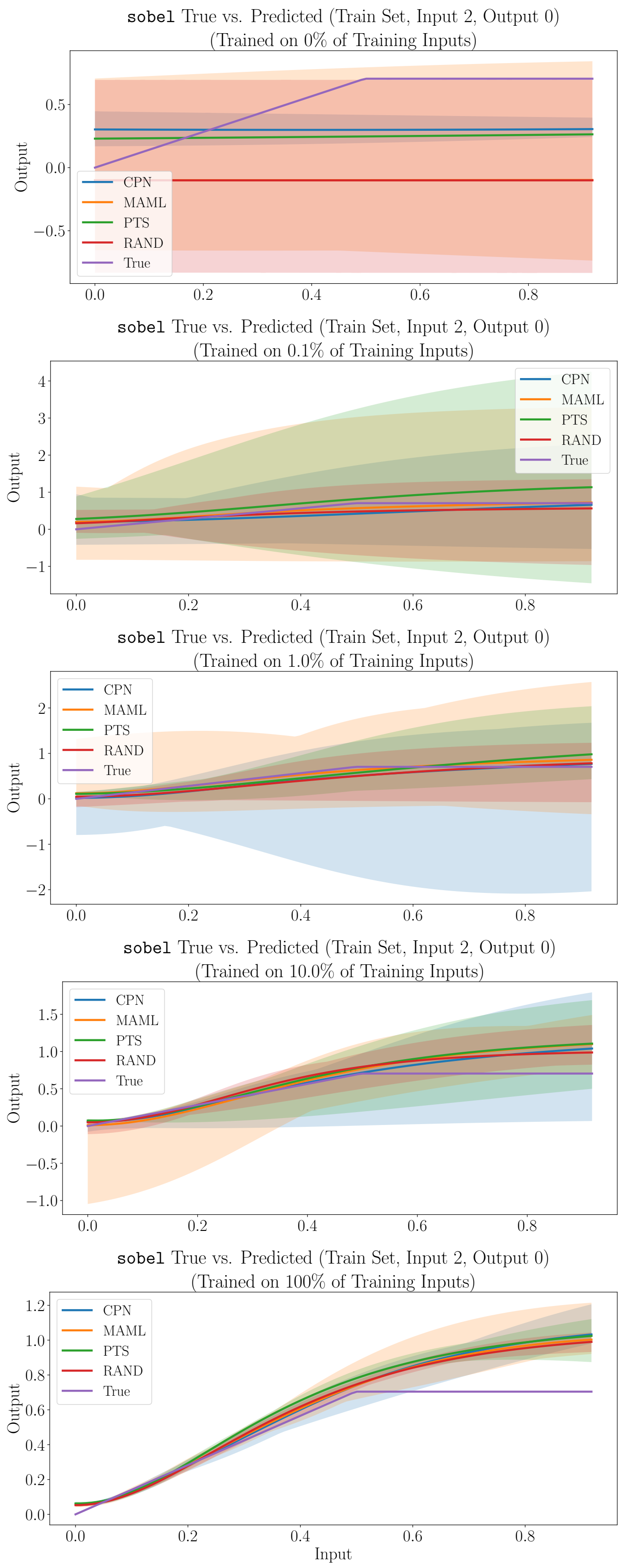


Figure 61: Visual comparisons of the ground-truth `sobel` function from ParrotBenchCPN and neural surrogate approximations thereof, when the third input is varied. We include results for all dataset sizes evaluated in Section 5.2.

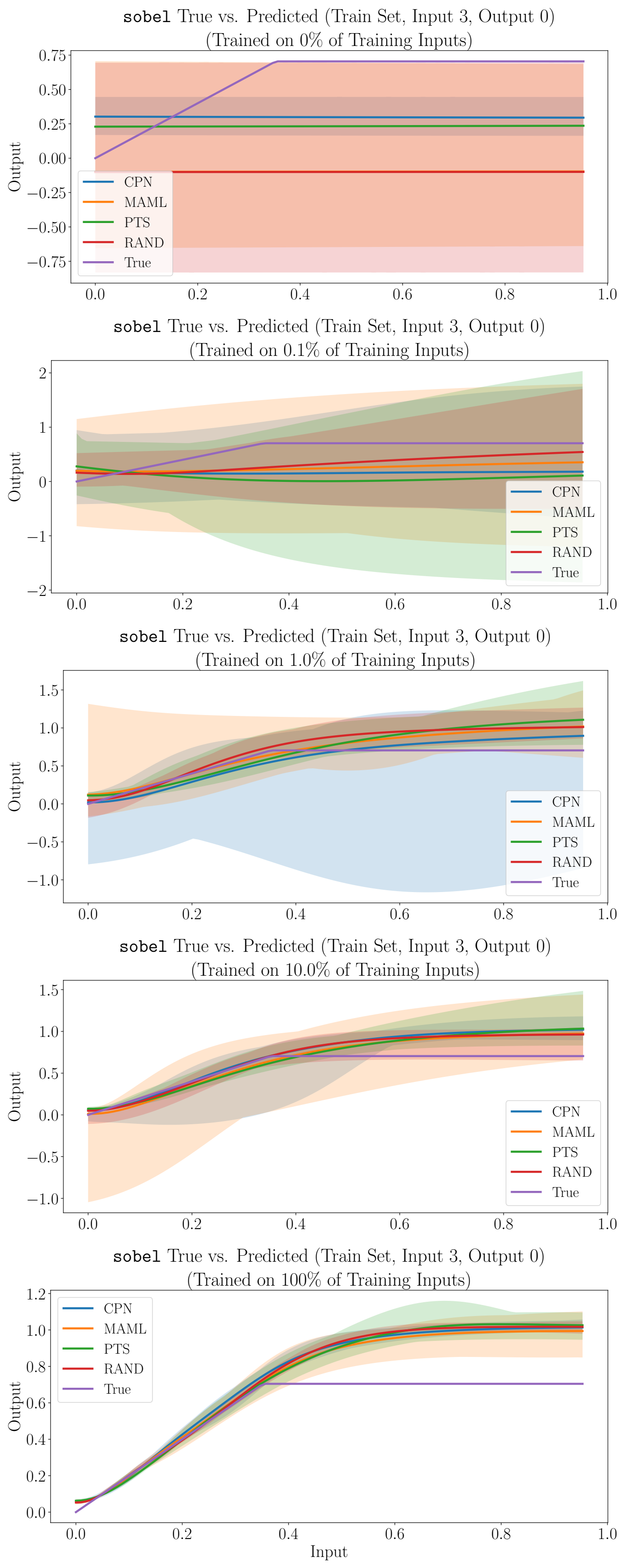


Figure 62: Visual comparisons of the ground-truth `sobel` function from ParrotBenchCPN and neural surrogate approximations thereof, when the fourth input is varied. We include results for all dataset sizes evaluated in Section 5.2.

