Scalable Reinforcement-Learning-Based Neural Architecture Search for Cancer Deep Learning Research

Prasanna Balaprakash∗
pbalapra@anl.gov
Argonne National Laboratory

Romain Egele∗
regele@anl.gov
Argonne National Laboratory

Misha Salim
msalim@anl.gov
Argonne National Laboratory

Stefan Wild
wild@anl.gov
Argonne National Laboratory

Venkatram Vishwanath
venkat@anl.gov
Argonne National Laboratory

Fangfang Xia
fangfang@anl.gov
Argonne National Laboratory

Tom Brettin
brettin@anl.gov
Argonne National Laboratory

Rick Stevens
stevens@anl.gov
Argonne National Laboratory

ABSTRACT
Cancer is a complex disease, the understanding and treatment of which are being aided through increases in the volume of collected data and in the scale of deployed computing power. Consequently, there is a growing need for the development of data-driven and, in particular, deep learning methods for various tasks such as cancer diagnosis, detection, prognosis, and prediction. Despite recent successes, however, designing high-performing deep learning models for nonimage and non-text cancer data is a time-consuming, trial-and-error, manual task that requires both cancer domain and deep learning expertise. To that end, we develop a reinforcement-learning-based neural architecture search to automate deep-learning-based predictive model development for a class of representative cancer data. We develop custom building blocks that allow domain experts to incorporate the cancer-data-specific characteristics. We show that our approach discovers deep neural network architectures that have significantly fewer trainable parameters, shorter training time, and accuracy similar to or higher than those of manually designed architectures. We study and demonstrate the scalability of our approach on up to 1,024 Intel Knights Landing nodes of the Theta supercomputer at the Argonne Leadership Computing Facility.

KEYWORDS
cancer, deep learning, neural architecture search, reinforcement learning

1 INTRODUCTION
Cancer is a disease that drastically alters the normal biological function of cells and damages the health of an individual. Cancer is estimated to be the second leading cause of death globally and was responsible for 9.6 million deaths in 2018 [10]. A thorough understanding of cancer remains elusive because of challenges due to the variety of cancer types, heterogeneity within a cancer type, structural variation in cancer-causing genes, complex metabolic pathways, and nontrivial drug-tumor interactions [31, 48, 57, 64, 67].

∗Both authors contributed equally to this research.

Recently, as a result of coordinated data management initiatives, the cancer research community increasingly has access to a large volume of data. This has led to a number of promising large-scale, data-driven cancer research efforts. In particular, machine learning (ML) methods have been employed for tasks such as identifying cancer cell patterns; modeling complex relationships between drugs and cancer cells; and predicting cancer types.

With the sharp increases in available data and computing power, considerable attention has been devoted to deep learning (DL) approaches. The first wave of success in applying DL for cancer stems from adapting the convolutional neural network (CNN) and recurrent neural network (RNN) architectures that were developed for image and text data. For example, CNNs have been used for cancer cell detection from images, and RNNs and its variants have been used for analyzing clinical reports.

These adaptations are possible because of the underlying regular grid nature of image and text data [24]. For example, images share the spatial correlation properties, and convolution operations designed to extract features from natural images can be generalized for detecting cancer cells in images with relatively minor modifications. However, designing deep neural networks (DNNs) for nonimage and non-text data remains underdeveloped in cancer research. Several cancer predictive modeling tasks deal with tabular data comprising an output and multidimensional inputs. For example, in the drug response problem, DNNs can be used to model a complex nonlinear relationship between the properties of drugs and tumors in order to predict treatment response [84]. Here, the properties of drugs and tumors cannot easily be expressed as images and text and cast into classical CNN and RNN architectures. Consequently, cancer researchers and DL experts resort to manual trial-and-error methods to design DNNs. Tabular data types are diverse; consequently, designing DNNs with shared patterns such as CNNs and RNNs is not meaningful unless further assumptions about the data are made. Fully connected DNNs are used for many modeling tasks with tabular data. However, they can potentially lead to unsatisfactory performance because they can have large numbers of parameters, overfitting issues, and difficult optimization landscape with low-performing local optima [33]. Moreover, tabular data often is obtained from multiple sources and modes, where combining certain inputs using problem-specific domain knowledge
can lead to better features and physically meaningful and robust models, thus preventing the design of effective architectures similar to CNNs and RNNs.

Automated machine learning (AutoML) [1, 4, 22, 37, 88] automates the development of ML models by searching over appropriate components and their hyperparameters for preprocessing, feature engineering, and model selection to maximize a user-defined metric. AutoML has been shown to reduce the amount of human effort and time required for a number of traditional ML model development tasks. Although DL reduces the need for feature engineering, extraction, and selection tasks, finding the right DNN architecture and its hyperparameters is crucial for predictive accuracy. Even on image and text data, DNNs obtained by using AutoML approaches have outperformed manually engineered DNNs that took several years of development [59, 86, 88].

AutoML approaches for DNNs can be broadly classified into hyperparameter search and neural architecture search (NAS). Hyperparameter search approaches try to find the best values for the hyperparameters for a fixed neural architecture. Examples include random search [20], Bayesian optimization [21, 42, 70], bandit-based methods [44, 70], metaheuristics [51, 54], and population-based training [38] approaches. NAS methods search over model descriptions of neural network specifications. Examples include discrete search-space traversal [49, 56], reinforcement learning (RL) [13, 61, 88], and evolutionary algorithms [34, 73, 75, 81].

We focus on developing scalable neural-network-based RL for NAS, which offers several potential advantages. First, RL is a first-order method that leverages gradients. Second, RL-based NAS construction is based on a Markov decision process: decisions that are made to construct a given layer depend on the decisions that were made on the previous layers. This exploits the inherent structure of DNNs, which are characterized by hierarchical data flow computations; While traditional RL methods pose several challenges [76] such as exploration-exploitation tradeoff, sample inefficiency, and long-term credit assignment, recent developments [35, 68] in the field are promising.

Although hyperparameter search work has been done on cancer data [83], to our knowledge scalable RL-based NAS has not been applied to cancer predictive modeling tasks. An online bibliography of NAS [6] and a recent NAS survey paper [32] did not list any cancer-related articles. The reasons may be twofold. First, AutoML, and in particular NAS, is still in its infancy. Most of the existing work in NAS focuses on image classification tasks on benchmark data sets. Since convolutions and recurrent cells form the basic building blocks for CNNs and RNNs, respectively, the problem of defining the search space for CNN and RNN architectures has become relatively easy [32]. However, no such generalized building block exists for non-image and non-text data. Second, large-scale NAS and hyperparameter search require high-performance computing (HPC) resources and appropriate software infrastructure [17]. These requirements are attributed to the fact that architecture evaluations (training and validation) are computationally expensive and parallel evaluation of multiple architectures on multiple compute nodes through scalable search methods is critical to finding DNNs with high accuracy in short computation time. We note that the time needed for NAS can be more than the training time of a manually designed network. However, designing the network by manually intensive trial-and-error approaches can take days to weeks even for ML experts [37].

