EvoStore: Towards Scalable Storage of Evolving Learning Models

Robert Underwood
runderwood@anl.gov
Argonne Nat. Laboratory
Lemont, Illinois, USA

Meghana Madhyastha
mmadhya1@jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

Randal Burns
randal@cs.jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

Bogdan Nicolae
bnicolae@anl.gov
Argonne Nat. Laboratory
Lemont, Illinois, USA

ABSTRACT

Deep Learning (DL) has seen rapid adoption in all domains. Since training DL models is expensive, both in terms of time and resources, application workflows that make use of DL increasingly need to operate with a large number of derived learning models, which are obtained through transfer learning and fine-tuning. At scale, thousands of such derived DL models are accessed concurrently by a large number of processes. In this context, an important question is how to design and develop specialized DL model repositories that remain scalable under concurrent access, while addressing key challenges: how to query the DL model architectures for specific patterns? How to load/store a subset of layers/tensors from a DL model? How to efficiently share unmodified layers/tensors between DL models derived from each other through transfer learning? How to maintain provenance and answer ancestry queries? State of art leaves a gap regarding these challenges. To fill this gap, we introduce EvoStore, a distributed DL model repository with scalable data and metadata support to store and access derived DL models efficiently. Large-scale experiments on hundreds of GPUs show significant benefits over state-of-art with respect to I/O and metadata performance, as well as storage space utilization.

CCS CONCEPTS

• **Software and its engineering** → *Checkpointing*; • **Computing methodologies** → **Neural networks**; • **Information systems** → **Parallel and distributed DBMSs**.

KEYWORDS

AI, Model Repository, Network Architecture Search, Regulartized Evolution, Distributed, AI for HPC

ACM Reference Format:
Robert Underwood, Meghana Madhyastha, Randal Burns, and Bogdan Nicolae. 2024. EvoStore: Towards Scalable Storage of Evolving Learning Models . In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3625549.3658679

1 INTRODUCTION

Deep learning (DL) applications are rapidly transforming all aspects of our society. As the data sizes, data pattern complexity and scope of the problems keep increasing, the DL models have evolved from all perspectives: size (number of parameters), depth (number of layers/tensors) and structure (layers with multiple inputs/outputs interconnected using directed graphs that feature divergent branches, fork-join, etc.). Training DL models from scratch is an expensive task, both in terms of resources and time [14].

As a consequence, alternatives such as the reuse of DL models through transfer learning [35] and fine-tuning [8] are becoming increasingly popular. Specifically, instead of training a DL model from scratch, we can start from a previously trained DL model by reusing its model parameters. If necessary, the architecture can be adjusted by adding or removing layers (and initializing the new layers with random parameters). Then, the model can be refined further with fresh input data through additional training. Using this approach, in a typical setup, the derived DL models can be trained much faster, not only because of faster convergence (thanks to a better starting point), but also because of techniques that accelerate each training iteration (e.g., freezing some of the layers and excluding them from the backward pass [6]).

Motivation: To facilitate transfer learning and fine-tuning, it is necessary to store promising snapshots of DL models in a scalable repository, from where they can be later reused to obtain derived DL models. This is challenging for several reasons:

Storage space efficiency: When the transferred layers are frozen during the training, only a subset of the tensors change, which leads to a large number of DL models that end up sharing the same tensors. In this case, writing a full copy of a derived DL model back to the repository wastes storage space, as unmodified tensors are duplicated unnecessarily. On the other hand, if we store each tensor only once, multiple DL models may have references to it. Thus, removing a DL model from the repository is a challenging operation that requires specialized garbage collection techniques.

Scalable I/O access performance under concurrency: AI workflows running on HPC systems are often composed of a large number of distributed processes that need to perform transfer learning under concurrency. For example, ensemble learning and network architecture search (NAS) [13][38] evaluate a large number of related candidate models in parallel. Such approaches can benefit from transfer learning [24]. As a consequence, the repository needs to remain scalable despite concurrent reads and writes of DL models.

Scalable query support for model architectures: As more DL models keep accumulating in the repository (either completely new or derived from other DL models through transfer learning), more candidates that can serve as a source for transfer learning become available. Combined with the growing complexity of the DL model architectures, the problem of how to identify the best candidate for transfer learning in a scalable fashion is challenging. In particular, there is a need to formulate and efficiently respond to queries that look for specific architectural features and patterns in the whole collection of DL models stored on the repository.

HPDC '24, June 03–07, 2024, Pisa, Italy

ACM ISBN 979-8-4007-0413-0/24/06.
https://doi.org/10.1145/3625549.3658679

Ancestry and provenance: When transfer learning is applied repeatedly to obtain a long chain of DL models that are derived from each other, it is important to study the ancestry and provenance of the DL models in order to be able to explain their behavior and reason about their reliability. For example, we may be interested in questions such as: what is the most recent common ancestor of two DL models obtained through transfer learning? Which ancestor "owns" a given frozen layer that remain unchanged in other DL models derived from it?

Limitations of State-of-Art: Each of the challenges introduced above is insufficiently addressed by the state-of-art approaches. With respect to storage space efficiency, most approaches serialize DL models and independent, self-contained objects (typically files) that duplicate any frozen layers that are reused across derived models. Only a small number of checkpointing approaches adopt incremental storage techniques (notably Check-n-Run [12]) during the training of the *same* model. They do not support different models that share common parts. With respect to scalable I/O access under concurrency, traditional DL model serialization formats (e.g., HDF5 [17] or SavedModel [1]) have significant overheads and typically use parallel file systems to store the resulting files. These are optimized for bulk I/O access, which does not match the need for partial I/O to enable fine-grain access to individual tensors needed to transfer individual layers and to store incremental differences. With respect to query support, the most common solution is to use a metadata server that catalogs the models based on search criteria and annotations. Such a solution does not scale, since the metadata server becomes a bottleneck when serving a large number of concurrent queries. To our best knowledge, the problem of ancestry and provenance has received very little attention and is solved using inefficient ad-hoc techniques that iterate over all DL models and extract metadata about their architecture on-demand.