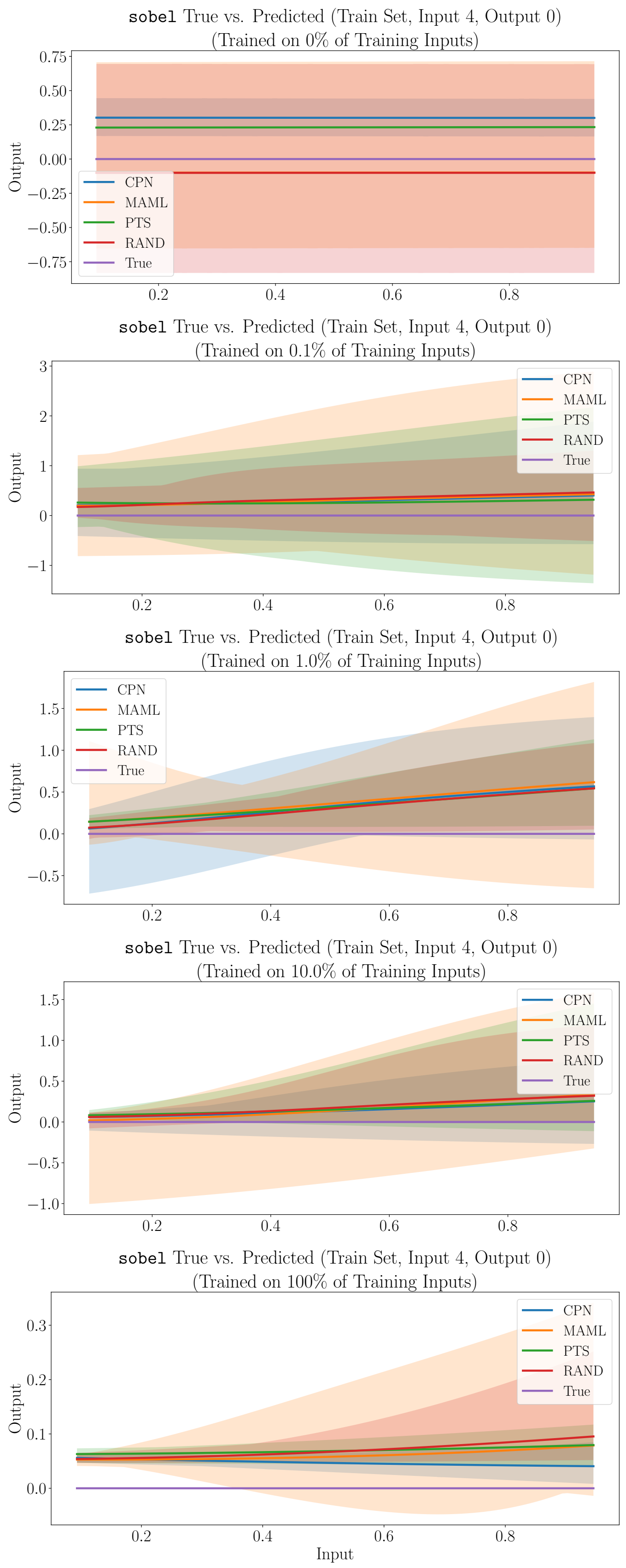


Figure 63: Visual comparisons of the ground-truth `sobel` function from ParrotBenchCPN and neural surrogate approximations thereof, when the fifth input is varied. We include results for all dataset sizes evaluated in Section 5.2.

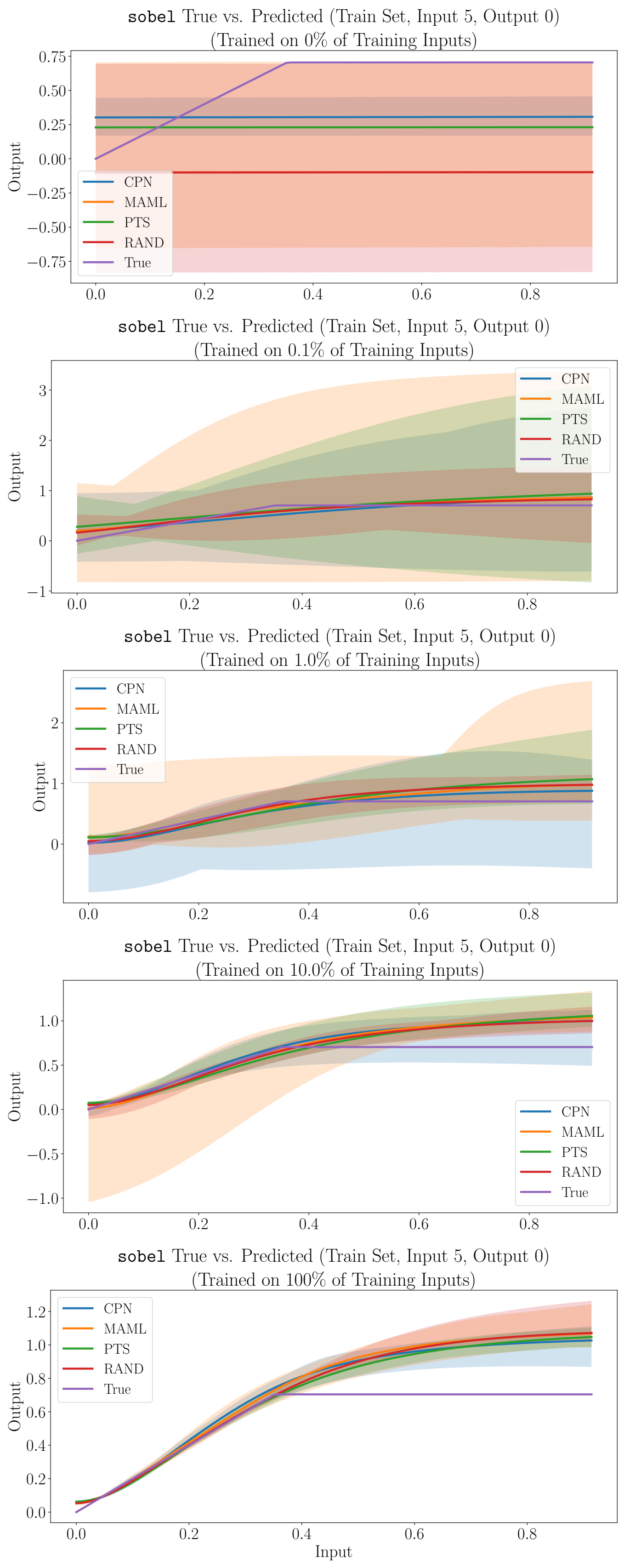


Figure 64: Visual comparisons of the ground-truth `sobel` function from ParrotBenchCPN and neural surrogate approximations thereof, when the sixth input is varied. We include results for all dataset sizes evaluated in Section 5.2.