We develop a scalable RL-based NAS infrastructure to automate DL-based predictive model development for a class of cancer data. The contributions of the paper are as follows:

- We develop a DL NAS search space with new types of components that take into account characteristics specific to cancer data. These include multidrug and cell line inputs, 1D convolution for traversing large drug descriptors, and nodes that facilitate weight sharing between drug descriptors.
- We demonstrate a scalable RL-based NAS on 1,024 Intel Knights Landing (KNL) nodes of Theta, a leadership-class HPC system, for cancer DL using a multiagent and multiworker approach.
- We scale asynchronous and synchronous proximal policy optimization, a state-of-the-art RL approach for NAS. Of particular importance is the convergence analysis of search methods at scale. We demonstrate that RL-based NAS achieves high accuracy on architectures because of the search strategy and not by pure chance as in random search.
- We show that the scalable RL-based NAS can be used to generate multiple accurate DNN architectures that have significantly fewer training parameters, shorter training time, and accuracy similar to or higher than those of manually designed networks.
- We implement our approach as a neural architecture search module within DeepHyper [14, 15], an open-source software package, that can be readily deployed on leadership-class machines for cancer DL research.

2 PROBLEM SETS AND MANUALLY DESIGNED DEEP NEURAL NETWORKS

We focus on a set of DL-based predictive modeling problem sets from the CANcer Distributed Learning Environment (CANDLE) project [5] that comprises data sets and manually designed DNNs for drug response; RAS gene family pathways; and treatment strategy at molecular, cellular, and population scales. Within these problem sets, we target three benchmarks, which represent a class of predictive modeling problems that seek to predict drug response based on molecular features of tumor cells and drug descriptors. An overview of these open-source [2] benchmarks (i.e., data set plus manually designed DNN) is given below.

2.1 Predicting tumor cell line response (Combo)

In the Combo benchmark [3], recent paired drug screening results from the National Cancer Institute (NCI) are used to model drug synergy and understand how drug combinations interact with tumor molecular features. Given drug screening results on NCI60 cell lines available at the NCI-ALMANAC database, Combo’s goal is to build a DNN that can predict the growth percentage from the cell line molecular features and the descriptors of drug pairs. The manually designed DNN comprises three input layers: one for cell expression (of dimension $d=942$) and two for drug descriptors ($d=3,820$). The two input layers for the drug pairs are connected by a shared submodel of three dense layers ($d=1,000$). The cell expression layer is connected to a submodel of three dense layers each with $d=1,000$. The outputs of these submodels are concatenated and connected to three dense layers each with $d=1,000$. The scalar
output layer is used to predict the percent growth for a given drug concentration. The training and validation input data are given as matrices of sizes 248,650 (number of data points) × 4,762 (total input size) and 62,164 × 4,762, respectively. The training and validation output data are matrices of size 248,650 × 1 and 62,164 × 1, respectively.

2.2 Predicting tumor dose response across multiple data sources (Uno)

The Uno benchmark [9] integrates cancer drug screening data from 2.5 million samples across six research centers to examine study biases and to build a unified drug response model. The associated manually designed DNN has four input layers: a cell RNA sequence layer (d = 942), a dose layer (d = 1), a drug descriptor layer (d = 5,270), and a drug fingerprints layer (d = 2,048). It has three feature-encoding submodels for cell RNA sequence, drug descriptor, and drug fingerprints. Each submodel is composed of three hidden layers, each with d = 1,000. The last layer for each of the submodels is connected to the concatenation layer along with the dose layer. This is connected to three hidden layers each with d = 1,000. The scalar output layer is used to predict tumor dose. We used the single drug paclataxel, a simplified indicator, for this study. The training and validation input data are given as matrices of sizes 9,588 × 8,261 and 2,397 × 8,261, respectively. The training and validation output data are given as matrices of sizes 9,588 × 1 and 2,397 × 1, respectively.

2.3 Classifying RNA-seq gene expressions (NT3)

The NT3 benchmark [8] classifies tumors from normal tissue by tracking gene-expression-level tumor signatures. The associated manually designed DNN has an input layer for RNA sequence gene expression (d = 60,483). This is connected to a 1D convolutional layer of 128 filters with kernel size 20 and a maximum pooling layer of size 1. This is followed by a 1D convolutional layer of 128 filters with kernel size 10 and a maximum pooling layer of size 10. The output of the pooling layer is flattened and given to the dense layer of size 200 and a dropout layer with 0.1%. This is followed by a dense layer of size 20 and a dropout layer with 0.1%. The output layer of size 2 for the two classes with softmax activation is used to predict the tissue type. The training and validation input data are given as matrices of sizes 1,120 × 60,483 and 280 × 60,483, respectively. The training and validation output data are given as matrices of sizes 1,120 × 1 and 280 × 1, respectively.

3 RL-BASED NAS

NAS comprises (1) a search space that defines a set of feasible architectures, (2) a search strategy to search over the defined search space, and (3) a reward estimation strategy that describes how to evaluate the quality of a given neural architecture.

3.1 Search space

We describe the search space of a neural architecture using a graph structure. The basic building block is a set of nodes N with possible choices; typically these choices are nonordinal (i.e., values that cannot be ordered in a numeric scale). For example, \{Dense(10, sig), Dense(50, relu), and Dropout(0.5)\} respectively represent a dense layer with 10 units and sigmoid activation, a dense layer with 50 units with relu activation, and a layer with 50% dropout. A block B is a directed acyclic graph: (N = (N_I, N_O, N_R, R_N), where the set of nodes is differentiated by input nodes N_I, intermediate nodes N_R, and output nodes N_O and where R_N ⊆ (N_I ∪ N_R) × (N_R ∪ N_O) is a set of binary relations that describe the connections among nodes in N. A cell C is a set of li blocks {B_1, ..., B_{li−1}} and a rule RC_out to create the output of C. The structure S is given by the set \{(I_l, ..., I_{l−1}), (C_0, ..., C_{K−1}), R_{S_out}\}, where \{(I_l, ..., I_{l−1})\} is a tuple of P inputs, \{(C_0, ..., C_{K−1})\} is a tuple of K cells, and R_{S_out} is a rule to create the output of S. Users can define cell-specific blocks and block-specific input, intermediate, and output nodes.

Figure 1 shows a sample search space. The structure S is made up of three cells (C_0, C_1, C_2). Cell C_0 has one block B_0 that has one input (N_0) and one output node (N_1). The rule to create the output is concatenation. Since there is only one block, the output from N_1 is the output layer for C_0; cell C_1 is similar to cell C_0. Since N_0 is a dense layer, the output of C_0 is connected as an input to the C_1.

