Accelerating DNN Architecture Search at Scale Using Selective Weight Transfer

Hongyuan Liu†, Bogdan Nicolae*, Sheng Di*, Franck Cappello*, Adwait Jog†
†*William & Mary, USA*
Email: hliu08@email.wm.edu, ajog@wm.edu
**Argonne National Laboratory, USA*
Email: {bnicolae,sdi1}@anl.gov, cappello@mcs.anl.gov

***Abstract*—Deep learning applications are rapidly gaining traction both in industry and scientific computing. Unsurprisingly, there has been significant interest in adopting deep learning at a very large scale on supercomputing infrastructures for a variety of scientific applications. A key issue in this context is how to find an appropriate model architecture that is suitable to solve the problem. We call this the neural architecture search (NAS) problem. Over time, many automated approaches have been proposed that can explore a large number of candidate models. However, this remains a time-consuming and resource expensive process: the candidates are often trained from scratch for a small number of epochs in order to obtain a set of top-K best performers, which are fully trained in a second phase. To address this problem, we propose a novel method that leverages checkpoints of previously discovered candidates to accelerate NAS. Based on the observation that the candidates feature high structural similarity, we propose the idea that new candidates need not be trained starting from random weights, but rather from the weights of similar layers of previously evaluated candidates. Thanks to this approach, the convergence of the candidate models can be significantly accelerated and produces candidates that are statistically better based on the objective metrics. Furthermore, once the top-K models are identified, our approach provides a significant speed-up (1.4∼1.5× on the average) for the full training.**

***Index Terms*—Deep Learning; Neural Architecture Search; Checkpointing**

I. Introduction

Deep learning applications are rapidly gaining traction both in industry and scientific computing: vision [1], [2], [3], financial technology [4], [5], business intelligence [6], [7], fusion energy science [8], computational fluid dynamics [9], and cancer research [10].

A key driver for this trend has been the unprecedented accumulation of big data, which exposes plentiful learning opportunities thanks to its massive size and variety. Unsurprisingly, there has been significant interest to adopt deep learning at a very large scale on supercomputing infrastructures. In this context, deep learning workflows are becoming increasingly complex, featuring multiple steps: pre-processing and augmentation of the training data, finding a suitable DNN model architecture, training, post-processing (e.g., sensitivity analysis to understand the robustness of the model).

A particularly laborious step is finding a suitable DNN model architecture. This has traditionally been performed by domain experts, using trial-and-error approaches. However, with increasing complexity, such an approach is not feasible anymore. Over time, automated approaches called NAS (Neural Architecture Search) [11] have been proposed as an alternative, however, this is a time-consuming and resource expensive process.

A typical NAS explores a large number of candidate models from a search space that is based on a set of specifications that define the set of possible choices that can be combined to obtain valid candidates. This process happens in two steps: (1) the candidates are trained from scratch for a small number of epochs in order to obtain a set of top-K best models; (2) the top-K best models are fully trained until they converge (or otherwise reach a maximum number of epochs) [12], [13], [14], [15], [16]. Finally, the domain experts pick the most suitable models among the fully trained top-K best models based on a trade-off between various criteria (e.g., accuracy, size of the model, inference speed).

A key problem in this context is the "training from scratch" approach adopted by state-of-the-art NAS, which presents two challenges. First, the accuracy after a small number of epochs may not be representative of the accuracy obtained after full training. Therefore, the score of a candidate may not accurately reflect its long-term potential, negatively impacting the ranking of the top-K best performers. Second, the training of the top-K best models is time-consuming, because each model eventually needs to be fully trained from scratch until it converges, which normally takes a large number of epochs.

To address this problem, we propose an alternative approach that is based on a key insight: candidates from the same search space are *functionally* and *structurally* similar with each other. Functionally, because they have the same target problem to solve (e.g., building a predictive model for a fixed dataset), and structurally, because many candidates differ only in a small number of choices (e.g., they may have a large number of identical layers or the beginning of convolutional layers have the same filters).

Starting from this insight, we propose the idea that new candidates need not be trained starting from random weights, but rather from the weights of similar layers of previously evaluated candidates. To this end, we checkpoint the candidate models after evaluating their scores and use them as *providers* for new candidates (*receivers*) by transferring the weights

of similar layers. As a consequence, we face two important challenges: first, it is not clear under what circumstances weight transfer accelerates the convergence of the receiver model. If these circumstances are not right, the receiver model may converge slower than being trained from random weights. Second, if weight transfer is beneficial, it is not clear what provider would bring the maximum benefit. This challenge increases in difficulty as more candidates are explored, because all of them could become potential providers.

Our work aims to address the aforementioned challenges. We solve the first challenge by proposing two simple yet efficient weight transfer techniques that feature different time complexities and transferring scope. We address the second challenge by revisiting existing NAS strategies and leveraging synergies that allow us to choose the provider models for weight transfer in a straightforward fashion. To the best of our knowledge, in the context of NAS, no prior work has considered this avenue. We summarize our contributions as follows:

- We propose two efficient techniques: LP (longest prefix) and LCS (longest common subsequence) for transferring weights selectively from a provider model to a receiver model (with a potentially different structure) that are specifically designed for NAS (Section IV).
- We study the circumstances under which weight transfer from the provider model to the receiver model is beneficial and establish criteria for the selection of the provider model (Section V).
- We integrate the weight transfer techniques and provider selection strategy into existing NAS approaches in a lightweight fashion (Section VI).
- We evaluate our proposal using 32 GPUs and four deep learning applications from various domains. The results show that our approach can accelerate the convergence of the models significantly, thereby enabling a full training of the top-K best performers 1.4$\sim$1.5$\times$ faster than training from scratch. Furthermore, our approach discovers better models based on the objective metrics (Section VIII).

II. Background

In this section, we introduce the fundamentals of Neural Architecture Search (NAS). In a nutshell, NAS is defined by three major components: (1) search space, (2) search strategy, and (3) candidate estimation.

Search Space: Search space is the set of rules that specify all possible choices that can be combined to obtain valid candidates. It is typically defined by domain experts empirically. More formally, it is a graph (V, E), where V contains p input nodes, one or multiple output nodes, and k *variable nodes*. Each *variable node* contains a set of valid choices. For example, a variable node may choose from a skip connection, a dense layer with 50 neurons and a ReLU activation function, a dense layer with 10 neurons with a sigmoid activation function, or a layer with 50% dropout, which can be denoted as $Identity$, $Dense(50, relu)$, $Dense(10, sig)$, and $Dropout(0.5)$, respectively. E encodes the connections between the nodes, with $E \subseteq V \times V$.

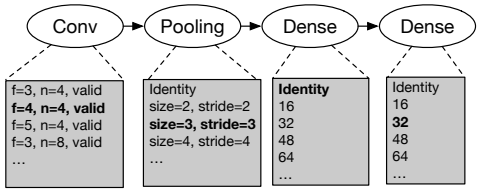


Fig. 1: A simplified search space for NT3. Circles show variable nodes; Boxes show the operations of variable nodes.

Figure 1 shows the simplified search space for NT3 [17], a cancer research application we evaluate in our paper. Each candidate model has four layers, each of which has a corresponding variable node (represented as a circle) that is associated with a set of choices (shown in shaded boxes). A candidate model is generated by fixing the choices of each variable node. If each choice is denoted by its index in the list of choices, then a candidate model can be uniquely identified by a sequence of indices, which we denote as *architecture sequence*. For example, the choices highlighted in a bold font in Figure 1) form the architecture sequence $[1, 2, 0, 2]$.