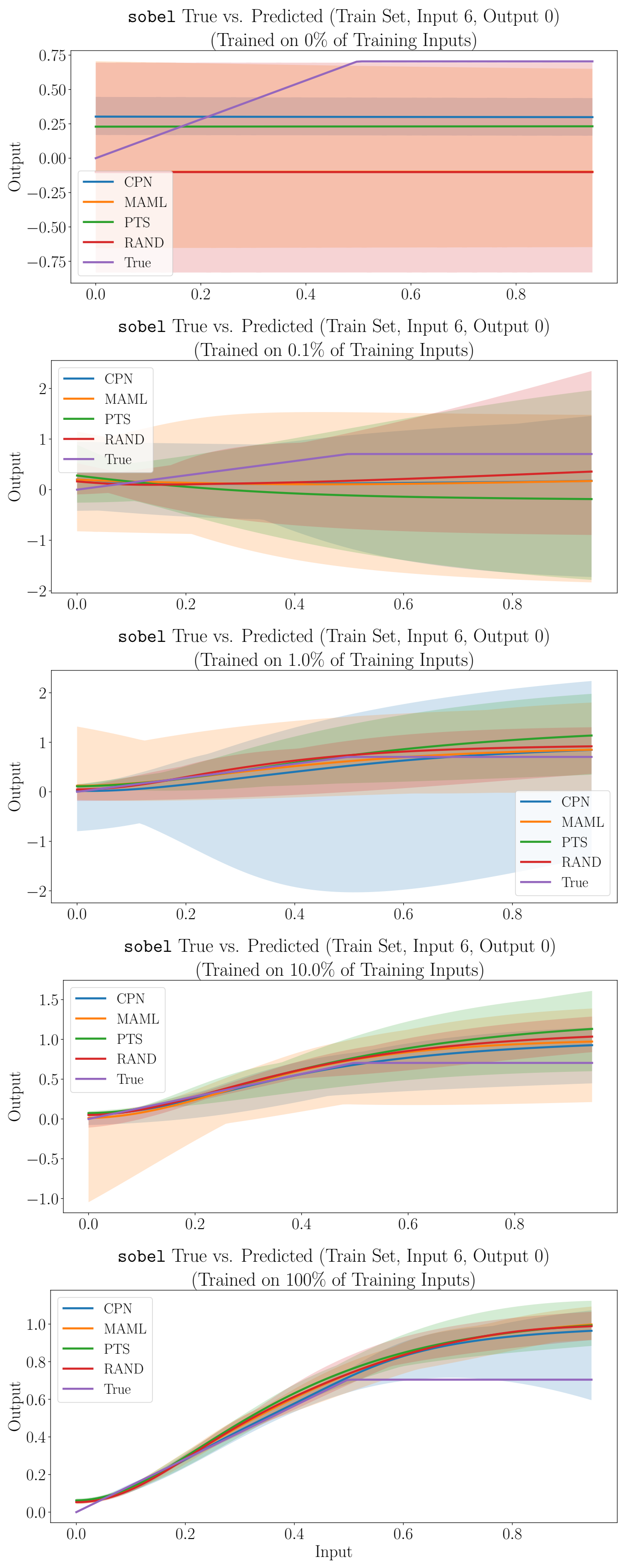


Figure 65: Visual comparisons of the ground-truth `sobel` function from ParrotBenchCPN and neural surrogate approximations thereof, when the seventh input is varied. We include results for all dataset sizes evaluated in Section 5.2.