The search space definition that we have differs in two ways from existing chain-structured neural networks, multibranch networks, and cells blocks for designing CNNs and RNNs. The first is the flexibility to define multiple input layers \{(I_1, ..., I_{l−1})\} (e.g., to support cell expression, drug descriptors in Combo; and RNA sequence, dose, drug descriptor, and drug fingerprints in Uno) and a cell for each of them. The second is the node types. By default, each node is a VariableNode, which is characterized by a set of possible choices.

In addition, we define two types of nodes. ConstantNode, with a particular operation, is excluded from the search space but will be used in neural architecture construction. This allows for domain knowledge encoding—for example, if we want the dose value in Uno in every block, we can define a constant node for every block and connect them to the dose input layer. MirrorNode is used to reuse an existing node. For example, in Combo, drug1.descriptor's and

\[\text{Uno} \quad \text{multiple data sources}\]

Figure 1: Example search space for NAS

\[\text{Uno} \quad \text{benchmark}\]

\[\text{Uno} \quad \text{tumor dose response}\]

\[\text{Uno} \quad \text{RNA-seq gene expressions}\]

\[\text{Uno} \quad \text{benchmark}\]

\[\text{Uno} \quad \text{tumor dose response}\]

\[\text{Uno} \quad \text{RNA-seq gene expressions}\]
drug2.descriptors share the same submodel for feature encoding. To support such shared submodel construction, we define a cell with variable nodes for drug1.descriptors and a cell with mirror nodes drug2.descriptors. Consequently, the mirror nodes are not part of the specified search space.

Using the search space formalism, we define the search spaces for Uno, Combo, and NT3. We consider a small and a large search space for each of Combo and Uno. For NT3, we define only a small search space because the baseline DNN obtains 98% accuracy on the validation data.

3.1.1 Combo. We define a VariableNode as a node consisting of options representing identity operation, a dense layer with x units and activation function y (Dense(x, y)), and a dropout layer Dropout(r) where r is a fraction of input units to drop (e.g., Identity, Dense(100, relu), Dense(100, tanh), Dense(100, sigmoid), Dropout(0.05), Dense(500, relu), Dense(500, tanh), Dense(500, sigmoid), Dropout(0.1), Dense(1000, relu), Dense(1000, tanh), Dense(1000, sigmoid), Dropout(0.2)). We refer to this VariableNode as MLP_Node, where MLP stands for multilayered perceptron.

For the small Combo search space, we define cells C0, C1, and C2. Cell C0 receives input from three input layers, cell expression, drug1 descriptors, and drug2 descriptors, and has three blocks, ⟨C0, B0⟩, ⟨C0, B1⟩, and ⟨C0, B2⟩. The block ⟨C0, B0⟩ receives input from the cell expression layer and comprises three MLP_Nodes connected sequentially in a feed-forward manner. The block ⟨C0, B1⟩ receives input from drug1 descriptors and has three MLP_Nodes similar to ⟨C0, B0⟩. The block ⟨C0, B2⟩ receives input from drug2 descriptors but has three Mirror_Nodes that reuse the MLP_Nodes of ⟨C0, B1⟩ to enable sharing the same submodel between drug1 descriptors and drug2 descriptors.

The output from C0 is used as input to the cell C1 that contains two blocks, ⟨C1, B0⟩, ⟨C1, B1⟩. The former has three MLP_Nodes with feed-forward connectivity. ⟨C1, B1⟩ has one VariableNode with a Connect operation that includes options to create skip-connections (i.e., Null, Cell expression, Drug 1 descriptors, Drug 2 descriptors, Cell 1 output, Inputs, Cell expression & Drug 1 descriptors, Cell expression & Drug 2 descriptors, Drug 1 & 2 descriptors).

The output from C1 is used as input to the cell C2 that has one block ⟨C2, B0⟩ with three MLP_Nodes with feed-forward connectivity. The Concatenate operation is used to combine the outputs from C0, C1, and C2 to form the final output. The size of the architecture space is ≈ 2.0968 × 10^14.

For the large search space, we replicate C1 8 times. For C1 for i ∈ [1, . . . , 8], we update the set of Connect operations by adding outputs of C1i−1 . . . C1−1 (i.e., outputs of previous cells). The size of the architecture space is ≈ 2.3298 × 10^13.

For the large search space, we have nine cells. The cell C0 is the same as the one we used for the small search space. Each cell Ci for i ∈ [1, 8], has two blocks. The block ⟨Ci, Bi⟩ has one MLP_Node. The block ⟨Ci, Bi⟩ has one VariableNode with the following set of Connect operations to create skip connections: Null, all combinations of inputs (i.e., 15 possibilities), all outputs of previous cells, and all N0 of previous cells except C0. Each Ci takes as input the output of the cell Ci−1 for i ∈ [1, . . . , 8]. The size of the architecture space is ≈ 5.7408 × 10^29.

3.1.3 NT3. We define five types of nodes: Conv_Node, Act_Node, Pool_Node, Dense_Node, and Dropout_Node. The Conv_Node has the following options, where x in Conv1D(x) is the filter size. Here, the number of filters and the stride are set to 8 and 1, respectively: Identity, Conv1D(3), Conv1D(4), Conv1D(5), Conv1D(6). The Act_Node has the following options, where x in Activation(x) is a specific type of activation function: Identity, Activation(tan), Activation(sigmoid). The Pool_Node has the following options, where x in MaxPooling1D(x) represents the pooling size: Identity, MaxPooling1D(3), MaxPooling1D(4), MaxPooling1D(5), MaxPooling1D(6). The Dense_Node has the following options: Identity, Dense(10), Dense(50), Dense(100), Dense(200), Dense(250), Dense(500), Dense(750), Dense(1000).

The Dropout_Node has the following options: Identity, Dropout(0.5), Dropout(0.4), Dropout(0.3), Dropout(0.2), Dropout(0.1), Dropout(0.05).

For the small search space, we define four cells: C0, C1, C2, and C3. Each cell Ci has one block ⟨Ci, Bi⟩, which takes the output of the previous cell as input except for the first block ⟨C0, B0⟩, which takes RNA-seq gene expression as input. This is followed by Conv_Node, Act_Nodes, and Pool_Node, which are connected sequentially. The blocks ⟨C0, B0⟩ and ⟨C1, B0⟩ have three sequentially connected VariableNodes: Conv_Node, Act_Nodes, and Pool_Node. The blocks ⟨C2, B0⟩ and ⟨C3, B0⟩ have three sequentially connected VariableNodes: Dense_Node, Act_Nodes, and Drop_Node. The size of the architecture space is 6.3504 × 10^8.