Search Strategy: Search strategy guides the exploration of the search space by deciding what new candidate models to try next. One simple example is random search, where the candidate models are always selected randomly. Other alternative approaches also have been proposed such as Bayesian optimization [18], [19], evolutionary methods [20], [21], [22], [23], [24], [25], and reinforcement learning [13], [26].

Candidate Estimation: Candidate estimation assigns a score to the candidate, and the score indicates how promising the candidate is. While it is possible to fully train the candidate models to get their actual objective metrics (e.g., validation accuracy), this approach is not feasible in practice due to prohibitively expensive amount of time and computational resources. Therefore, candidate models need to be evaluated quickly by means of partial training. For the purpose of this work, partial training means to use a small number of epochs [12], [13], after which we evaluate the objective metrics to obtain the score. However, it is important to note our proposal is general and can be applied to other estimation approaches, because they suffer from similar issues (e.g., using a subset of the original dataset [27], a small proxy of the network [28], or a smaller proxy dataset [29]).

III. Motivation

Prior work [12], [13], [14], [15], [16] that estimates the candidate models by training them from scratch for a few epochs suffers from inaccurate candidate estimation due to slow convergence. Our work aims to address this issue by means of weight transfer. To understand why this works, consider an extreme case where all candidate models in the search space are *identical*. In this case, if we checkpoint and evaluate a candidate model after a few epochs, initializing the weights of the next candidate from the checkpoint is equivalent to resuming the training of the provider model. In other words,

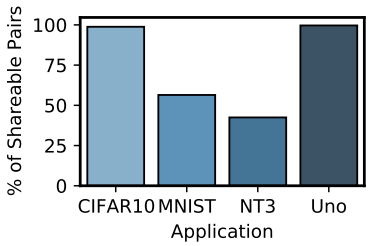


Fig. 2: A large portion of candidate models are structurally similar.

we are training the new candidate for two times more epochs, which has two advantages: (1) it increases the accuracy of the estimation; (2) if the candidate is selected among the top-K best performers, training it fully until it converges will be much faster.

It is important to note that the models generated by the same search space are similar both functionally (because they are part of a family of related models that aim to solve the *same* problem using the same training data) and structurally (because their architecture sequences overlap to a large degree). Therefore, in the general case, we expect that the weight transfer will lead to a transfer of the learning patterns, which, just like the case of identical models, will lead to faster convergence and more accurate estimations. Moreover, we expect the effectiveness of the weight transfer to increase for an increasingly higher degree of overlap between the architecture sequences of the provider and receiver models.

To verify this intuition, our first step is to study the structural similarity of the models. Specifically, we use DEEPHYPER [13] to generate NAS traces of four applications, each of which contains at least 672 candidate models obtained by searching for 16 node-hours in our clusters (by training each candidate model from scratch for one epoch). The applications and experimental setup are presented in greater detail in Section VII.

We randomly sample 10,000 pairs from the traces without replacement and count the number of pairs that have at least one tensor with an identical shape. Figure 2 shows the percentage of pairs that respect this condition (which we refer to as "shareable"). CIFAR-10 and Uno have almost 100% percent of shareable pairs, while MNIST and NT3 also have 54% and, respectively, 40% of shareable pairs. Thus, for each new candidate model, there is a large number of potential provider models that can be considered for weight transfer.

However, just because there is a large number of potential provider models does not mean the weight transfer is trivial to perform, not that it is beneficial. First, for each layer of the provider model, there can be a large number of similarly shaped layers in the receiver model for which weight transfer is possible, which leads to a large number of possible combinations. Second, a random transfer between two similarly shaped layers does not consider their functionality, which may lead to anomalies that cause the model to converge slower or not at all compared with a random weight initialization. Therefore, our goal is to design weight transfer strategies that are both fast and that take structural patterns into consideration to maximize the likelihood of beneficial weight transfers.

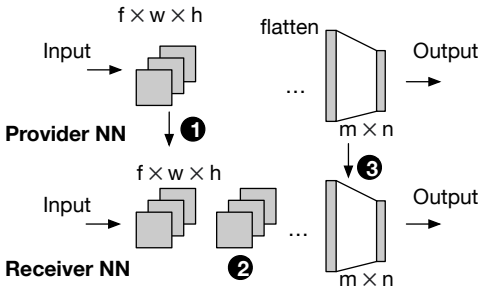


Fig. 3: Illustrating weight transfer mechanisms of two convolutional neural networks

IV. WEIGHT TRANSFER PROPOSAL

In this section, we propose two weight transfer techniques that are both quick to evaluate and that leverage the structure of the models to maximize the impact of weight transfer on the convergence of the receiving model.

A. Tensor Matching

DNN models are typically expressed using tensor operators, which form a computation graph where each node is a mathematical tensor operator (e.g., matrix multiplication, convolution, etc.) [30]. The input nodes of the computation graph do not have predecessors, while the output nodes do not have successors.

A provider and a receiver model can have many tensors of identical shapes (which we refer to as *transferable*). Therefore, finding a mapping of the transferable tensors from the provider to the receiver is equivalent to mapping sub-topologies of the two corresponding computation graphs, which is related to the graph homomorphism problem, known to be NP-complete.

To reduce the computational complexity, we need to simplify the mapping problem. To this end, similarly to how we define the architecture sequence as the string of choices taken by the variable nodes, we define the shape sequence as the string of tensor shapes corresponding to a computation graph. Thus, the mapping problem can be cast as a string matching problem where the elements of the shape sequence corresponding to the provider model are mapped to their counterparts of the receiver model.

Figure 3 shows an illustrative example of a convolutional NN and its shape sequence. The neural network begins with $f \times w \times h$ filters, followed by several layers such as pooling layer, batch normalization layer, or another convolutional layer (omitted in the figure). Before feeding into the dense layer with m neurons, the feature volume is flattened to a m-length vector. The last dense layer has n neurons, so the shape of the last tensor is (m, n). The shape sequence of the depicted NN is `[(f, w, h), ..., (m, n)]`.

In this context, we propose to use two string matching heuristics: longest prefix (LP) and longest common subsequence (LCS). They differ both in terms of performance and impact on the convergence of the receiver model.

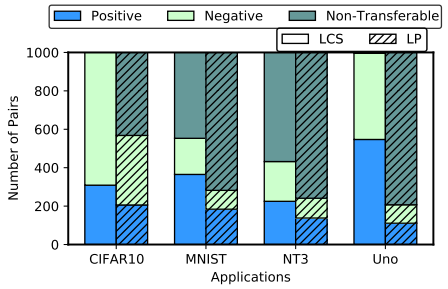


Fig. 4: Illustrating the scope and the effectiveness of LP and LCS for random pairs of provider and receiver models. 1) We have a large portion of transferable pairs; 2) Weight transfer from a random provider model may lead to slower convergence (e.g., CIFAR-10).

Longest Prefix (LP): Empirically, the first layer is often considered shareable in the machine learning community [31], [32]. For example, the first convolutional layer often identifies coarse-grained patterns in an image, which can be shared if there are common patterns in the datasets [33]. Therefore, we assume that as long as the beginning layers have identical shapes, weight transfer will be beneficial, hence the proposal to use the longest prefix of the shape sequences. As an example, Figure 3 (❶) illustrates the tensor that could be transferred using LP. However, since the receiver model has an additional convolutional layer (❷), even if the shape of the last dense layer is the same between the provider and the receiver (❸), LP cannot transfer the weights of the last dense layer. The time complexity of calculating an LP of two shape sequences is $O(\min(n, m))$, where n and m are lengths of the two sequences.