Key Insights and Contributions: To address the aforementioned limitations, in this paper we introduce EvoStore, a distributed DL model repository with scalable, fine-grain I/O access and incremental storage allowing reading/writing only data that is changed at the tensor-level, which is complemented by a decentralized metadata infrastructure to provide efficient best candidate and provenance query support. Unlike state of art approaches, EvoStore distributes the tensors among a large number of storage providers and caches them in-memory, which introduces an opportunity to break free from the I/O bottlenecks of parallel file systems, to eliminate the redundant storage of shared tensors and to provide lightweight metadata management. We summarize our contributions as follows:

(1) We introduce several key design principles that underline the novelty of our approach: (1) incremental tensor storage and garbage collection techniques based on the notion of *owner maps*; (2) consolidated sets of tensors that are distributed at scale and accessed using low-overhead RDMA techniques to enable fine-grained I/O access under concurrency; (3) techniques to organize and query the architecture metadata of DL models for the *longest common directed graph prefix*, which we introduce as a best-match pattern for transfer learning; (4) a distributed metadata query engine that searches for the best match among all DL models in the repository. We present these in Section 4.1.
(2) We introduce several algorithms and considerations that enable an efficient implementation of the design principles (Section 4.2). We design and implement EvoStore, a DL model repository research prototype that illustrates the design principles and algorithms in real life. EvoStore integrates with the Tensorflow and DeepHyper runtimes (Section 4.3). Furthermore, we introduce the application of such a scalable repository in the context of a network architecture search scenario that is based on transfer learning (Section 2).
(3) We evaluate our approach in a series of scalability experiments that involve both micro-benchmarks and real-life network architecture search applications based on transfer learning. We show significant speed-up and better scalability compared with other approaches for metadata queries, and I/O requests involving subsets of tensors and resulting in better end-to-end application runtime. These contributions are presented in Section 5.

Limitations of the Proposed Approach: We focus on storing derived DL models by capturing low-overhead checkpoints of the model parameters into a lineage. Specifically, each checkpoint includes a compact representation of the model architecture, a collection of tensors that represent the differences with respect to the original model parameters, and an owner map that enables reading the inherited parameters. This is sufficient to address a majority of transfer learning and fine tuning scenarios. However, under certain circumstances, it is necessary to capture additional information, notably the optimizer state used during training. This is relevant in scenarios that need to adjust a model after a setback (e.g., failures or lack of convergence) and continue the original training from where it left. We will add support for such scenarios in future work.

2 MOTIVATING SCENARIO: NAS

To illustrate the need for a flexible repository such as EvoStore that satisfies the requirements introduced above, we choose to focus on *network architecture search* (NAS) [13], which can take advantage of transfer learning extensively at a large scale. However, it is important to note that EvoStore is a generic DL model repository that can be leveraged to facilitate many other scenarios related to transfer learning and fine-tuning.

NAS Fundamentals: A typical NAS explores a large number of candidate models from a search space that is based on a set of rules that define what choices are possible. The set of all choices that define the architecture of a valid candidate model is called a *candidate sequence*.

A common approach to finding good candidates is to simply sample the search space randomly [21]. This process is illustrated in Figure 1: a controller process is responsible to randomly sample the search space and produce candidate sequences, which are then distributed among worker processes in an embarrassingly parallel fashion. Each worker process constructs a DL model from the candidate sequence, trains it superficially (e.g., for a single epoch) , and reports the quality metric of the DL model (e.g., classification accuracy) to the controller. The controller retains the top-K best performers, which are then further refined (e.g., fully trained for many more epochs) and their viability is assessed based on various criteria (e.g., accuracy, size, inference speed, etc.).

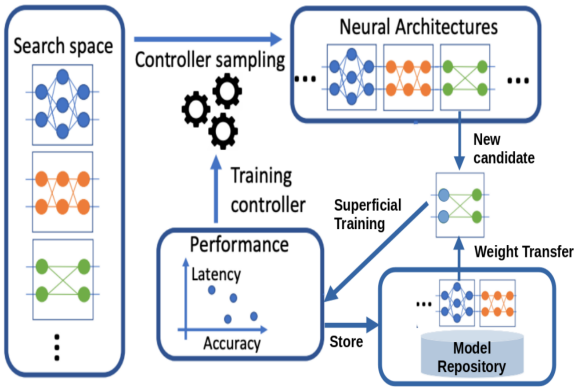

Figure 1: Network architecture search based on transfer learning facilitated by a DL model repository.

However, if the search space is large, such an approach takes a long time to complete, uses large amounts of resources and may produce poor results [3]. Thus, a better approach is to have the controller guide the exploration of the search space rather than randomly sample from it. A common approach is to decide what new candidate sequences to try next based on the quality metrics of the previously explored candidates. For example, the aged evolution algorithm [10] starts with an initial random population of candidate sequences and then generates new candidate sequences by mutating the best candidate out of a randomly chosen subset of the population, much like genetic algorithms. The size of the population can be limited to a maximum of N candidates by dropping the worst performers as the population evolves.

Assuming each DL model is trained from scratch, it is enough to simply store N candidate sequences and their quality metrics on the controller to fully describe the entire population. However, this comes at the expense of high training overhead, which limits the number and quality of the explored DL model candidates within a given time limit and/or resource budget [24].

NAS with Transfer Learning: NAS introduces an opportunity to leverage transfer learning find better candidates and/or accelerate the search [24]. For example, in the case of aged evolution, each DL model candidate is obtained as a mutation of a previously explored DL model candidate. Therefore, the population is composed of a large number of candidates that share a similar architecture. Thus, we can initialize the weights of a new candidate from a previously explored ancestor whose architecture is similar. Doing so enables the new candidate to benefit from the experience of the entire lineage of ancestors, which makes the superficial training more accurate as an estimation of the quality metric. This improves the odds of better quality models being selected in the top-K.

One of the most commonly used transfer learning strategies is to fine-tune the last layers of an ancestor model by inheriting and freezing the first layers [6]. This approach accelerates superficial training, because each backward pass needs to be applied only to the last layers that are updated. For sequential models, the first layers correspond to the *longest common prefix* of the new candidate and ancestor sequences.

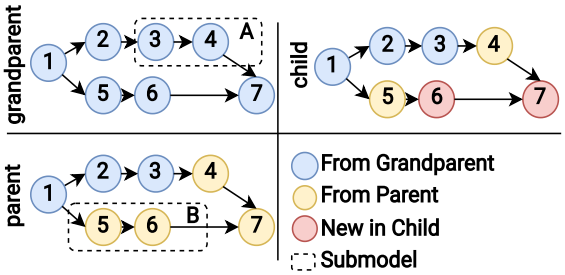

Figure 2: Example of three DL model architecture graphs. The longest common prefix (LCP) between the parent and the grandparent is $\{1, 2, 3\}$ and between the parent and the child is $\{1, 2, 3, 4, 5\}$. By transferring and freezing the LCP during training, only 13 unique layers need to be stored compared with 21 layers in the case when the three DL models are stored independently.

