Building High-Throughput Neural Architecture Search Workflows via a Decoupled Fitness Prediction Engine

Ariel Keller Rorabaugh, Silvina Caíno-Lores, Travis Johnston, and Michela Taufer, *Senior Member, IEEE*

Abstract—Neural networks (NN) are used in high-performance computing and high-throughput analysis to extract knowledge from datasets. Neural architecture search (NAS) automates NN design by generating, training, and analyzing thousands of NNs. However, NAS requires massive computational power for NN training. To address challenges of efficiency and scalability, we propose *PENGUIN*, a decoupled fitness prediction engine that informs the search without interfering in it. *PENGUIN* uses parametric modeling to predict fitness of NNs. Existing NAS methods and parametric modeling functions can be plugged into *PENGUIN* to build flexible NAS workflows. Through this decoupling and flexible parametric modeling, *PENGUIN* reduces training costs: it predicts the fitness of NNs, enabling NAS to terminate training NNs early. Early termination increases the number of NNs that fixed compute resources can evaluate, thus giving NAS additional opportunity to find better NNs. We assess the effectiveness of our engine on 6,000 NNs across three diverse benchmark datasets and three state of the art NAS implementations using the Summit supercomputer. Augmenting these NAS implementations with *PENGUIN* can increase throughput by a factor of 1.6 to 7.1. Furthermore, walltime tests indicate that *PENGUIN* can reduce training time by a factor of 2.5 to 5.3.

Index Terms—Machine learning, artificial intelligence, performance prediction, neural networks

1 Introduction

Neural networks (NN) are powerful models that are increasingly used in traditional high-performance computing (HPC) scientific simulations and new research areas, such as high-performance artificial intelligence and high-throughput data analytics, to solve problems in physics [1], materials science [2], neuroscience [3], and medical imaging [4] among other domains. Finding suitable NNs is a time-consuming process involving several rounds of hyperparameter selection, training, validation, and manual inspection. Neural architecture search (NAS) automates the process of finding near-optimal models for a given dataset, but it comes at a high training cost involving thousands of NNs on a large number of HPC resources. For instance, conventional NAS algorithms exhibit prohibitive computational demand where training of each NN to convergence is the main bottleneck [5], [6]. A single NAS run on a small dataset of one or two petabytes can require HPC systems with thousands of accelerators [1]. A training run of a large language model such as BERT takes more than 80 hours on 16 TPUv3 AI accelerator hardware [7], and training a visual transformer requires decades of compute time on a TPUv3 [8]. For even larger datasets, scientists must allocate significant time on the largest compute resources available (in the range of tens of thousands of GPU hours [1]) to conduct a single search of NN models. Furthermore, in the early stages of NAS, up to 88% of NNs fail to learn [9], wasting expensive compute resources.

In current literature there are multiple strategies to conduct NAS, including random searches, grid searches, hyper-parameter sweeps, reinforcement learning, evolutionary optimization, gradient-based optimization, and Bayesian optimization [10], [11]. Many NAS implementations rely on built-in "truncated training"—a fixed termination criterion where each NN is trained for a set number of epochs; fixed termination criteria result in wasting expensive HPC resources [12], [13], [14], [15]. Advanced NAS implementations use a fitness prediction strategy and dynamically terminate training each NN once the fitness prediction is calculated [16], [17], [18]. NAS implementations that employ fitness prediction embed their prediction strategy in their search process, resulting in a tightly coupled search and prediction solution. Tight coupling between search and prediction strategies mean that NAS cannot be optimized without heavily interfering in a given NAS implementation. By decoupling search and prediction and creating a flexible fitness prediction method, we make NAS optimizations portable across problems and datasets, and

- *Ariel Keller Rorabaugh, Silvina Caíno-Lores, and Michela Taufer are with the University of Tennessee at Knoxville, Knoxville, TN 37996 USA. E-mail: ariel.keller@gmail.com, {scainolo, mtaufer}@utk.edu.*
- *Travis Johnston is with Striveworks, Austin, TX 78731 USA. E-mail: j.travis.johnston@gmail.com.*

Manuscript received 27 Aug. 2021; revised 15 Nov. 2021; accepted 20 Dec. 2021. Date of publication 7 Jan. 2022; date of current version 23 May 2022.
This work was supported in part by National Science Foundation (NSF) under Grants 1741057, 1740990, 1741040, and 1841758, in part by Joint Directed Research Development (JDRD) program at UTK, and by IBM through a Shared University Research Award. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which supported by the Office of Science of the U.S. Department of Energy under Grant DE-AC05-00OR22725.
(Corresponding author: Michela Taufer.)
Recommended for acceptance by A. J. Peña, M. Si, and J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2022.3140681

TABLE 1
Examples of NAS Implementations That Could Plug Into PENGUIN

NAS	Type	Open source
EvoCNN [12]	evolutionary	✓
MENNDL [19]	evolutionary	×
NAS for image reconstruction [20]	evolutionary	×
psoCNN [13]	particle swarm	✓
Hierarchical representation [14]	evolutionary	×
NSGA-Net [15]	evolutionary	✓
Large-scale evolution [21]	evolutionary	×
Genetic CNN [22]	evolutionary	×
NASNet [23]	reinforcement learning	×
Auto-Keras [24]	Bayesian	✓

TABLE 2
Examples of Parametric Functions for Learning Curve Modeling

Formula	Trend	Formula	Trend
ax^b	increasing	$c-(-ax+b)^{-d}$	increasing
ax^b+c	increasing	$c-(a/\log(x))$	increasing
$a-b^{(c-x)}$	increasing	$c-(c-a)e^{(-bx)}$	increasing
$a\log(x)+c$	increasing	ax^{-b}	decreasing
$ae^{(bx)}+c$	increasing	$ax^{-b}+c$	decreasing
$e^{(a+\frac{b}{x}+c\log(x))}$	increasing	$-a\log(x)+c$	decreasing
$\frac{ab+cx^d}{b+x^d}$	increasing	$ae^{(-bx)}+c$	decreasing
$c-be^{(-ax^d)}$	increasing	$ae^{(-bx)}$	decreasing
$c-e^{(-ax^d+b)}$	increasing	$-ax+b$	decreasing
$c-e^{(x-b)^a}$	increasing		

Functions with an increasing trend model fitness types that increase as the NN learns (e.g., accuracy). Functions with a decreasing trend model fitness types that decrease as the NN learns (e.g., loss).

simultaneously increase NAS implementations' throughput and efficiency. We implement this decoupling approach and its flexible fitness prediction via our engine *PENGUIN*, which plugs into existing NAS implementations, informing each search without interfering in it.

Our solution decouples search and prediction strategies by augmenting NAS implementations with our *PENGUIN* engine to build flexible and composable NAS workflows. *PENGUIN* uses a parametric modeling approach to predict NN fitness early in the training phase; it functions independently of the NN architecture, the particular NAS implementation, and the target dataset. NAS can leverage *PENGUIN*'s fitness predictions to terminate training NNs early, thus increasing throughput and exploring a larger space of candidate models in a scalable way. *PENGUIN* can plug into existing NAS implementations to augment their searches. Any NAS whose search strategy includes training NNs and making decisions that depend on NN fitness can use *PENGUIN* (see Table 1 for examples).

Not only can *PENGUIN* be plugged into a variety of different NASes, but *PENGUIN* itself can use any parametric function for modeling NN fitness. By tailoring the function parameter values to the NN's fitness data, *PENGUIN* constructs a fitness model; it then extrapolates from the model to predict future fitness of the NN. Table 2, adapted from Viering and Loog [25], lists a variety of parametric functions, including several exponential and power functions, that have been successfully used for modeling learning curves in a variety of machine learning problems. Any of these functions can be plugged into *PENGUIN*'s fitness modeling method, as can custom user-defined parametric functions.

We perform a case study in which we evaluate the accuracy of *PENGUIN*'s predictions, and its throughput gain, as compared to three different unaugmented NAS implementations (i.e., MENNDL [19], EvoCNN [12], and NSGA-Net [15]). By augmenting each of these NAS implementations with *PENGUIN* and a parametric function from Table 2, we can save 39% to 86% of training epochs, resulting in an increase in throughput by a factor of 1.6 to 7.1. Furthermore, we measure the actual training walltime speedup that can be attained by augmenting MENNDL with *PENGUIN* and observe a decrease in training time by a factor of 2.5 to 5.3 compared to the unaugmented NAS.

2 Building a Decoupled NAS Workflow

Designing an NN manually is time-consuming and prone to human bias, often resulting in sub-optimal models. NAS exploits supercomputing resources to automate the NN architecture design, alleviating the architecture tuning barrier. NAS selects NN models from a search space, trains them on the target dataset, evaluating their fitness (e.g., validation accuracy or loss), and uses the fitness information to generate new NNs. NAS implementations can be supported by strategies that provide predictions for NN fitness at a given epoch in the future, enabling a NAS to truncate the training process, compare the fitness predictions for different NNs, and steer the search towards NNs that yield better results faster. NAS implementations as found in the literature either do not use a prediction strategy, opting instead for fixed truncated training where NNs are trained for a statically-defined, fixed number of epochs and then compared; or the NAS design involves a tight coupling between the NAS search strategy and a particular prediction strategy, as depicted in Fig. 1. The tight coupling imposes restrictions to the generalization of predictions for different NAS, datasets, and problems, requiring non-trivial, ad-hoc tuning of the NAS implementation.