3.2 Search strategy

Different approaches have been developed to explore the space of neural architectures described by graphs. These approaches include random search, Bayesian optimization, evolutionary methods, RL, and other gradient-based methods. We focus on RL-based NAS, where an agent generates a neural architecture, trains the generated neural architecture on training data, and computes an accuracy metric on validation data. The agent receives a positive (negative) reward when the validation accuracy of the generated architecture increases (decreases). The goal of the agent is to learn to generate neural architectures that result in high validation accuracy by maximizing the agent’s reward.

Policy gradient methods have emerged as a promising optimization approach for leveraging DL for RL problems [76, 77]. These methods alternate between sampling and optimization using a loss
function of the form

\[ J_t(\theta) = \hat{J}_t[\log \pi_{\theta}(a_t|s_t) \hat{A}_t], \]

where \( \pi_{\theta}(a_t|s_t) \) is a stochastic policy given by the action probabilities of a neural network (parameterized by \( \theta \)) that, for given a state \( s_t \), performs an action \( a_t; \hat{A}_t \) is the advantage function at time step \( t \) that measures goodness of the sampled actions from \( \pi_{\theta} \); and \( \hat{A}_t \) denotes the empirical average over a finite batch of sampled actions. The gradient of \( J_t(\theta) \) is used in a gradient ascent scheme to update the neural network parameters \( \theta \) to generate actions with high rewards.

Actor-critic methods [35, 76] improve the stability and convergence of policy gradient methods by using a separate critic to estimate the value of each state that serves as a state-dependent baseline. The critic is typically a neural network that progressively learns to predict the estimate of the reward given the current state \( s_t \). The difference between the rewards collected at the current state from the policy network \( \pi_{\theta}(a_t|s_t) \) and the estimate of the reward from the critic is used to compute the advantage. When the reward of the policy network is better (worse) than the estimate of a critic, the advantage function will be positive (negative), and the policy network parameters \( \theta \) will be updated by using the gradient and the advantage function value.

Proximal policy optimization (PPO) is a policy gradient method for RL [68] that uses a loss function of the form

\[ J_t(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \]

where \( r_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta}(a_t'|s_t)} \) is the ratio of action probabilities under the new and old policies; the clip/median operator ensures that the ratio is in the interval \([1-\epsilon, 1+\epsilon]\), and \( \epsilon \in (0, 1) \) is a hyperparameter (typically set to 0.1 or 0.2). The clipping operation prevents the sample-based stochastic gradient estimator of \( \nabla_{\theta} J_t(\theta) \) from making extreme updates to \( \theta \).

We used two algorithmic approaches to scale up the NAS: synchronous advantage actor-critic (A2C) and asynchronous advantage actor-critic (A3C). Both approaches use a manager-worker distributed learning paradigm as shown in Fig. 2. In A2C, all \( N \) agents start with the same policy network. At each step \( t \), agent \( i \) generates \( M \) neural architectures, evaluates them in parallel (training and validation), and computes the gradient estimate \( \nabla_{\theta} J_t(\theta) \) using the PPO method. Once the parameter server (PS) receives the PPO gradients from the \( N \) agents, it averages the gradients and sends the result to each agent. The parameters of the policy network for each agent are updated by using the averaged gradient. The A3C method is similar to the A2C method except that an agent \( i \) sends the PPO gradients to the PS, which does not wait for the gradients from all the agents before computing the average. Instead, the PS computes the average from a set of recently received gradients and sends it to the agent \( i \).

The synchronous update of A2C guarantees that the gradient updates to the PS are coordinated. A drawback of this approach is that the agents must wait until all \( M \times N \) tasks have completed in a given iteration. Given a wide range of training times for the generated networks, A2C will underutilize nodes and limit parallel scalability. On the other hand, A3C increases the node utilization at the expense of gradient staleness due to the asynchronous gradient updates, a staleness that grows with the number of agents. While several works address synchronous and asynchronous updates in large batch supervised learning, studies in RL settings are limited, and none exists for the RL-based NAS methods studied here.

### 3.3 Reward estimation strategy

Crucial to the effectiveness of RL-based NAS is the way in which rewards are estimated for the agent-generated architectures. A naive approach of training each architecture from scratch on the full training data is computationally expensive even at scale and can require thousands of single-GPU days [32]. A common approach to overcome this challenge is low-fidelity training, where the rewards are estimated by using a smaller number of training epochs [87], a subset of original data [43], a smaller proxy network for the original network [89], and a smaller proxy data set [27]. In this paper, we use a smaller number of training epochs, a subset of the full training data, and timeout strategies to reduce the training time required for estimating the reward for architectures generated by NAS agents.

For the image data sets, research has showed that low-fidelity training can introduce a bias in reward estimation, which requires a gradual increase in fidelity as the search progresses [44]. Whether this is the case for nonimage and nontext cancer data is not clear, however. Moreover, the impact of low-fidelity training on NAS at scale is not well understood. As we show next, at scale the RL-agent behavior (and consequently the generated architectures) exhibits different characteristics based on the fidelity level employed.

### 4 SOFTWARE DESCRIPTION

Our open source software comprises three Python subpackages: benchmark, a collection of representative NAS test problems; evaluator, a model evaluation interface with several execution backends; and search, a suite of parallel NAS implementations that are implemented as distributed-memory mpi4py applications, where each MPI rank represents an RL agent.

As new architectures are generated by the RL agents, the corresponding reward estimation tasks are submitted via the evaluator.
get finished evals

them onto idle nodes using a pilot-job mechanism. The launcher
sam launcher, in turn, pulls new reward estimation tasks and launches
therefore each agent has a separate database connection. The Bal-
agent interacts with an environment that contains a
PostgreSQL database through the Django model API. Each NAS
NAS is shown in Fig. 3. The
BalsamEvaluator
tasks at any given time
the fraction of allocated compute nodes actively running evaluation
the benefit of agent-specific random weight initialization. Balsam’s
uated architectures is not maintained because that would nullify
to avoid repeating reward estimation tasks. A global cache of eval-
task launching and monitoring for supercomputing platforms. Each
interconnected by using an Aries fabric with a file system capacity
of 10 PB.

Figure 3: Distributed NAS architecture. The Balsam service runs on a designated node, providing a Django interface to
a PostgreSQL database and interfacing with the local batch scheduler for automated job submission. The launcher is a
pilot job that runs on the allocated resources and launches
tasks from the database. The multiagent search runs as a sin-
gle MPI application, and each agent submits model evalu-
tion tasks through the Balsam service API. As these evalua-
tions are added, the launcher continually executes them by dispatch onto idle worker nodes.

interface. The evaluator exposes a three-function API, that gener-
cally supports parallel asynchronous search methods. In the con-
text of NAS, add_eval_batch submits new reward estimation tasks,
while get_finished_evals is a nonblocking call that fetches newly
completed completed reward estimations. This API enforces a complete sep-
aration of concerns between the search and the backend for par-
allel evaluation of generated architectures. Moreover, a variety of
evaluator backends, ranging from lightweight threads to mas-
vously parallel jobs using a workflow system, allow a single search
code to scale from toy models on a laptop to large DNNs running
across leadership-class HPC resources.