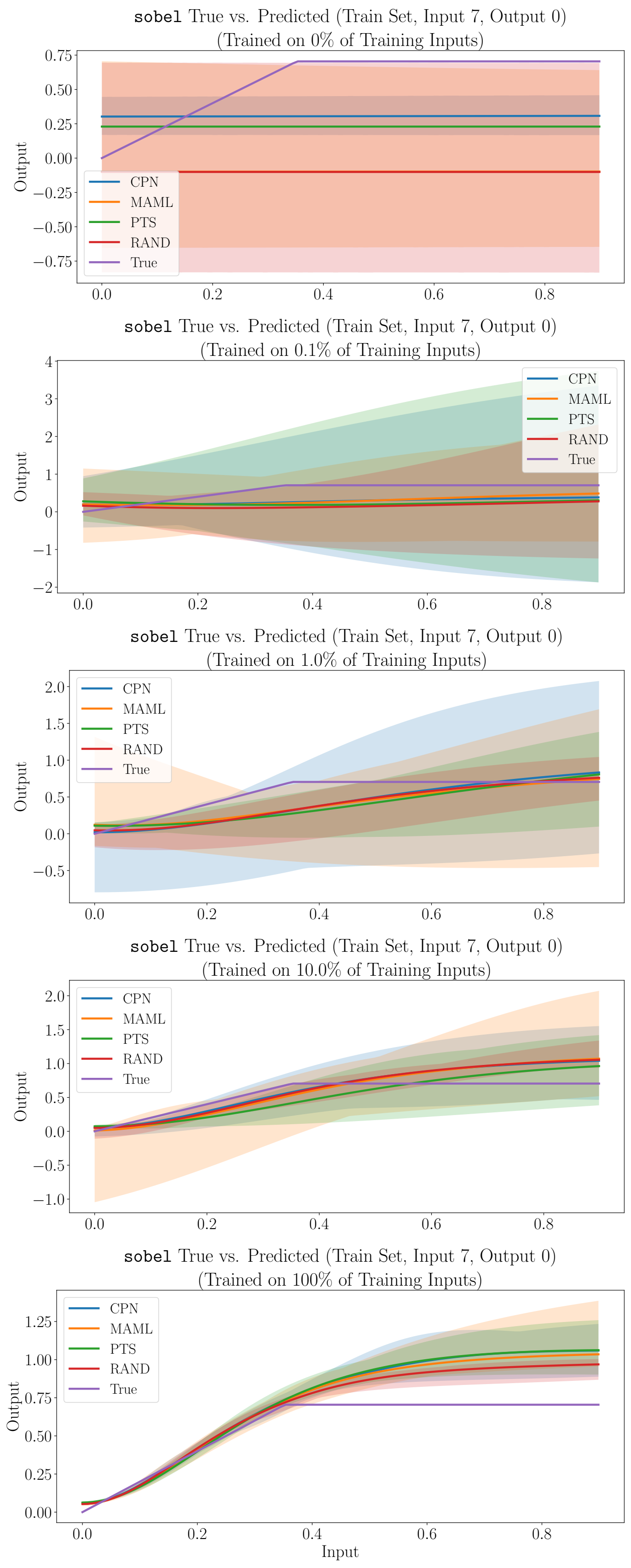


Figure 66: Visual comparisons of the ground-truth `sobel` function from PARROTBENCHCPN and neural surrogate approximations thereof, when the eighth input is varied. We include results for all dataset sizes evaluated in Section 5.2.

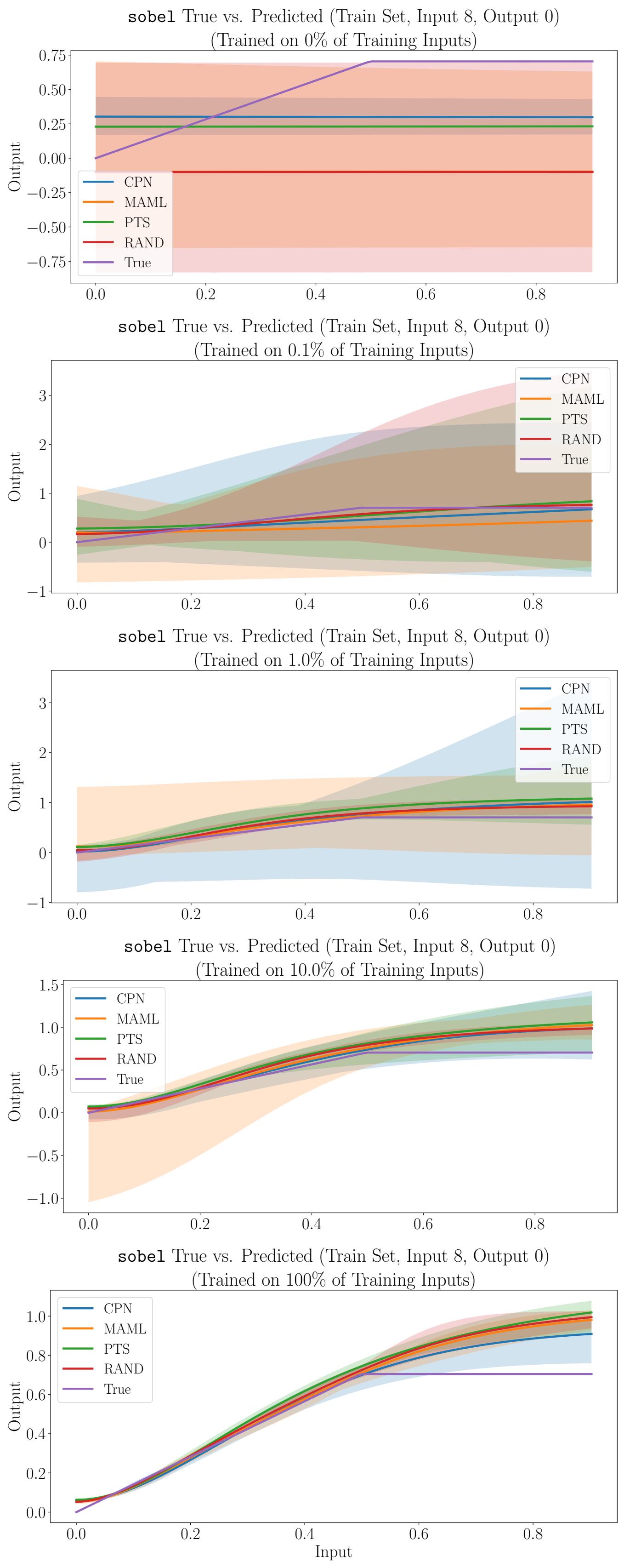


Figure 67: Visual comparisons of the ground-truth `sobel` function from ParrotBenchCPN and neural surrogate approximations thereof, when the ninth input is varied. We include results for all dataset sizes evaluated in Section 5.2.

Benchmark	CPN	MAML	PTS	RND	PRT	E2E Error
`fft`	$4.3 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$	$5.3 \cdot 10^{-6}$	$3.2 \cdot 10^{-6}$	$2.0 \cdot 10^{-5}$	7.22%
`invk2j`	$3.3 \cdot 10^{-3}$	$3.3 \cdot 10^{-3}$	$3.4 \cdot 10^{-3}$	$3.1 \cdot 10^{-3}$	$5.6 \cdot 10^{-3}$	7.50%
`kmeans`	$3.4 \cdot 10^{-3}$	$1.3 \cdot 10^{-2}$	$5.2 \cdot 10^{-3}$	$8.3 \cdot 10^{-3}$	$1.7 \cdot 10^{-3}$	6.18%
`sobel`	$5.4 \cdot 10^{-4}$	$4.5 \cdot 10^{-4}$	$6.3 \cdot 10^{-4}$	$4.2 \cdot 10^{-4}$	$2.3 \cdot 10^{-3}$	3.44%

Figure 68: MSE on PARROTBENCHCPN testing set for each initialization method, MSE of the neural surrogates Esmaeilzadeh et al. (2012a) train (PRT), and end-to-end error achieved by the surrogates of Esmaeilzadeh et al. (E2E Error).

Benchmark	`float` vs. `double` MSE
`fft`	$1.2 \cdot 10^{-14}$
`invk2j`	$1.6 \cdot 10^{-11}$
`kmeans`	0.0
`sobel`	$6.51 \cdot 10^{-8}$

Figure 69: MSE between PARROTBENCHCPN implementations that solely use the `float` datatype and implementations that solely use the `double` datatype. To calculate MSE, each program is evaluated on all inputs from double-precision versions of the training and testing set of PARROTBENCHCPN, and MSE is computed using the programs' outputs.

leads to the greatest geometric mean test loss improvement of $1.29\times$ over random initialization.