By generalization, when the model architecture is an arbitrary directed graph, the longest common prefix is the set of vertices V, for which $v \in V$ if and only if: (1) the choice of v is identical both for the ancestor and the new candidate; (2) all vertices whose outputs are inputs for v are also included in V. This is a recursive definition: if the input layer matches, it is included in V; any other matching layers connected to the input layer are also included in V, etc. For example, Figure 2 illustrates the longest common prefix for a chain of two transfer learning operations: parent and grandparent share the same architecture for layers $\{1, 2, 3\}$ but not $\{4, 5\}$. Even if 7 also shared the same architecture, the longest common prefix would have remained $\{1, 2, 3\}$. Similarly, the child shares the same architecture with the parent for all layers except 6. In this case, the longest common prefix is $\{1, 2, 3, 4, 5\}$. For the rest of this paper, we use longest common prefix to refer to the generalized form.

Problem Formulation: To enable a NAS transfer learning strategy as discussed in Section 2, we need a DL model repository capable of (1) storing the entire population of N candidates (both architecture and layer weights); (2) retiring DL models that are replaced in the population; (3) given a new candidate, identifying the ancestor with the longest common prefix (and preferring the one with the highest quality metrics in case of a tie); (3) selectively read only the tensors corresponding to the longest common prefix from the ancestor and transfer them to the new candidate; (4) selectively writing only the tensors that have changed after training the new candidate; (5) given a set of DL models (e.g., top-K best performers after the search), extracting the lineage of each DL model (chain of ancestors) and/or find the most recent common ancestor of a DL model pair. An illustration of how this repository fits in the NAS workflow is depicted in Figure 1.

Given a large number of workers that need to load and store the DL model candidates under concurrency, this NAS scenario illustrates the challenges discussed in Section 1: the need for incremental storage and garbage collection to achieve space efficiency, scalable I/O access performance under concurrency, scalable metadata query support to efficiently answer longest common prefix queries under concurrency, ancestry, and provenance support. Our goal is to design such a repository that addresses these challenges.

3 RELATED WORK

DL Model Checkpointing: Popular runtimes such as Tensorflow and PyTorch serialize DL Model into various formats (e.g. HDF5 [17] or SavedModel [1]) used to checkpoint and resume the training later. These formats include additional unnecessary information such as the optimizer state and implement inefficient serialization that incurs high I/O overheads. Optimized checkpointing approaches exist for data-parallel training [29]. They take advantage of multiple identical replicas to parallelize the writes of different shards. Other checkpointing efforts such as CheckFreq and Check-n-Run [12, 28] focus on determining the optimal checkpointing interval through systematic online profiling, which is important for resilience but not our scenario. FlameStore [34] aims to reduce the serialization overheads by directly capturing and storing the tensors at their final destination (in-memory, local file system). DStore [27] extends this principle by adopting fine-grained RDMA-enabled storage and efficient partial read support. Overall, the lack of important features (e.g., no metadata query support, no incremental storage for DStore) or different scope (e.g., only incremental storage of the same model for Check-n-Run) makes such approaches insufficient in the scenarios targeted by our approach.

Web-enabled DL Model Repositories: are optimized for collaborative rapid application development. They enable users to upload, classify, curate, and search for DL models. Prominent examples include open repositories such as Tensorflow Hub [36], PyTorch Hub [32], Caffe's Model Zoo [18]. Some efforts, such as DLHub [22] target scientific applications specifically. For web-enabled repositories, the emphasis is on providing rich features and ease-of-use, rather than high performance and scalability. One notable feature is the versioning of DL models. For example, Data Version Control (DVC) [20] relies on Git's version control capabilities to store metadata references to large binary and textual objects that represent training data and DL model checkpoints. However, the serialization and actual storage of the tensors is outside of the scope of the repository and delegated to web-based RPCs or parallel file systems that are not optimized for small non-contiguous data transfers.

Network Architecture Search: Several aspects such as search space, candidate estimation and search strategy. are summarized in various surveys [4, 13]. Search spaces are well documented in the image classification community [9, 41]. Other domains rely on expert input (e.g., cancer research [2]). Regarding candidate estimation, one-shot NAS techniques [25, 31] training a supernet that contains all possible architectures in the search space, which reduces the search duration. However, this approach suffers from poor candidate quality [42]. Regarding the search strategy, there are multiple approaches possible: random search [5], Bayesian optimization [7, 19], evolutionary methods [10, 23, 26, 33], and reinforcement learning [2, 43]. These approaches are subject to various trade-offs. Transfer learning based on our DL model repository approach can complement a majority of such approaches.

To our knowledge, we are the first to consider the problem designing a decentralized DL model repository that meets the scalability and performance requirements of HPC infrastructures under concurrency, while simultaneously focusing on fine-grain tensor-level access, incremental storage, and lineage.

4 EVOSTORE: SYSTEM DESIGN

4.1 Design Principles

Incremental Storage Based on Sharing Frozen Layers: Transfer learning typically results in unmodified layers in the derived DL models because of the practice of freezing some layers inherited from the ancestors. The proportion of frozen layers can be very high (e.g., 50% on the average in NAS scenarios [37]). Thus, it is important to share unmodified layers using incremental storage in order to save storage space and speed up writes. To this end, we organize the repository into a lineage of derived DL models, each of which has exactly one ancestor. The lineage may have many branches that evolve independently into potentially long chains. Each derived DL model uses its ancestor as a reference to compute a difference in terms of new/modified layers. The repository stores only this difference, which results in unique copies of unmodified layers that may be indirectly inherited by a large number of derived DL models, effectively resulting in large storage space savings.

In this context, an important challenge is how to identify the differences at fine granularity such as to maximize the deduplication potential. This is non-trivial because widely-used ML libraries (e.g., Keras [16]) consider layers as recursive structures that can share identical content at any level, from large structures to individual tensors. We devise fast algorithms (detailed in Section 4.2) to compute differences between derived DL models at tensor-level, at the finest possible granularity.

Lightweight Lineage Based on Owner Maps: A simple solution that simply persists the fine-grain difference between the derived DL model and its ancestor to the repository (together with a reference to the ancestor) is optimal with respect to write throughput, but introduces a high read throughput penalty later when the DL model needs to be reused. This is because the entire chain of incremental writes needs to be examined in order to reconstruct the DL model. Moreover, with an increasing chain length in the lineage, the overhead of the read requests grows proportionally. Thus, such a simple solution does not scale with an increasing number of derived DL models in the repository. To address this issue, we take the following approach: first, we extract the leaf layers and their tensors from the model architecture (a process detailed in Section 4.2). Then, we construct an *owner map* that assigns each tensor to an *owner*, i.e., the most recent ancestor that modified the tensor. Initially, a DL model not obtained through transfer learning owns all its tensors. Any other model obtained through transfer learning inherits the owner map of its ancestor, which it then updates to set itself as the owner only for the new/modified tensors. For example, in Figure 2, if we freeze the layers of the longest common prefix during training, then the grandparent needs to be stored entirely and the parent needs to store $\{4, 5, 6, 7\}$ and mark $\{1, 2, 3\}$ as belonging to the grandparent in the owner map. Similarly, the child needs to store $\{6, 7\}$ and mark $\{4, 5\}$ as belonging to the parent and $\{1, 2, 3\}$ as belonging to the grandparent in the owner map. Using this approach, we can efficiently serve read requests by consulting a single owner map, whose complexity does not depend on the number of ancestors.