Longest Common Subsequence (LCS): In many cases, the addition of an extra layer does not change the function of the neighboring layers. However, LP will not be able to capture such cases. To address this issue, we propose to compute the longest common subsequence of the shape sequences, which is well suited to handle insertions and additions. For example, in Figure 3, the LCS of the provider and the receiver models contains both (❶) and (❷) in the same order. We use dynamic programming to implement LCS [34] for the two shape sequences, whose time complexity is $O(nm)$, where n and m are lengths of the two sequences. Note that LP is a subset of LCS, therefore LCS will always transfer at least as many tensors as LP.

B. Scope and Effectiveness of Weight Transfer

Next, we evaluate the scope and the effectiveness of both LP and LCS. We focus on two questions:

1) How likely is it when the LP or LCS is non-empty for a pair of provider and receiver models (i.e., they are a *transferable* pair)?
2) Can our weight transfer techniques accelerate the convergence of the receiver model?

Experimental Setup. We sample 1,000 pairs of provider and receiver models uniformly at random from the DEEPHYPER traces (discussed in Section III) for each of the four applications. We train each receiver model in two ways: (1) starting from random weights and (2) starting from the weights of the provider model for the layers that are included in LP and LCS, and from random weights for the rest of the layers. The training lasts for one epoch. If after one epoch, the objective metrics (validation accuracy) are better when the receiver is trained from random weight initialization, we consider this case as *negative*, because it makes the training convergence slower. Conversely, when the objective metrics are better after weight transfer, we consider the case as *positive*.

Figure 4 illustrates our results:

Scope of Transferable Pairs. We observe that the transferable pairs are not uncommon, even if we select the provider model randomly in this experiment. For example, CIFAR-10 and Uno can have 100% transferable pairs using LCS to transfer the weights, while MNIST and NT3 have at least 42% transferable pairs. As we expected, the scope of transferable pairs using LP is less than LCS, but more than 20% of pairs are still transferable across all evaluated applications.

Effectiveness of Weight Transfer. Figure 4 also shows that weight transfer from a random provider model is not always beneficial. In CIFAR-10, the number of positive pairs is less than the number of negative pairs in both LCS and LP, indicating that *randomly* select a provider model to transfer from is likely harmful to the application. By contrast, MNIST has 65% and 64% percentage of positive pairs among the transferable pairs using LCS and LP, respectively. LCS and LP generate a similar level of positive transferrable pairs for NT3 and Uno, but all fall in the range of 53% to 57%.

In summary, by *randomly* picking a provider model and receiver model, we have a large percentage of transferable pairs. However, the weight transfer is not always guaranteed to accelerate the convergence of the receiver models. Thus, we need a strategy to decide what provider to pick for a receiver model. We will discuss such a strategy next.

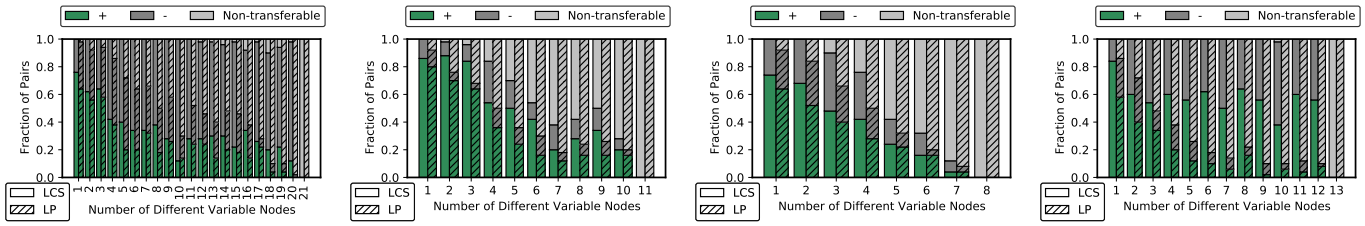
V. Provider Model Selection and Integration with the Search Strategy

In this section, we introduce an approach that integrates with the search strategy to provide a straightforward and highly efficient provider model selection and weight transfer.

A. Which provider model is suitable for weight transfer?

Our key insight (discussed in Section III) is that weight transfer from a *similar* model is likely to accelerate the convergence of the receiver model. Hence, we use the number of different variable node choices as the distance d to measure the similarity of two models in the same search space. The larger d is, the less similar the two models are. The choices made by variable nodes are recorded in the *architecture sequences* (discussed in Section II). Specifically, we define $d = \sum arch\_seq_A \oplus arch\_seq_B$, where A and B are a pair of models in the same search space. For example, $d = 1$ for the architecture sequences $[1, 2, 3]$ and $[0, 2, 3]$, because only the first position (variable node) differs.

To evaluate the impact of d on the convergence of the receiver model, we group the pairs (obtained from the DEEPHYPER traces) together based on d and run a series of experiments



(a) CIFAR-10 (b) MNIST (c) NT3 (d) Uno

Fig. 5: The effect of transferring weights from a provider model with different similarities to the receiver model. If the provider model and the receiver model have a large overlap in their architecture sequences, transferring the weights from the provider model is likely to accelerate the convergence of the receiver model. In this figure, $+$ stands for positive pairs, and $-$ stands for negative pairs, which are introduced in Section IV-B.

similar to the ones discussed in Section IV-B, in which we train each receiver model using (1) random initialization, (2) LP weight transfer (and random initialization of non-LP layers); (3) LCS weight transfer (and random initialization of non-LP layers).

Figure 5 illustrates our results. In general, we observe that for an increasing d, the fraction of transferable pairs decreases (shown in light gray color), while the fraction of positive pairs also decreases (shown in green pairs). One exception is Uno with LCS, where the fraction of positive pairs only decreases marginally because many variable nodes in its search space contain the same set of operations. Notably, when d is small (e.g., less than 3), the number of positive pairs is significantly larger than the number of negative pairs. Comparing LP and LCS, we observe that LP (in bars with hatches) has a smaller portion of transferable pairs in general, but its positive rate is similar to LCS.

In summary, *transferring the weights from a provider model with a small d is likely to accelerate the convergence of a receiver model.* We will use these findings in the design of the selection strategy, which is discussed next.

B. Zoom on the Selection Strategy

NAS employs a scheduler process that runs the search strategy (responsible to propose new candidate models), and also maintains the metadata that describes the candidates explored so far and their scores. To select a provider model with a small d, the scheduler must iterate over the previous checkpointed candidates, which can introduce a significant overhead given the large number of checkpointed candidates.

