# Learning to Compile Programs to Neural Networks

**Anonymous Authors**[1]

## Abstract

A *neural surrogate* is a neural network that mimics the behavior of a program. Neural surrogates of programs have been used to automatically tune program inputs, adapt programs to new settings, and accelerate computations. Neural surrogates have traditionally been developed by training on input-output examples for a single program. Language models present another approach wherein a model is trained on a single, large dataset then directly consumes program text, to act as a neural surrogate of the program. Having the language model as both the neural surrogate generator and the neural surrogate, however, poses a tradeoff of limited accuracy or excessive resource consumption. 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 they produce neural surrogates that are $2.56$-$5.51\times$ as data-efficient and train in $1.52$-$3.34\times$ 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 of programs 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., 2012).

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

**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 for a single program by collecting and curating a dataset of input-output pairs and training a neural network to predict the program output given the input (Renda et al., 2021).

Another point in the spectrum is to amortize the cost of training neural surrogates by training a language model to directly consume the text of a program and predict the program's output for a given input (Zaremba & Sutskever, 2015; Nye et al., 2021; Gu et al., 2024). A key benefit of this approach when compared to the traditional approach is that creating this dataset need only be done once, thereby enabling the creation of a neural surrogate for a given program without the need to curate a dataset of program-specific, input-output pairs as is required by the traditional approach.

However, these language model based approaches 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 (e.g., chain-of-thought reasoning) (Nye et al., 2021; Wei et al., 2022). These limitations pose challenges for successfully using such a model as a neural surrogate, as small models may not be able to capture complex programs (Zaremba & Sutskever, 2015) while 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., 2012; Mendis, 2020; Munk et al., 2022).

**Our Approach: Neural Surrogate Compilation.** To maintain the benefits of language model based approaches while bypassing the above limitations, we propose *neural surrogate compilation*. A neural surrogate compiler is a system, specialized to a given neural surrogate architecture, that accepts program text as input and produces an initial neural surrogate of the program. This initial neural surrogate can vary in behavioral quality, ranging from closely matching the behavior of the input program to only approximating the behavior of the program on a few inputs. We demonstrate in this work that this initial neural surrogate can be more quickly finetuned – as measured in both sample complexity and training time – to closely mimic the behavior of the program when compared to the traditional approach of training a neural surrogate from a random initialization.

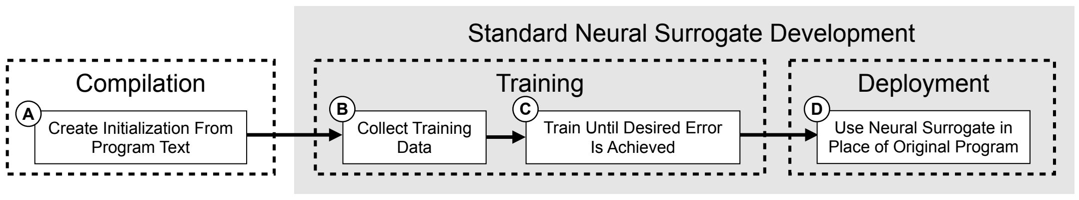
Standard Neural Surrogate Development
Compilation
A
Create Initialization From Program Text
Training
B
Collect Training Data
C
Train Until Desired Error Is Achieved
Deployment
D
Use Neural Surrogate in Place of Original Program

Figure 1: Neural surrogate development with neural surrogate compilation

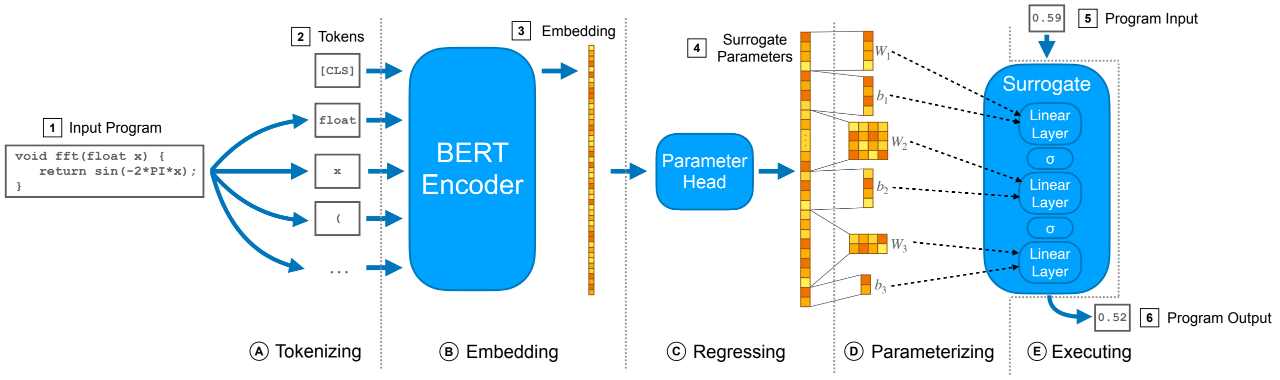

Figure 2: System diagram describing the HYBERTNET architecture, comprising five phases: (A) tokenizing an input program, (B) embedding the program using a BERT, (C) regressing the embeddings to a parameter weight vector using a parameter head, (D) parameterizing a neural network using the parameter weight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

**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 HYBERTNET.

To train HYBERTNETs, we develop EXESTACK, a dataset of executable C programs collected from The Stack (Kocetkov et al., 2022), a large corpus of source code. We then evaluate neural surrogates initialized via HYBERTNET on EXESTACK and PARROTBENCHSHORT, the latter being a set of benchmarks from prior work in approximate computing (Esmaeilzadeh et al., 2012).

Our technique produces neural surrogates that achieve 2.56-5.51× lower error than neural surrogates trained from scratch, with the same amount of data, and achieve a target error with 1.52-3.34× fewer epochs than neural surrogates trained from scratch.

## 2. Neural Surrogate Compilation

A *neural surrogate* is a neural network that models a subset of the observable behavior of a program. The typical strategy to train a neural surrogate is through supervised learning of a neural network with curated dataset of input-output pairs of the program (Renda et al., 2021).

### 2.1. The Neural 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 $\mathcal{I}_p$ 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 description $a : \mathcal{A}$, where $\mathcal{A} = \mathbb{R}^d \to \mathcal{I}_p \to \mathcal{O}_p$ is the space of neural network architectures, 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 $f : \mathcal{I}_p \to \mathcal{O}_p$ defined by $f(i) = a(\theta)(i)$ has low approximation error:

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

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

$$\mathbb{E}_{i \sim \mathcal{I}_p}[\ell(f(i), [\![p]\!](i))]$$

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

[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.

the number of epochs used to train the surrogate. We formalize these costs by defining a *training procedure* $t_a : \mathcal{P} \times \mathbb{R}^{\geq} \times \mathbb{N}^{\geq} \to \mathbb{R}^d$ for a given surrogate architecture $a$ as a random function that takes program text $p$, a training data budget $b : \mathcal{R}^{\geq}$, and training time budget $n : \mathcal{R}^{\geq}$ and produces a set of parameters $\theta : \mathbb{R}^d$ for the surrogate.