Overall this, approach uses relatively little metadata regardless of the access patterns. It is at most hundreds of KB (128 bits per

leaf-layer) which is negligible compared with the actual size of the models and layers themselves.

Owner Maps as a Foundation for Provenance: With minimal extensions, the owner maps introduced above are naturally fitted to answer several types of provenance queries. Specifically, since the owner of each tensor is simultaneously the most recent ancestor that modified the tensor, we can directly use the owner map to identify what ancestors contributed to the composition of a given DL model and what tensors they affected. Furthermore, by introducing a global ordering of the owners (e.g., based on a timestamp at which their write request occurred), we can also determine what chain of transfer learning operations resulted in a given DL model stored by the repository. Using this approach, only a single consolidated data structure needs to be accessed without any need to synchronize with other DL models and their metadata, which facilitates scalability under concurrent access.

Distributed RDMA-enabled Tensor Storage with Owner-Based Consolidation: to achieve scalability, the owner map of each new DL model and the corresponding collection of modified tensors are distributed among a set of providers, each of which employs a local key-value store to persist the tensors and the owner maps. The providers can either be co-located with the application processes on the same compute nodes or be deployed separately on dedicated nodes. There are two challenges in this context: (1) how to enable the clients to find out what providers to contact to recover the architecture and/or content of the DL model, and (2) how to distribute the requests of concurrent clients to different providers to achieve load balancing and scalability. To this end, we leverage the fact that each owner map fully describes the composition of the DL model. Thus, we use a simple static hashing scheme that maps a model ID to a provider ID. Then, to store a new DL model, we consolidate the new tensors and send them in bulk together with the owner map to the provider ID that corresponds to the new DL model ID. Note that the unmodified tensors are already stored by other providers. To reconstruct a DL model, we follow the reverse process: we contact the provider to obtain the owner map, then follow the owner map to recover the tensors from the corresponding providers. This way, we obtain a good balance between scattering the tensors among the providers while retaining locality (i.e., tensors likely to be accessed together end up on the same provider). Combined with low-latency RDMA operations, such as scheme enables high performance and scalability under concurrent accesses. Furthermore, since each provider stores both the owner maps and the tensors, it simultaneously acts as a data and metadata provider. This simplifies the deployment and management of the DL model repository.

Distributed garbage collection using reference counting: In NAS, it is possible to remove a model's layers when a model leaves the population of models that are being evolved. By modifying the search framework to indicate this to the model repository, it is possible to retire models to save space for new models.

If no tensors are shared across the DL models, then retiring a model is a trivial operation that simply needs to remove all tensors and metadata corresponding to the model. However, when tensors are shared across DL models, each model may feature many different ancestors in its owner map. Thus, retiring an ancestor cannot simply remove all the tensors it owns, because each tensor may be reused by an arbitrary number of descendants. To address this challenge, we take the following approach: each provider embeds each segment that it reuses for bulk RDMA read requests with a reference counter. This reference counter is incremented for all tensors involved in the owner map of a write request. Similarly, the reference counter is decremented for all tensors of a DL model that is being retired taking $O(k)$ time where k is the number of leaf layers in the model.Note that k is small for even advanced models. GPT-3 has only 96 layers [40]. While the metadata of the retired model is always fully removed, its owned tensors are only removed when the reference counter drops down to zero. Note that due to different ownership, the tensors may be scattered across multiple providers. In this case, we employ a similar strategy as above (the client issues multiple bulk operations in parallel to the providers), which results in similar benefits.

Distributed provider-side collective metadata queries: An increasing number of DL models stored in the repository results in an explosion of the metadata associated with them. Our decentralized approach distributes the coupled data/metadata, which enables concurrent clients to perform scalable queries about individual DL models because it is likely that the metadata of different DL models ends up on different providers. Most of the provenance queries fall into this category. However, the longest common prefix queries follow a different pattern: a client needs to scan the whole metadata in order to find the best ancestor for transfer learning. Furthermore, for each potential ancestor, it is non-trivial to determine the longest common prefix when the model architecture is a generic graph (we study this problem in-depth in Section 4.2). Therefore, a naive solution that iteratively collects the individual metadata about each DL model determines the longest common prefix and retains the best candidate does not scale. To address this issue, starting from the observation that the providers are mostly idle because the majority of I/O transfers are performed using bulk RDMA operations, we take a map-reduce-inspired solution instead: the client broadcasts the architecture of the given DL model to all providers, which in parallel determines the best match among their local metadata. This is followed by an asynchronous reduction step to find the global best match, which is then retained by the client. Using this approach not only avoids additional overheads involved in metadata transfers but also distributes the computational load.

4.2 Zoom on Architecture Graphs, Owner Maps and Longest Common Prefix Queries

As mentioned in Section 2, in the general case, the model architecture is an arbitrary graph. In practice, the layers are typically expressed recursively (e.g., Keras [16]). They may form a nested architecture of submodels that eventually are composed of leaf layers (holding relevant parameters such as weights, biases, etc.). Therefore, it is not enough to simply consider the original layers for the purpose of identifying the longest common prefix.

To illustrate this, we can revisit the DL models whose architecture graphs are depicted in Figure 2. The grandparent model is composed of the leaf layer $\{1, 2, 5, 6, 7\}$ and submodel $A = \{3, 4\}$.

```
Input  : Architecture of model M
Output : Best Ancestor and its LCP set Prefix
G ← graph of M based on unique vertex IDs
Prefix ← ∅
Ancestor ← None
foreach A ∈ Ancestors do
    Frontier ← Queue(G.root)
    P ← ∅
    foreach v ∈ G do
        Visits[v] ← 0
    end
    while Frontier ≠ ∅ do
        u ← Frontier.pop_front()
        P ← P ∪ {u}
        foreach v ∈ G[u].out_edges do
            if ∃v ∈ A[u].out_edges then
                Visits[v] ← Visits[v] + 1
                if Visits[v] =
                  max(G[v].in_degree, A[v].in_degree) then
                    Frontier.push_back(v)
                end
            end
        end
    end
    if |P| > |Prefix| then
        Prefix ← P
        Ancestor ← A
    end
end
return (Ancestor, Prefix)
```

Algorithm 1: Find Longest Common Prefix

The parent model uses the same leaf layer $\{1, 2\}$, but there's also a partial match with submodel A, as both parent and grandparent share leaf-layer 3. Therefore, if we do not decompose the A into leaf layers and instead consider it as a single unit, then the longest common prefix is shorter. Similarly, the child model uses the same leaf layers as the parent on both branches except for $\{6, 7\}$. However, 6 is part of submodel B. Again, the longest common prefix would have been shorter if we didn't decompose B into leaf layers.