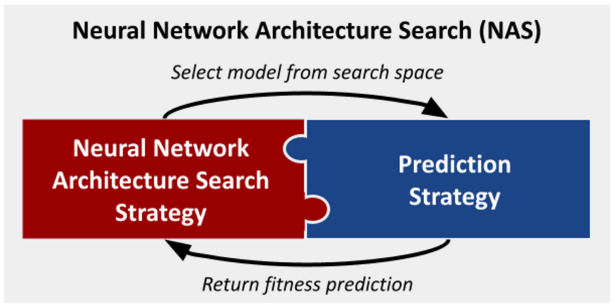

Neural Network Architecture Search (NAS)
Select model from search space
Neural Network Architecture Search Strategy
Prediction Strategy
Return fitness prediction

Fig. 1. Overview of current NAS solutions and their tightly-coupled search and prediction strategies.

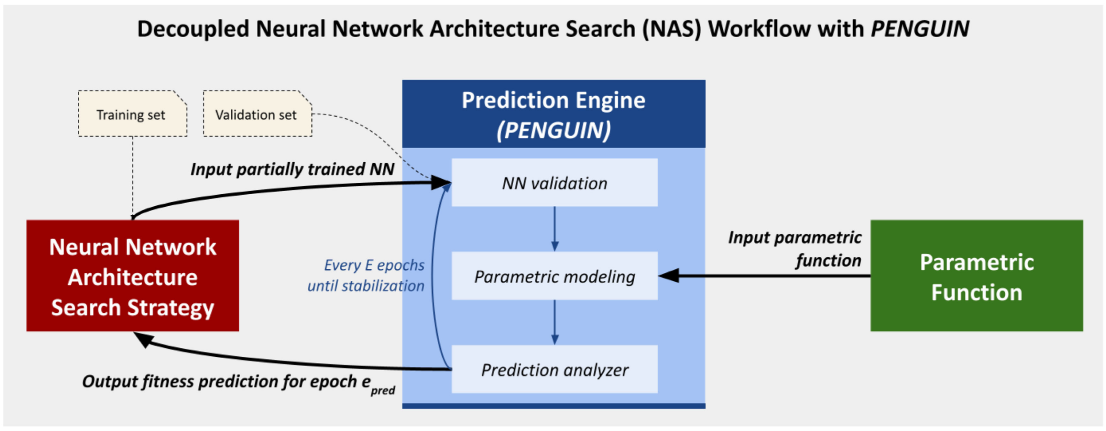

Decoupled Neural Network Architecture Search (NAS) Workflow with PENGUIN
Training set
Validation set
Input partially trained NN
Prediction Engine (PENGUIN)
NN validation
Every E epochs until stabilization
Parametric modeling
Prediction analyzer
Neural Network Architecture Search Strategy
Output fitness prediction for epoch e_pred
Input parametric function
Parametric Function

Fig. 2. Overview of our vision of a decoupled NAS workflow supported by *PENGUIN*. *PENGUIN* constitutes the core prediction engine that, given an input NN provided by a NAS and a parametric function, outputs a fitness prediction that can be leveraged by a NAS.

In contrast to current tightly-coupled NAS solutions, we envision a workflow-oriented approach to the NAS problem, as depicted in Fig. 2. To this end, *PENGUIN* is designed to be completely decoupled from any NAS, and its prediction strategy is not tailored to a specific search strategy or dataset, allowing portability across NAS and scientific domains. *PENGUIN* can be utilized even by NAS implementations that already employ fixed truncated training, allowing them to truncate training as soon as NN fitness predictions are calculated by our engine.

Besides being NAS-agnostic, *PENGUIN* integrates a framework for parametric modeling that adapts to different NN fitness metrics. Parametric modeling of an NN's fitness involves fitting a *parametric function* (i.e., a function with parameters to determine, for example, $\mathcal{F}(x) = ax^b + c$) to the NN's historical fitness data. This fitting process produces a function (e.g., $f(x) = 10x^2 + 5$) that models the fitness curve. The fitness model can then be used to extrapolate a prediction for the fitness value at a given epoch in the future, e_{pred}, by calculating the value of the model at that epoch, $f(e_{pred})$. As shown in Fig. 2, *PENGUIN* allows users to plug in any desired parametric function (some examples in Table 2).

PENGUIN's design and interfaces provide users with the building blocks to flexibly build NAS workflows from tools and methods already available. The decoupled nature of *PENGUIN* constitutes a crucial first step towards scalable high-performance NAS workflows, since this decoupling enables optimization of the resource allocation for each component of the workflow, the component orchestration and placement, and the mechanisms to exchange data efficiently amongst them.

3 The PENGUIN Engine

PENGUIN augments NAS to enable a modular NAS workflow in which the search strategy is fully decoupled from the prediction strategy. *PENGUIN* is designed to plug into any NAS method and run on dedicated resources concurrently with the NAS, informing the search without interfering in it. *PENGUIN* dynamically predicts the fitness each NN could attain during the NN training phase and informs the NAS of each NN's predicted fitness.

The NAS selects NNs from the search space to explore and trains those NNs. When a NAS plugs in *PENGUIN*, periodically during training of each NN, the NAS passes partially trained NNs to *PENGUIN*. *PENGUIN* then iteratively executes a three-step process depicted in Fig. 2. In each iteration of the process, *PENGUIN* first validates the NN and calculates its fitness (*NN validation*); then executes a parametric modeling method in order to construct a model for the NN's fitness curve, and uses the parametric model to extrapolate a prediction for the fitness the NN is expected to attain at a given epoch in the future (*Parametric modeling*); and finally determines whether the prediction has converged, deciding whether to output the NN fitness prediction to the NAS or continue the iterative process (*Prediction analyzer*).

3.1 NN Validation

When classifying a scientific dataset, the dataset is divided into a training set and a validation set. Both *PENGUIN* and the NAS have knowledge of the location of the training and validation sets, and they have capability to access these data. The NAS trains the NNs on the training set and passes partially trained NNs to *PENGUIN* every E epochs throughout training, where an epoch means one cycle through the training set. The value of the parameter E is user-defined. In our tests, we let $E = 0.5$, meaning that each NN is passed to *PENGUIN* every 0.5 epochs throughout its training. At each iteration, *PENGUIN* validates the NN M and calculates its fitness fit_V for the validation set at the current epoch (e). We build an ordered list of these (epoch, fitness) datapoints for the NN across the iterations of *PENGUIN*. In the first iteration, we initialize the list with the starting epoch and fitness values. In subsequent iterations, we append the current (epoch, fitness) datapoint to the ordered list.

3.2 Parametric Modeling

Given a parametric function, we attain a model for NN fitness by determining values for the function parameters using least squares regression. Specifically, we utilize SciPy's *optimize.curve_fit* (Curve Fit) method to find the parameter values that best fit the ordered list of the NN's

TABLE 3
Symbols for Parametric Modeling Algorithm (Algorithm 1)

Symbol	Definition
M	NN identifier
e	Current epoch in which the prediction is calculated
$\mathcal{H}$	Ordered list of $< M, e, fit_V, fit_P >$ tuples, each representing the fitness value fit_V of an NN M plus the fitness prediction fit_P, calculated at a specific epoch e
fit_V	Fitness value at current epoch e from NN *validation* component
$\mathcal{F}$	Parameteric function to use for modeling
h_e	Tuple of fitness metrics for NN M at epoch e
C_{min}	Minimum cardinality of the set $\mathcal{F}$ will be fitted to
r	Epoch rescaling factor
$(b_1, \ldots, b_m)$	Parameter values of the fitness model f at epoch e
f	Fitness model attained by determining parameters for the parametric function $\mathcal{F}$
e_{pred}	Epoch in the future for which *PENGUIN* predicts NN fitness
fit_P	Fitness NN is predicted to attain at epoch e_{pred}

epoch and fitness data. The result is a function that models fitness in terms of epochs. This function is then used to extrapolate a fitness prediction for the NN at a given epoch in the future.

3.2.1 Fitting the Parametric Function to Predict Fitness

Algorithm 1 shows the internal parametric modeling procedure. All inputs and outputs of Algorithm 1 are defined in Table 3. Given an NN M, the procedure takes as input the current training epoch (e); the ordered list of (epoch, fitness) values and predictions from previous iterations ($\mathcal{H}$), also called fitness history; the fitness value at the current epoch e (fit_V); and a parametric function ($\mathcal{F}$), along with optional bounds and initial values for the function's parameters.

The first step in our parametric modeling method is the creation of a new tuple of fitness metrics for the NN M at the specified epoch (h_e). This tuple is inserted in the fitness history $\mathcal{H}$ for M (Line 2) so that it is taken into consideration as a new datapoint for the Curve Fit method. The rest of the steps required to construct the fitness model f only take place if there are enough datapoints to conduct the Curve Fit method. The number of datapoints needed in $\mathcal{H}$ to proceed with curve fitting, denoted C_{min}, is equal to the degrees of freedom in the parametric function (Line 3). This is the same as the number of parameters to determine. For example, the parametric function $\mathcal{F}(x) = a - b^{c-x}$ has three parameters and degrees of freedom; thus it requires $C_{min} = 3$ datapoints.

Recall that the constant number of training epochs per iteration of *PENGUIN* (E) is user-defined (see Section 3.1). We multiply the epoch values in $\mathcal{H}$ by a rescaling factor $r = 1/E$ in order to rescale $\mathcal{H}$ as a new list $\mathcal{H}'$ in which the first epoch value is 1 (Line 4), since this enables assigning general bounds for the function parameters that do not depend on the value of E. For example, if the NN is validated every half epoch, the first epoch value is 0.5, so $\mathcal{H}$ is rescaled by multiplying every value of e in the list by $r = 2$. Then we feed $\mathcal{H}'$ to Sci-Py's *optimize.curve_fit* method to find the function f that best fits the historic fitness data for the NN (Line 6). The function f is used to extrapolate a prediction for the NN's fitness at a given epoch in the future (e_{pred}). The fitness the NN is predicted to attain at epoch e_{pred} is given by $f(e_{pred} \cdot r) = fit_P$, where r is the rescaling factor that was used for the epoch data. We append the value of fit_P to the fitness tuple for the current epoch (h_e) so that it will be reflected in the fitness history (Line 9). As a result of our modeling method, we output the updated history of fitness calculations and predictions ($\mathcal{H}$) for analysis by the prediction analyzer.