We used the DeepHyper [15] software module on Theta, our
target HPC platform, to dispatch reward estimation tasks to Balsam
[66], a workflow manager enabling high-throughput, asynchronous
task launching and monitoring for supercomputing platforms. Each
agent exploited DeepHyper’s evaluation cache and leveraged this to
avoid repeating reward estimation tasks. A global cache of eval-
uated architectures is not maintained because that would nullify
the benefit of agent-specific random weight initialization. Balsam’s
performance monitoring capabilities are used to infer utilization as
the fraction of allocated compute nodes actively running evaluation
tasks at any given time $t$; the maximum value of 1.0 indicates that
all worker nodes are busy evaluating configurations.

A schematic view of the NAS-Balsam infrastructure for parallel
NAS is shown in Fig. 3. The BalsamEvaluator queries a Balsam
PostgreSQL database through the Django model API. Each NAS
agent interacts with an environment that contains a BalsamEvaluator; therefore each agent has a separate database connection. The Bal-
sam launcher, in turn, pulls new reward estimation tasks and launches them onto idle nodes using a pilot-job mechanism. The launcher
monitors ongoing tasks for completion status and signals successful
evaluations back to the BalsamEvaluator.

For the implementations of A3C and A2C, we interfaced our NAS
software with OpenAI Baselines [29], open-source software with a set of high-performing state-of-the-art RL methods. We developed
an API for the RL methods in OpenAI Baselines so that we can leverage any new updates and RL methods that become available in
the package. We followed the same interface as in OpenAI Gym
[23] to create a NAS environment that encapsulates the Evaluator
interface of Balsam to submit jobs for reward estimation.

The interface to specify the graph search space comprises sup-
port for structure, cell, block, variable node, constant node, mirror
node, and operation. These are implemented as Python objects
that allow the search space module to be extensible for different applications. The different choices for a given variable node are
specified by using the add_op method. These choices can be any
set of Dense or Connect operations; the former creates a Keras [26]
dense layer and the latter creates skip connections. After a neural
architecture is generated, the corresponding Keras model is created automatically for training and inference.

The analytics module of the software can be used to analyze
the data obtained from NAS. This module parses the logs from the
NAS to extract the reward trajectory over time and to find the best
architectures, worker utilization from the Balsam database, and
number of unique architectures evaluated.

5 EXPERIMENTAL RESULTS

For the NAS search, we used Theta, a 4,392-node, 11.69-petaflop
Cray XC40–based supercomputer at the Argonne Leadership Com-
puting Facility (ALCF). Each node of Theta is a 64-core Intel Knights
Landing (KNL) processor with 16 GB of in-package memory, 192
GB of DDR4 memory, and a 128 GB SSD. The compute nodes are
interconnected by using an Aries fabric with a file system capacity
of 10 PB.

The reward estimation for a given architecture uses only a single
KNL node (no distributed learning) with the number of training
epochs set to 1 and timeout set to 10 minutes.

The reward estimation for a generated architecture comprises
two stages: training and validation. For Combo, the training is per-
formed by using only 10% of the training data. For Uno and NT3,
since the data sizes are smaller, the full training data are used. The
reward is computed by evaluating the trained model on the valida-
tion data set. For Combo and Uno, we use $R^2$ value as the reward;
for NT3, we use classification accuracy (ACC). While we focus on
accuracy in this paper, other metrics can be specified, such as model
size, training time, and inference time for a fixed accuracy using a
custom reward function. To increase the exploration among agents,
we used random weight initialization in the DNN training using
agent-specific seeds during the reward estimation. Consequently,
different agents generating the same architecture can have different
rewards. For the policy network and value networks, we used a
single-layer LSTM with 32 units and trained them with epochs=4,
clip=0.2, and learning_rate=0.001, respectively.

Once the NAS search was completed on Theta, we selected the
top 50 DNN architectures from the search based on the estimated
reward values. We performed post-training, where we trained the
The same values were used in the manually designed networks. A3C has a faster learning capability and a better system utilization than does A2C. In this section, we show that in spite of gradient staleness issue, the small search space. The utilization of the RDM on the small search space.

Figure 4: Search trajectory showing reward over time for A3C, A2C, and RDM on the small search space.

DNNs for a larger number of epochs (20 for all experiments), without a timeout, and on the full training data. Running post-training on the KNL nodes was slower; therefore we used Cooley, a GPU cluster at the ALCF. Cooley has 126 compute nodes; each node has 12 CPU cores, one NVIDIA Tesla K80 dual-GPU card, with 24 GB of GPU memory and 384 GB of DDR3 CPU memory. The manually designed Combo network took 2215.13 seconds and 705.26 seconds for training on KNL and K80 GPUs, respectively. We ran 50 models on 25 GPUs with two model trainings per K80 dual-GPU card, one per GPU. For both the reward estimation and post-training, we used the Adam optimizer with a default learning rate of 0.001. The batch size was set to 256, 32, and 20 for Combo, Uno, and NT3, respectively. The same values were used in the manually designed networks.

We evaluated the generated architectures after post-training with respect to three metrics: accuracy ratio ($R^2/R^2_b$ or $ACC/ACC_b$), given by the ratio of $R^2$ ($ACC$) of a NAS-generated architecture and the manually designed network for Combo and Uno (NT3); trainable parameters ratio ($P_b/P$), given by the ratio of number of trainable parameters of the manually designed network and the given NAS-generated architecture (this metric helps us evaluate the ability of NAS to build smaller networks, which have better generalization and fewer overfitting issues compared with larger networks; and training time ratio ($T_b/T$), given by the ratio of the post-training time (on a single NVIDIA Tesla K80 GPU) of the manually designed network and the given NAS-generated architecture (this metric allows us to evaluate the ability of the NAS to find faster-to-train networks, which are useful for hyperparameter search and subsequent training with additional data).

The Theta environment consists of Cray Python 3.6.1, TensorFlow 1.13.1 [11]. Based on ALCF recommendations, we used the following environment variable settings to increase the performance of TensorFlow:

- `KMP_BLOCKTIME=0`
- `KMP_AFFINITY=granularity=fine,compact,1,0`
- `intra_op_parallelism_threads=62`

The Cooley environment consists of Intel Python 3.6.5, Tensorflow-GPU 1.13.1.

5.1 Evaluation of the search strategy

In this section, we show that in spite of gradient staleness issue, A3C has a faster learning capability and a better system utilization than does A2C, synchronized gradient updates and the consequent node idleness adversely affect the efficacy of A2C.