Moreover, since the child is part of a transfer learning chain and inherits leaf layer from both grandparent and parent, the opportunity to identify longer common prefixes by considering matches at leaf-layer granularity grows proportionally to the number of transfers in the chain. In turn, this increases the effectiveness of transfer learning. Similarly, by organizing the owner maps around leaf layer instead of arbitrary layers, we improve the effectiveness of the de-duplication that can be leveraged by incremental storage since it is strongly correlated with the longest common prefix.

Decomposing a DL model into leaf layers and extracting the architecture graph is non-trivial. We cannot simply match the names of the layers, because identical names may be used for different layer configurations, and, conversely, different names may be used for identical layer configurations. Thus, we "flatten" the model architecture into a single hierarchy of leaf layers. Flattening recursively visits all complex layers starting from the input layer in a deterministic fashion (e.g., a breath-first-search). During this process, we construct two data structures: (1) a compact architecture graph of the leaf layers that assigns unique IDs to the vertices and retains the edges between the vertices; (2) the owner map that initially assigns the new DL model ID as the owner of each vertex.

Then, based on the compact architecture graph of the new DL model (denoted G) we broadcast each longest common prefix query to all providers. The providers work in parallel to identify the best match according to Algorithm 1. Specifically, each provider checks all locally stored DL models that can act as ancestors for transfer learning. These compact architecture graphs of the ancestors are part of the *Ancestors* set. We start with an empty $Prefix$. Then, for each ancestor $A \in Ancestors$, we expand a set of frontier vertices (retained in a queue denoted $Frontier$) starting from the root of G, by visiting all matching vertices that can be reached from a frontier vertex both in A and G. A vertex of G has a match in A if their leaf-layer architectures are identical. We maintain a visit counter $Visits$ for each vertex. When the counter reaches the maximum in-degree of a vertex in both A and G, then the frontier can be expanded by this vertex, which is now eligible to be added to $Prefix$. This process repeats until the frontier cannot be expanded any longer. The worst-case complexity of a single longest common prefix calculation between G and A is $O(min(|V_G|, |V_A|)$, because we may need to visit the all vertices of one of the two graphs and because a directed acyclic graph can have only $O(|V|)$ edges. However, in practice, the longest common prefix is often smaller than the full graph [37] resulting in the complexity only increasing linearly in the number of locally stored ancestors A.

The results returned by the providers are reduced to obtain the best overall match for transfer learning. Then, to obtain the final owner map of G, we need to consult the corresponding owner map of A and assign to all vertices belonging to the longest common prefix the corresponding owner ID from A. The rest of the vertices will retain G as their owner. At this point, we can separately contact the providers to read the tensors that are part of the longest common prefix and to perform the transfer learning. Later, once we are ready to store the new DL model back to the repository, we need to contact a single provider, which will store the compact architecture graph, owner map, and the consolidated, modified tensors.

4.3 Implementation Details

Core implementation of EvoStore: We designed and built a research prototype based on the principles in Section 4.1. Specifically, we follow a client-server design. The clients are libraries that the applications link with and expose a C++ low-level API to issue longest common prefix (LCP) queries (which transparently broadcast and reduce the results) and to read/write subsets of tensors. The client is responsible for interpreting the owner maps and optimizing the RDMA communication with multiple providers in parallel. To this end, we rely on optimized HPC-oriented remote procedure calls (RPCs), as provided by the Mochi [34] collection of composable building blocks. Specifically, we use Thallium, which is a C++ wrapper on top of Mercury and Argobots. The providers use an extensible key-value store abstraction that can be used to group and store tensors and fine granularity either in-memory and persistently using underlying backends such as C++ synchronized memory pools or RocksDB [30].

Most DL applications construct and use DL models through high-level AI runtimes such as Tensorflow and PyTorch. Thus, we implemented a Python library that relies on the low-level C++ client API to expose higher level primitives that handle both LCS queries, transfer learning and retiring of DL models in a user-friendly fashion. At the same time, this approach overcomes the multi-threaded

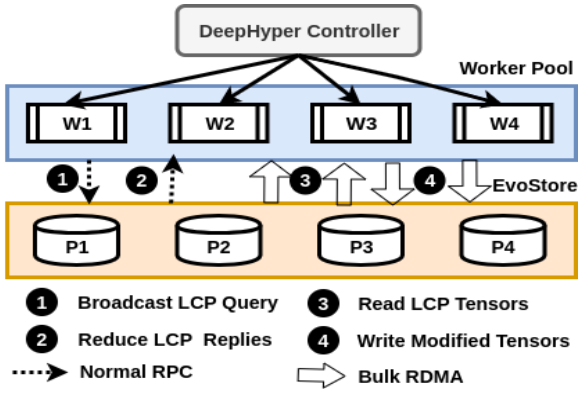

Figure 3: EvoStore architecture diagram.

limitations of Python due to the global interpreter lock, which enables the C++ clients to fully take advantage of asynchrony and parallelism.

Integration with DeepHyper: We integrated EvoStore with *DeepHyper* [2, 3], a NAS framework specifically designed for use on HPC systems. It relies on MPI communications to synchronize the controller with a large number of workers. The design of DeepHyper is modular and allows a plugin-able search strategy on the controller. For this work, we use the aged evolution search strategy [10], which is briefly summarized in Section 2. The workers also support plugin-able evaluation and training of the DL model candidates. We use the default plugin that trains the DL model candidate for one epoch and reports the training accuracy to the controller.

The original DeepHyper implementation does not support transfer learning. Thus, we contributed extensions to enable transfer learning for NAS. Specifically, this involves two aspects. First, we modified the candidate evaluation function used by the workers to (1) broadcast a longest common prefix query to the EvoStore providers, (2) use the reduced result to perform transfer learning from the best match available in EvoStore; (3) train the DL model using a modified strategy that freezes the transferred layers (for one epoch); (4) write back the new DL model to EvoStore; (5) report the training accuracy to the controller. Second, we modified the search strategy of the controller to retire DL model candidates from EvoStore when they are dropped from the population of active DL model candidates. The overall architecture is depicted in Figure 3.