Algorithm 1. Procedure for Parametric Modeling Method

Input: Current training epoch (e); history of fitness values and predictions ($\mathcal{H}$); fitness value at epoch e (i.e., fit_V); parametric function to fit ($\mathcal{F}$); and minimum required cardinality (C_{min})
Output: History of validation accuracy and predicted accuracy ($\mathcal{H}$), updated with the latest prediction fit_P

```
1:  procedure Predictor ($e, \mathcal{H}, fit_V, \mathcal{F}$)
2:      $h_e \leftarrow < M, e, fit_V, 0 >$
3:      $\mathcal{H} \leftarrow insert(h_e)$
4:      if $|\mathcal{H}| \geq C_{min}$ then
5:        $\mathcal{H}' \leftarrow rescale\_epochs(\mathcal{H})$
6:        $(b_1, b_2, \ldots, b_m) \leftarrow optimize.curve\_fit(\mathcal{H}', \mathcal{F})$
7:        $f \leftarrow \mathcal{F}(b_1, b_2, \ldots, b_m)$
8:        $fit_P \leftarrow f(e_{pred} \cdot r)$
9:        $\mathcal{H} \leftarrow update\_prediction(\mathcal{H}, h_e, fit_P)$
10:     return $\mathcal{H}$
```

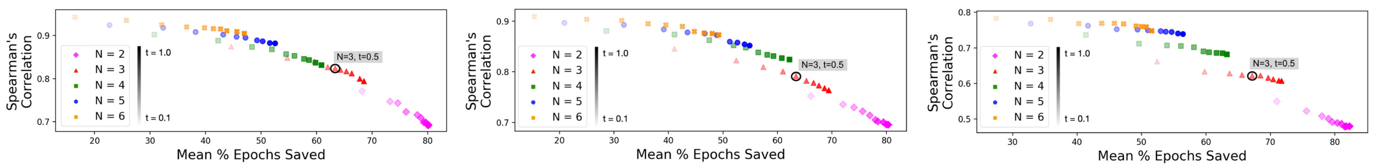
3.3 Prediction Analyzer

Each time the parametric modeling method receives a new fitness measurement for an NN M at epoch e, it generates a new fitness model f from the parametric function $\mathcal{F}$, and it uses the fitness model to compute a new prediction (fit_P) for the fitness the NN will attain at epoch e_{pred}. This fitness prediction fit_P is appended to the history of measured and predicted fitness values $\mathcal{H}$. The fitness model and its associated fitness prediction fit_P vary from one iteration to the next because new fitness measurements are calculated and appended to $\mathcal{H}$ at each iteration, hence the parametric modeling method has one more datapoint to use to construct the fitness model with each iteration. The goal of the prediction analyzer is to determine whether the iterative fitness predictions for M have converged to a stable value, which we denote by FIT_P. If so, the analyzer outputs the final fitness prediction FIT_P, and *PENGUIN*'s iterative process terminates. If not, the next iteration begins, and the NN resumes training.

3.3.1 Parameterizing the Analysis

To guide the behavior of the prediction analyzer and its convergence criteria, we rely on four parameters:

- N, the number of most recent fitness predictions to consider
- t, the threshold describing how much variability is tolerated to establish convergence and report a final fitness prediction



(a) CIFAR-100 (b) Fashion MNIST (c) SVHN

Fig. 3. Trade-off for different choices of N and t, indicating the values we use ($N = 3$ and $t = 0.5$) for the three benchmark datasets used in this paper (i.e., CIFAR-100, Fashion MNIST, and SVHN).

- E, the number of epochs per iteration
- e_{pred}, the epoch in the future for which *PENGUIN* predicts NN fitness; also, the maximum number of epochs an NN will train.

These engine parameters, summarized in Table 4, constitute part of the input arguments for the prediction analyzer procedure and can be adjusted in *PENGUIN* to tune the overall iterative process. Fig. 3 shows the trade-off for different choices of the parameters N and t on sample subsets of each of the benchmark datasets introduced in Section 4. This figure motivates our choice of $N = 3$ and $t = 0.5$ because these points are Pareto-optimal or near-Pareto-optimal on all the trade-off curves.

The parameter E affects the ability of the parametric modeling to successfully approximate the NN accuracy curve. E impacts the granularity of the set of datapoints used for fitting the parametric function; the smaller E is, the more datapoints we will have. Therefore, E must meet a trade-off between two constraints: it must be as small as possible to maximize the number of datapoints to use for constructing the parametric model, and it must be large enough to avoid extrapolating local behavior to the whole curve. This motivates our selection of $E = 0.5$.

The parameter e_{pred} corresponds to the future epoch for which *PENGUIN* predicts NN fitness. One can set e_{pred} to any arbitrary value to match the epoch at which the user would like to assess NN fitness. We set the parameter to match the epoch of truncated training of the NAS that *PENGUIN* is augmenting. By default, each of the NAS methods evaluated in this paper (i.e., MENNDL [19], EvoCNN [12], and NSGA-Net [15]) incorporate truncated training, terminating training at 20, 10, and 25 epochs respectively. If epoch e_{pred} is reached during *PENGUIN*'s iterative process, the process terminates by outputting the actual fitness corresponding to that epoch.

TABLE 4
Symbols for Prediction Analyzer Algorithm (Algorithm 2)

Symbol	Definition
e	Current epoch in which the predictions are analyzed
$\mathcal{H}$	Ordered list of $< M, e, fit_V, fit_P >$ tuples, each representing the validation fitness of an NN M plus the fitness prediction fit_P, calculated at a specific epoch e
N	Number of most recent fitness predictions to consider in the analysis
E	Number of epochs per iteration
e_{pred}	Epoch in the future for which *PENGUIN* predicts NN fitness
t	Maximum fitness prediction variability allowed for convergence
FIT_P	Most recent fitness prediction for the NN M on convergence

3.3.2 Analyzing the Convergence of the Predictions

Algorithm 2 shows the internal procedure in our prediction analyzer. Given an NN M, the procedure takes as input the current training epoch (e), the history of fitness values and predictions ($\mathcal{H}$), and the four engine configuration parameters described above. We first check if we have reached the minimum number of epochs to assess convergence (Line 2). If the answer is no, the number of tuples in $\mathcal{H}$ is not sufficient to analyze whether the predictions are stable; we exit the validation and continue with the next iteration. If the answer is yes, we check for convergence of the iterative fitness predictions. If convergence is achieved prior to reaching epoch e_{pred} in training, then the final output fitness prediction (FIT_P) corresponds to the fitness predicted by the tailored function in the current epoch (fit_P) (Lines 5, 8). We define three conditions for reaching convergence:

- *Condition 1:* NN training must not have already reached the epoch, e_{pred}, for which to predict fitness.
- *Condition 2:* The most recent fitness prediction (FIT_P) must be within the range of valid fitness values. For example, if fitness is measured by validation accuracy, the accuracy prediction must be less than or equal to 100%. Otherwise, the prediction is not valid to establish convergence (Line 6). This condition ensures that we have a realistic value for FIT_P.
- *Condition 3:* The N most recent fitness predictions are all within the threshold t of their mean (Line 7). In other words, for each of the N most recent predictions p_i, we checks if $mean - t \leq p_i \leq mean + t$.

Algorithm 2. Procedure for Prediction Analyzer

Input: Current training epoch (e); history of fitness values and predictions ($\mathcal{H}$); configuration parameters
Output: Convergence status (*converged*)
Output: Fitness prediction (if convergence is acheived, predicted fitness (FIT_P); if e_{pred} is reached, actual fitness calculated at epoch e_{pred} (fit_V))

```
1: procedure Analyzer $e, \mathcal{H}, N, E, e_{pred}, t$
2:     $converged \leftarrow false$
3:     if $e \leq N \cdot E$ then exit
4:     if $e = e_{pred}$ then return $fit_V$
5:     $FIT_P \leftarrow fit_P$ from $\mathcal{H}$ in epoch $e$
6:     if $FIT_P$ is a valid fitness value then exit
7:     if last $N$ $fit_V \in \mathcal{H}$ are within $t$ then $converged \leftarrow true$
8:     if $converged$ is $true$ then return $FIT_P$
9:     else exit
```

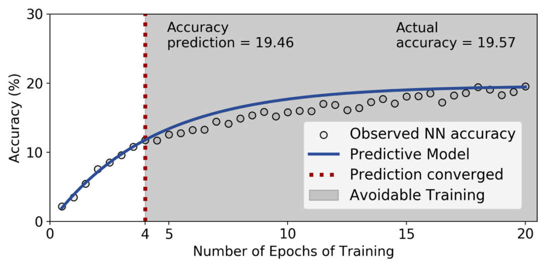


Fig. 4. Example of the fitness (accuracy) prediction and tailored function given by *PENGUIN* for an NN trained on CIFAR-100.

There are three possible outcomes based on these conditions. First, if all three conditions are true, it means the prediction has converged. In this case, the prediction analyzer returns the final prediction FIT_P with convergence status *true* (Line 8). Second, if the first condition is not satisfied, then the NN has already trained for e_{pred} epochs. In this case, there is no need for a fitness prediction from *PENGUIN*. Instead, we return as our final fitness prediction the actual fitness value at epoch e_{pred}, fit_V, with convergence status *false*, indicating that *PENGUIN*'s prediction did not converge prior to training for e_{pred} epochs (Line 4). Last, if the first condition is satisfied , but either of the second or third conditions is not met, then the prediction analyzer procedure ends without returning a fitness prediction and with convergence status *false* (Line 9).