Across all finetuning and evaluation modes, COMPNET initializations trained on random-padded inputs outperform COMPNET initializations trained on zero-padded inputs. When COMPNET initializations are trained on random-padded inputs, finetuning and evaluating on zero-padded inputs leads to the greatest geometric mean test loss improvement of $1.96\times$ over random initialization.

Conclusion. In light of these results, we make the following decisions. We choose COMPNETs that are trained on random-padded inputs and the surrogates they produce are finetuned and evaluated on zero-padded inputs. We choose MAML initializations that are trained, finetuned, and evaluated on zero-padded inputs. We choose pretrained surrogates that are pretrained on random-padded inputs and finetuned and evaluated on zero-padded inputs. We choose standard random initialization over any of the padded variants (i.e., we make the topology match the program's input-output signature).

The reason why some initialization methods perform better when training on random-padded inputs and others perform better when training on zero-padded inputs is unclear and we believe deserves further study.

O. Variable-Output Support for Initialization Methods

Recall, all programs in EXESTACKCPN have a single output (Section 4). However, the `fft` and `invk2j` benchmarks in PARROTBENCHCPN have multiple outputs. In this appendix, we propose and evaluate a set of strategies to adapt initialization methods trained on EXESTACKCPN to support variable-output programs.

Methodology. For each initialization method, we produce a neural surrogate initialization, then we apply one of the following strategies:

- **Grow:** Use the initialization produced by the method and extend the final layer with randomly initialized weights to reach the target number of outputs.
- **Reinitialize:** Use the initialization produced by the initialization method but randomly initialize the final layer, sized to match the target number of outputs.
- **Clone:** Use the initialization produced by the initialization method but duplicate the weights for the one active output in the final layer of the initialization, to generate weights for the target number of outputs.

To decide which strategy to use for each initialization method, we performed the PARROTBENCHCPN data efficiency evaluation of Section 5.2, and we swept over a set of variable-output strategies applied to each initialization method. We used initialization methods that support variable-input programs, using the best strategies from Appendix N. For each initialization method, we choose the strategy that achieves the greatest overall test loss improvement over random initialization.

Results. We present the results for COMPNETs, MAML, and pretrained surrogates in Figures 77, 78, and 79.

Program	RND	RND FT-R EV-R	RND FT-R EV-Z	RND FT-Z EV-Z
`fft` (0)	**1.00**$\times$	0.02$\times$	0.02$\times$	0.19$\times$
`fft` (1)	1.00$\times$	0.18$\times$	0.23$\times$	**1.06**$\times$
`invk2j` (0)	**1.00**$\times$	0.45$\times$	0.53$\times$	**1.00**$\times$
`invk2j` (1)	**1.00**$\times$	0.31$\times$	0.32$\times$	0.82$\times$
`kmeans`	**1.00**$\times$	0.64$\times$	0.65$\times$	0.83$\times$
`sobel`	**1.00**$\times$	**1.00**$\times$	**1.00**$\times$	**1.00**$\times$

Dataset Size	RND	RND FT-R EV-R	RND FT-R EV-Z	RND FT-Z EV-Z
0%	**1.00**$\times$	0.80$\times$	0.80$\times$	0.80$\times$
0.1%	**1.00**$\times$	0.05$\times$	0.08$\times$	0.51$\times$
1%	**1.00**$\times$	0.10$\times$	0.10$\times$	0.39$\times$
10%	1.00$\times$	0.22$\times$	0.23$\times$	**1.01**$\times$
100%	1.00$\times$	1.22$\times$	**1.29**$\times$	1.18$\times$

Statistic	RND	RND FT-R EV-R	RND FT-R EV-Z	RND FT-Z EV-Z
0th	**1.00**$\times$	$4.46 \cdot 10^{-4}\times$	$6.03 \cdot 10^{-4}\times$	0.01$\times$
25th	**1.00**$\times$	0.27$\times$	0.30$\times$	0.92$\times$
50th	**1.00**$\times$	0.83$\times$	0.86$\times$	**1.00**$\times$
75th	1.00$\times$	1.00$\times$	1.00$\times$	**1.09**$\times$
100th	1.00$\times$	5.64$\times$	**6.48**$\times$	1.66$\times$
MPI	**0**th	68th	63rd	50th
GM	**1.00**$\times$	0.26$\times$	0.28$\times$	0.72$\times$

Figure 70: Data efficiency results for PARROTBENCHCPN programs using variants of random initialization. FT-R and FT-Z mean the surrogate initialization was finetuned using random-padded and zero-padded inputs, respectively. EV-R and EV-Z mean the surrogate initialization was evaluated using random-padded and zero-padded inputs, respectively.

Program	PTS-R FT-R EV-R	PTS-R FT-R EV-Z	PTS-R FT-Z EV-Z
`fft` (0)	$0.06\times$	$0.06\times$	$\mathbf{1.53}\times$
`fft` (1)	$0.17\times$	$0.19\times$	$\mathbf{0.76}\times$
`invk2j` (0)	$0.50\times$	$0.59\times$	$\mathbf{1.18}\times$
`invk2j` (1)	$0.28\times$	$0.29\times$	$\mathbf{0.77}\times$
`kmeans`	$1.77\times$	$1.80\times$	$\mathbf{2.28}\times$
`sobel`	$\mathbf{0.85}\times$	$\mathbf{0.85}\times$	$\mathbf{0.85}\times$

Dataset Size	PTS-R FT-R EV-R	PTS-R FT-R EV-Z	PTS-R FT-Z EV-Z
0%	$\mathbf{1.38}\times$	$\mathbf{1.38}\times$	$\mathbf{1.38}\times$
0.1%	$0.05\times$	$0.06\times$	$\mathbf{1.26}\times$
1%	$0.11\times$	$0.12\times$	$\mathbf{1.03}\times$
10%	$0.60\times$	$0.62\times$	$\mathbf{0.94}\times$
100%	$1.33\times$	$\mathbf{1.45}\times$	$1.07\times$

Statistic	PTS-R FT-R EV-R	PTS-R FT-R EV-Z	PTS-R FT-Z EV-Z
0th	$3.50 \cdot 10^{-4}\times$	$4.76 \cdot 10^{-4}\times$	$\mathbf{0.15}\times$
25th	$0.26\times$	$0.35\times$	$\mathbf{0.75}\times$
50th	$0.73\times$	$0.77\times$	$\mathbf{1.07}\times$
75th	$1.38\times$	$1.39\times$	$\mathbf{1.65}\times$
100th	$28.05\times$	$\mathbf{28.24}\times$	$28.03\times$
MPI	66th	65th	**47**th
GM	$0.36\times$	$0.39\times$	$\mathbf{1.13}\times$

Figure 71: Data efficiency results for ParrotBenchCPN using pretrained surrogates trained with random-padded inputs. FT-R and FT-Z mean the surrogate initialization was finetuned using random-padded and zero-padded inputs, respectively. EV-R and EV-Z mean the surrogate initialization was evaluated using random-padded and zero-padded inputs, respectively.