We evaluated the learning and convergence capabilities of A3C and A2C by comparing them with random search (RDM), where agents perform actions at random and will not compute and synchronize gradients with the parameter server. This comparison was to ensure that the search space was the same as A3C, A2C, and RDM and allowed us to evaluate the search capabilities of A3C and A2C with all other settings remaining constant. We used 256 Theta nodes for A3C, A2C, and RDM with 21 agents and 11 workers per agent:

21 agent nodes, 231 worker nodes, 1 Balsam workflow node, and 3 unused nodes.

Figure 4 shows rewards obtained over time for A3C, A2C, and RDM. We observe that A3C outperforms A2C and RDM with respect to both time and rewards obtained. A3C exhibits a faster learning trajectory than does A2C and reaches a higher reward in a shorter wall-clock time. A3C reaches reward values of 0.5 and 0.4 in approximately 70 and 35 minutes for Combo and Uno, respectively, after which the increase in the reward values is small. On Combo and NT3, A3C ends in 250 and 285 minutes, respectively, because all the agents generate the same architecture for which the agent-specific cache returns the same reward value. We detected this and stopped the search since it could not proceed in a meaningful way. On Uno, A3C generates different architectures and does not end before the wall-clock time limit. A2C shows a slower learning trajectory; it eventually reaches the reward values of A3C on Combo and Uno, but its reward value on NT3 is poor. As expected, RDM shows neither learning capability nor the ability to reach higher reward values. On NT3, we found an oscillatory behavior with A3C toward the end. After finding higher rewards, A3C did not converge as expected. After a closer examination of the architectures and their reward values, we found that although the agents are producing similar architectures, the reward estimation is sensitive to a random initializer with one training epoch and a batch size of 20. Consequently, the search produced by two different agents had significantly different rewards (e.g., 1.0 and 0.4).

Figure 5 shows the utilization over time for A3C, A2C, and RDM on the small search space. The utilization of the RDM on Combo stays at 1.0 in the initial search stages, but after that it averages 0.75. Although RDM lends itself to an entirely asynchronous search, the estimation of $M$ rewards per agent was blocking in our implementation. This per-agent synchronization, combined with variability.

\[ \text{We want to set number of agents} \times \text{number of workers}: 21 \text{ agents and 11 workers} \]

\[ \text{satisfy the constraint with minimal unused nodes. We requested 256 instead of 253 to get 6 hours of running time.} \]
of the reward estimation times, leads to suboptimal utilization. The utilization of A3C is similar to that of RDM until 100 minutes, after which there is a steady decrease due to an increase in the caching effect; this is just a manifestation of the convergence of A3C, which stops after 160 minutes.

On Uno, the utilization of RDM becomes high, with an average of 0.9. This is because randomly sampled DNNs in this space have a smaller variance of reward estimation times. The utilization of A3C is similar to that of RDM in the beginning of the search, but it decreases after 50 minutes because it learns to generate architectures that have a shorter training time with higher rewards. On NT3, the utilizations of RDM and A2C are similar to that of Combo but with even lower values because per batch several architectures have a shorter reward estimation time. The utilization of A2C shows a sawtooth shape; because of the synchronous nature, at the start of each batch the utilization goes to 1, then drops off and becomes zero when all agents finish their batch evaluation.

5.2 Comparison of A3C-generated networks with manually designed networks

Here, we show that A3C discovers architectures that have significantly fewer trainable parameters, shorter training time, and accuracy similar to or higher than those of manually designed architectures.

Figure 6 shows the search trajectory and utilization of A3C on Combo with the large search space. We observe that A3C finds higher rewards faster than do A2C and RDM. The utilization of A3C is similar to that of RDM (75% average) until 200 minutes, after which there is a gradual decrease because of the caching effect. Nevertheless, the search did not converge and stop as it did in the small search space.

Figure 7 shows the post-training results of A3C on the 50 best architectures selected based on the estimated reward during the NAS. From the accuracy perspective, on Combo, five A3C-generated architectures have

Figure 8: Post-training results on top-50 A3C architectures from the large search space run on 256 nodes

Figure 7 shows the post-training results of A3C on the top 50 A3C architectures from the small search space run on 256 nodes. Accuracy ratios $(R^2/R^2_b, Acc/Acc_b) > 1.0$ indicate a A3C-generated architecture outperforming the manually designed network. Trainable parameter ratios $(P_b/P) > 1.0$ indicate that a A3C-generated architecture has fewer trainable parameters than the manually designed network. Training time ratios $(T_b/T) > 1.0$ indicate that a A3C-generated architecture is faster than the manually designed network.
architectures obtain $R^2$ values that are competitive with the manually designed network ($R^2_b/R^2_0 > 0.98$); on Uno, more than forty A3C-generated architectures obtain $R^2$ values that are better than the manually designed network value ($R^2_b/R^2_0 > 1.0$); on NT3, three A3C-generated architectures obtain accuracy values that are higher than that of the manually designed network ($ACC/ACC_0 > 1.0$). From the trainable parameter ratio viewpoint, A3C-generated architectures completely outperform the manually designed networks on all three data sets. On Combo, A3C-generated architectures have 5x to 15x fewer trainable parameters than the manually designed network has; on Uno this is between 2x to 20x; on NT3, A3C-generated architectures have up to 800x fewer parameters than the manually designed network has. The significant reduction in the number of trainable parameters is reflected in the training time ratio, where we observed up to 2.5x speedup for Combo and Uno and up to 20x for NT3.

Figure 8 shows the post-training results of A3C with the large search space on Combo and Uno. On Combo, use of the large search space allowed A3C to generate a number of architectures with accuracy values higher than those generated with the small search space. Among them, five architectures obtained $R^2_b/R^2_0 > 0.99$; one was better than the manually designed network. The large search space increases the number of training parameters and training time significantly. Nevertheless, on Uno, we found that the larger search space decreases the accuracy values significantly, which can be attributed to the overparameterization given the relatively small amount of data, and additional improvement in accuracy was not observed after a certain number of epochs.

5.3 Scaling A3C on Combo with large search space

In this section, we demonstrate that increasing the number of agents and keeping the number of workers to a smaller value in A3C results in better scalability and improvement in accuracy.

We ran A3C on Combo with the large search space on 512 and 1,024 KNL nodes. We studied two approaches to scaling. In the first approach, called worker scaling, we fixed the number of agents at 21 and varied the number of workers per agent. For 512 and 1,024 nodes, we tested 23 and 47 workers per agent, respectively. In the second approach, called agent scaling, we fixed the number of workers per agent at 11 and increased the number of agents. For 512 and 1,024 nodes, we used 42 and 85 agents, respectively.