5 EXPERIMENTAL EVALUATION

5.1 Setup

Our experiments were conducted on the Polaris HPC system hosted at Argonne National Laboratory. It comprises 560 compute nodes, each of which is equipped with a 32-core AMD Zen 3 processor, 512 GB of DDR4 RAM, 4 Nvidia A100 GPUs (40 GB high bandwidth memory per GPU), and 2 TB of local SSD storage. The nodes are interconnected with a dual Slingshot 10 network fabric. The compute nodes have access to a Lustre parallel file system of 100 PB capacity. It features 150 object storage targets (OSTs) and 40 metadata targets, with an aggregate data transfer rate of up to 650 GB/s. For software, we use Deephyper 0.4.2, TensorFlow 2.9, Mercury 2.1.0 using OFI verbs provided in libfabric, Argobots 1.1, and Thallium 0.10.0 compiled using GCC-11.2 and Cuda 11.6.5.

5.2 Compared Approaches

We compare our approach with several state-of-the-art approaches and alternative implementations, as described below.

DL Model Repository, Denoted HDF5+PFS. This approach stores and loads DL models to/from HDF5 [17] files. It is part of the standard TensorFlow distribution and is implemented in Keras. The store primitive first copies the content of the tensors into NumPy arrays (by launching a separate Tensorflow execution context), then writes the arrays into an HDF5 file using the HDF5 Python bindings. The load primitive implements the reverse process. This is the fastest available format (alternatives such as *SavedModel* are 2× to 17× slower [27]). The underlying backend for the HDF5 files is the Lustre parallel file system. All aspects of I/O under concurrency are handled transparently by Lustre.

Query Processing Using Redis, Denoted Redis–Queries. This approach implements a centralized DL model metadata repository that enables longest common prefix queries (LCP). We use Redis [15] to store the DL model architectures as key-value pairs. It serves LCP queries by iterating over all key-value pairs and retaining the best match. To handle concurrent queries efficiently, we use native support for atomic put/get operations, which we extended with an optimized transactional support using specialized reader-writer locks (needed to handle concurrent LCP queries and add/retire operations). Redis–Queries is used with HDF5–PFS to provide a full end-to-end equivalent solution comparable with our approach for the end to end comparisons.

When Redis-queries is used with a model repository, several locks are required to handle concurrent read and write operations. We breifly describe these here. For Redis-queries, to add a model a global writer metadata lock is acquired, and then a model architecture specific writer lock is attempted to be acquired. If this succeeds, the reference count for the model is incremented and the metadata lock is dropped. After dropping the metadata lock, the weights are written. When the weights are persisted to the PFS, the metadata writer lock is re-acquired and the model is published to the list of architectures stored. If grabbing the model architecture specific lock fails, the model architecture is already registered and the add procedure ends without storing weights after incrementing the model reference count. Retiring does the inverse. It grabs a writer lock for the metadata, and then it decrement the reference count. If the reference count hits zero, the model architecture specific lock is acquired, the model is unpublished from the list of models, and the metadata lock is freed. Finally the store is storage is freed and the model architecture specific lock is freed. Lastly, are the queries themselves. For queries, a reader lock is acquired for the metadata. After this lock is acquired, the set of published models is iterated over to identify the longest common prefix match. The best identified model has it reference count incremented, and the reader lock is released. After the weights are transferred, the reference count is decremented, if necessary retirement of the model is preformed as described above beginning with the reference count hitting zero.

This careful coordination ensures that only the required locks and types of locks acquired are held only when strictly necessary. The simpler process described above for EvoStore is possible because the model weights and storage are managed jointly by EvoStore and cannot be out of sync with the external storage.

NAS: DeepHyper Without Transfer Learning, Denoted DH–NoTransfer. This approach is the original DeepHyper [2] implementation that uses the same regularized evolution search strategy [10] as described in Section 4.3. Unlike our extensions to DeepHyper, it simply trains each DL model candidate with random weights from scratch, without applying transfer learning. We use this approach as a baseline in our real-life NAS application scenarios, to showcase the benefits of adopting transfer learning from the previous models identified in the context of NAS.

5.3 Methodology

We evaluate the performance and scalability of the approaches described in Section 5.2 both with micro-benchmarks and a real-life NAS application.

Mirco-benchmarks: We designed and implemented a series of micro-benchmarks that make it possible to isolate and study separately both the aspects of incremental storage and query processing. Regarding incremental storage, our micro-benchmark includes an architecture generator that can be configured with a variety of parameters: total model size, number of leaf layers, and variations between the layers. Using this approach, we can simulate different LCP lengths, which enables us to control the total size of the modified tensors as a fraction of the total size of the DL model being written to the DL repository. Regarding query processing, we rely on the DeepSpace library (which is part of DeepHyper) to generate a variety of DL model architectures that are both diverse and showcase complex architectural features with alternative branches and submodels. Thus, we can generate complex leaf-layer architecture graphs that stress the query processing algorithms.

Real-Life Application: NAS for CANDLE:. To show the impact of our approach in end-to-end experiments, we have chosen the *ATTN* problem from the ECP CANDLE project [39]. CANDLE aims to address a series of related cancer research problems using large-scale deep learning. In particular, the ATTN problem aims to model cancer responses to drugs. To this end, it relies on DL models to infer the responses. In this context, NAS is used to find a model architecture that results in extremely high inference accuracy. We use the search space for ATTN from [11] with 3.1×10^{57} candidates.

5.4 Results: Scalability of Incremental Model Storage

Our first series of experiments focuses on the performance and the scalability of incremental storage. To this end, we deploy EvoStore on an increasing number of compute nodes, each of which hosts one provider and four workers that are assigned to a dedicated GPU. EvoStore was configured to use the in-memory KV store backend based on C++ synchronized pools (as mentioned in Section 4.3).

The experiment proceeds in two phases. First, each worker process uses our architecture generator to obtain a model that is 4 GB large and is comprised of 100 evenly-sized layers. Then, each worker simulates a partial write of a new DL model by assuming a fixed number of layers remain frozen, while the rest are modified and need to be written to the DL repository. The number of frozen layers remains the same for all workers. Then, in the next phase, the workers are synchronized using a barrier to start writing their DL models to the DL repository, which produces a highly concurrent write pattern.