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

$$\arg\min_{t_a} \mathbb{E}_{\theta \sim t_a(p,b,n)} \left[ \mathbb{E}_{i \sim \mathcal{I}_p} [\ell(a(\theta)(i), [\![p]\!](i))] \right]$$

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 of the program (Renda et al., 2021).

### 2.2. Neural Surrogate Compilation

A *neural surrogate compiler* is a system that, given the source code of a program, produces a neural surrogate of the program, with the neural surrogate architecture fixed in advance. We use neural surrogate compilers as better solutions to the efficient surrogate training problem than traditional neural surrogate development, as they can produce an initialization for a neural surrogate that requires less data and fewer epochs to converge to a target error than random initialization.

Figure 1 illustrates how a neural surrogate compiler augments the traditional neural surrogate development workflow. 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 compliation workflow in which a neural surrogate compiler maps the program text to a neural network initialization for use in the the training of the neural surrogate.

**The Neural Surrogate Compiler Problem.** We formalize the development of a neural surrogate compiler as an optimization problem. The goal is to implement a system $\phi_a \in \mathcal{P} \to \mathbb{R}^d$ that accepts program text $p$ and produces neural network parameters $\theta$ for architecture description $a$ such that the surrogate $f = a(\phi_a(p))$ needs as little data as possible to reach low approximation error. We use the approximation error of the surrogate with no training as a proxy for the approximation error of the surrogate with additional training. The optimization problem is then:

$$\arg\min_{\phi_a \in \mathcal{P} \to \mathbb{R}^d} \mathbb{E}_{p \sim \mathcal{P}} [\mathbb{E}_{i \sim \mathcal{I}_p} [\ell(a(\phi_a(p))(i), [\![p]\!](i))]]$$

## 3. HyBERTNet

A HyBERTNet is an implementation of a neural surrogate compiler using hypernetworks. We first explain the architecture, then we explain how to train it.

### 3.1. Architecture

Figure 2 presents the architecture of a HyBERTNet.

A HyBERTNet accepts program text as input and produces parameters for a neural surrogate architecture $a : \mathbb{R}^d \to \mathcal{I} \to \mathcal{O}$ that is fixed in advance. A consequence of this design is that, to apply a given HyBERTNet to arbitrary programs, one requires a methodology for interpreting programs with potentially different type signatures as functions from $\mathcal{I}$ to $\mathcal{O}$.

Ⓐ First, HyBERTNet tokenizes an input program ($\boxed{1}$), resulting in a sequence of tokens ($\boxed{2}$) that includes a distinguished *classification token* `[CLS]`, as is common in BERT-based architectures.

Ⓑ HyBERTNet 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 ($\boxed{3}$); HyBERTNet discards the embeddings of the other tokens.

Ⓒ Next, HyBERTNet uses a linear layer to map the classification token embedding to a neural surrogate parameter vector ($\boxed{4}$).

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

Ⓔ Finally, HyBERTNet 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 HyBERTNet 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 HyBERTNet is used to compile.

We detail our collection of such a dataset, ExeStack, in Section 4. Since we would like to evaluate the generalization capabilities on both unseen programs and/or unseen program inputs, we do not use a simple train/test split; we use a train-program/test-program split and a train-input/test-input split.

With such a dataset in hand, a batch of the forward pass of training proceeds by selecting a batch of programs and input-output pairs for those programs, generating neural surrogate parameters for each program, then executing the corresponding neural surrogate with the corresponding inputs.

During the forward pass, this neural network interprets the parameters according to the target neural surrogate architecture and processes the inputs using these parameters. The loss is calculated as the mean squared error between the neural surrogates' predicted outputs and the true outputs.

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 HyBERTNet. Appendix B presents additional training details.

## 4. ExeStack

Learning a neural surrogate compiler requires a dataset of programs and input-output examples describing the behavior of each program (see Section 3.2). To fill this need, we developed ExeStack, a dataset of numerical, executable, deterministic C programs and corresponding input-output examples. ExeStack is based on The Stack, a dataset of 3 TB of permissively licensed source code written in various programming languages scraped from GitHub (Kocetkov et al., 2022). The following filters produce a dataset of numerical, executable, deterministic C programs, along with a set of input-output examples for each program.

**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 "`#`", we have run it 2 times, or an invocation fails. Once one of these conditions is met, we pass the file to the next filter in the pipeline.

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

**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 possiblity 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.

**Filtering for Executable Functions and Collecting Outputs.** To simultaneously check for executability and collect outputs from a function, we first generate 2048 sets of inputs by sampling from the uniform distribution $\mathcal{U}(-1, 1)$ and 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. The harness is compiled 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. We provide an example of the execution harness instantiated for a function in Appendix A.

To target a fixed neural surrogate architecture, a methodology is required for interpreting functions of varying type signatures as a single, fixed type signature. For a neural surrogate architecture with $m$ inputs and $n$ outputs, we distinguish the first $m$ arguments of a function as the input for the neural surrogate, and we initialize all other arguments to the constant 1.0. When a function has more than $n$ outputs, we only collect the first $n$ outputs. When a function has fewer than $n$ outputs, we pad the function outputs with the constant 0.0.

**Filtering for Deterministic Functions.** Since a neural surrogate is often a deterministic function of its inputs and weights, 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.

## 5. Evaluation

We answer the following research questions.

**RQ 1:** On average, does a neural surrogate initialized by a HyBERTNet converge to a lower test error than a neural surrogate initialized randomly, for a fixed training set size?

**RQ 2:** On average, does a neural surrogate initialized by a HyBERTNet converge to a target test error in fewer epochs than a neural surrogate initialized randomly?

Our results demonstrate that HyBERTNets lead to improvements in data efficiency (Section 5.2) and training time (Section 5.3).

### 5.1. Methodology

To develop and evaluate HyBERTNets, we choose a BERT architecture and a neural surrogate architecture, produce datasets HyBERTNets can be trained and evaluated on, and introduce a baseline initialization method to compare against.

| 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 PARROTBENCHSHORT (Esmaeilzadeh et al., 2012).

### 5.1.1. HYBERTNET ARCHITECTURE

As our base BERT architecture, we use BERT-Tiny (Turc et al., 2019).

We adopt one of the neural surrogate architectures used for the Parrot transformation by Esmaeilzadeh et al. (2012). This neural surrogate architecture is a multilayer perceptron consisting of a single input, a hidden layer of 4 neurons, another hidden layer of 4 neurons, and 2 outputs. The activation function is sigmoid.

### 5.1.2. DATASETS

We train HYBERTNETS on programs from EXESTACK variant. We test the effectiveness of HYBERTNETS on test programs from bothEXESTACK and PARROTBENCHSHORT, a subset of the suite of benchmarks introduced by Esmaeilzadeh et al. (2012) (Table 1).

**EXESTACK For HYBERTNETS** We applied additional filters to EXESTACK to produce a variant that is compatible with HYBERTNETS.

- **Filtering Long Programs**. Since BERT-Tiny has a maximum context length of 512 tokens, we remove functions with more than 512 tokens. We 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.

In total, we are left with 23,064 programs in this EXESTACK variant, each with 2048 input-output examples.

From the full set of programs, we created a train set and test set. The train set consists of 80% of the full set of programs and uses 50% of the input-output examples of each program. The test set consists of 1,000 programs selected uniformly at random from the 20% of programs that we withheld from the train set. We use all of 1,024 intput-output examples of each program to form the test set. We do not use the remainder the withheld programs due to the extensive cost of our evaluation methodology for programs in the test set.