The utilization of A3C is shown in Fig. 9. We observe that scaling the number of agents is more efficient than scaling the number of workers per agent. In particular, the utilization values of agent scaling, 512(a) and 1,024(a), are similar to those measured at 256 nodes; there is no significant loss in utilization by going to higher node counts. On the other hand, utilization suffers as the number of workers per agent is increased. The reason is that the worker evaluations are batch synchronous and the increase in workers results in an increase in the number of idle nodes within a batch. The decreasing overall trend in utilization can be attributed to the increased cache effect.

Figure 10 shows the post-training results of the 50 best architectures from the 512- and 1,024-node agent scaling experiments.  

![Figure 9: Utilization of A3C on Combo with the large search space run on 512 and 1,024 nodes with agent and worker scaling; 256 nodes are used as reference.](image)

![Figure 10: Post-training results of A3C on Combo with the large search space run on 512 and 1,024 nodes with agent scaling.](image)

5.4 Impact of fidelity in reward estimation

Here, we show that, at scale, the fidelity of the reward estimation affects agent learning behavior in different ways and can generate diverse architectures.

We analyzed the impact of fidelity in the reward estimation strategy by increasing the training data size in A3C from the default of 10% to 20%, 30%, and 40% on Combo. We ran the experiments on 256 nodes and used the default values for the training epochs and the timeout.

Figure 11 shows the search trajectory of A3C. We can observe that on 10%, 20%, and 30% training data, A3C generates architectures with high rewards within 80 minutes. With 40% training data, the improvement in the reward is slow. The reason is that the large

---

We did not use more than 1,024 nodes in this experiment because of a system policy limiting the total number of concurrent application launches to 1,000.
number of architectures generated by A3C cannot complete training before the timeout. Consequently, it takes 80 minutes to reach reward values greater than 0. Nevertheless, it slowly learns to generate architectures that can be trained within the timeout—within 160 minutes, A3C with 40% of the training data reaches the reward values found by A3C with less training data.

The post-training results are shown in Fig. 12. As we increase the training data size in the reward estimation, we can observe a trend in which the best architectures generated by A3C have fewer trainable parameters and shorter post-training time. In the 10% case, the training time in reward estimation is not a bottleneck. Consequently, the agents generate networks with fewer trainable parameters to increase the reward, and the post-training time of the best architectures often exceeds that of the manually designed network. Increasing the training data size to 20% results in the best architectures that have smaller trainable parameters and longer post-training time than the manually designed network has. We found that the agents that can achieve faster rewards by using fewer parameters update the parameter server and bias the search. In the 30% case, the training time affects the best architectures. Consequently, several best architectures have fewer trainable parameters and shorter post-training time than the 10% and 20% cases have. In the 40% case, the training time in the reward estimation becomes a bottleneck. As a result, the agent learns to maximize the reward by generating architectures with faster training time in the reward estimation by using fewer trainable parameters.

### 5.5 Impact of randomness in A3C

The A3C strategy that we used in NAS is a randomized method. The randomness stems from several sources, including random weight initialization of the neural networks, asynchronicity, and stochastic gradient descent for reward estimation. Here, we analyze the impact of randomness on the search trajectory of A3C. We repeated A3C 10 times on the Combo benchmark with the small search space. The results are shown in Fig. 13. Given a time stamp, we compute 10%, 50% (median), and 90% quantiles from 10 values—this removes both the best and worst values (outliers) for a given time stamp. In the beginning of the search, the differences in the search trajectory of A3C are noticeable, where the quantiles of the reward range between 0.2 and 0.4. Nevertheless, the variations become smaller as the search progresses. At the end of the search, all the quantile values are close to 0, indicating that the search trajectories of different replications are similar and the randomness does not have a significant impact on the search trajectory of A3C.

### 5.6 Summary of the best A3C architectures

Table 1 summarizes the results of the best A3C-generated architectures with respect to the manually designed networks on three data sets. On Combo, the accuracy of the best A3C-generated architecture is slightly better than that of the manually designed network. However, it has 7.3x fewer trainable parameters and 2.5x faster training time. On Uno, the best A3C-generated architecture outperforms the manually designed network with respect to all three factors: trainable parameters, training time, and accuracy. The best NAS architecture obtained an $R^2$ value of 0.729 with 11.5x fewer trainable parameters and 2.5x faster training time than that of the manually designed network.
We refer the reader to [6, 17, 32] for a detailed exposition on NAS network is the first application of NAS for cancer predictive modeling. This is followed by recurrent neural nets for text classification tasks.

Methods, and reinforcement learning [12, 13, 16, 18, 19, 28, 36, 72, 74, 78, 86], gradient-based Bayesian optimization [21, 30, 41, 42, 65, 70, 80, 82], evolutionary algorithms, using evolutionary algorithms, but it does not show that A3C, despite gradient staleness due to asynchronous gradient update, can find high-performing DNNs in a short computation time. We evaluated the efficacy of the RL-based NAS at scale with respect to accuracy, training time, parameters, and reward estimation fidelity.

RL-based NAS scalability: In [88], RL-based NAS was scaled on 800 GPUs using 20 parameter servers, 100 agents, and 8 workers per agent. This approach was run for three to four weeks. In [89], a single RL agent generated 450 networks and used 450 GPUs for concurrent training across four days. In both these works, the primary goal was to demonstrate that the NAS can outperform manually designed networks on image and text classification tasks. We demonstrated RL-based NAS experiments on up to 1,024 KNL nodes and for a much shorter wall-clock time of 6 hours on cancer data.

Open source software: AutoKeras [39] is an open source automated machine learning package that uses Bayesian optimization and network morphism for NAS. The scalability of the package is limited because it is designed to run on single node with multiple GPUs that can evaluate few architectures in parallel. Microsoft’s Neural Network Intelligence [7] is a open source AutoML package designed primarily to tune the hyperparameters of a fixed DNN architecture by using different types of search methods; it lacks capabilities for architecture templates. Ray [35] is a open source high-performance distributed execution framework that has modules for RL and hyperparameter search but does not have support for NAS. AMLA [40] is a framework for implementing and deploying AutoML neural network generation algorithms. Nevertheless, it has not been demonstrated on benchmark applications or at scale. TPOT [58] optimizes scikit-learn [60], a library of classical machine learning algorithms, using evolutionary algorithms, but it does not have support for NAS. Our package differs from the existing ones with respect to customized NAS search space for cancer data and scalability.