Based on the output of the prediction analyzer, we can establish the termination criteria for *PENGUIN*. If a fitness prediction is returned, *PENGUIN* terminates and outputs this value to the NAS. This can occur in two scenarios: either we reached convergence and the output is the fitness prediction FIT_P, or we reached e_{pred} and the output is fit_V—the actual fitness attained at epoch e_{pred}. We can distinguish these cases by inspecting the value of the convergence status. If no value is returned, then *PENGUIN* begins the next iteration and resumes training the NN. As an example, Fig. 4 depicts the end result of this process for one of the NNs we trained on the CIFAR-100 benchmark dataset [26]. The x-axis indicates the number of training epochs, and the y-axis indicates the fitness of the NN (accuracy in this example). The dotted red line indicates the point at which the prediction converges; this is when our engine outputs the fitness prediction and terminates the iterative training process. In this example, our engine terminates the iterative process and outputs the fitness prediction after 4 epochs. The fitness model from the final iteration is graphed, along with the fitness datapoints (validation accuracy in this example) for the NN across e_{pred} epochs. Note that only the fitness values from the first 4 epochs were used to construct the fitness model.

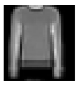
4 Case Study

We present a case study in which we use *PENGUIN* to answer two questions from the perspective of a NAS plugging in *PENGUIN*: (i) "How accurate are the predictions?" and (ii) "What savings are gained?"

4.1 Experimental Setup

We apply *PENGUIN* to a set of approximately 6,000 NNs—2,000 each trained on one of three widely used benchmark datasets: CIFAR-100, Fashion MNIST, and SVHN—using a parametric function of the form $f(x) = a - b^{(c-x)}$. Using the Summit supercomputer at the Oak Ridge National Laboratory, we train our set of diverse NNs across these benchmark datasets.

TABLE 5
Characterization of C-100, F-MNIST, and SVHN

	C-100	F-MNIST	SVHN
Number of samples	60,000	70,000	99,289†
Training set	50,000 (83%)	60,000 (86%)	73,257 (75%)
Validation set	10,000 (17%)	10,000 (14%)	26,032 (26%)
Number of classes	100	10	10
Samples per class	600	7,000	6,500 to 19,000
Balanced	Yes	Yes	No
Number of channels	3	1	3
Sample size (pixels)	32×32	28×28	32×32
Example image			

†*SVHN also contains 531,131 samples that can be used as additional, easier training data, which we did not use in our evaluation.*

4.1.1 Benchmark Dataset Characterization

To the best of our knowledge, there is no systematic characterization of datasets used in evaluating ML methods in HPC. This makes the assessment of the generality and applicability of a method in the area of NN research a case-driven discussion [27], [28], [29], [30], [31], [32] that is also found in works tackling the problem of evolutionary NN design [33], [34]. We contribute to the discussion by characterizing three widely used datasets (i.e., CIFAR-100, Fashion MNIST, and SVHN) in terms of the diversity of their attributes. Based on this diversity we use these benchmark datasets to evaluate our engine's capability to predict the accuracy of NNs. We identify five key attributes that demonstrate the diverse nature of these benchmark datasets: the number of samples, the number of classes, the number of image color channels, the sample size [35], and the internal balancing of the sample distribution per class, which is known to affect the accuracy of NNs [36]. Table 5 summarises these attributes and shows a sample from each benchmark dataset.

CIFAR-100 (C-100): This dataset was introduced in 2009 [26] as a subset of the *80 Million Tiny Images* dataset [37], aimed towards improving tasks of unsupervised training of deep generative models. It is still one of the most popular benchmark datasets in the field of computer vision due to the manageable size of the dataset, the resolution of its images, and its challenges for NN models [38], [39], [40].

Fashion MNIST (F-MNIST): This benchmark dataset [41] serves as a replacement for the original MNIST dataset comprising ten classes of handwritten digits [42]. It shares the same image and dataset size, data format, and structure of training and testing splits with MNIST, making it a popular benchmark dataset for NN models targeting computer vision problems [43], [44], [45].

TABLE 6
Parameters and Value Intervals for NN Generation

Parameter	Values*
Number of convolutional layers	$[1, 10]$
Kernel	$[1, dimension\ of\ input]$
Stride	$[1, kernel]$
Padding	$[0, 5]$
Number of filters	$[0, 400]$
Number of non-linear layers	$[10, 30]$
Type	ReLU, dropout, pooling
Dropout rate for dropout layers	$[0.1, 0.7]$
Pool kernel for pooling layers	$[1, dimension\ of\ input]$
Stride for pooling layers	$[1, pool\ kernel]$
Padding for pooling layers	$[0, \lfloor pool\ kernel/2 \rfloor]$
Number of fully connected layers	$[1, 5]$
Number of filters	$[0, 400]$
Dropout rate for dropout layers†	$[0.1, 0.7]$
Learning rate	$10^p, p \in [-6.0, 0.0]$
Momentum†	$10^p, p \in [-6.0, 0.0]$
Dampening†	$10^p, p \in [-6.0, 0.0]$
Weight decay†	$10^p, p \in [-6.0, 0.0]$
Training batch size	$[25, 250]$

*Values are uniformly randomized in the specified intervals.
†These parameters are only taken into account if they are randomized to be true.

TABLE 7
Breakdown of Models in the Full NN Set

	C-100	F-MNIST	SVHN
Total number of NNs	1980	1910	1956
Never-learns	1673	1340	1684
Learns - anomalies	79	126	149
Learns - not anomalies	228	444	123
Anomalies as a percent of learning NNs	26%	22%	55%

SVHN: Like MNIST, SVHN [46] contains digits, but in this case they are obtained from real-world house numbers, and thus contain color information, various natural backgrounds, overlapping digits, and other distracting features. These characteristics make SVHN a more difficult benchmark dataset than MNIST, and as a result, SVHN is a very popular benchmark dataset for NNs [47].

4.1.2 NN Generation

In the first stage of the search, many NAS implementations generate the initial set of NNs randomly. We generate each NN with uniformly randomized parameter values from the intervals defined in Table 6 Our set of NNs represents the NNs such a NAS would select from the search space to explore. Each NN begins with a number of convolutional layers. We randomize kernel, stride, and padding values for each convolutional layer, as well as the number of filters. This is followed by randomized non-linear layers of the pooling, ReLU (rectified linear unit activation function), or dropout types; we add fully connected layers at the end of the network. For each NN, we generate a random boolean that determines whether or not to include any dropout layers between the fully connected layers. If the boolean is $true$, then we generate a value for the dropout rate for these layers. After each fully connected layer except the final one, we randomly decide whether or not to add a dropout layer. Finally, we randomize the learning rate, momentum, dampening, and weight decay. In addition, we randomize the batch size to use for training. We choose an integer uniformly between 25 and 250 and truncate the training benchmark dataset to be divisible by batch size. Because we validate every half epoch, we need number of samples in the truncated dataset to be divisible by twice the batch size. If this divisibility condition is not met, we re-randomize batch size until it is. This process assures the diversity of our NN sets. As a result, we generate and train a set of approximately 2,000 NNs for each one of the three benchmark datasets (i.e., C-100, F-MNIST, and SVHN). We produce a publicly available NN dataset containing architecture descriptions of these random NNs and metadata describing each NN's accuracy and loss at each half epoch of training [48], [49].

4.1.3 NN Classification

When searching the space of possible NNs for a given dataset, we deal with a wide range of parameter values that define those NNs and that affect their capability to learn the benchmark dataset. We classify our randomly generated NNs based on their learning capability. A large number of randomly generated NNs may be unable to learn. We call such NNs *never-learns*. Generally, never-learns are able to classify a single class from the benchmark dataset on which they are trained, and they never learn to classify more than one class. As a result, the accuracy value of never-learns depends on the number of classes in the benchmark dataset and whether or not those classes are balanced. We observe that never-learns on C-100 have final accuracy values of about 1% because C-100 contains 100 balanced classes; never-learns on F-MNIST have final accuracy values of about 10% because F-MNIST contains 10 balanced classes; and never-learns on SVHN have varying final accuracy values up to about 20% because SVHN contains unbalanced classes, and the largest class contains about $1/5$ of the data (in both the training and testing sets). Such behavior is consistent with early stages of many NAS methods, when a NAS generates a large set of random NNs for exploration, and many of these NNs have very low accuracy.

We also observe that many NNs do not learn for several epochs but eventually do begin to learn, in some cases attaining quite high final accuracy. Because the accuracy of these NNs does not increase for many epochs, they can be incorrectly predicted to never learn. We call these NNs that are incorrectly predicted to never learn because of a long initial learning delay *anomalies*. Table 7 breaks down the generated random NNs into these categories.

As noted earlier, the full set of trained NNs with many never-learns reflects the type of networks expected in early stages of NAS. As a NAS progresses, we expect to see fewer and fewer never-learns and more better performing NNs. To simulate later generations of NNs in a NAS, we create a subset of NNs from this full NN set, keeping all learning NNs and removing all the never-learns. Using Summit, we evaluate our engine on both of these sets (i.e., with and without never-learns) in order to see results that reflect how our engine would perform in different stages of a NAS (e.g.,

TABLE 8
Sensitivity of the Predicted Best x NNs

	Full NN set			Learning NNs only		
x	C-100	F-MNIST	SVHN	C-100	F-MNIST	SVHN
50	0.70	0.56	0.70	0.70	0.58	0.70
100	0.85	0.77	0.77	0.86	0.77	0.77
150	0.87	0.81	0.73	0.87	0.81	0.73

TABLE 9
Specificity of the predicted best x NNs

	Full NN set			Learning NNs only		
x	C-100	F-MNIST	SVHN	C-100	F-MNIST	SVHN
50	0.99	0.99	0.99	0.94	0.96	0.93
100	0.99	0.99	0.99	0.93	0.95	0.87
150	0.99	0.98	0.98	0.88	0.93	0.66

at the very beginning of the search and in subsequent iterations).