**Parrot Benchmarks.** We refer to the original benchmark suite of Esmaeilzadeh et al. as PARROTBENCH. Table 1 shows the programs we include in PARROTBENCHSHORT, including descriptions of the computations and datasets.

Due to methodological choices in EXESTACK and architectural choices for HYBERTNETS, we omit some PARROTBENCH benchmarks from PARROTBENCHSHORT and modify others. The benchmarks `jmeint` and `jpeg` in PARROTBENCH are significantly longer than the context length of a BERT-Tiny (512 tokens), so we do not include them in our evaluation.

HYBERTNETS do not apply directly to `fft` and `invk2j`. Both benchmarks use pointer arguments to store the two outputs of the function. To make these functions pointer-free and thus compatible with our HYBERTNETS, we split each into two functions, each function computing one component of the outputs. Additionally, the `sobel` benchmark uses pointer inputs, so we rewrite it to only use scalar inputs. Finally, the `kmeans` benchmark uses custom structs to pass arguments, so we rewrite the benchmark to desugar these structs into their scalar components. The modified code for each benchmark is listed in Appendix C.

### 5.1.3. DATASET-TAILORED INITIALIZATION

As a program-text-agnostic baseline, we develop *dataset-tailored neural surrogate initializations* (DTI). That is, we train a single neural surrogate on all input-output examples of EXESTACK, ignoring program text. We use the resulting neural surrogate as an initialization for finetuning. If that initialization performs worse than HYBERTNET initializations, then we gain confidence hypernetworks were useful in developing an effective neural surrogate compiler, rather than just learning a dataset-tailored initialization.

To produce dataset-tailored neural surrogate initializations, we train a single neural surrogate—using the same neural surrogate architecture as the HYBERTNETS in this evaluation—on all input-output examples from training programs and training inputs in EXESTACK. We train the neural surrogate for 1,200 epochs, with a learning rate

| Dataset Size | Dataset-Tailored Init | HyBERTNet |
|---|---|---|
| 0% | 2.71× | 33.17× |
| 0.1% | 0.35× | 5.71× |
| 1% | 0.01× | 1.37× |
| 10% | 0.07× | 4.31× |
| 100% | 0.08× | 4.55× |

| Statistic | Dataset-Tailored Init | HyBERTNet |
|---|---|---|
| GM | 0.14× | 5.51× |
| 0th | $1.53 \cdot 10^{-9}\times$ | $3.58 \cdot 10^{-7}\times$ |
| 25th | 0.02× | 1.05× |
| 50th | 0.25× | 5.78× |
| 75th | 2.18× | 33.15× |
| 100th | $1.80 \cdot 10^{3}\times$ | $8.56 \cdot 10^{6}\times$ |
| MPI | 66th | 25th |

Figure 3: Geometric mean test-input loss improvement over random initialization on EXESTACK test programs, grouped by dataset sizes (left), as well as test-input loss improvements over all programs and dataset sizes (right). The table on the right reports overall geometric mean improvements, percentiles from 0th to 100th, and the minimum percentile at which an initialization method improves over random initialization.

of 0.01. We perform 3 trials with varying random seeds, producing 3 dataset-tailored surrogate initializations.

#### 5.1.4. QUANTIFYING IMPROVEMENTS

We define the improvement for a given configuration (consisting of an initialization, program, and budget) as the ratio of the arithmetic mean of the test losses in that configuration to the arithmetic mean of the test losses of that program and budget when initialized randomly. For each initialization method, we report the geometric mean of the improvements grouped by program, grouped by budget, and overall.

### 5.2. HYBERTNET Improvements To Data Efficiency

To assess whether HYBERTNETs improve data efficiency, we use HYBERTNETs to initialize neural surrogates, finetune on subsets of training data of various sizes, then compare the results to those of other initialization methods.

#### 5.2.1. METHODOLOGY

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

**Experiment Configurations.** We sweep over a number of configurations in this experiment, each consisting of a program, a dataset size, and an initialization method (e.g., a HYBERTNET).

Each dataset size specifies the percentage of the training data to train neural surrogates on. We sweep over the following percentages: $\{0\%, 0.1\%, 1\%, 10\%, 100\%\}$.

Given a configuration consisting of a program, dataset size percentage $c$, 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 using the Adam optimizer with no weight decay and a learning rate of 0.01[2]. The final test loss we report for a trial is the test error at the epoch closest to the epoch with the lowest validation error.[3] When the dataset size is 0, we use the test loss at the final epoch.

#### 5.2.2. RESULTS

Figures 3 and 4 show the results of finetuning for a sample of 1,000 EXESTACK test programs and PARROT-BENCHSHORT, respectively, using DTI to refer to dataset-tailored initializations and HBN to refer to HYBERTNET initializations. More data is available in Appendix D.

**EXESTACK Test Programs.** The improvement due to HYBERTNETs on EXESTACK test programs is most pronounced in the zero-shot regime, where the improvement is 33.17× over random initialization. The zero-shot regime is also the only regime where dataset-tailored initializations show an improvement, achieving 2.71× better test loss than random initializations. The worst performance for both HYBERTNETs and dataset-tailored initializations is in the middle of the dataset sizes we evaluated, at a dataset size of 0.1. Here, HYBERTNETs achieved only a 1.37× improvement and dataset-tailored initializations achieved only a 0.01× improvement (i.e., a 100× degradation).

Over all sampled EXESTACK programs, HYBERTNETs achieve a data efficiency of 5.5× over randomly initailized

[2] The number of epochs and learning rate are in accordance with Esmaeilzadeh et al. (2012).

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

| Program | DTI | MAML | HBN |
|---|---|---|---|
| `fft` (0) | 0.05× | 0.86× | 2.89× |
| `fft` (1) | 0.21× | 1.10× | 2.66× |
| `invk2j` (0) | 0.19× | 0.98× | 1.13× |
| `invk2j` (1) | 2.83× | 0.99× | 16.07× |
| `kmeans` | 0.54× | 1.04× | 1.05× |
| `sobel` | 0.70× | 1.00× | 1.89× |

| Dataset Size | DTI | MAML | HBN |
|---|---|---|---|
| 0% | 1.45× | 1.00× | 1.53× |
| 0.1% | 0.16× | 0.97× | 1.17× |
| 1% | 0.28× | 1.03× | 3.22× |
| 10% | 0.23× | 1.13× | 3.78× |
| 100% | 0.42× | 0.86× | 5.04× |

| Statistic | DTI | MAML | HBN |
|---|---|---|---|
| GM | 0.36× | 0.99× | 2.56× |
| 0th | $3.87 \cdot 10^{-5}\times$ | 0.10× | 0.07× |
| 25th | 0.19× | 0.92× | 1.01× |
| 50th | 0.73× | 1.00× | 1.65× |
| 75th | 1.69× | 1.13× | 4.52× |
| 100th | 21.19× | 5.78× | 787.50× |
| MPI | 65th | 55th | 25th |

Figure 4: Test-input loss improvement over random initialization on PARROTBENCHSHORT, grouped by dataset sizes (left) and by programs (middle), as well as test-input loss improvements over all programs, dataset sizes, and train trials (right). DTI refers to dataset-tailored initializations and HBN to HYBERTNET. The suffixes (0) and (1) denote the first and second outputs of a program, respectively. The rightmost table reports overall geometric mean improvements, percentiles from 0th to 100th, and the minimum percentile at which an initialization method improves over random initialization.

neural surrogates, while dataset-tailored neural surrogate initializations achieve a data efficiency of $0.14\times$, worsening performance. For both initialization methods, the minimums and maximums are extreme, with HYBERTNETs having a minimum and maximum of $3.58 \cdot 10^{-7}\times$ and $8.56 \cdot 10^{6}\times$, respectively, while dataset-tailored initializations have a minimum and maximum of $1.53 \cdot 10^{-9}\times$ and $1.80 \cdot 10^{3}\times$, respectively. As low as the 25th percentile, HYBERTNETs improve over random initialization by $1.05\times$.