To address this issue, we revisit a popular set of genetic (evolutionary) search algorithms [25], [22], [23], [24]. In particular, Algorithm 1 illustrates the basic idea of a regularized evolutionary algorithm [25]. The algorithm maintains the population that contains N evaluated candidates. When a new candidate is scored, it is pushed into the active population (Line 4). If the size of the population is larger than N, it pops the model that it pushed the earliest. The search strategy works by sampling S out of N models, choosing the best provider model out of the subset S, and then mutating it to generate a receiver model (child) (Line 8). The mutation is done by randomly modifying one of the choices in the architecture sequence. Therefore, d between the provider and the receiver

Algorithm 1 Integrating Weight Transfer Mechanisms with Genetic Algorithm

```
1: procedure GENETIC SEARCH STRATEGY
2:     P ⇐ ∅                          ▷ P is a population of evaluated candidate models.
3:     while NAS is within time budget do
4:         Get scored models from the evaluators and push them to P
5:         if |P| ≥ N then
6:             S ⇐ sample(P)                    ▷ S is a sampled subset of P
7:             parent ⇐ best of S in scores
8:             child ⇐ mutate a variable node choice from the architecture sequence
   of parent
9:             Initialize child with the weights of parent   ▷ d between the parent
   and the child is always one!
10:            Evaluate the score of child
11:        end if
12:    end while
13: end procedure
```

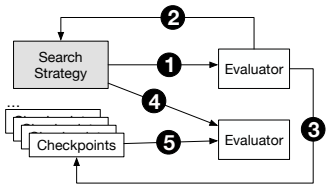


Fig. 6: Software Architecture of Our Implementation

model is always one. We take advantage of this situation to enhance Algorithm 1 to perform weight transfer whenever possible, which by construction will solve both the problem of deciding when to perform weight transfer (always, because $d = 1$ and identifying the provider candidate fast (it's the parent). Nevertheless, it is important to note that our approach can be extended to other search strategies, in which case the aforementioned issues are not straightforward.

VI. System Implementation

We build our approach on top of DeepHyper [13], [14], a state-of-the-art NAS framework that works for different domains. DeepHyper has three components, 1) problem, 2) search strategy, and 3) evaluator. The search space of an application is defined in the "problem" component. When NAS starts, the search strategy which works as a scheduler launches multiple evaluators. DeepHyper providers different backend for the evaluators such as Ray [35] evaluator, MPI evaluator, or Balsam [36] evaluator. The evaluators can be distributed to different nodes.

Figure 6 shows the architecture of our implementations. As discussed in Section V, we augment the regularized evolution

search strategy [25] to make use of LP and LCS based weight transfer. We use Ray-based evaluators such that each cluster node has a number of Ray evaluators equal to the number of GPUs. The scheduler runs the search strategy, which begins with generating random candidate models to be scored (❶) by the evaluators. When an evaluator finishes scoring a candidate model, it returns the results to the scheduler (❷), while also checkpoints the model on the parallel file system (❸). Since the efficiency of checkpointing is not the focus of this paper, we use a normal HDF5 format [37] to store the model checkpoints. When the search strategy has trained enough new candidates from scratch, it generates child models by mutating one variable node of the provider models acting as parents. The scheduler passes the architecture sequences of both parent and the child model to the evaluators (❹). Compared to the original DEEPHYPER implementation, the only additional overhead is due to passing the architecture sequence of the parent, so the overhead on the scheduler is negligible. To score a child model, each evaluator reads the checkpoint of the parent model from the parallel file system (❺).

VII. EVALUATION

This section provides an experimental evaluation of our proposal. To summarize, we aim to answer the following questions:

Q1. Can weight transfer accelerate the convergence of the candidate models? (Section VIII-A)

Q2. Can we speed up the full training for the top-K candidate models? (Section VIII-B)

Q3. How do the models discovered using weight transfer compare with those discovered by training from scratch? (Section VIII-C)

Q4. Are the proposed weight transfer techniques scalable? (Section VIII-E)

A. Applications and Search Spaces

Throughout our evaluations, we use four applications ranging from computer vision to cancer research. Table I shows a summary of our evaluated applications and their search spaces. The smallest search space contains three million models, which requires over 8,000 node hours to evaluate them all.

TABLE I: Summary of evaluated applications and their search spaces. CE: categorical cross-entropy; ACC: accuracy; Loss: loss function. Obj.: objective metrics. MAE: minimum absolute error. VN: Variable Node (introduced in Section II).

	Dataset Size		Search Space		Problem	
App.	Training	Validation	Size	#VNs	Loss	Obj.
CIFAR10	50K $\times$ 32^2 $\times$ 3	10K $\times$ $32^2$$\times$3	2558T	21	CE	ACC
MNIST	60K $\times$ 28^2	10K $\times$ 28^2	120M	11	CE	ACC
NT3	1120$\times$60483	280$\times$ 60483	3M	8	CE	ACC
Uno	9588$\times$ 1 9588$\times$ 942 9588$\times$5270 9588$\times$2048	2397$\times$ 1 2397$\times$942 2397$\times$5270 2397$\times$2048	302T	13	MAE	R^2

CIFAR-10. CIFAR-10 [38] dataset contains 60,000 images in 10 classes such as airplane, automobile, bird, etc. Each image is 32 $\times$ 32 with three channels. Inspired by the VGG networks [3], we stack three VGG blocks at the beginning of the search space, followed by three "Dense" variable nodes choosing an identity operation or a dense layer with a different number of neurons. Each block has six variable nodes of three types: 1) A "Convolution" variable node varies the number of filters, "valid" or "same" paddings, and whether it has a kernel regularizer (L2 with 0.0005 weight decay) of a convolutional layer; 2) A "Pooling" variable node chooses from an identity operation or a pooling layer with different sizes and strides. 3) A "BatchNorm" variable node chooses either apply or not to apply a batch normalization operator. A VGG block stacks Convolution, Pooling, and BatchNorm variable nodes and repeats them twice.

MNIST. MNIST [39] database of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples. Each image of it is 28 $\times$ 28 with one channel. We create a search space inspired by LeNet-5 [2]. There are five types of variable nodes in our search space: 1) Convolution, which selects a convolution layer with different filter sizes, the number of filters, and "valid" or "same" padding; 2) Activation, which selects an activation function from relu, tanh, or sigmoid; 3) Pooling, which selects an operation from identity and pooling layers with different sizes or strides from 2 to 5; 4) Dense, which selects an operation from identity or dense layers with 32, 64, ..., 512 neurons; 5) Dropout, which selects an operation from identity, or dropout 2%, 5%, 10%, 20%, 30%, 40%, 50% connections. The variable nodes are connected following this order: Convolution, Activation, Pooling, Convolution, Activation, Pooling, Dense, Activation, Dense, Activation, Dropout.

NT3. NT3 [17] is a benchmark from ECP CANDLE (Cancer Distributed Learning Environment) [10] that addresses a series of cancer research problems using large scale deep learning. NT3 classifies RNA-sequence gene-expression profiles into normal or tumor tissue categories. We use the same search space of NT3 defined by prior work [40]. Similar to MNIST, five categories of variable nodes in NT3 are connected following this order: Convolution, Activation, Pooling, Dense, Activation, Dropout, Dense, Activation, Dropout. Different from MNIST, the convolution is 1D for the gene-sequence data.

Uno. Uno [41], a benchmark that is also part of ECP CANDLE, integrates multiple data sources of 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. It predicts tumor dose-response across multiple data sources by a DNN for regression. We use the single drug paclitaxel, a simplified indicator, for this study. We use the same search space defined by prior work [40]. Three neural networks are set up in the beginning by stacking three variable nodes, each of which accepts an input dataset. Then, the outputs of the beginning three NNs are concatenated with the fourth input dataset, which serves as the input of the bottom NN. The bottom NN consists of four variable nodes. All variable nodes have mixed layer choices: Identity, a dense layer with 100, 500, or 1,000 neurons, or a dropout layer with 30%, 40%, and 50% dropout connections.

We use the Adam optimizer [42] for all applications. The learning rate is set to 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-7}$. The batch size is set to 64 for CIFAR-10 and MNIST, and set to 32 for NT3 and Uno as prior work [13].