4.1.4 Parameterized Function

In our case study, NN fitness is measured by validation accuracy. We selected the parameterized function $f(x) = a - b^{(c-x)}$ to use in *PENGUIN*'s parametric modeling method. Our empirical observation indicates that during the initial phase of training, NN accuracy curves tend to be concave down and increasing, with the accuracy values approaching a horizontal asymptote; the parametric function $f(x) = a - b^{(c-x)}, b \geq 1$ shares these properties. We choose this function because a preliminary comparison with other functions showed that this specific function provides good results across the spectrum of datasets we are considering, when fitness is measured by accuracy. Future work will study function selection for more datasets and types of fitness measurements (e.g., loss).

4.2 Accuracy

When addressing the first question of our case study, *"How accurate are the predictions?"*, we must recall that a NAS process involves generating NN models, evaluating their accuracy, selecting the best models at each step, and using these best models to inform the next generation of models (Section 1).

PENGUIN reports fitness predictions for all the generated NNs (Section 2), and a NAS can use these predictions to select the best models. In other words, it is important for *PENGUIN* to accurately identify the best NNs from among the set generated by a NAS, as these are used to create the next generation of NNs. Thus, in evaluating the accuracy of our predictions, we compare the ground truth best x NNs from all our generated NNs with the predicted best x NNs as identified by *PENGUIN* for $x = 50, 100,$ and 150.

We use *PENGUIN*'s fitness predictions for all the NNs to create a set of our engine's predicted best x NNs. Recall that *PENGUIN* predicts the fitness each NN is expected to attain in the future, at epoch e_{pred} (Section 3.3.1). Thus, the ground truth to compare with *PENGUIN*'s predictions is the actual fitness of each NN at epoch e_{pred}. In our tests, in order to identify the ground truth best x NNs, we allow all the NNs to continue training and validating for e_{pred} epochs, even after *PENGUIN* has reported fitness predictions for them. The NNs that achieve the best validation fitness at epoch e_{pred} of training are the ground truth best NNs. These are also the NNs that the NAS *PENGUIN* is augmenting would select as the best.

In this case study, we measure the accuracy of *PENGUIN*'s predictions for the selected parametric function using the *sensitivity* and *specificity* metrics, which measure respectively the true positive rate and true negative rate of the predicted best x NNs.

4.2.1 Sensitivity and Specificity

We measure the sensitivity (i.e., true positive rate) and the specificity (i.e., true negative rate) of *PENGUIN*'s predicted best x NNs for different values of x, for both the full NN set and the subset of learning NNs only. Table 8 shows the sensitivity values of *PENGUIN*'s predicted best x NNs; the sensitivity values are similar for both NN sets. Across all our experiments, for x values ranging from 50 to 150, the sensitivity of *PENGUIN*'s predictions ranges between 0.56 and 0.87. In fact, in almost all of our experiments, the sensitivity values are greater than or equal to 0.70. The exception to this is F-MNIST when $x = 50$, which has sensitivity values of 0.56 and 0.58 for the full NN set and the set of learning NNs only, respectively.

Table 9 shows the specificity of *PENGUIN*'s predicted best x NNs for the full NN set and the subset of learning NNs only. The specificity values show more variance for these two NN sets because their sizes are different. Across all our experiments on the full NN set, for x values ranging from 50 to 150, the specificity of *PENGUIN*'s predictions ranges between 0.98 and 0.99. For the subset of learning NNs only, the specificity ranges between 0.66 and 0.97.

Fig. 5 zooms into the second row of Table 8 and the second row of Table 9 for the full NN set depicting the accuracy predictions and actual accuracy values of all NN models for each benchmark dataset. Specifically, the figure indicates the true positives, true negatives, false positives, and false negatives resulting from our prediction of the best 100 models. *PENGUIN*'s output accuracy predictions are on the y-axis; actual accuracy is on the x-axis. *PENGUIN*'s accuracy predictions are predicted using our parametric modeling. Note that *PENGUIN*'s output prediction is the accuracy each NN is expected to attain for epoch e_{pred}. The actual accuracy is given by the observed accuracy values of each NN at the epoch e_{pred}. In this case, we set $e_{pred} = 20$ because the value is representative of the truncated training used by cutting-edge NAS implementations like MENNDL. Throughout our case study, actual and predicted accuracy values are always compared for the same epoch; should $e_{pred} = 10$, then PENGUIN would predict the accuracy each NN is expected to attain for epoch 10, and we would compare these predictions with the actual values of the NNs at epoch 10.

In Fig. 5, the purple triangles depict true positives (i.e., NNs that are correctly predicted to be in the top 100). The grey plus signs depict true negatives (i.e., NNs that are

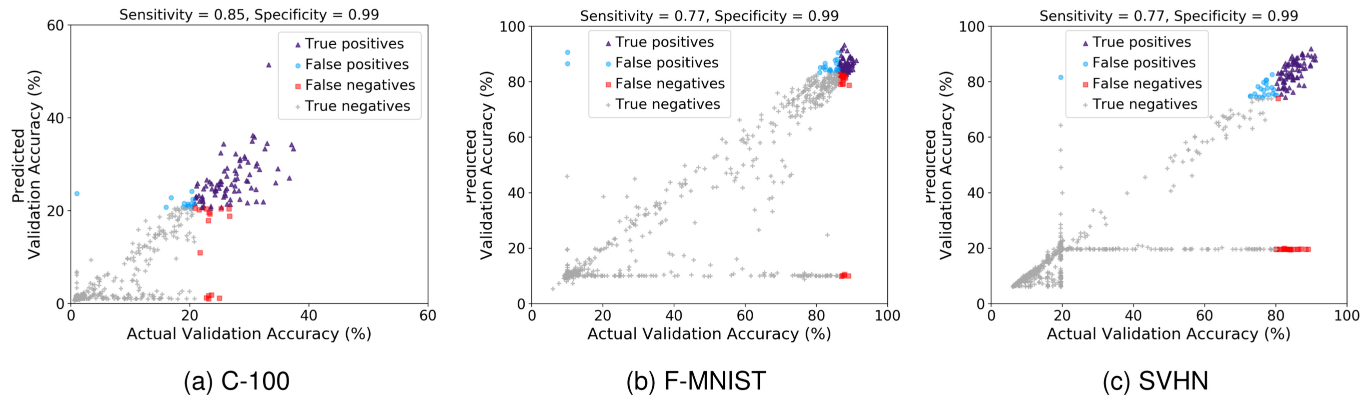


Fig. 5. Predicted accuracy and actual accuracy for all NNs, indicating true positives, true negatives, false positives, and false negatives for the predicted best 100 models.

correctly predicted not to be in the top 100). The blue circles depict false positives (i.e., NNs that are incorrectly predicted to be in the top 100). The red squares depict false negatives (i.e., NNs that are incorrectly predicted not to be in the top 100). Because the x-axis indicates actual accuracy, the most accurate NNs are the furthest to the right of the figure, and the least accurate NNs are the furthest to the left. We observe that the most accurate NNs from the set of ground truth best NNs are true positives, and the ground truth best NNs that are false negatives tend to be the least accurate ones, with the exception of a few of the anomalies. In the lower left corner of the figure, we see NNs with very low actual accuracy and predicted accuracy; these are the never-learns (i.e., NNs that never learn to classify more than a single class). Parallel to the x-axis we see the anomalies; as discussed in Section 4.1.2, these are NNs that have varying levels of actual accuracy but are predicted to have very low accuracy. They are NNs whose accuracy does not begin to increase for many epochs, and therefore they are incorrectly predicted to never learn. Part of our ongoing work involves studying ways to distinguish between NNs that never learn and these anomalies.

4.3 Gain

We answer the second question of our case study, “What savings are gained?”, by using three metrics—number of training epochs saved, throughput gain, and walltime speedup—for each of the three benchmark datasets. We calculate our gain using these metrics as compared to the training required to assess NN accuracy by three different state-of-the-art NAS methods (i.e., MENNDL [19], EvoCNN [12], and NSGA-Net [15]). MENNDL, EvoCNN, and NSGA-Net are NAS implementations that all use built-in truncated training for early termination of NN training. MENNDL terminates training at 20 epochs, EvoCNN terminates training always at 10 epochs, and NSGA-Net terminates training always at 25 epochs. Additionally, MENNDL incorporates its own dynamic built-in early termination criterion on top of its truncated training, terminating training earlier if the NN’s minimum training loss has not decreased for at least 10 epochs. We use the built-in training termination methods of these NAS implementations as a baseline to compare with the same NAS implementations when augmented by *PENGUIN*.

4.3.1 Training Epochs Saved

We compare the epoch at which MENNDL, EvoCNN, and NSGA-Net would terminate NN training if they were augmented by *PENGUIN* versus the epoch at which each of these NAS implementations would terminate training without *PENGUIN*, using their built-in termination methods. Any of the three NAS implementations, when augmented by our engine, terminates training each NN when either *PENGUIN*’s fitness prediction converges (as in Fig. 4), avoiding the full training, or in the worst scenario at e_{pred} epochs, where we set e_{pred} equal to the number of epochs each NAS’s existing built-in truncated training ends (i.e., 20, 25, and 10 for MENNDL, EvoCNN, and NSGA-Net respectively).