**PARROTBENCHSHORT Programs.** HYBERTNETs improve data efficiency on each PARROTBENCHSHORT program, with the smallest improvement on `kmeans` (1.05×) and the largest improvement on the second component of `invk2j` (16.07×). The reasons for the disparity between these programs is unclear, with both having a comparable length (`kmeans` being 122 tokens and `invk2j` (1) being 84 tokens) and one transcendental function (`kmeans` using `sqrt` and `invk2j` (1) using `acos`). Dataset-tailored initializations only improve on `invk2j` (1), with an improvement of 2.83×.

The improvement due to HYBERTNETs is most pronounced at a dataset size of 100%, unlike the EXESTACK results, with HYBERTNETs achieving a $5.04\times$ improvement. Dataset-tailored initializations, however, perform best in the zero-shot regime, with a $1.45\times$ improvement. The worst performance for both initialization methods is again in the middle, at a dataset size of 0.1%, with HYBERTNETs achieving only a $1.1\times$ improvement and dataset-tailored initializations achieving only $0.16\times$.

Over all PARROTBENCHSHORT programs, HYBERTNETs achieve a data efficiency of $2.56\times$ over randomly initialized neural surrogates, while dataset-tailored neural surrogate initializations achieve a data efficiency of $0.36\times$, worsening performance. The minimum and maximum for HYBERTNET improvements (0.07× and 787.50×, respectively) only span 4 orders of magnitude, unlike those for EXESTACK. However, dataset-tailored initializations span 7 orders of magnitude, with a minimum of $3.87 \cdot 10^{-5}\times$ and a maximum of $21.19\times$. As low as the 25th percentile, HYBERTNETs improve over random initialization by $1.01\times$.

Since HYBERTNETs improve data efficiency over random initialization on both EXESTACK and PARROTBENCHSHORT, we answer yes to RQ 1.

### 5.3. HYBERTNET Improvements To Training Time

To assess whether HYBERTNETs improve training time, we sweep over initialization methods and finetune initialized surrogates until they reach a target test error. We first detail the methodology of this experiment, then present results.

| Statistic | DTI | HBN |
|---|---|---|
| GM | 0.64× | 3.34× |
| 0th | 0.04× | 0.03× |
| 25th | 0.68× | 0.97× |
| 50th | 0.86× | 2.30× |
| 75th | 0.88× | 8.46× |
| 100th | 15.08× | $1.66 \cdot 10^3$× |
| MPI | 92nd | 27th |

(a) Geometric mean and percentile improvements to training time over random initialization on a sample of 1,000 EXESTACK test programs. MPI is the minimum percentile at which an initialization method improves over random initialization.

| Program | DTI | HBN |
|---|---|---|
| `fft` (0) | 0.46× | 3.45× |
| `fft` (1) | 1.02× | 3.76× |
| `invk2j` (0) | 0.81× | 1.12× |
| `invk2j` (1) | 0.58× | 1.03× |
| `kmeans` | 0.24× | 0.86× |
| `sobel` | 0.61× | 0.96× |

| Statistic | DTI | HBN |
|---|---|---|
| GM | 0.57× | 1.52× |
| 0th | 0.13× | 0.65× |
| 25th | 0.41× | 0.94× |
| 50th | 0.58× | 1.21× |
| 75th | 0.76× | 1.64× |
| 100th | 6.39× | 9.88× |
| MPI | 92nd | 37th |

(b) Geometric mean training time improvements over random initialization on PARROTBENCHSHORT, grouped by programs (left), as well as training time improvements over all programs and training trials (right). The table on the right includes geometric mean improvement, percentiles from 0th to 100th, and the minimum percentile at which an initialization method improves over random initialization.

Figure 5: Training time improvements on EXESTACK test programs (left) and PARROTBENCHSHORT programs (right). DTI refers to dataset-tailored initializations and HBN refers to HYBERTNET initializations.

#### 5.3.1. METHODOLOGY

We now describe how we set a target error to use as a stopping condition and the configurations we sweep over.

**Setting a Target Error.** We set a target test error for each program by training 3 randomly initialized surrogates for 5,000 epochs with learning rate 0.01. The average final test error is the target test error for all initialization methods.

**Experiment Configurations.** We sweep over a number of configurations in this experiment, each 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 error. We repeat the process above 3 times with different random seeds.[4]

#### 5.3.2. RESULTS

The results are summarized in Table 5a and Figure 5b for EXESTACK test and PARROTBENCHSHORT, respectively. Additional data is available in Appendix E.

**EXESTACK Test Programs.** Over a sample of EXESTACK test programs, HYBERTNETs produce neural surrogates that train 3.34× faster than randomly initialized surrogates, whereas dataset-tailored initializations train 0.64× faster (i.e., 1.56× slower). HYBERTNETs see improvements of 0.97× as low as the 25th percentile, whereas dataset-tailored initializations see improvements of at most 0.88× up to the 75th percentile.

[4] For the dataset-tailored surrogate and HYBERTNET initializations, this just changes the training data order.

**PARROTBENCHSHORT Programs.** Over PARROTBENCHSHORT programs, HYBERTNETs range between improvements of 0.86× on `kmeans` to 3.76× on `fft` (1), whereas dataset-tailored initializations range between improvements of 0.24× on `kmeans` to 1.02× on `fft`(1). The worst HYBERTNET performance for data efficiency was also `kmeans`, suggesting this benchmark is difficult for HYBERTNETs to target.

Averaged over all PARROTBENCHSHORT programs, HYBERTNETs achieve a 1.52× training time improvement over random initialization, and dataset-tailored initializations achieve a 0.57× improvement. The worst trial over all HYBERTNETs achieved a 0.65× improvement and the best trial achieved a 9.88× improvement. The worst trial over all dataset-tailored initializations achieved a 0.13× improvement and the best trial achieved a 6.39× improvement.

Since HYBERTNETs improve training time over random initialization on both EXESTACK and PARROTBENCHSHORT, we answer yes to RQ 2.

## 6. Related Work

There are bodies of work on both neural surrogates of programs and meta-learning. Our work draws on both fields and takes inspiration from compilers.

**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.** A notable approach to few-shot learning that does not use hypernetworks is *model-agnostic meta-learning (MAML)*. MAML is a framework for developing neural network initializations that can be finetuned to new tasks with a small amount of data and a small number of SGD steps (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.