Program	PTS-Z FT-R EV-R	PTS-Z FT-R EV-Z	PTS-Z FT-Z EV-Z
`fft` (0)	$0.06\times$	$0.06\times$	$\mathbf{0.95}\times$
`fft` (1)	$0.24\times$	$0.33\times$	$\mathbf{1.04}\times$
`invk2j` (0)	$0.53\times$	$0.62\times$	$\mathbf{1.25}\times$
`invk2j` (1)	$0.35\times$	$0.37\times$	$\mathbf{1.08}\times$
`kmeans`	$0.95\times$	$0.93\times$	$\mathbf{1.31}\times$
`sobel`	$\mathbf{1.07}\times$	$\mathbf{1.07}\times$	$\mathbf{1.07}\times$

Dataset Size	PTS-Z FT-R EV-R	PTS-Z FT-R EV-Z	PTS-Z FT-Z EV-Z
0%	$\mathbf{1.58}\times$	$\mathbf{1.58}\times$	$\mathbf{1.58}\times$
0.1%	$0.06\times$	$0.09\times$	$\mathbf{1.17}\times$
1%	$0.10\times$	$0.10\times$	$\mathbf{1.09}\times$
10%	$0.80\times$	$0.84\times$	$\mathbf{1.12}\times$
100%	$0.91\times$	$\mathbf{0.96}\times$	$0.74\times$

Statistic	PTS-Z FT-R EV-R	PTS-Z FT-R EV-Z	PTS-Z FT-Z EV-Z
0th	$5.52 \cdot 10^{-4}\times$	$6.16 \cdot 10^{-4}\times$	$\mathbf{0.07}\times$
25th	$0.37\times$	$0.47\times$	$\mathbf{0.87}\times$
50th	$0.84\times$	$0.84\times$	$\mathbf{1.10}\times$
75th	$1.13\times$	$1.19\times$	$\mathbf{1.41}\times$
100th	$10.94\times$	$\mathbf{12.86}\times$	$7.41\times$
MPI	66th	65th	**41**st
GM	$0.37\times$	$0.41\times$	$\mathbf{1.11}\times$

Figure 72: Data efficiency results for ParrotBenchCPN using pretrained surrogates trained with zero-padded inputs. FT-R and FT-Z mean the surrogate initialization was finetuned using random-padded and zero-padded inputs, respectively. EV-R and EV-Z mean the surrogate initialization was evaluated using random-padded and zero-padded inputs, respectively.

Program	MAML-R FT-R EV-R	MAML-R FT-R EV-Z	MAML-R FT-Z EV-Z
`fft` (0)	$0.06\times$	$0.07\times$	$\mathbf{1.30}\times$
`fft` (1)	$0.32\times$	$0.45\times$	$\mathbf{1.11}\times$
`invk2j` (0)	$0.39\times$	$0.48\times$	$\mathbf{0.82}\times$
`invk2j` (1)	$0.64\times$	$0.75\times$	$\mathbf{1.13}\times$
`kmeans`	$0.63\times$	$\mathbf{0.65}\times$	$\mathbf{0.65}\times$
`sobel`	$\mathbf{0.44}\times$	$\mathbf{0.44}\times$	$\mathbf{0.44}\times$

Dataset Size	MAML-R FT-R EV-R	MAML-R FT-R EV-Z	MAML-R FT-Z EV-Z
0%	$\mathbf{0.97}\times$	$\mathbf{0.97}\times$	$\mathbf{0.97}\times$
0.1%	$0.06\times$	$0.08\times$	$\mathbf{1.12}\times$
1%	$0.12\times$	$0.15\times$	$\mathbf{0.82}\times$
10%	$0.51\times$	$0.54\times$	$\mathbf{0.56}\times$
100%	$1.11\times$	$\mathbf{1.33}\times$	$0.90\times$

Statistic	MAML-R FT-R EV-R	MAML-R FT-R EV-Z	MAML-R FT-Z EV-Z
0th	$7.16 \cdot 10^{-4}\times$	$7.54 \cdot 10^{-4}\times$	$\mathbf{0.07}\times$
25th	$0.27\times$	$0.32\times$	$\mathbf{0.52}\times$
50th	$0.54\times$	$0.58\times$	$\mathbf{0.87}\times$
75th	$0.95\times$	$0.99\times$	$\mathbf{1.39}\times$
100th	$12.03\times$	$\mathbf{16.94}\times$	$15.18\times$
MPI	79th	76th	**65**th
GM	$0.33\times$	$0.39\times$	$\mathbf{0.85}\times$

Figure 73: Data efficiency results for PARROTBENCHCPN using MAML initializations trained with random-padded inputs. FT-R and FT-Z mean the surrogate initialization was finetuned using random-padded and zero-padded inputs, respectively. EV-R and EV-Z mean the surrogate initialization was evaluated using random-padded and zero-padded inputs, respectively.

Program	MAML-Z FT-R EV-R	MAML-Z FT-R EV-Z	MAML-Z FT-Z EV-Z
`fft` (0)	$0.18\times$	$0.21\times$	$\mathbf{2.85}\times$
`fft` (1)	$0.54\times$	$0.69\times$	$\mathbf{1.16}\times$
`invk2j` (0)	$0.62\times$	$0.77\times$	$\mathbf{1.35}\times$
`invk2j` (1)	$0.50\times$	$0.56\times$	$\mathbf{1.11}\times$
`kmeans`	$0.88\times$	$0.91\times$	$\mathbf{1.04}\times$
`sobel`	$\mathbf{0.89}\times$	$\mathbf{0.89}\times$	$\mathbf{0.89}\times$

Dataset Size	MAML-Z FT-R EV-R	MAML-Z FT-R EV-Z	MAML-Z FT-Z EV-Z
0%	$\mathbf{1.35}\times$	$1.34\times$	$1.34\times$
0.1%	$0.06\times$	$0.08\times$	$\mathbf{1.40}\times$
1%	$0.14\times$	$0.17\times$	$\mathbf{1.38}\times$
10%	$1.34\times$	$\mathbf{1.44}\times$	$1.16\times$
100%	$2.60\times$	$\mathbf{2.89}\times$	$1.19\times$