7 CONCLUSION AND FUTURE WORK

We developed scalable RL-based NAS to automate DNN model development for a class of cancer data. We designed a NAS search space that takes into account characteristics specific to nonimage and nontext cancer data. We scaled the proximal policy optimization, a state-of-the-art RL approach, using a manager-worker approach on up to 1,024 Intel Knights Landing nodes and evaluated its efficacy using cancer DL benchmarks. We demonstrated the efficacy of this method at scale and showed that the asynchronous actor critic method (A3C) outperforms its synchronous and random variants. The results showed that A3C can discover architectures that have significantly fewer training parameters, shorter training time, and accuracy similar to or higher than those of manually designed architectures. We experimented with the volume of training data in reward estimation, analyzed the impact of fidelity in reward estimation, and found that the NAS can outperform manually designed networks on image and text classification tasks.
estimation on the agent learning capabilities, and showed that it can be used to discover different types of network architectures.

Our future work will include applying NAS on a broader class of cancer data, conducting an architecture search for transformer and attention-based networks [52, 79], adapting NAS for multiple objectives, developing adaptive reward estimation approaches, developing multiparameter servers to improve scalability, integrating hyperparameter search approaches, and comparing our approach with extremely scalable evolutionary approaches such as MENDEL [59, 86] and Bayesian optimization methods [71]. Our NAS framework is designed to be flexible for developing surrogate DNN models for tabular data. We will explore NAS for reduced-order modeling in scientific application areas such as climate and fluid dynamics simulations.

NAS has the potential to accelerate cancer deep learning research. A scalable open-source NAS package such as ours can allow cancer researchers to automate neural architecture discovery using HPC resources and to experiment with diverse DNN architectures. This will be of paramount importance in order to tackle cancer as we incorporate more diverse and complex data sets.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility.

REFERENCES

[1] [n. d.]. AutoML Workshops. https://www.ml4aad.org/automl/
[2] [n. d.]. CANDLE Exascale Computing Program Application. https://github.com/ECN-CANDELBenchmarks/tree/master/Pilot1/Combo
[3] [n. d.]. Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer. https://candle.els.aon.gov
[4] [n. d.]. Literature on Neural Architecture Search. https://www.ml4aad.org/automl/literature-on-neural-architecture-search/
[5] [n. d.]. NT3 Benchmark. https://github.com/ECN-CANDELBenchmarks/tree/master/Pilot1/NT3
[6] [n. d.]. Uno Benchmark. https://github.com/ECN-CANDELBenchmarks/tree/master/Pilot1/Uno
[7] [n. d.]. World Health Organization: Cancer key facts. https://www.who.int/continuous-concurrency-analysis. CoRR abs/1802.09941 (Feb. 2018).
[8] Gabriel Bender, Pieter-Jan Kindermans, Barrett Zoph, Vijay Vasudevan, and Quoc Le. 2018. Understanding and simplifying one-shot architecture search. In International Conference on Machine Learning. 549–558.
[9] Gabriel Bender, Pieter-Jan Kindermans, Barrett Zoph, Vijay Vasudevan, and Quoc Le. 2018. Understanding and Simplifying One-Shot Architecture Search. In Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research), Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 550–559.
[10] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. Journal of Machine Learning Research 13, Feb (2012), 281–305.
[11] James Bergstra, Dan Yamins, and David D Cox. 2013. Hyperopt: A Python library for optimizing the hyperparameters of machine learning algorithms. In Proceedings of the 12th Python in Science Conference. 13–20.
[12] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:arXiv:1606.01540
[13] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. 2017. Geometric deep learning: Going beyond euclidean data. IEEE Signal Processing Magazine 34, 4 (2017), 18–42.
[14] Yukang Chen, Qian Zhang, Chang Huang, Lisen Mu, Gaofeng Meng, and Xing-Gang Wang. 2018. Reinforced Evolutionary Neural Architecture Search. arXiv preprint 1808.00193 (2018).
[15] François Chollet et al. 2017. Keras (2015).
[16] Patrik Chrabaszcz, Ilya Loshchilov, and Frank Hutter. 2017. A downsampled variant of ImageNet as an alternative to the CIFAR datasets. arXiv preprint 1707.08819 (2017).
[17] Xiaoxiang Chu, Bo Zhang, Hailong Ma, Ruijun Xu, Jixiang Li, and Qingyuan Li. 2019. Fast, Accurate and Lightweight Super-Resolution with Neural Architecture Search. arXiv preprint 1901.07261 (2019).
[18] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. 2017. OpenAI baselines. https://github.com/openai/baselines.
[19] Georgi Dikov, Patrick van der Smagt, and Justin Bayer. 2019. Bayesian Learning of Neural Network Architectures. arXiv preprint 1901.04436 (2019).
[20] Jesse R Dixon, Jie Xu, Vishnu Dileep, Ye Zhan, Fan Song, Victoria T Le, Galip Girikcan Yardumcu, Abhijit Kabra, Darrin V Bann, Yanli Wang, et al. 2018. Integrative detection and analysis of structural variation in cancer genomes. Nature Genetics 50, 10 (2018), 1388.
[21] Thomas Eldon, Jan Hendrik Metzen, and Frank Hutter. 2018. Neural architecture search: A survey. arXiv preprint 1808.05377 (2018).
[22] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. 2014. Do we need hundreds of classifiers to solve real world classification problems? The Journal of Machine Learning Research 15, 1 (2014), 3133–3181.
[23] Dario Floreano, Peter Durr, and Claudia Mattiussi. 2008. Neuroevolution: From architectures to learning. Evolutionary Intelligence 1, 1 (2008), 47–62.
[24] Dan Grundman, Lucian Borchani, Gabriel AD Lopes, and Robert Babuska. 2012. A survey of actor-critic reinforcement learning: SStandard and natural policy gradients. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42, 6 (2012), 1291–1307.
[25] Minghao Guo, Zhao Zheng, Wei Wu, Dahua Lin, and Junjie Yan. 2018. IRLAS: Inverse Reinforcement Learning for Architecture Search. arXiv preprint 1812.05285 (2018).
[26] F. Hutter, L. Kotthoff, and J. Vanschoren (Eds.). 2019. Automated Machine Learning: Methods, Systems, Challenges. Springer International Publishing.
[27] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ari Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. 2017. Population Based Training of Neural Networks. arXiv:1711.09846 (2017).
[28] Haifeng Jin, Qingguang Song, and Xia Hu. 2018. Efficient neural architecture search with network morphism. arXiv preprint 1806.10282 (2018).
[29] Purushotham Kamath, Abhishek Singh, and Debo Dutta. [n. d.]. AMLA: An AutoML Framework for Neural Network Design. (n. d.).
[30] Kirthivasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. 2018. Neural architecture search with Bayesian optimisation and optimal transport. In Advances in Neural Information Processing Systems. 2020–2029.
[31] A. Klein, S. Falkner, N. Mansur, and F. Hutter. 2017. RoBO: A flexible and robust Bayesian Optimization framework in Python. In NeurIPS 2017 Bayesian Optimization Workshop.
[32] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. 2016. Learning curve prediction with Bayesian neural networks. (2016).