## 7. Conclusion

In this paper, we presented the concept of a neural surrogate compiler and demonstrated how a neural surrogate compiler can be implemented with HyBERTNets. We provided a dataset, ExeStack, that one can use to learn neural surrogate compilers. We demonstrated the effectiveness of HyBERTNets on ExeStack programs and ParrotBenchShort, a suite of numerical benchmarks spanning various application domains. Specifically, we showed HyBERTNets achieve a loss $2.56$-$5.51\times$ lower than randomly initialized neural surrogates and train in $1.52$-$3.34\times$ fewer epochs than randomly initialized neural surrogates.

The key insight of our work is 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.

## 8. 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.

## References

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 *IEEE/ACM International Symposium on Microarchitecture*, 2012.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 1126–1135. JMLR.org, 2017.

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

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

Ïpek, 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 *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020.

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. *Preprint*, 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. *IEEE Transactions on Plasma Science*, 48:46–53, 2020.

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

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 *The 38th Conference on 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, 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.

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 *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2021.

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*, 72:185–190, 2018. ISSN 2212-8271. 51st CIRP Conference on Manufacturing Systems.

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. *ACM Transactions on Graphics (TOG)*, 38(4), 7 2019. doi: 10.1145/3306346.3322996.

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

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.

Zaremba, W. and Sutskever, I. Learning to execute, 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.

```c
#include <math.h>

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

int main() {
    return 0;
}
```

Figure 6: Source code template used for checking compilability, instantiated with the source of `fftSin`.

## A. Execution Harness Example

Figure 6 shows an example of the code we use to check whether functions can be compiled. Figure 7 shows an example of the execution harness we use to collect outputs from functions.

## B. Training Details

| | |
|---|---|
| Architecture | BERT-Tiny |
| Surrogate topology | $1 \to 4 \to 4 \to 2$ |
| Hypernet/program batch size | 32 |
| Surrogate/input batch size | 1024 |
| Max program length (in tokens) | 512 |
| Tokenizer vocab size | 30,522 |
| Total number of programs in dataset | 23,064 |
| Number of tokens in dataset | 1,035,021 |
| Number of training programs | 18,451 |
| Number of testing programs | 4,613 |
| Number of train I/O pairs per program | 1024 |
| Number of test I/O pairs per program | 1024 |
| Number of epochs | 1200 |
| Learning rate | 2e-5 |
| GPU | NVIDIA V100 16GB |

Table 2: Experimental Setup

## C. Parrot Benchmark Code

We present the code used for PARROTBENCHSHORT in our evaluation (Section 5), which we adapted from PARROTBENCH to be pointer-free. The code for the benchmarks is shown in Figures 8, 9, 10, and 11.

## D. Data Efficiency (Extended)

Figures 12 and 13 show the final test losses achieved by each initialization method on each configuration, averaged over trials.

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

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

float fftSin(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 7: Source code template used for checking executability and collecting outputs, instantiated with the source of `fftSin`.

## E. Training Time Evaluation (Extended)

Here we collect additional information, for both EXESTACK test programs and PARROTBENCHSHORT programs, about the number of epochs (Tables 3 and 7), initial train losses (Tables 4 and 8), and initial test losses (Tables 5 and 9) for each initialization method, as well as the baseline train and test losses for each program (Tables 6 and 10).

## F. MAML Implementation

In our implementation of MAML, we maintain two copies of EXESTACK training data. In each epoch, we randomly sample a batch without replacement from one copy and another batch from the other copy. We use one batch to collect the $\theta'_i$. We use the other batch to evaluate collect losses with respect to each $\theta'_i$, which we then use to update $\theta$.

We use $\alpha = 0.01$ and $\beta = 0.01$.

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

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

Figure 8: Code for the `fft` benchmark in PARROTBENCHSHORT.

```
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 9: Code for the `invk2j` benchmark in PARROTBENCHSHORT.