Table 10 lists the mean epoch when *PENGUIN* would terminate training for each of our NN sets across the three benchmark datasets with MENNDL. For the full NN set (including never-learns), *PENGUIN* terminates training at about epoch 4 on average. For the set of learning NNs only, *PENGUIN* terminates training at about epoch 7.5 on average. Fig. 6 depicts the distribution of NNs according to the percentage of training epochs saved by augmenting MENNDL with *PENGUIN*, both for the full NN set and for the subset of learning NNs only. Percentage of epochs saved is denoted on the x-axis and percent of total NN samples on the y-axis. The height of a rectangle denotes the percent of total samples that save the indicated portion of training epochs: taller rectangles to the right indicate that more samples save a larger percentage of epochs (i.e., there is a larger region of avoidable training for these samples). The dashed line indicates the mean percentage of training epochs saved. We observe that for the full NN set (including never-learns), across all three benchmark datasets, 60% to 75% of models save more than 80% of the training epochs that would be required by MENDDL’s early termination criterion. In fact,

TABLE 10
Mean Epoch When NN Training Would be Terminated for *PENGUIN* Augmenting MENNDL

	C-100	F-MNIST	SVHN
Full NN set	3.5	4	4.5
Learning NNs only	7.5	7.5	7

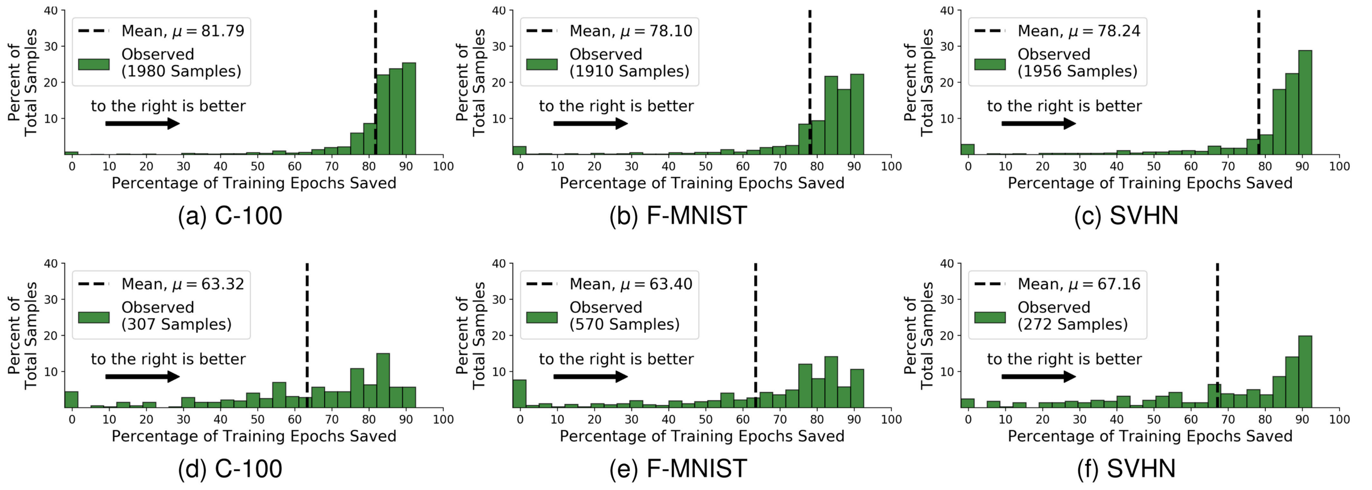


Fig. 6. Percentage of training epochs that would be saved by augmenting MENNDL with *PENGUIN*. (a) - (c): Full NN set, representing early stages of NAS. (d) - (f): Learning NNs only, representing later stages of NAS.

for all three benchmark datasets, we cut the mean number of training epochs needed to evaluate the networks by more than 75%. We demonstrate respective mean savings of 82%, 78%, and 78% of the training epochs on C-100, F-MNIST, and SVHN (indicated by the dashed lines in Figs. 6a, 6b, and 6c), as compared to MENNDL unaugmented by *PENGUIN*. For the set of learning NNs only, across all three benchmark datasets, more than 40% of models save over 80% of the training epochs that would be required by MENDDL's early termination criterion. Overall, for all three benchmark datasets, C-100, F-MNIST, and SVHN, we cut the mean number of training epochs needed to evaluate the networks by 63%-67%.

Similarly, we calculate the percentage of training epochs saved by augmenting EvoCNN and NSGA-Net with *PENGUIN*, for the same NN models and benchmark datasets as above, and for both our full NN set and the set of learning NNs only. For the full NN set (including never-learns), across all three benchmark datasets, augmenting EvoCNN with *PENGUIN* would save 64% to 70% of the mean training epochs needed, and augmenting NSGA-Net with *PENGUIN* would save 82% to 86% of the mean training epochs needed. For the set of learning NNs only, across all three benchmark datasets, augmenting EvoCNN with *PENGUIN* would save 39% to 40% of the mean training epochs needed, and augmenting NSGA-Net with *PENGUIN* would save 70% to 74% of the mean training epochs needed. These outcomes are not presented in a figure because of space constraints.

Across both NN sets (i.e., the full NN set and the subset of learning NNs only), all three benchmark datasets, and all three NAS implementations, we measure average savings of 39% or more as compared to the training epochs needed without the use of *PENGUIN*. We expect to observe larger savings in the earliest stages of NAS because we encounter many never-learns and *PENGUIN*'s predictions stabilize quickly. This is reflected in the average savings ranging between 64% and 86% across all benchmark datasets and NAS implementations on the full NN set. In the set of learning NNs only, which better represents later stages of NAS, we see average savings ranging between 39% to 74%.

4.3.2 Throughput Gain

The ability of *PENGUIN* to predict NN model fitness early in the training process has immediate implications in NAS. As summarized in Section 4.3.1, we can anticipate reducing required training epochs by about 64%-86% in early stages of NAS and by about 39%-74% in later stages of NAS, depending on the benchmark dataset and the NAS method. The reduction in training epochs for individual NNs increases the number of networks that can be explored and evaluated using the same amount of wall time and compute resources. Exploring more network structures gives a NAS additional opportunity to find better NN models. Alternatively, if a NAS has a fixed problem size (i.e., a set number of models to explore), then one could evaluate those models using fewer computational resources.

For the full NN set on C-100 in Fig. 6a, the mean savings is about 80% or $\frac{4}{5}$; this means *PENGUIN* evaluates the models in $\frac{1}{5}$ the computation needed by MENNDL. This corresponds to a throughput gain of about $5\times$, allowing $5\times$ the number of networks to be explored with the same computational resources. In Fig. 6d, for the learning NNs on C-100, the mean savings is about 63%; this means *PENGUIN* evaluates the models in $\frac{37}{100}$ times the computation needed by MENNDL. This corresponds to a throughput gain of about $\frac{100}{37}$, or $2.7\times$, allowing $2.7\times$ the number of networks to be explored with the same computational resources. Thus, for C-100, augmenting MENNDL with *PENGUIN* yields a throughput gain of between $2.7\times$ and $5\times$. Similarly, augmenting MENNDL with *PENGUIN* yields a throughput gain of between $2.7\times$ and $4\times$ on F-MNIST and SVHN.

The same manner of computation demonstrates a throughput gain of between $1.6\times$ and $3.3\times$ for EvoCNN, across all benchmark datasets and both NN datasets, and a throughput gain of between $3.3\times$ and $7.1\times$ for NSGA-Net, across all benchmark datasets and both NN datasets.

4.3.3 Walltime Speedup

MENNDL's built-in training termination leverages truncated training in conjunction with a dynamic early

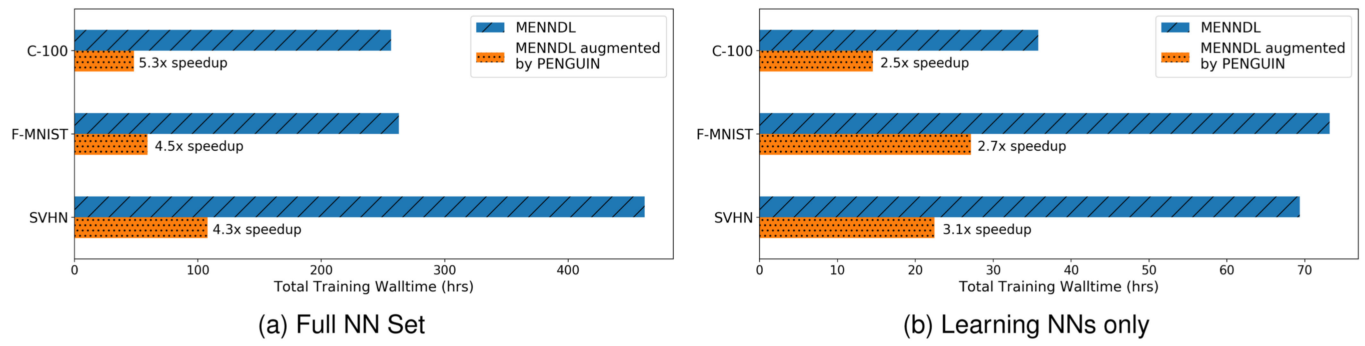


Fig. 7. Training walltime for MENNDL versus training walltime for MENNDL augmented by PENGUIN.

termination criterion. It ends training at 20 epochs, or earlier if loss is stable over the past 10 epochs, making this NAS implementation one of the most effective on HPC systems [1], [4]. We put *PENGUIN* to the test by comparing the actual walltime of training our 6,000 NNs using MENNDL with two termination scenarios: (i) with MENNDL's built-in training termination; and (ii) with *PENGUIN* augmenting the termination decision. Fig. 7 shows the training walltime in hours for the full NN set (Fig. 7a) and the set of learning NNs only (Fig. 7b) for the two scenarios listed above and the three benchmark datasets. When augmented by our engine, MENNDL gains speedups of up to 5.3 times for the full NN set and up to 3.1 times for the learning NN set compared to its dynamic built-in termination. As noted in Section 4.3.1, we expect to observe larger savings in the earliest stages of NAS, represented by the full NN set, and smaller savings in later stages of NAS, represented by the set of learning NNs only. This is consistent with our observations in Fig. 7—we see speedups of $4.3\times$ to $5.3\times$ on the full NN set and speedups of $2.5\times$ to $3.1\times$ on the set of learning NNs only.