B. Experimental Setup

TABLE II: Hardware Configurations of Our Experiments

HW.	Node Type A	Node Type B
CPU	4 × AMD EPYC 7742	Intel Xeon CPU E5-2620 v3
RAM	1 TB	384 GB
GPU	8 × NVIDIA Ampere A100 GPU	2 × NVIDIA Tesla K80
GPU Mem.	40 GB HBM2	12 GB GDDR5

We use two types of cluster nodes, referred to as A and B, which are shown in Table II. The experiments in Section VIII are performed on up to four type A nodes. The experiments in Section III, IV-B, and V are performed on type B nodes.

Our software environment is built with Python 3.7, Tensorflow [43] 2.4.1, DeepHyper [13] 0.2.2, Ray [35] 1.2.0 and Keras [44] 2.4.3.

C. Methodology

Our work proposes two weight transfer techniques and integrates them into an existing search strategy. To this end, we compare our approach with a baseline that always trains the candidates from scratch. Specifically, the baseline is DEEPHYPER with the regularized evolution [25] search strategy. The population size is set to 64 and the sample size is set to 32.

In addition to the baseline, we first modify the search strategy to make use of LP and LCS based weight transfer. Then, before training the candidate model, the evaluator finishes the following steps: 1) checks the parent's architecture sequence, 2) reads the checkpoint of the parent, 3) calculate LP/LCS between the parent and the current model, and 4) if they have shareable tensors, initialize the weights of the current model with the weights of the parent's model.

VIII. RESULTS

In this section, we discuss the experimental results.

A. Convergence Speed of Candidate Models

We study the evolution of the estimated objective metrics (scores) of the candidate models during NAS runtime. The settings we used are aligned with DEEPHYPER [13], i.e., all candidate models are trained for an epoch. We repeat the experiment five times and report the average. Each experiment evaluates a total of 400 candidate models. Since the evolution algorithm we use is randomized, we use different random seeds for the five experiments. The experiments may end at different times, therefore we focus only on the duration of the shortest experiment.

We group the time into 50-second slots[1] to smooth the fluctuations and emphasize the trends more clearly. Specifically, after a candidate model is evaluated and returns at time t with an associated score r, we plot the following point for it: $(50 \times \lceil t/50 \rceil, r)$. Multiple points can be in the same 50-second time slots, therefore we depict their mean with a 95% confidence interval (shown in bands).

Figure 7 shows the mean scores of the discovered models during NAS runtime. We make the following observations. 1) The curves of LP and LCS are significantly higher than the baseline in CIFAR-10, NT3, and Uno, after the beginning stage. As it is very easy to get high accuracy in MNIST, the scores acquired by LCS or LP are comparable to the baseline but with fewer fluctuations. 2) The scores from LCS are slightly higher than the scores from LP in CIFAR-10, and Uno. As NT3 has many fluctuations in scores, it is not clear whether LP or LCS works better. The reason for fluctuations in NT3 is that NT3 has very few observations and large dimensions, which is harder to converge.

Answer to Q1. Since our experiments keep the search strategy and the search space the same with baseline, the increased scores of the candidate models indicate that our weight transfer proposal accelerates their convergence, bringing them closer to fully-trained models. Ultimately, this accelerates the second step of NAS, in which the top-K best performers need to be fully trained.

B. Speedup of Full Model Training

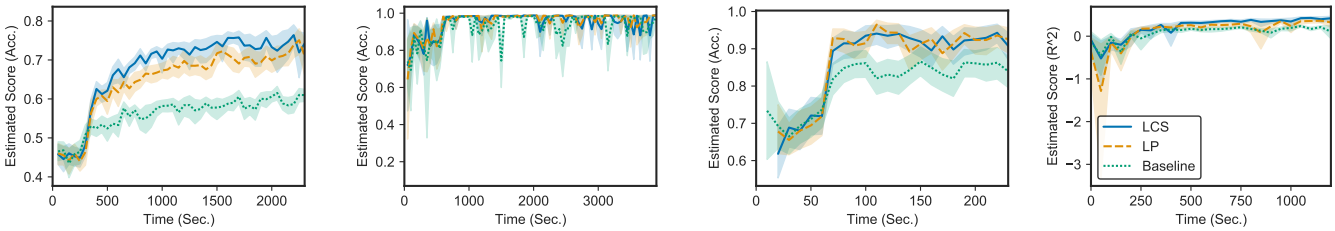
After obtaining the top-K best performers, NAS typically proceeds with a full training of all K models. This is necessary for two reasons: (1) the scores are not estimated perfectly, therefore the top candidate may not be the overall winner after the full training; (2) the criteria for choosing the best model may include trade-offs between accuracy and complexity (e.g. users may want to go for the second-best model if it is significantly simpler and can infer faster or can be transferred faster). Therefore, a diverse set of fully trained models with acceptable objective metrics give the user more choices.

We fully train the top-10 (K=10) models for each of the five experiments. We apply early stopping, which means if the objective metrics do not change by more than a given threshold for a fixed number of epochs (two in our case), the training stops. Early stopping during the full training implies the model has converged. We set up the threshold for NT3, MNIST, CIFAR-10, and Uno to 0.005, 0.001, 0.01, and 0.02, respectively. We also train the models for 20 epochs without early stopping to compare whether the results of early stopping are close to the results of full training.

Figure 8 shows the average number of epochs needed to fully train the 50 models (10 top-scored models × 5 experiments), and the associated objective metrics obtained from early stopping and full training. The objective metrics with early stopping are shown in blue lines, and without early stopping are shown in orange lines.

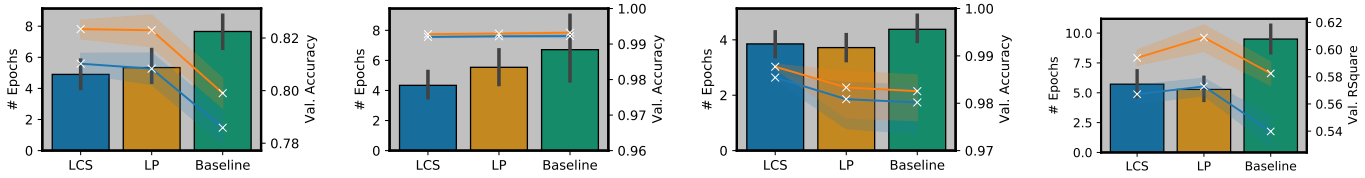
We make the following observations. (1) fewer epochs are needed by the models generated from our approach to achieve convergence. Since the training time is proportional to the number of epochs, overall, LCS achieves 1.5× speedup, and LP achieves 1.4× speedup on geometric mean across all applications; (2) the achieved final objective metrics are

[1]NT3 uses 10-second slots, as its training time is very short.



(a) CIFAR-10 (b) MNIST (c) NT3 (d) Uno

Fig. 7: Estimated objective metrics (scores) of the candidate models during NAS runtime. After the beginning phase, with our weight transfer techniques, the estimated objective metrics increase significantly compared to the baseline.