Statistic	MAML-Z FT-R EV-R	MAML-Z FT-R EV-Z	MAML-Z FT-Z EV-Z
0th	$7.14 \cdot 10^{-4}\times$	$1.08 \cdot 10^{-3}\times$	$\mathbf{0.29}\times$
25th	$0.41\times$	$0.46\times$	$\mathbf{0.86}\times$
50th	$0.88\times$	$0.91\times$	$\mathbf{1.10}\times$
75th	$1.42\times$	$1.43\times$	$\mathbf{1.50}\times$
100th	$57.67\times$	$\mathbf{76.44}\times$	$13.85\times$
MPI	56th	55th	**39**th
GM	$0.53\times$	$0.61\times$	$\mathbf{1.29}\times$

Figure 74: Data efficiency results for PARROTBENCHCPN using MAML initializations trained with zero-padded inputs. FT-R and FT-Z mean the surrogate initialization was finetuned using random-padded and zero-padded inputs, respectively. EV-R and EV-Z mean the surrogate initialization was evaluated using random-padded and zero-padded inputs, respectively.

Program	CPN-R FT-R EV-R	CPN-R FT-R EV-Z	CPN-R FT-Z EV-Z
`fft` (0)	$0.88\times$	$0.99\times$	$\mathbf{7.18}\times$
`fft` (1)	$0.48\times$	$0.73\times$	$\mathbf{1.17}\times$
`invk2j` (0)	$0.56\times$	$0.73\times$	$\mathbf{1.09}\times$
`invk2j` (1)	$0.55\times$	$0.62\times$	$\mathbf{1.04}\times$
`kmeans`	$4.12\times$	$4.26\times$	$\mathbf{5.22}\times$
`sobel`	$\mathbf{1.14}\times$	$\mathbf{1.14}\times$	$\mathbf{1.14}\times$

Dataset Size	CPN-R FT-R EV-R	CPN-R FT-R EV-Z	CPN-R FT-Z EV-Z
0%	$\mathbf{1.42}\times$	$\mathbf{1.42}\times$	$\mathbf{1.42}\times$
0.1%	$0.09\times$	$0.14\times$	$\mathbf{2.33}\times$
1%	$1.11\times$	$1.44\times$	$\mathbf{2.56}\times$
10%	$1.89\times$	$1.99\times$	$\mathbf{2.18}\times$
100%	$2.42\times$	$\mathbf{2.57}\times$	$1.57\times$

Statistic	CPN-R FT-R EV-R	CPN-R FT-R EV-Z	CPN-R FT-Z EV-Z
0th	$1.18 \cdot 10^{-3}\times$	$1.47 \cdot 10^{-3}\times$	$\mathbf{0.19}\times$
25th	$0.49\times$	$0.60\times$	$\mathbf{0.86}\times$
50th	$0.99\times$	$1.01\times$	$\mathbf{1.22}\times$
75th	$2.17\times$	$2.20\times$	$\mathbf{2.31}\times$
100th	$171.49\times$	$191.02\times$	$\mathbf{1478.96}\times$
MPI	51st	48th	**35**th
GM	$0.92\times$	$1.08\times$	$\mathbf{1.96}\times$

Figure 75: Data efficiency results for PARROTBENCHCPN programs using COMPNETS trained on random-padded inputs. FT-R and FT-Z mean the surrogate initialization was finetuned using random-padded and zero-padded inputs, respectively. EV-R and EV-Z mean the surrogate initialization was evaluated using random-padded and zero-padded inputs, respectively.

Program	CPN-Z FT-R EV-R	CPN-Z FT-R EV-Z	CPN-Z FT-Z EV-Z
`fft` (0)	$0.02\times$	$0.02\times$	$\mathbf{0.58}\times$
`fft` (1)	$0.24\times$	$0.28\times$	$\mathbf{1.06}\times$
`invk2j` (0)	$0.47\times$	$0.61\times$	$\mathbf{1.07}\times$
`invk2j` (1)	$0.60\times$	$0.67\times$	$\mathbf{1.62}\times$
`kmeans`	$1.45\times$	$1.51\times$	$\mathbf{1.78}\times$
`sobel`	$\mathbf{0.91}\times$	$\mathbf{0.91}\times$	$\mathbf{0.91}\times$

Dataset Size	CPN-Z FT-R EV-R	CPN-Z FT-R EV-Z	CPN-Z FT-Z EV-Z
0%	$1.50\times$	$\mathbf{1.51}\times$	$\mathbf{1.51}\times$
0.1%	$0.05\times$	$0.07\times$	$\mathbf{0.88}\times$
1%	$0.10\times$	$0.12\times$	$\mathbf{1.33}\times$
10%	$0.40\times$	$0.45\times$	$\mathbf{0.66}\times$
100%	$1.65\times$	$\mathbf{1.75}\times$	$1.36\times$

Statistic	CPN-Z FT-R EV-R	CPN-Z FT-R EV-Z	CPN-Z FT-Z EV-Z
0th	$3.05 \cdot 10^{-4}\times$	$\mathbf{5.88 \cdot 10^{-4}}\times$	$1.52 \cdot 10^{-4}\times$
25th	$0.39\times$	$0.42\times$	$\mathbf{0.77}\times$
50th	$0.79\times$	$0.82\times$	$\mathbf{1.14}\times$
75th	$1.31\times$	$1.37\times$	$\mathbf{1.60}\times$
100th	$70.29\times$	$\mathbf{70.97}\times$	$69.18\times$
MPI	63rd	59th	**38**th
GM	$0.35\times$	$0.39\times$	$\mathbf{1.10}\times$

Figure 76: Data efficiency results for PARROTBENCHCPN programs using COMPNETS trained on zero-padded inputs. FT-R and FT-Z mean the surrogate initialization was finetuned using random-padded and zero-padded inputs, respectively. EV-R and EV-Z mean the surrogate initialization was evaluated using random-padded and zero-padded inputs, respectively.