Table 11 summarizes the gain from augmenting MENNDL with *PENGUIN*. The first column gives the percentage of mean epochs saved. The second column shows the theoretical speedup (i.e., throughput gain), we calculated based on the percentage of epochs saved—note how the estimates are conservative because of rounding down. The third column shows the actual wallclock time speedup measured—in all cases the measured speedup is very close to the theoretical speedup. To sum up, for the full NN set, we observe theoretical speedup of $5\times$ on C-100, $4\times$ on F-MNIST, $4\times$ on SVHN, and measured speedup of $5.3\times$ on C-100, $4.5\times$ on F-MNIST, and $4.3\times$ on SVHN. For the set of learning NNs only, we observe theoretical speedup of $2.7\times$ on C-100, F-MNIST, and SVHN, and measured speedup of $2.5\times$ on C-100, $2.7\times$ on F-MNIST, and $3.1\times$ on SVHN. Our measurements empirically confirm the effectiveness of *PENGUIN* in drastically cutting the NN training time and consequently increasing the training throughput on HPC systems.

4.4 Applications of *PENGUIN*

The decoupling of search and prediction, and the flexible fitness prediction method in *PENGUIN*, allow us to apply this work in two directions.

First, we can plug different parametric functions into *PENGUIN* to handle other types of fitness measurements beyond accuracy. For example, a function based on exponential decay rather than exponential growth could be used when fitness is measured by loss.

Second, we can tune *PENGUIN*'s parametric modeling for different problems with different datasets and fitness measurements. There is no perfect alignment of NAS and parametric function that works for every dataset and fitness measurement. *PENGUIN* enables a user to test different combinations of NAS methods and parametric functions at small scale in order to find the best pairing of NAS and parametric function for a given dataset and fitness measurement. The search can then be scaled up using the best NAS and parametric function for a given problem.

Furthermore, by extending *PENGUIN*, we can capture edge cases (i.e., models that never learn, models with anomalies, models with false negatives, and models that never converge). Fig. 8 presents NN accuracy curves for four edge cases, together with the predictions given by *PENGUIN*. Specifically, Fig. 8a depicts a never-learn on C-100. In this example, there is noise in the data, causing the accuracy to alternate between values slightly larger than 1 and values slightly smaller than 1, but at no point does the accuracy

TABLE 11
Gain from Augmenting MENNDL With PENGUIN

	Full NN Set			Learning NNs Only		
	Epochs Saved	Theoretical Speedup	Measured Speedup	Epochs Saved	Theoretical Speedup	Measured Speedup
C-100	82%	5x	5.3x	63%	2.7x	2.5x
F-MNIST	78%	4x	4.5x	63%	2.7x	2.7x
SVHN	78%	4x	4.3x	67%	2.7x	3.1x

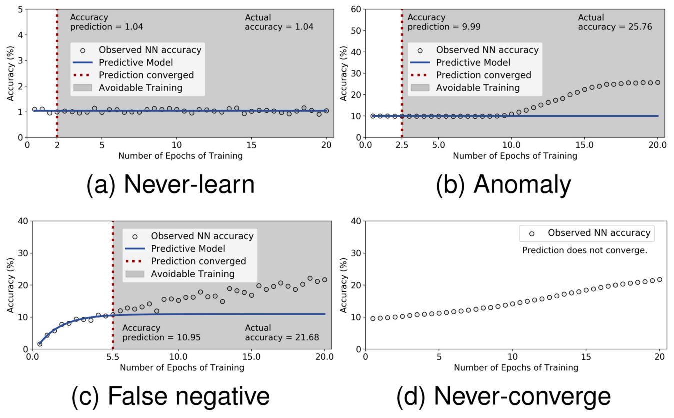


Fig. 8. Accuracy curve and prediction for edge cases (a) never-learn, (b) anomaly, (c) false negative and (d) never-converge. NNs in (a) and (c) trained on C100; NNs in (b) and (d) trained on FMNIST.

begin to increase steadily. This NN is correctly classified by *PENGUIN* as a never-learn. Fig. 8b depicts an anomaly on the F-MNIST benchmark dataset. This NN is incorrectly classified by *PENGUIN* as a never-learn because its accuracy does not begin to increase until after 9.5 epochs, and *PENGUIN* stabilizes earlier (at 2 epochs). Fig. 8c is an example of a false negative on C-100 (indicated by the red squares in Fig. 5a). Fig. 8d shows an NN on F-MNIST where the prediction of *PENGUIN* never converges. In this case, the NN is not terminated early by *PENGUIN* and instead trains for e_{pred} epochs—here 20 epochs as used for MENNDL—and the actual observed maximum accuracy is taken as the accuracy estimate for the NN. The cases where *PENGUIN*'s predictions do not converge are the samples in Fig. 6 with 0 percent of training epochs. The frequency of each of these cases may be observed in Table 7, Figs. 6, and 5. These are examples of edge cases that we will target in future work.

5 Related Works

Recent works have proposed different prediction strategies. The two most common approaches for truncating training are to train for a fixed but significantly reduced number of epochs (e.g., 20 instead of 100s) [21], [50] or to train until the maximum accuracy or minimum loss has not improved for a user-specified amount of time (e.g., if the maximum accuracy does not improve for 5 epochs). Such methods are commonly included in NN software like Keras [51], and are built-in to many NAS implementations, including the three NAS implementations in our case study.

There are a variety of works in the area of performance prediction and extrapolating NN learning curves [52], [53], [54], [55]. Some methods target hyperparameter search, which involves finding the best hyperparameter configuration for a given human-designed network but does not explore different architectures [52]. Domhan *et al.* [52] and Klein *et al.* [53] use probabilistic methods involving computationally expensive Markov Chain Monte Carlo sampling. The approach of Swersky *et al.* [55] automatically pauses and restarts training of models based on the predicted trajectory of the loss curves. Baker *et al.* [54] extract features from NNs to use in training a series of regression models to estimate fitness. There are also a variety of works that use parametric modeling of learning curves for machine learning problems.

The solution presented in this paper amplifies these approaches with the design of a prediction engine, *PENGUIN*, that allows the use of any parametric function to model learning curves. Furthermore, *PENGUIN* is fully decoupled from any NAS and can be plugged into existing NAS to provide fitness models and predictions, regardless of the NAS method or target datset.

6 Conclusion

This paper introduces *PENGUIN*, an engine that enables the decoupling of NAS from fitness prediction strategies. *PENGUIN* increases the computational efficiency of high-performance and high-throughput NAS workflows and enables portability of fitness prediction across NAS implementations and scientific domains. We present a case study using three diverse datasets (i.e., CIFAR-100, FashionMNIST, and SVHN) and three NAS implementations (i.e., MENNDL, Evo-CNN, and NSGA-Net). We compared the number of epochs, throughput, and walltime when these NAS methods would terminate training of the NNs using their built-in truncated training versus when augmented by *PENGUIN*. *PENGUIN* enabled a reduction in needed training epochs by 39% to 86%, depending on the NN sets and the benchmark dataset used. *PENGUIN* increases throughput of explored NNs by a factor of $1.6\times$ to $7.1\times$. Compared to MENNDL, which serves as a cutting edge NAS implementation on HPC systems such as the Summit supercomputer, *PENGUIN* achieves a walltime speedup between 2.5x and 5.3x.

In future work we will further quantify the impact of *PENGUIN* for a broader range of NAS implementations, a more diverse set of parametric functions, and a larger suite of datasets.

Supplemental material available on Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2022.3140681.

Acknowledgments

The authors would like to thank Danny Rorabaugh, Catherine D Schuman, and Michael R. Wyatt II for their valuable feedback.

References

[1] R. M. Patton *et al.*, "167-PFlops deep learning for electron microscopy: From learning physics to atomic manipulation," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, pp. 638–648.
[2] J.-P. Correa-Baena *et al.*, "Accelerating materials development via automation, machine learning, and high-performance computing," *Joule*, vol. 2, no. 8, pp. 1410–1420, 2018.
[3] S. Chen *et al.*, "How big data and high-performance computing drive brain science," *Genomic., Proteomic. Bioinf.*, vol. 17, no. 4, pp. 381–392, 2019.
[4] R. M. Patton *et al.*, "Exascale deep learning to accelerate cancer research," 2019, *arXiv:1909.12291*.
[5] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," 2018, *arXiv:1812.00332*.

[6] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, “Efficient neural architecture search via parameter sharing,” 2018, *arXiv:1802.03268*.

[7] Y. You *et al.*, “Large batch optimization for deep learning: Training bert in 76 minutes,” 2019, *arXiv:1904.00962*.

[8] A. Dosovitskiy *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” 2020, *arXiv:2010.11929*.

[9] T. Johnston, S. R. Young, D. Hughes, R. M. Patton, and D. White, “Optimizing convolutional neural networks for cloud detection,” in *Proc. Mach. Learn. HPC Environ.*, 2017, pp. 1–9.