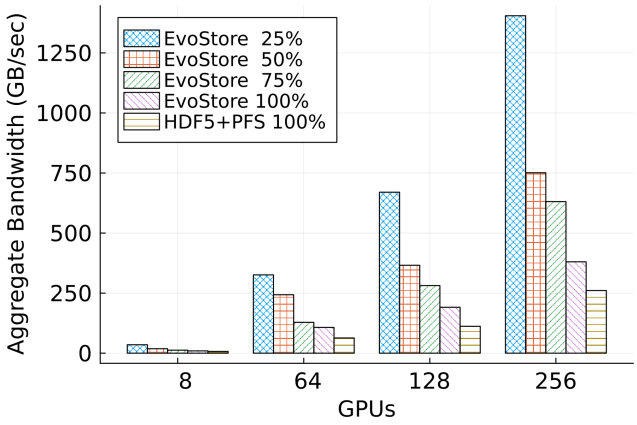

Figure 4: Incremental storage: EvoStore vs. HDF5+PFS. EvoStore takes advantage of incremental storage to achieve up to 5x higher aggregated write bandwidth compared with HDF5+PFS. Furthermore, it consistently achieves 25% higher write bandwidth for full DL model writes. In this case, HDF5+PFS is used without the Redis metadata server.

We vary the number of GPUs from 8 to 256 (weak scalability) and the fraction of the tensors being written (which coincides with the total size of the modified tensors) from 25% to 100% (i.e., full writes). We measure the aggregated write bandwidth, which is calculated as the sum of the individual write bandwidths observed by the workers under concurrency. In turn, the individual write bandwidth is normalized to the total model size (i.e., total model size 4 GB divided by the time taken to store the DL model). The normalization enables us to reason about how effective incremental storage is at accelerating the writing of a DL model for a decreasing fraction of modified tensors.

We compare EvoStore with HDF5+PFS. Recall that HDF5+PFS is not capable of incremental storage. Therefore, regardless of the number of modified tensors, HDF5+PFS always writes the full model, which is why we only need to evaluate a single 100% setting for it. The results are depicted in Figure 4. As can be observed, our approach shows high scalability for an increasing number of GPUs for all fractions ranging from 25% to 100%. This effect is observable despite the decreasing size of the modified tensors, which emphasizes the latency of the writes. We attribute this to our RDMA-centric design that consolidates the modified tensors, which in turn minimizes the write latency. Furthermore, another trend is visible: even for full model writes (100%), our approach is significantly faster than HDF5+PFS at all scales, retaining a 25% higher aggregated write bandwidth. As expected, in the case of partial writes, EvoStore is up to 5x faster. We attribute this difference to the higher overheads of HDF5 serialization and the lower I/O bandwidth of the OSTs.

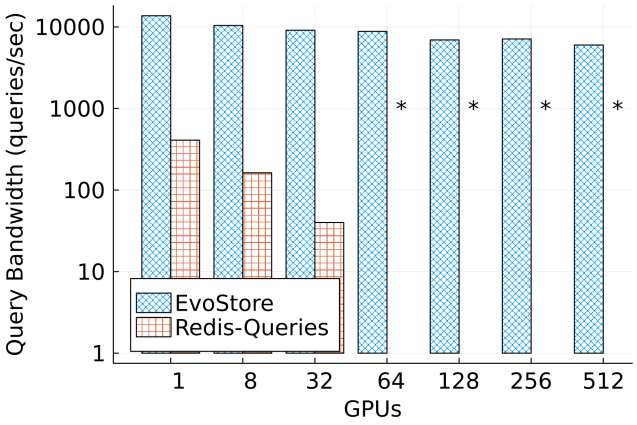

Figure 5: Strong Scalability of LCP Query Processing: 10k queries issued by a variable number of workers against a catalog of 60k DL model architectures. EvoStore maintains high scalability over Redis-Queries and provides up to 3 orders of magnitude faster query processing. Redis-Queries does not scale beyond 32 GPUs (marked with an asterisk). In this figure, no model weights are read/written in either case so HDF5+PFS is not used in the Redis-queries .

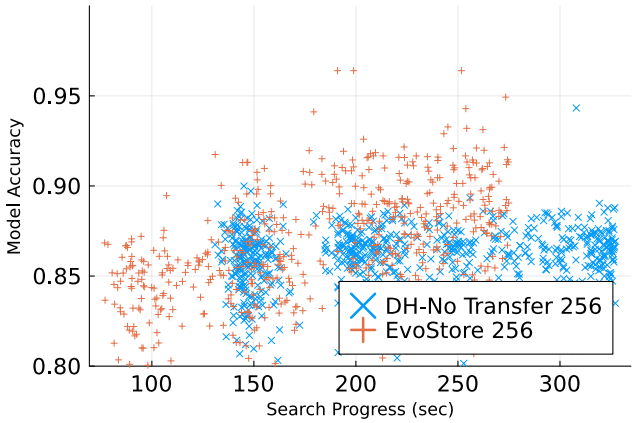

Figure 6: Accuracy of the DL model candidates over time: DeepHyper based on transfer learning facilitated by EvoStore produces candidates with high accuracy (>80%) much faster than DH-NoTransfer. Furthermore, the average accuracy of the candidates is significantly higher and the overall runtime is significantly shorter.

5.5 Results: Scalability of Metadata Query Processing

Next we focuse on the longest common prefix (LCP) queries under concurrency. EvoStore is deployed in the same configuration as above. We compare EvoStore with Redis-Queries, which is deployed on a dedicated compute node.

The experiment consists of two phases. In the first phase, we use the DeepSpace library to generate a large number (60k) of DL model architectures, as described in Section 5.3. These architectures are serialized in JSON format and used to populate the metadata of EvoStore and Redis-Queries. Since the experiment focuses on the queries, the actual DL model tensors are not stored. Then, in the second phase, we synchronize the workers to perform a large number (10k) of LCP queries under concurrency.

We study the strong scalability of both approaches by varying the number of concurrent workers from 1 up to 512 while keeping the total number of queries fixed (10k). Each worker issues the same number of queries, which enables us to enforce load balancing during our study.

Figure 5 shows the results. As can be observed, even for a single worker under no concurrency, EvoStore is more than an order of magnitude faster. We attribute this difference to our optimized compact architecture graph in-memory representation, which enables our LCP algorithm to quickly iterate over the entire catalog of 60k DL models. On the other hand, Redis-Queries uses the Redis API to iterate over the same catalog, whose performance is significantly worse. Furthermore, our approach maintains excellent scalability: despite an increasing number of concurrent workers, the query processing throughput remains stable. This observation underlines the efficient implementation of broadcasts and reductions in EvoStore, which effectively distributes the LCP query workload among the providers. On the other hand, the Redis-Queries throughput sharply drops even at moderate scales and stops beyond 32 GPUs.

5.6 Results: Scalability of NAS for CANDLE

Our last series of experiments evaluate the real-life end-to-end benefits of our approach for NAS. Specifically, we solve the *ATTN* problem of CANDLE detailed in Section 5.3.

To enable a fair comparison, we configure the DeepHyper controller to use a fixed pseudo-random number generator seed, which improves reproducibility of results. We limit the maximum number of explored DL model candidates (population size) to 1000 and consider two scales: 128 and 256 workers (each worker on a dedicated GPU).