(a) CIFAR-10 (b) MNIST (c) NT3 (d) Uno

Fig. 8: LCS and LP achieve 1.5× and 1.4× speedup vs. training from scratch for full training (using early stopping). LCS and LP also achieve better or comparable objective metrics. The bars show the average number of epochs before early stopping. The blue lines show the objective metrics with early stopping. The orange lines show the objective metrics of the fully trained models.

higher in CIFAR-10, NT3, and Uno when applying LCS or LP, regardless of early stopping or not. For MNIST on the other hand, the baseline has marginally better accuracy (≈ 0.001); (3) the objective metrics with early stopping are very close to fully trained metrics. (4) LCS works better for CIFAR-10 and NT3, while LP generates better models for Uno. We will further analyze this in Section VIII-C.

Answer to Q2. Thanks to our approach, full training of the top-K best performers is significantly faster (1.5× and 1.4×). The fully trained models also have better or comparable objective metrics.

C. Quality of the Discovered Models

TABLE III: Top scored models after full training. Our approach provides a significantly better or comparable set of fully-trained models.

		Obj. Metrics (Accuracy or R^2)	
Application	Scheme	Fully Trained	Early Stopped
CIFAR-10	Baseline	0.799 ± 0.025	0.786 ± 0.028
	LCS	**0.823 ± 0.016**	**0.810 ± 0.017**
	LP	**0.823 ± 0.026**	0.808 ± 0.027
MNIST	Baseline	0.993 ± 0.001	0.992 ± 0.001
	LCS	0.993 ± 0.001	0.992 ± 0.001
	LP	0.993 ± 0.001	0.992 ± 0.001
NT3	Baseline	0.976 ± 0.067	0.974 ± 0.066
	LCS	**0.988 ± 0.003**	**0.985 ± 0.004**
	LP	0.987 ± 0.003	**0.985 ± 0.004**
Uno	Baseline	0.582 ± 0.038	0.540 ± 0.047
	LCS	0.594 ± 0.024	0.567 ± 0.030
	LP	**0.609 ± 0.041**	**0.573 ± 0.028**

We study whether the top-K best models discovered by our approach are statistically better than those discovered by the baseline. To this end, we fully train the top-10 models for each of the five experiments, so each scheme has 50 models. To enable a fair comparison with the baseline, we only select the top-10 models from all models discovered before the duration of the shortest experiment. In other words, all the approaches have the same time budget for the candidate estimation phase.

TABLE IV: Model Complexity of the Top-Scored Models

		Number of Parameters ($/10^6$)		
Application	Scheme	Mean	Max	Min
CIFAR-10	Baseline	12.389 ± 10.974	34.179	1.158
	LCS	9.555 ± 10.970	35.675	0.871
	LP	8.739 ± 11.651	34.519	0.505
MNIST	Baseline	2.807 ± 1.196	4.102	0.756
	LCS	2.043 ± 1.412	4.235	0.458
	LP	1.782 ± 0.784	3.346	0.63
NT3	Baseline	11.603 ± 8.914	38.711	1.291
	LCS	6.918 ± 5.060	15.485	1.291
	LP	14.034 ± 9.646	46.456	1.293
Uno	Baseline	6.191 ± 3.126	11.906	1.832
	LCS	6.062 ± 2.593	12.372	1.709
	LP	5.059 ± 2.265	8.453	1.587

Table III shows the results. We observe that (1) LCS and LP achieve better objective metrics than the baseline in both full training and early stopping cases. (2) LCS performs the best for CIFAR-10 and NT3. LP performs slightly better than LCS for Uno. The reason is that the variable nodes of CIFAR-10 and NT3 choose a different set of operations, while the variable nodes of Uno choose the same set of operations. A conservative approach that transfers the beginning layers only may be more beneficial for Uno. (3) All schemes achieve similar results for MNIST, which shows that our scheme does not bring negative effects even if the application is simple and the baseline performs well.

Model Complexity. As we discussed in Section VIII-B, the user may also prefer simpler models with acceptable objective metrics. Therefore, we study whether our weight transfer techniques have an impact on the diversity of the top-scored models, especially the model complexity. Table IV considers the number of parameters as a proxy for model complexity. We

observe that for most cases, our schemes have a similar range of parameters as the baseline does. NT3 with LCS and Uno with LP, however, have a fewer number of parameters than the baseline. Therefore, we conclude that our approach has the potential to reduce the model complexity without sacrificing the objective metrics.

Answer to Q3. Our approaches lead to statistically better models than the baseline without a negative impact on the model complexity characteristics.

D. Discussion: Why Weight Transfer Mechanisms Discovered Better Models?

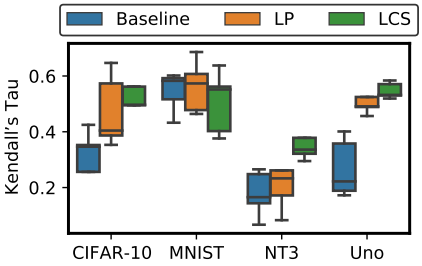


Fig. 9: Kendall's Tau between the estimated scores and the ground truth objective metrics. Applying weight transfer mechanisms improve the quality of candidate estimation significantly.

Next, we study why our approach discovered better models. Since our schemes do not change the search space and search strategy, we evaluate whether our schemes affect the candidate estimation.

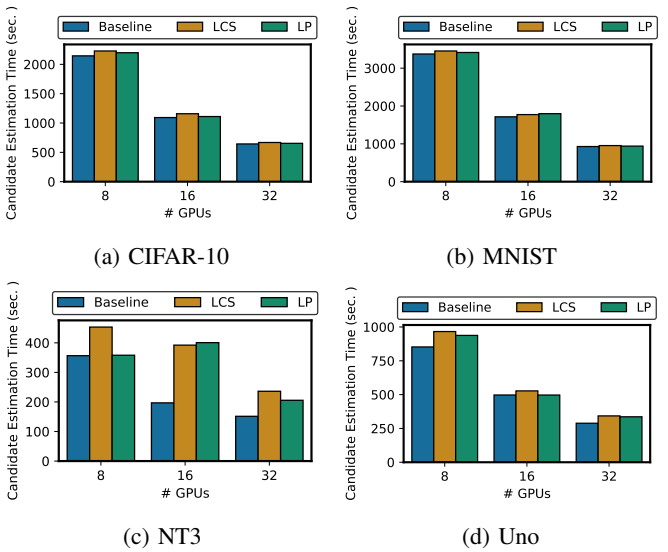
We sample 100 models out of 400 models in the candidate estimation phase of each experiment and then fully train the sampled 100 models. Therefore, we have a vector estimated scores $x_0, x_1, x_2, ..., x_n$ and a vector objective metrics $y_0, y_1, y_2, ..., y_n$ of the same models, where $n = 100$.

We use Kendall's τ to measure how accurate the estimated scores are. Kendall's τ correlation coefficient (τ) [45] measures the correlation between two rankings. Specifically, $\tau = \frac{2(N_c - N_d)}{n(n-1)}$, where $(x_1, y_1), (x_2, y_2), ...(x_n, y_n)$ are pairs of an estimated score and a fully trained objective metrics. For a pair of observations (x_i, y_i) and (x_j, y_j) and $i < j$, if $x_i < x_j \wedge y_i < y_j$, or $x_i > x_j \wedge y_i > y_j$, this pair is concordant. Otherwise, this pair is discordant. N_c is the number of concordant pairs. N_d is the number of discordant pairs. The range of Kendall's τ falls in $[-1, 1]$. Larger τ indicates the rank of estimated scores is closer to the rank of fully-trained objective metrics.