[10] T. Elsken, J. H. Metzen, and F. Hutter, “Neural architecture search: A survey,” *J. Mach. Learn. Res.*, vol. 20, no. 55, pp. 1–21, 2019.

[11] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, “Neural architecture search without training,” 2020, *arXiv:2006.04647*.

[12] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, “Evolving deep convolutional neural networks for image classification,” in *IEEE Trans. Evol. Comput.*, vol. 24, no. 2, pp. 394–407, Apr. 2020. [Online]. Available: https://github.com/yn-sun/evocnn

[13] F. E. Fernandes Junior and G. G. Yen, “Particle swarm optimization of deep neural networks architectures for image classification,” *Swarm Evol. Comput.*, vol. 49, pp. 62–74, 2019.

[14] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, “Hierarchical representations for efficient architecture search,” in *Proc. Int. Conf. Learn. Representations*, 2018.

[15] Z. Lu *et al.*, “Nsga-net: Neural architecture search using multiobjective genetic algorithm,” in *Proc. Genet. Evol. Comput. Conf.*, 2019, pp. 419–427. [Online]. Available: https://github.com/ianwhale/nsga-net

[16] C. Liu *et al.*, “Progressive neural architecture search,” in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 19–35.

[17] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, “Neural architecture optimization,” in *Proc. 32nd Int. Conf. Adv. Neural Inf. Process. Syst. 31*, 2018, pp. 7827–7838.

[18] K. Kandasamy, W. Neiswanger, J. Schneider, B. Póczos, and E. P. Xing, “Neural architecture search with bayesian optimisation and optimal transport,” in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 2020–2029.

[19] S. R. Young *et al.*, “Evolving deep networks using HPC,” in *Proc. Mach. Learn. HPC Environ.*, 2017, pp. 1–7.

[20] G. J. van Wyk and A. S. Bosman, “Evolutionary neural architecture search for image restoration,” in *Proc. Int. Joint Conf. Neural Netw.*, 2020, pp. 1–8.

[21] E. Real *et al.*, “Large-scale evolution of image classifiers,” in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 2902–2911.

[22] L. Xie and A. Yuille, “Genetic CNN,” in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 1388–1397.

[23] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 8697–8710.

[24] H. Jin, Q. Song, and X. Hu, “Auto-keras: An efficient neural architecture search system,” in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 1946–1956. [Online]. Available: https://autokeras.com/

[25] T. Viering and M. Loog, “The shape of learning curves: A review,” 2021, *arXiv:2103.10948*.

[26] A. Krizhevsky, “Learning multiple layers of features from tiny images,” Univ. Toronto, Toronto, Tech. Rep. TR-2009, 2009.

[27] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, “Maxout networks,” in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1319–1327.

[28] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, “Deeply-supervised nets,” in *Proc. 18th Int. Conf. Artifi. Intell. Statist.*, 2015, pp. 562–570.

[29] W. Shi, Y. Gong, and J. Wang, “Improving CNN performance with min-max objective,” in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 2004–2010.

[30] C. Song *et al.*, “MAT: A multi-strength adversarial training method to mitigate adversarial attacks,” in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2018, pp. 476–481.

[31] X. Han and Q. Dai, “Batch-normalized Mlpconv-wise supervised pre-training network in network,” *Appl. Intell.*, vol. 48, no. 1, pp. 142–155, 2018.

[32] S. Li, W. Song, H. Qin, and A. Hao, “Deep variance network: An iterative, improved CNN framework for unbalanced training datasets,” *Pattern Recognit.*, vol. 81, pp. 294–308, 2018.

[33] F. Assunçao, N. Lourenço, P. Machado, and B. Ribeiro, “Denser: deep evolutionary network structured representation,” *Genet. Program. Evolvable Mach.*, vol. 20, no. 1, pp. 5–35, 2019.

[34] D. Tan, W. Zhong, X. Peng, Q. Wang, and V. Mahalec, “Accurate and fast deep evolutionary networks structured representation through activating and freezing dense networks,” *IEEE Trans. Cogn. Devel. Syst.*, to be published, doi: 10.1109/TCDS.2020.3017100.

[35] B. Bilalli, A. Abelló, and T. Aluja-Banet , “On the predictive power of meta-features in openml,” *Int. J. Appl. Math. Comput. Sci.*, vol. 27, no. 4, pp. 697–712, 2017.

[36] J. M. Johnson and T. M. Khoshgoftaar, “Survey on deep learning with class imbalance,” *J. Big Data*, vol. 6, no. 1, 2019, Art. no. 27.

[37] A. Torralba, R. Fergus, and W. T. Freeman, “80 million tiny images: A large data set for nonparametric object and scene recognition,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970, Nov. 2008.

[38] L. N. Darlow, E. J. Crowley, A. Antoniou, and A. J. Storkey, “Cinic-10 is not imagenet or cifar-10,” 2018, *arXiv:1810.03505*.

[39] F. Scheidegger, R. Istrate, G. Mariani, L. Benini, C. Bekas, and C. Malossi, “Efficient image dataset classification difficulty estimation for predicting deep-learning accuracy,” *The Vis. Comput.*, pp. 1–18, 2020.

[40] B. Barz and J. Denzler, “Do we train on test data? purging CIFAR of near-duplicates,” *J. Imag.*, vol. 6, no. 6, 2020, Art. no. 41.

[41] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms,” 2017, *arXiv:1708.07747*.

[42] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[43] K. Meshkini, J. Platos, and H. Ghassemain, “An analysis of convolutional neural network for fashion images classification (fashion-mnist),” in *Proc. Int. Conf. Intell. Inf. Technol. Ind.*, 2019, pp. 85–95.

[44] M. Kayed, A. Anter, and H. Mohamed, “Classification of garments from fashion MNIST dataset using CNN LeNet-5 architecture,” in *Proc. Int. Conf. Innov. Trends Commun. Comput. Eng.*, 2020, pp. 238–243.

[45] S. Bhatnagar, D. Ghosal, and M. H. Kolekar, “Classification of fashion article images using convolutional neural networks,” in *Proc. 4th Int. Conf. Image Inf. Process.*, 2017, pp. 1–6.

[46] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, “Reading digits in natural images with unsupervised feature learning,” in *Proc. Workshop Deep Learn. Unsupervised Feature Learn.*, 2011.

[47] P. Sermanet, S. Chintala, and Y. LeCun, “Convolutional neural networks applied to house numbers digit classification,” in *Proc. 21st Int. Conf. Pattern Recognit.*, 2012, pp. 3288–3291.

[48] A. Keller Rorabaugh, S. Caíno-Lores, M. R. Wyatt II, T. Johnston, and M. Taufer, “Architecture Descriptions and High Frequency Accuracy and Loss Data of Random Neural Networks Trained on Image Datasets,” 2021. [Online]. Available: https://doi.org/10.7910/DVN/ZXTCGF

[49] A. Keller Rorabaugh, S. Caíno-Lores, T. Johnston, and M. Taufer, “High Frequency Accuracy and Loss Data of Random Neural Networks Trained on Image Datasets,” *Data Brief*, vol. 40, 2022, Art. no. 107780.

[50] B. Zoph and Q. V. Le, “Neural architecture search with reinforcement learning,” in *Proc. 5th Int. Conf. Learn. Representations*, 2017.

[51] MIT, Keras, Mar. 2015. [Online]. Available: https://keras.io/api/callbacks/early_stopping/

[52] T. Domhan, J. T. Springenberg, and F. Hutter, “Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves,” in *Proc. 24th Int. Joint Conf. Artif. Intell.*, 2015, pp. 3460–3468.

[53] A. Klein, S. Falkner, J. Springenberg, and F. Hutter, “Learning curve prediction with bayesian neural networks,” in *Proc. 5th Int. Conf. Learn. Representations*, 2017.

[54] B. Baker, O. Gupta, R. Raskar, and N. Naik, “Accelerating neural architecture search using performance prediction,” in *Proc. NIPS Workshop Meta-Learn.*, 2017.

[55] K. Swersky, J. Snoek, and R. P. Adams, “Freeze-thaw bayesian optimization,” 2014, *arXiv:1406.3896*.

Ariel Keller Rorabaugh received the MS and PhD degrees in mathematics from Emory University, in 2017 and 2018 respectively. She currently is a postdoctoral research associate with the University of Tennessee, Knoxville, as a member of the Global Computing Laboratory directed by Dr. Michela Taufer. Her research interests include machine learning and artificial intelligence for high-performance computing.

Silvina Caíno-Lores received the BSc and MSc degrees in computer science and technology from the Carlos III University of Madrid, in 2014 and 2015, respectively, and the PhD degree in computer science and technology from the Carlos III University of Madrid, Spain, in 2019. She is currently a postdoctoral research associate with the University of Tennessee, Knoxville, as a member of the Global Computing Laboratory directed by Dr. Michela Taufer. Her research interests include cloud computing, in-memory computing and storage, HPC scientific simulations, and data-centric paradigms.

Travis Johnston received the BS degree from the University of South Carolina, and PhD degree in mathematics from the University of Nebraska-Lincoln. He is currently a senior scientist with Striveworks in Austin, Texas. His research interests include the intersection of machine learning and high-performance computing with applications to areas of national interest including fundamental science, energy, health, and national security.

Michela Taufer (Senior Member, IEEE) received the PhD degree in computer science from the Swiss Federal Institute of Technology (ETH), in 2002. She is currently an ACM Distinguished Scientist and holds the Jack Dongarra professorship in high-performance computing with the Department of Electrical Engineering and Computer Science, University of Tennessee Knoxville. Her research interests include high-performance computing, volunteer computing, scientific applications, scheduling and reproducibility challenges, and in situ data analytics.