```
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 10: Code for the `kmeans` benchmark in PARROTBENCHSHORT.

```
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 11: Code for the `sobel` benchmark in PARROTBENCHSHORT.

| Dataset Size $0\%$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| 3858 | 0.30 | 0.12 | 0.15 | 0.17 | 0.02 | 0.37 | 0.24 |
| 379 | 1.03 | 0.40 | 0.35 | 0.32 | 0.63 | 0.93 | 0.68 |
| 4141 | 0.37 | 0.12 | 0.15 | 0.17 | 0.37 | 0.23 | 0.16 |
| 619 | 0.32 | 0.06 | 0.07 | 0.07 | 0.46 | 0.04 | 0.23 |
| 4570 | 0.21 | 0.09 | 0.13 | 0.14 | $1.20 \times 10^{-4}$ | $1.11 \times 10^{-4}$ | $6.22 \times 10^{-6}$ |

| Dataset Size $0.1\%$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| 3858 | 0.26 | 0.09 | 0.05 | 0.07 | $1.69 \times 10^{-3}$ | 0.22 | 0.10 |
| 379 | 0.06 | 0.40 | 0.76 | 0.24 | 0.42 | 0.96 | 0.54 |
| 4141 | 0.22 | 0.09 | 0.06 | 0.07 | 0.48 | 0.32 | 0.22 |
| 619 | 0.07 | 0.08 | 0.09 | 0.09 | 0.58 | 0.05 | 0.10 |
| 4570 | $1.19 \times 10^{-4}$ | 0.05 | 0.06 | 0.05 | $9.61 \times 10^{-7}$ | $1.52 \times 10^{-5}$ | $3.93 \times 10^{-6}$ |

| Dataset Size $1\%$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| 3858 | $7.20 \times 10^{-5}$ | 0.08 | 0.01 | 0.01 | $1.54 \times 10^{-5}$ | $4.71 \times 10^{-4}$ | $2.30 \times 10^{-3}$ |
| 379 | $5.42 \times 10^{-5}$ | 1.01 | 0.03 | 0.01 | $2.47 \times 10^{-4}$ | $3.14 \times 10^{-4}$ | $1.73 \times 10^{-4}$ |
| 4141 | $1.09 \times 10^{-4}$ | 0.01 | 0.01 | $3.75 \times 10^{-3}$ | $3.37 \times 10^{-4}$ | $4.38 \times 10^{-5}$ | $5.73 \times 10^{-4}$ |
| 619 | $2.58 \times 10^{-3}$ | 0.07 | 0.03 | 0.03 | 0.01 | $2.55 \times 10^{-4}$ | $4.61 \times 10^{-4}$ |
| 4570 | $2.72 \times 10^{-8}$ | $5.22 \times 10^{-4}$ | $5.10 \times 10^{-6}$ | $1.46 \times 10^{-6}$ | $4.37 \times 10^{-10}$ | $1.78 \times 10^{-9}$ | $1.05 \times 10^{-10}$ |

| Dataset Size $10\%$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| 3858 | $2.60 \times 10^{-6}$ | $9.95 \times 10^{-6}$ | $3.13 \times 10^{-5}$ | $4.31 \times 10^{-5}$ | $6.68 \times 10^{-8}$ | $4.45 \times 10^{-6}$ | $4.17 \times 10^{-6}$ |
| 379 | $5.34 \times 10^{-6}$ | $5.01 \times 10^{-6}$ | $4.41 \times 10^{-3}$ | $5.99 \times 10^{-6}$ | $7.82 \times 10^{-6}$ | $3.46 \times 10^{-6}$ | $6.00 \times 10^{-6}$ |
| 4141 | $2.53 \times 10^{-6}$ | $7.84 \times 10^{-6}$ | $3.07 \times 10^{-5}$ | $4.18 \times 10^{-5}$ | $1.92 \times 10^{-6}$ | $6.25 \times 10^{-7}$ | $1.81 \times 10^{-6}$ |
| 619 | 0.01 | 0.02 | 0.03 | 0.02 | 0.03 | $1.04 \times 10^{-6}$ | $1.30 \times 10^{-6}$ |
| 4570 | $3.19 \times 10^{-9}$ | $9.80 \times 10^{-7}$ | $9.24 \times 10^{-7}$ | $2.91 \times 10^{-7}$ | $4.76 \times 10^{-11}$ | $6.06 \times 10^{-9}$ | $1.33 \times 10^{-11}$ |

| Dataset Size $100\%$ | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| 3858 | $2.18 \times 10^{-6}$ | $4.58 \times 10^{-6}$ | $2.05 \times 10^{-5}$ | $4.00 \times 10^{-5}$ | $4.11 \times 10^{-8}$ | $3.55 \times 10^{-6}$ | $3.56 \times 10^{-6}$ |
| 379 | $5.32 \times 10^{-6}$ | $8.59 \times 10^{-7}$ | $4.56 \times 10^{-6}$ | $4.80 \times 10^{-6}$ | $7.00 \times 10^{-6}$ | $2.76 \times 10^{-6}$ | $5.38 \times 10^{-6}$ |
| 4141 | $2.10 \times 10^{-6}$ | $4.62 \times 10^{-6}$ | $2.04 \times 10^{-5}$ | $3.99 \times 10^{-5}$ | $1.63 \times 10^{-6}$ | $5.46 \times 10^{-7}$ | $1.46 \times 10^{-6}$ |
| 619 | $3.73 \times 10^{-6}$ | 0.01 | 0.03 | 0.03 | 0.03 | $7.74 \times 10^{-7}$ | $1.03 \times 10^{-6}$ |
| 4570 | $1.98 \times 10^{-9}$ | $7.40 \times 10^{-7}$ | $1.26 \times 10^{-6}$ | $2.49 \times 10^{-7}$ | $5.18 \times 10^{-11}$ | $5.30 \times 10^{-10}$ | $4.22 \times 10^{-11}$ |

Figure 12: Average final test loss for each initialization method, program, and dataset size on a sample of 5 EXESTACK test programs, from the 1,000 we evaluated on.

| Dataset Size 0% | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| `fft` (0) | 0.52 | 0.91 | 1.00 | 1.03 | 1.66 | 0.46 | 0.44 |
| `fft` (1) | 0.70 | 0.66 | 0.70 | 0.71 | 0.63 | 0.38 | 0.57 |
| `invk2j` (0) | 1.47 | 1.73 | 1.80 | 1.82 | 1.94 | 1.58 | 1.93 |
| `invk2j` (1) | 6.24 | 4.60 | 4.47 | 4.42 | 4.44 | 3.87 | 3.56 |
| `kmeans` | 0.76 | 0.19 | 0.20 | 0.20 | 0.23 | 0.27 | 0.24 |
| `sobel` | 0.74 | 0.19 | 0.19 | 0.19 | 0.29 | 0.22 | 0.21 |

| Dataset Size 0.1% | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| `fft` (0) | $8.39 \times 10^{-5}$ | $1.15 \times 10^{-3}$ | 0.06 | 0.02 | $1.55 \times 10^{-4}$ | $9.40 \times 10^{-5}$ | $6.24 \times 10^{-6}$ |
| `fft` (1) | $2.30 \times 10^{-4}$ | $1.20 \times 10^{-3}$ | 0.53 | $2.30 \times 10^{-5}$ | $1.79 \times 10^{-4}$ | $1.69 \times 10^{-4}$ | $6.09 \times 10^{-5}$ |
| `invk2j` (0) | 0.31 | 0.45 | 1.56 | 0.71 | 0.45 | 0.40 | 0.82 |
| `invk2j` (1) | 0.07 | 0.13 | 0.05 | 0.09 | 0.02 | 0.03 | 0.10 |
| `kmeans` | $8.20 \times 10^{-7}$ | $1.70 \times 10^{-5}$ | $3.59 \times 10^{-6}$ | $7.77 \times 10^{-6}$ | $1.45 \times 10^{-6}$ | $4.88 \times 10^{-7}$ | $1.08 \times 10^{-5}$ |