Figure 9 illustrates that τ is significantly improved for both LCS and LP for CIFAR-10, NT3, and Uno. LCS also works better than LP for the three applications, as it has more scope to transfer weights (refer to Figure 4), which makes the convergence faster. On the other hand, for MNIST, LP and LCS have a similar τ as the baseline, which explains why the discovered models are comparable with the baseline.

Explanation of Q3's Answer. Our approach discovers better models *because the weight transfer significantly improves the candidate estimation.*

E. Overhead and Scalability Analysis



(a) CIFAR-10 (b) MNIST

(c) NT3 (d) Uno

Fig. 10: Scalability for 8, 16, and 32 NVIDIA A100 GPUs. Our approach scales perfectly for CIFAR-10, MNIST, and Uno, showing a constant time overhead. For NT3, our approach has comparable scalability but incur notable checkpointing overhead compared with NT3's short training time (∼6 seconds).

We study whether our approaches affect the scalability of NAS. Since our weight transfer techniques only impact the candidate estimation due to checkpointing overheads, we perform the candidate estimation for 400 models on different numbers of GPUs (8, 16, and 32) for each application. Each training task runs on one GPU independently. Other configurations are the same as in Section VIII-A.

Figure 10 shows the execution time of the candidate estimation for an increasing number of GPUs. We make the following observations: (1) Our schemes have linear scalability for CIFAR-10, MNIST, and Uno. For these applications, the overhead of our schemes is very small. In addition, the percentage of overhead does not change as the number of GPUs increases, indicating NAS remains scalable. LCS often uses slightly more time than LP, because they have different time complexity and also have different memory footprints. (2) NT3, on the other hand, is not perfectly scalable. From 16 GPUs to 32 GPUs, the reduction of time is not linear. This happens in both baseline and our approaches. Our approaches increase the time of NT3 a lot because training NT3 is very fast as it has a tiny dataset (i.e., only has 1,120 observations). Also, we noticed that the time of NT3 fluctuates a lot because the randomness of the candidate estimation affects NT3 the most (i.e., even if we fix the random seed, due to the randomness of GPU training [46], [47], the search goes to diverged directions).

Sources of Overhead. Weight transfer mechanisms at most take 150 ms in the training process across all applications, which is negligible to the training model. However, loading checkpoints can take on average 4 seconds in NT3. We observe that this effect is related to the training time and the model

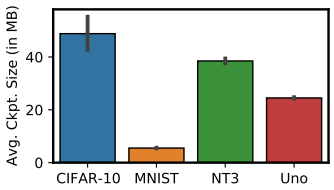


Fig. 11: Average Checkpoint Sizes of Evaluated Applications

size. Figure 11 shows the average checkpoint sizes for each application. We observe that NT3 has a large checkpoint (∼40MB) compared to its training time (∼6 seconds on average). When the checkpoint is loaded, the model is recovered in the memory. Since NT3 takes short time to train, the Ray evaluator frequently changes the objects in its local store, leading to worse performance.

We highlight that all of our earlier evaluations use the same time budget for the candidate estimation. Therefore, even if our approach has additional overheads and manages to explore fewer candidates, it ultimately assembles a better top-K set of models (Section VIII-C) and reduces the number of epochs needed for full training using early stopping (Section VIII-B).

Answer to Q4. Our approach is scalable and incurs a low overhead for a majority of the evaluated applications.

IX. Related Work

Neural Architecture Search (NAS). NAS is a technique that finds suitable neural network architectures automatically. The proposed research of NAS follows three aspects, 1) search space, 2) candidate estimation and 3) search strategy. Several survey papers [11], [48] focus on these issues. In the image classification domain, there are a few well-known proposed search spaces [15], [16], [49]. However, in other domains such as cancer research, or prediction tasks for tabular data, the domain experts need to design their search spaces manually [13], [12]. Regarding the estimation of candidate models, previous methods [26], [14], [13], [15], [16] train many models independently from scratch. Such methods are prohibitively expensive because the candidate models converge slowly. To reduce the execution time, one-shot NAS [49], [50] proposes to train a supernet that contains all possible architectures in the search space. The weights of a model in the search space could be inherited from the supernet. However, one-shot NAS suffers from degraded performance [12], [51]. The reason is that the estimated objective metrics is poorly correlated (in Kendall's τ rank coefficient) with the fully trained models [52], [53], [54]. Our work falls into the candidate estimation aspect. Compared to the existing works, our weight transfer proposal focuses on the normal NAS. Our approach accelerates the convergence of the models, and also significantly improves Kendall's τ of the candidate estimation.

Several prior works focus on designing new search strategies to navigate the search space. These works include random search [55], Bayesian optimization [18], [19], evolutionary methods [20], [21], [22], [23], [24], [25], and reinforcement learning [13], [26]. Currently, however, no clear winner outperforms all the other approaches in all settings [25]. While the proposed weight transfer mechanisms are implemented with an evolutionary search strategy [25], our approaches are not limited to it if we can select the provider model fast. Our implementation is also extensible to support different provider model selectors.

Efficient checkpointing for DNNs. Weight transfer is based on checkpointing previously trained candidate models, which becomes an important source of overhead. To reduce this overhead, DNN checkpointing techniques have been extensively explored before. Here, we summarize two categories of related work. First, in most large scale HPC infrastrucutures, I/O bandwidth a scarce resource. To this end, *multi-level* checkpointing can be used to leverage I/O strategies adapted for hybrid storage hierarchies. VELOC [56], [57] takes this approach further by introducing asynchronous I/O strategies. Specifically for DNNs, approaches such as DeepFreeze [58] exploit the fine-grain parallelism of the backward pass in order to augment the execution graph with additional operations that capture individual tensors asynchronously. Other approaches such as CheckFreq [59] focus on determining the optimal checkpointing frequency through systematic online profiling of the overhead. The second category explores compression of checkpoints to save unnecessary I/O bandwidth. In this regard, DeepSZ [60] leverages bounded lossy compression for DNNs. Check-N-Run [61] coordinates incremental and quantization compression that achieved negligible overhead compared to the training. We envision our proposal to be complemented by such approaches.

X. Conclusions

NAS is an important step of deep learning workflows that involves two stages: (1) candidate exploration to obtain top-K best models; (2) full training of the top-K models to select a winner based on various trade-offs. Unlike state-of-the-art approaches that evaluate new candidates by training them from scratch, we propose the idea of training new candidates by performing weight transfer from previous candidates with similar structures using two techniques: LP and LCS. We demonstrated the effectiveness of weight transfer for an evolutionary search strategy, which produced better top-K models that are 1.4∼1.5× faster to train fully in the second stage. Furthermore, despite involving additional checkpointing overheads, weight transfer remains scalable in the first stage. Encouraged by these results, in future work, we plan to extend our approach by complementing it with efficient DNN checkpointing techniques, which we expect to further emphasize the benefits of our proposal.

Acknowledgments

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. We acknowledge the computing resources provided on Theta and JLSE (operated by Argonne Leadership Computing Facility).

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.

[3] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[4] Q. Wu, Z. Zhang, A. Pizzoferrato, M. Cucuringu, and Z. Liu, "A Deep Learning Framework for Pricing Financial Instruments," *CoRR*, 2019.

[5] Q. Wu, F. M. Wong, Y. Li, Z. Liu, and V. Kanade, "Adaptive Reduced Rank Regression," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[6] Q. Wu, W.-L. Hsu, T. Xu, Z. Liu, G. Ma, G. Jacobson, and S. Zhao, "Speaking with Actions—Learning Customer Journey Behavior," in *Proceedings of the IEEE International Conference on Semantic Computing (ICSC)*, 2019.