Program	CPN-R Z/Z (Grow)	CPN-R Z/Z (Reinit)	CPN-R Z/Z (Clone)
`fft`	$0.95\times$	$\mathbf{1.49}\times$	$1.47\times$
`invk2j`	$0.86\times$	$\mathbf{1.01}\times$	$\mathbf{1.01}\times$
`kmeans`	$\mathbf{7.85}\times$	$1.77\times$	$\mathbf{7.85}\times$
`sobel`	$\mathbf{1.14}\times$	$1.12\times$	$\mathbf{1.14}\times$

Dataset Size	CPN-R Z/Z (Grow)	CPN-R Z/Z (Reinit)	CPN-R Z/Z (Clone)
0%	$\mathbf{1.86}\times$	$0.95\times$	$1.81\times$
0.1%	$1.61\times$	$1.46\times$	$\mathbf{1.98}\times$
1%	$1.49\times$	$1.40\times$	$\mathbf{1.77}\times$
10%	$2.13\times$	$1.93\times$	$\mathbf{2.38}\times$
100%	$1.26\times$	$1.05\times$	$\mathbf{1.68}\times$

Statistic	CPN-R Z/Z (Grow)	CPN-R Z/Z (Reinit)	CPN-R Z/Z (Clone)
0th	$0.17\times$	$\mathbf{0.33}\times$	$0.22\times$
25th	$0.79\times$	$0.85\times$	$\mathbf{0.88}\times$
50th	$1.05\times$	$1.13\times$	$\mathbf{1.23}\times$
75th	$1.96\times$	$1.76\times$	$\mathbf{2.96}\times$
100th	$\mathbf{106.91}\times$	$31.55\times$	$\mathbf{106.91}\times$
MPI	42nd	**33**rd	36th
GM	$1.64\times$	$1.31\times$	$\mathbf{1.91}\times$

Figure 77: Data efficiency results for PARROTBENCHCPN programs using COMPNETs trained on various variable-output strategies. CPN-R means we train the COMPNETs on random-padded inputs. Z/Z means we finetune and evaluate COMPNET-initialized surrogates on zero-padded inputs (see Appendix N).

Program	MAML-Z Z/Z (Grow)	MAML-Z Z/Z (Reinit)	MAML-Z Z/Z (Clone)
`fft`	$0.65\times$	$\mathbf{0.98}\times$	$0.63\times$
`invk2j`	$1.06\times$	$\mathbf{1.07}\times$	$0.88\times$
`kmeans`	$\mathbf{0.94}\times$	$0.68\times$	$\mathbf{0.94}\times$
`sobel`	$0.89\times$	$\mathbf{1.06}\times$	$0.89\times$

Dataset Size	MAML-Z Z/Z (Grow)	MAML-Z Z/Z (Reinit)	MAML-Z Z/Z (Clone)
0%	$\mathbf{1.42}\times$	$0.90\times$	$\mathbf{1.42}\times$
0.1%	$0.92\times$	$\mathbf{0.94}\times$	$0.73\times$
1%	$0.73\times$	$\mathbf{0.93}\times$	$0.51\times$
10%	$0.88\times$	$\mathbf{1.11}\times$	$0.94\times$
100%	$0.60\times$	$\mathbf{0.81}\times$	$0.75\times$

Statistic	MAML-Z Z/Z (Grow)	MAML-Z Z/Z (Reinit)	MAML-Z Z/Z (Clone)
0th	$0.15\times$	$\mathbf{0.28}\times$	$0.05\times$
25th	$0.64\times$	$\mathbf{0.82}\times$	$0.64\times$
50th	$0.92\times$	$\mathbf{0.97}\times$	$0.85\times$
75th	$1.14\times$	$1.14\times$	$\mathbf{1.16}\times$
100th	$4.01\times$	$1.99\times$	$\mathbf{8.00}\times$
MPI	58th	**54**th	66th
GM	$0.87\times$	$\mathbf{0.93}\times$	$0.82\times$

Figure 78: Data efficiency results for PARROTBENCHCPN programs using MAML initializations trained on various variable-output strategies. MAML-Z means we train the MAML initializations on zero-padded inputs. Z/Z means we finetune and evaluate MAML-initialized surrogates on zero-padded inputs (see Appendix N).

Program	PTS-R Z/Z (Grow)	PTS-R Z/Z (Reinit)	PTS-R Z/Z (Clone)
`fft`	$0.61\times$	$\mathbf{0.88}\times$	$0.46\times$
`invk2j`	$1.05\times$	$0.95\times$	$\mathbf{1.12}\times$
`kmeans`	$\mathbf{2.24}\times$	$0.65\times$	$\mathbf{2.24}\times$
`sobel`	$0.85\times$	$\mathbf{0.92}\times$	$0.85\times$

Dataset Size	PTS-R Z/Z (Grow)	PTS-R Z/Z (Reinit)	PTS-R Z/Z (Clone)
0%	$1.56\times$	$0.79\times$	$\mathbf{1.65}\times$
0.1%	$\mathbf{0.98}\times$	$0.93\times$	$0.81\times$
1%	$0.79\times$	$\mathbf{0.87}\times$	$0.75\times$
10%	$\mathbf{1.23}\times$	$1.00\times$	$1.00\times$
100%	$0.86\times$	$0.67\times$	$\mathbf{0.98}\times$

Statistic	PTS-R Z/Z (Grow)	PTS-R Z/Z (Reinit)	PTS-R Z/Z (Clone)
0th	$\mathbf{0.23}\times$	$0.22\times$	$0.21\times$
25th	$0.75\times$	$\mathbf{0.77}\times$	$0.65\times$
50th	$\mathbf{0.97}\times$	$0.93\times$	$0.85\times$
75th	$1.26\times$	$1.04\times$	$\mathbf{1.40}\times$
100th	$\mathbf{38.18}\times$	$1.86\times$	$\mathbf{38.18}\times$
MPI	**54**th	69th	63rd
GM	$\mathbf{1.05}\times$	$0.84\times$	$1.00\times$

Figure 79: Data efficiency results for PARROTBENCHCPN programs using pretrained initializations trained on various variable-output strategies. PTS-R means we train the pretrained initializations on random-padded inputs. Z/Z means we finetune and evaluate pretrain-initialized surrogates on zero-padded inputs (see Appendix N).

The best-performing strategy for CompNets is cloning, with a geometric mean test loss improvement of $1.91\times$, the best-performing strategy for MAML is reinitialization, with a geometric mean test loss improvement of $0.93\times$, and the best-performing strategy for pretrained surrogates is growing, with a geometric mean test loss improvement of $1.05\times$. Note that the `fft` and `invk2j` benchmarks are the only programs where the variable-output strategies are necessary, but we perform each strategy indiscriminately. This indiscriminate application harms performance for the reinitialization strategy on `kmeans` and `sobel` when using CompNets and pretrained surrogates. For CompNets in particular, if we only applied each strategy where necessary, reinitialization would have outperformed cloning by a small margin.

Conclusion. In light of these results, we make the following decisions. We choose the cloning strategy for CompNets, the reinitialization strategy for MAML, and the growing strategy for pretrained surrogates.