| `sobel` | $3.75 \times 10^{-4}$ | $1.81 \times 10^{-3}$ | $2.35 \times 10^{-4}$ | $4.20 \times 10^{-4}$ | $7.59 \times 10^{-5}$ | $4.57 \times 10^{-4}$ | $3.00 \times 10^{-4}$ |

| Dataset Size 1% | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| `fft` (0) | $3.80 \times 10^{-5}$ | $9.09 \times 10^{-5}$ | 0.06 | $2.01 \times 10^{-4}$ | $1.56 \times 10^{-5}$ | $8.29 \times 10^{-6}$ | $5.83 \times 10^{-7}$ |
| `fft` (1) | $1.17 \times 10^{-4}$ | $9.25 \times 10^{-4}$ | 0.49 | $1.16 \times 10^{-5}$ | $1.02 \times 10^{-4}$ | $1.25 \times 10^{-4}$ | $1.67 \times 10^{-5}$ |
| `invk2j` (0) | 0.07 | 0.05 | 1.29 | 0.07 | 0.03 | 0.05 | 0.02 |
| `invk2j` (1) | 0.02 | $3.29 \times 10^{-3}$ | 0.01 | 0.01 | $1.20 \times 10^{-3}$ | $1.05 \times 10^{-3}$ | $7.49 \times 10^{-4}$ |
| `kmeans` | $6.36 \times 10^{-7}$ | $2.29 \times 10^{-5}$ | $2.75 \times 10^{-6}$ | $3.83 \times 10^{-6}$ | $1.13 \times 10^{-6}$ | $4.83 \times 10^{-7}$ | $8.51 \times 10^{-6}$ |
| `sobel` | $1.07 \times 10^{-4}$ | $1.68 \times 10^{-4}$ | $9.82 \times 10^{-5}$ | $6.07 \times 10^{-5}$ | $1.10 \times 10^{-5}$ | $1.28 \times 10^{-4}$ | $2.33 \times 10^{-5}$ |

| Dataset Size 10% | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| `fft` (0) | $9.38 \times 10^{-6}$ | $5.46 \times 10^{-6}$ | 0.06 | $1.27 \times 10^{-5}$ | $3.74 \times 10^{-6}$ | $1.08 \times 10^{-6}$ | $1.85 \times 10^{-7}$ |
| `fft` (1) | $1.89 \times 10^{-5}$ | $2.59 \times 10^{-5}$ | 0.49 | $5.08 \times 10^{-6}$ | $1.64 \times 10^{-5}$ | $8.09 \times 10^{-5}$ | $5.50 \times 10^{-7}$ |
| `invk2j` (0) | 0.01 | 0.02 | 1.28 | 0.04 | 0.01 | 0.01 | 0.01 |
| `invk2j` (1) | 0.02 | $2.92 \times 10^{-3}$ | 0.01 | 0.01 | $8.54 \times 10^{-4}$ | $7.75 \times 10^{-4}$ | $5.59 \times 10^{-4}$ |
| `kmeans` | $3.79 \times 10^{-7}$ | $1.86 \times 10^{-6}$ | $4.90 \times 10^{-7}$ | $6.47 \times 10^{-7}$ | $6.58 \times 10^{-8}$ | $3.85 \times 10^{-7}$ | $2.24 \times 10^{-7}$ |
| `sobel` | $1.78 \times 10^{-5}$ | $1.07 \times 10^{-4}$ | $3.02 \times 10^{-5}$ | $1.97 \times 10^{-5}$ | $8.26 \times 10^{-6}$ | $1.22 \times 10^{-5}$ | $1.34 \times 10^{-5}$ |

| Dataset Size 100% | | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
| `fft` (0) | $7.26 \times 10^{-7}$ | $5.52 \times 10^{-7}$ | $3.61 \times 10^{-3}$ | $1.60 \times 10^{-6}$ | $6.60 \times 10^{-6}$ | $2.18 \times 10^{-7}$ | $6.95 \times 10^{-8}$ |
| `fft` (1) | $8.16 \times 10^{-6}$ | $1.02 \times 10^{-5}$ | $1.81 \times 10^{-5}$ | $1.20 \times 10^{-6}$ | $1.87 \times 10^{-6}$ | $1.29 \times 10^{-6}$ | $1.11 \times 10^{-7}$ |
| `invk2j` (0) | $3.23 \times 10^{-4}$ | $1.62 \times 10^{-3}$ | 0.10 | 0.01 | $2.00 \times 10^{-4}$ | $2.13 \times 10^{-4}$ | $2.04 \times 10^{-4}$ |
| `invk2j` (1) | 0.02 | $1.04 \times 10^{-3}$ | 0.01 | $2.52 \times 10^{-3}$ | $4.87 \times 10^{-5}$ | $5.91 \times 10^{-5}$ | $2.81 \times 10^{-5}$ |
| `kmeans` | $9.14 \times 10^{-7}$ | $1.05 \times 10^{-6}$ | $1.09 \times 10^{-7}$ | $4.75 \times 10^{-7}$ | $2.84 \times 10^{-6}$ | $3.07 \times 10^{-7}$ | $4.64 \times 10^{-7}$ |
| `sobel` | $1.27 \times 10^{-6}$ | $1.71 \times 10^{-5}$ | $1.13 \times 10^{-5}$ | $4.49 \times 10^{-6}$ | $1.18 \times 10^{-6}$ | $1.87 \times 10^{-6}$ | $1.62 \times 10^{-6}$ |

Figure 13: Average final test loss for each initialization method, program, and dataset size on ParrotBenchShort.

| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
|---|---|---|---|---|---|---|---|
| 3858 | 4247.0 | 4998.0 | 4998.0 | 4998.0 | 1314.0 | 4998.0 | 4998.0 |
| 379 | 4466.0 | 3897.0 | 4998.0 | 4998.0 | 4998.0 | 4998.0 | 4998.0 |
| 4141 | 4300.0 | 4998.0 | 4998.0 | 4998.0 | 3654.0 | 2430.0 | 4350.0 |
| 619 | 4330.0 | 4998.0 | 4998.0 | 4998.0 | 4998.0 | 1742.0 | 3864.0 |
| 4570 | 3249.0 | 4998.0 | 4998.0 | 4998.0 | 132.0 | 4998.0 | 285.0 |
| 1708 | 2362.0 | 4998.0 | 4998.0 | 4998.0 | 4998.0 | 4998.0 | 4998.0 |
| 252 | 4405.0 | 4998.0 | 4998.0 | 4998.0 | 2117.0 | 615.0 | 450.0 |
| 4046 | 4372.0 | 4998.0 | 4998.0 | 4998.0 | 240.0 | 447.0 | 1272.0 |
| 4222 | 4372.0 | 4998.0 | 4998.0 | 4998.0 | 1281.0 | 2469.0 | 1191.0 |
| 2384 | 4001.0 | 4998.0 | 4998.0 | 4998.0 | 801.0 | 381.0 | 96.0 |

Table 3: Number of epochs required for each initialization method to achieve the baseline test error on each of a sample of 10 EXESTACK test programs, from the 1,000 we evaluated on.

| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
|---|---|---|---|---|---|---|---|
| 3858 | 0.4 | 0.1 | 0.2 | 0.2 | $1.9 \times 10^{-2}$ | 0.4 | 0.2 |
| 379 | 0.9 | 0.4 | 0.4 | 0.3 | 0.6 | 0.9 | 0.7 |
| 4141 | 0.4 | 0.1 | 0.2 | 0.2 | 0.4 | 0.2 | 0.2 |
| 619 | 0.3 | 0.1 | 0.1 | 0.1 | 0.5 | $3.8 \times 10^{-2}$ | 0.2 |
| 4570 | 0.2 | 0.1 | 0.1 | 0.1 | $1.2 \times 10^{-4}$ | $1.1 \times 10^{-4}$ | $5.9 \times 10^{-6}$ |
| 1708 | 0.2 | 0.1 | 0.1 | 0.1 | 0.1 | 0.3 | 0.1 |
| 252 | 25.0 | 22.0 | 21.5 | 21.3 | 3.9 | 4.7 | 0.8 |
| 4046 | 0.4 | 0.1 | 0.2 | 0.2 | $9.6 \times 10^{-3}$ | $3.6 \times 10^{-3}$ | $1.2 \times 10^{-2}$ |
| 4222 | 0.4 | 0.1 | 0.2 | 0.2 | $4.3 \times 10^{-2}$ | 0.1 | $1.7 \times 10^{-2}$ |
| 2384 | 0.9 | 0.3 | 0.2 | 0.2 | $2.8 \times 10^{-2}$ | $3.4 \times 10^{-2}$ | $1.2 \times 10^{-3}$ |