[7] S. Zhao, W.-L. Hsu, G. Ma, T. Xu, G. Jacobson, and R. Rustamov, "Characterizing and Learning Representation on Customer Contact Journeys in Cellular Services," in *Proceedings of the International Conference on Knowledge Discovery & Data Mining (KDD)*, 2020.

[8] "Artificial Intelligence and Deep Learning Accelerate Efforts to Develop Clean, Virtually Limitless Fusion Energy," https://science.osti.gov/fes/Highlights/2019/FES-2019-04-b, accessed: 2021-05-22.

[9] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer, "Machine Learning Accelerated Computational Fluid Dynamics," 2021.

[10] Wozniak, Justin M and Jain, Rajeev and Balaprakash, Prasanna and Ozik, Jonathan and Collier, Nicholson T and Bauer, John and Xia, Fangfang and Brettin, Thomas and Stevens, Rick and Mohd-Yusof, Jamaludin and others, "CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research," *BMC bioinformatics*, vol. 19, no. 18, pp. 59–69, 2018.

[11] T. Elsken, J. H. Metzen, and F. Hutter, "Neural Architecture Search: A Survey," *Journal of Machine Learning Research (JMLR)*, 2019.

[12] R. Egele, P. Balaprakash, V. Vishwanath, I. Guyon, and Z. Liu, "AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data," 2020.

[13] P. Balaprakash, R. Egele, M. Salim, S. Wild, V. Vishwanath, F. Xia, T. Brettin, and R. Stevens, "Scalable Reinforcement-learning-based Neural Architecture Search for Cancer Deep Learning Research," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[14] P. Balaprakash, M. Salim, T. D. Uram, V. Vishwanath, and S. M. Wild, "DeepHyper: Asynchronous Hyperparameter Search for Deep Neural Networks," in *Proceedings of the International Conference on High Performance Computing (HiPC)*, 2018.

[15] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "NAS-bench-101: Towards reproducible neural architecture search," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2019.

[16] X. Dong and Y. Yang, "NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search," in *International Conference on Learning Representations (ICLR)*, 2020.

[17] "NT3 Benchmark," https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/NT3, accessed: 2021-04-22.

[18] G. Dikov and J. Bayer, "Bayesian Learning of Neural Network Architectures," in *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2019.

[19] K. Kandasamy, W. Neiswanger, J. Schneider, B. Póczos, and E. P. Xing, "Neural Architecture Search with Bayesian Optimisation and Optimal Transport," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS)*, 2018.

[20] J. Liang, E. Meyerson, B. Hodjat, D. Fink, K. Mutch, and R. Miikkulainen, "Evolutionary Neural AutoML for Deep Learning," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019.

[21] P. R. Lorenzo, J. Nalepa, L. S. Ramos, and J. R. Pastor, "Hyper-parameter selection in deep neural networks using parallel particle swarm optimization," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017.

[22] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing Neural Networks Using Genetic Algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[23] M. Suganuma, S. Shirakawa, and T. Nagao, "A Genetic Programming Approach to Designing Convolutional Neural Network Architectures," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

[24] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-Scale Evolution of Image Classifiers," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2017.

[25] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

[26] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[27] A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter, "Learning Curve Prediction with Bayesian Neural Networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[28] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning Transferable Architectures for Scalable Image Recognition," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[29] P. Chrabaszcz, I. Loshchilov, and F. Hutter, "A downsampled variant of imagenet as an alternative to the CIFAR datasets," *CoRR*, vol. abs/1707.08819, 2017.

[30] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[31] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How Transferable are Features in Deep Neural Networks? ," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., 2014.

[32] "Convolutional Neural Networks," https://www.coursera.org/learn/convolutional-neural-networks, accessed: 2021-05-22.

[33] M. Aygün, Y. Aytar, and H. K. Ekenel, "Exploiting Convolution Filter Patterns for Transfer Learning," in *IEEE International Conference on Computer Vision Workshops (ICCVW)*, 2017.

[34] R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM*, 1974.

[35] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[36] M. A. Salim, T. D. Uram, J. T. Childers, P. Balaprakash, V. Vishwanath, and M. E. Papka, "Balsam: Automated Scheduling and Execution of Dynamic, Data-Intensive HPC Workflows," 2019.

[37] The HDF Group. (2000-2010) Hierarchical data format version 5. [Online]. Available: http://www.hdfgroup.org/HDF5

[38] "The CIFAR-10 Dataset," https://www.cs.toronto.edu/~kriz/cifar.html, accessed: 2021-05-15.

[39] "The MNIST Database," http://yann.lecun.com/exdb/mnist/, accessed: 2021-04-22.

[40] "Problem Definition for NAS Linked with CANDLE Project." https://github.com/deephyper/candlepb, accessed: 2021-04-22.

[41] "Uno Benchmark," https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/Uno, accessed: 2021-04-22.

[42] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[43] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[44] "Keras: the Python Deep Learning API," https://keras.io/, accessed: 2021-05-15.

[45] N. N. Liu and Q. Yang, "EigenRank: a ranking-oriented approach to collaborative filtering," in *Proceedings of the international ACM*

SIGIR conference on Research and development in information retrieval (SIGIR), 2008.

[46] D. Riach, "Determinism in Deep Learning," http://bit.ly/dl-determinism-slides-v2, 2019, accessed: 2021-05-22.

[47] H. Jooybar, W. W. Fung, M. O'Connor, J. Devietti, and T. M. Aamodt, "GPUDet: A Deterministic GPU Architecture," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[48] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *ACM Computing Surveys*, 2019.

[49] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *International Conference on Learning Representations (ICLR)*, 2019.

[50] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient Neural Architecture Search via Parameters Sharing," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

[51] Y. Zhao, L. Wang, Y. Tian, R. Fonseca, and T. Guo, "Few-shot Neural Architecture Search," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2021.

[52] S. Hu, S. Xie, H. Zheng, C. Liu, J. Shi, X. Liu, and D. Lin, "DSNAS: Direct Neural Architecture Search Without Parameter Retraining," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[53] C. Sciuto, K. Yu, M. Jaggi, C. Musat, and M. Salzmann, "Evaluating the Search Phase of Neural Architecture Search," *CoRR*, vol. abs/1902.08142, 2019.

[54] R. Luo, T. Qin, and E. Chen, "Balanced One-shot Neural Architecture Optimization," 2020.

[55] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research (JMLR)*, 2012.

[56] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[57] B. Nicolae, A. Moody, G. Kosinovsky, K. Mohror, and F. Cappello, "VELOC: VEry Low Overhead Checkpointing in the Age of Exascale," 2021.

[58] B. Nicolae, J. Li, J. M. Wozniak, G. Bosilca, M. Dorier, and F. Cappello, "DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models," in *Proceedings of the International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2020.

[59] J. Mohan, A. Phanishayee, and V. Chidambaram, "CheckFreq: Frequent, Fine-Grained DNN Checkpointing," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[60] S. Jin, S. Di, X. Liang, J. Tian, D. Tao, and F. Cappello, "DeepSZ: A Novel Framework to Compress Deep Neural Networks by Using Error-Bounded Lossy Compression," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.

[61] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, M. Annavaram, K. Nair, and M. Smelyanskiy, "Check-n-run: A checkpointing system for training recommendation models," *arXiv preprint arXiv:2010.08679*, 2020.