Table 4: Initial train losses for neural surrogates produced by each initialization method on a sample of 10 EXESTACK test programs, from the 1,000 we evaluated on.

| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
|---|---|---|---|---|---|---|---|
| 3858 | 0.4 | 0.1 | 0.2 | 0.2 | $2.0 \times 10^{-2}$ | 0.4 | 0.2 |
| 379 | 0.9 | 0.4 | 0.3 | 0.3 | 0.6 | 0.9 | 0.7 |
| 4141 | 0.4 | 0.1 | 0.2 | 0.2 | 0.4 | 0.2 | 0.2 |
| 619 | 0.3 | 0.1 | 0.1 | 0.1 | 0.5 | $3.7 \times 10^{-2}$ | 0.2 |
| 4570 | 0.2 | 0.1 | 0.1 | 0.1 | $1.2 \times 10^{-4}$ | $1.1 \times 10^{-4}$ | $6.2 \times 10^{-6}$ |
| 1708 | 0.2 | 0.1 | 0.1 | 0.1 | 0.1 | 0.3 | 0.1 |
| 252 | 25.1 | 22.1 | 21.5 | 21.3 | 3.9 | 4.7 | 0.8 |
| 4046 | 0.4 | 0.1 | 0.2 | 0.2 | $9.4 \times 10^{-3}$ | $3.8 \times 10^{-3}$ | $1.2 \times 10^{-2}$ |
| 4222 | 0.4 | 0.1 | 0.2 | 0.2 | $4.2 \times 10^{-2}$ | 0.1 | $1.6 \times 10^{-2}$ |
| 2384 | 0.9 | 0.3 | 0.2 | 0.2 | $2.7 \times 10^{-2}$ | $3.3 \times 10^{-2}$ | $1.2 \times 10^{-3}$ |

Table 5: Initial test losses for neural surrogates produced by each initialization method on a sample of 10 EXESTACK test programs, from the 1,000 we evaluated on.

| Program | Final Baseline Train Loss | Final Baseline Test Loss |
|---|---|---|
| 3858 | $3.7 \times 10^{-6} \pm 4.1 \times 10^{-7}$ | $3.7 \times 10^{-6} \pm 4.3 \times 10^{-7}$ |
| 379 | $1.9 \times 10^{-6} \pm 4.1 \times 10^{-7}$ | $2.0 \times 10^{-6} \pm 4.1 \times 10^{-7}$ |
| 4141 | $3.5 \times 10^{-6} \pm 3.7 \times 10^{-7}$ | $3.6 \times 10^{-6} \pm 3.7 \times 10^{-7}$ |
| 619 | $3.2 \times 10^{-6} \pm 5.2 \times 10^{-7}$ | $3.5 \times 10^{-6} \pm 6.3 \times 10^{-7}$ |
| 4570 | $5.2 \times 10^{-10} \pm 1.3 \times 10^{-10}$ | $5.2 \times 10^{-10} \pm 1.3 \times 10^{-10}$ |
| 1708 | $8.8 \times 10^{-10} \pm 4.3 \times 10^{-12}$ | $8.8 \times 10^{-10} \pm 1.7 \times 10^{-12}$ |
| 252 | $7.6 \times 10^{-5} \pm 2.7 \times 10^{-5}$ | $7.4 \times 10^{-5} \pm 2.8 \times 10^{-5}$ |
| 4046 | $3.5 \times 10^{-6} \pm 5.1 \times 10^{-7}$ | $3.5 \times 10^{-6} \pm 5.4 \times 10^{-7}$ |
| 4222 | $3.5 \times 10^{-6} \pm 5.1 \times 10^{-7}$ | $3.5 \times 10^{-6} \pm 5.4 \times 10^{-7}$ |
| 2384 | $4.7 \times 10^{-6} \pm 5.4 \times 10^{-7}$ | $4.8 \times 10^{-6} \pm 6.3 \times 10^{-7}$ |

Table 6: Final baseline train and test losses for a sample of 10 EXESTACK test programs, from the 1,000 we evaluated on.

| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
|---|---|---|---|---|---|---|---|
| `invk2j` (0) | 4066.0 | 4998.0 | 4998.0 | 4998.0 | 3083.0 | 4236.0 | 3698.0 |
| `invk2j` (1) | 2914.0 | 4998.0 | 4998.0 | 4998.0 | 3132.0 | 4087.0 | 1768.0 |
| `kmeans` | 117.0 | 898.0 | 268.0 | 460.0 | 138.0 | 101.0 | 181.0 |
| `fft` (1) | 2026.0 | 4998.0 | 4998.0 | 317.0 | 572.0 | 1259.0 | 217.0 |
| `sobel` | 3067.0 | 4998.0 | 4998.0 | 4998.0 | 3119.0 | 4324.0 | 2436.0 |
| `fft` (0) | 751.0 | 658.0 | 4998.0 | 1320.0 | 555.0 | 244.0 | 76.0 |

Table 7: Number of epochs required for neural surrogates produced by each initialization method to achieve the baseline test error on each PARROTBENCHSHORT program.

| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
|---|---|---|---|---|---|---|---|
| `invk2j` (0) | 1.5 | 1.7 | 1.8 | 1.8 | 1.9 | 1.6 | 1.9 |
| `invk2j` (1) | 6.2 | 4.6 | 4.4 | 4.4 | 4.4 | 3.8 | 3.5 |
| `kmeans` | 0.7 | 0.2 | 0.2 | 0.2 | 0.3 | 0.3 | 0.3 |
| `fft` (1) | 0.7 | 0.7 | 0.7 | 0.7 | 0.6 | 0.4 | 0.6 |
| `sobel` | 0.7 | 0.2 | 0.2 | 0.2 | 0.3 | 0.2 | 0.2 |
| `fft` (0) | 0.5 | 0.9 | 1.0 | 1.0 | 1.6 | 0.5 | 0.4 |

Table 8: Initial train losses for neural surrogates produced by each initialization method on PARROTBENCHSHORT programs.

| Program | Random | DTI 1 | DTI 2 | DTI 3 | HBN 1 | HBN 2 | HBN 3 |
|---|---|---|---|---|---|---|---|
| `invk2j` (0) | 1.4 | 1.7 | 1.8 | 1.8 | 1.9 | 1.6 | 1.9 |
| `invk2j` (1) | 6.2 | 4.6 | 4.5 | 4.4 | 4.4 | 3.9 | 3.6 |
| `kmeans` | 0.8 | 0.2 | 0.2 | 0.2 | 0.2 | 0.3 | 0.2 |
| `fft` (1) | 0.7 | 0.7 | 0.7 | 0.7 | 0.6 | 0.4 | 0.6 |
| `sobel` | 0.7 | 0.2 | 0.2 | 0.2 | 0.3 | 0.2 | 0.2 |
| `fft` (0) | 0.5 | 0.9 | 1.0 | 1.0 | 1.7 | 0.5 | 0.4 |

Table 9: Initial test losses for neural surrogates produced by each initialization method on PARROTBENCHSHORT programs.

| Program | Final Baseline Train Loss | Final Baseline Test Loss |
|---|---|---|
| `invk2j` (0) | $1.2 \times 10^{-4} \pm 2.9 \times 10^{-5}$ | $2.1 \times 10^{-4} \pm 5.4 \times 10^{-5}$ |
| `invk2j` (1) | $9.1 \times 10^{-5} \pm 2.7 \times 10^{-5}$ | $7.1 \times 10^{-5} \pm 2.2 \times 10^{-5}$ |
| `kmeans` | $2.1 \times 10^{-6} \pm 8.4 \times 10^{-7}$ | $4.7 \times 10^{-7} \pm 2.7 \times 10^{-8}$ |
| `fft` (1) | $1.0 \times 10^{-5} \pm 7.9 \times 10^{-7}$ | $7.1 \times 10^{-6} \pm 1.9 \times 10^{-8}$ |
| `sobel` | $4.2 \times 10^{-6} \pm 4.7 \times 10^{-7}$ | $2.7 \times 10^{-6} \pm 2.8 \times 10^{-8}$ |
| `fft` (0) | $5.6 \times 10^{-6} \pm 2.2 \times 10^{-6}$ | $2.3 \times 10^{-6} \pm 3.4 \times 10^{-7}$ |

Table 10: Final baseline train and test losses for PARROTBENCHSHORT programs.