It is important to compare the accuracy over time of both DeepHyper without transfer to a method that uses transfer. To our knowledge, our approach is the first to fully implement a transfer learning pipeline in network architecture search. Prior to our work, it is unclear how much benefit there is to model quality vs simply not performing transfers which introduces I/O overhead. It was also possible that models discovered in the same NAS process could decrease or not effect the quality of future models. However, as we will show in subsequent paragraphs, it both improves quality of model over all, but also decrease the time to identify a high quality model.

First, we discuss the overall DL model accuracy and end-to-end runtime of EvoStore vs. DH-NoTransfer for the maximum scale (256 GPUs). The results are depicted in Figure 6. As can be observed, with EvoStore, DeepHyper produces high-quality (with an accuracy of over 80%) DL model candidates much faster than the original version (without transfer learning). Specifically, with transfer learning, high-quality candidates are found almost immediately after the search process has started, while in the case of DH-NoTransfer, this happens no sooner than 1/3 into the search process. Furthermore, the overall quality of the DL model candidates is much higher with

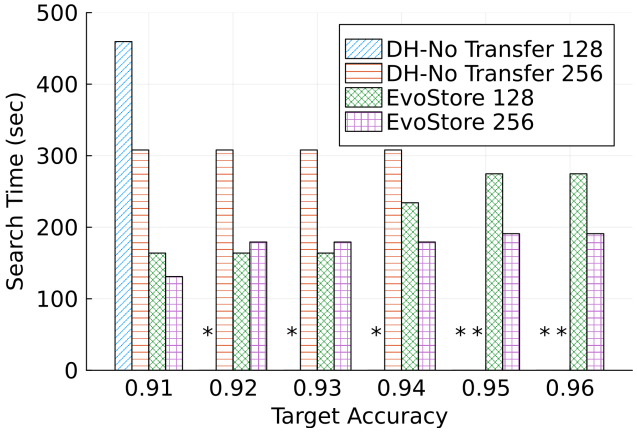

Figure 7: Time To Target Objective: DeepHyper based on transfer learning facilitated by EvoStore produces DL model candidates above target accuracy faster than DH–NoTransfer at different scales (128, 256 GPUs). In many cases, DH–NoTransfer cannot reach a given target accuracy at all (marked with asterisk).

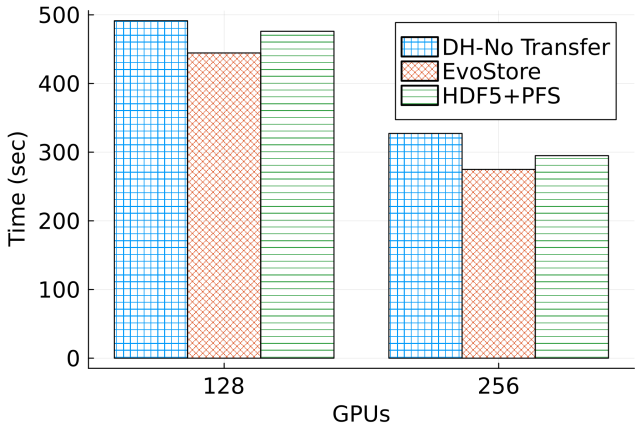

Figure 8: End-to-end runtime: transfer learning with EvoStore reduces the end-to-end runtime compared with DH–NoTransfer. The gap increases as the number of GPUs increases. In this figure, HDF5+PFS uses a metadata server implemented in Redis

transfer learning, both in terms of averages and top performers. Finally, we observe that transfer learning reduces the end-to-end runtime to explore 1000 candidates by almost 30%. Thus, we conclude our approach has a double advantage: it simultaneously increases the overall population quality and significantly reduces the end-to-end NAS runtime.

Next, we discuss the runtime needed to reach a given target objective: finding a DL model candidate with an accuracy above a given threshold. We vary both the scale (128, 256 GPUs) and the accuracy threshold. The results are depicted in Figure 7. As can be observed, in the case of 128 GPUs, transfer learning based on EvoStore is 3x faster than DH–NoTransfer at finding a DL model candidate above 90% accuracy. After that, DH–NoTransfer cannot find a better candidate at all. Moving to 256 GPUs, DH–NoTransfer takes significantly less than in the case of 128 GPUs to find a candidate above 90% accuracy. Even so, transfer learning is 2.5x faster in achieving the same objective. As the target accuracy increases, DH–NoTransfer manages to find suitable candidates with up to 94% accuracy. On the other hand, our approach keeps finding suitable candidates faster above 96% accuracy both with 128 and 256 GPUs.

A careful observer may notice that the search time of EvoStore-256 vs EvoStore-128 longer in Figure 7 only at the .91 accuracy threshold. Parallelism plays some role in how the candidates are generated. Even with a fixed seed, there is non-determinism in the order that model evaluations are completed. This non-determinism means that the search algorithm may have a different set of candidates to evolve from even when using the same seed. Thus, 128 GPUs can beat 256 GPUs because low accuracy thresholds may be reachable with fewer luckier candidates. At a higher accuracy threshold, luck plays a smaller role and the number of candidates needed to reach it is similar for both 128 and 256 GPUs.

Next, we focus on the weak scalability of the end-to-end runtime to completely evaluate all 1000 candidates for an increasing number of GPUs (128, 256). In addition to DH–NoTransfer, we also compare EvoStore with transfer learning based on HDF5+PFS (which uses Redis–Queries for metadata management). As can be observed in Figure 8, HDF5+PFS is close to DH–NoTransfer, which implies that the training speed-up enabled by freezing the transferred layers are offset by high metadata and I/O overheads, whose effect is amplified under concurrency. This makes HDF5+PFS a poor choice as a DL model repository compared to our approach under realistic scenarios. On the other hand, our approach exhibits a significant reduction in the end-to-end runtime compared with both approaches, which is close to the lower bound (i.e., assuming an infinitely fast DL model repository). Specifically, the interactions with EvoStore (LCS queries, load/store) cause an overhead that is less than 2% of the end-to-end runtime. Thus, EvoStore shows negligible overheads and remains scalable under realistic scenarios, enabling the training to fully take advantage of frozen layers.

We studied the above findings in greater detail by plotting the evolution of the training tasks in the case of 128 GPUs. Figure 9 depicts the start and finish times (X axis) of each task running on each GPU (Y axis) for all three approaches. As expected, in the case of DH–NoTransfer, the training tasks roughly start and finish in waves. On the other hand, both in the case of EvoStore and HDF5+PFS the pattern formed by the training tasks becomes irregular, hinting at uneven training durations due to a variable number of frozen layers [1]. In the case of HDF5+PFS, the training tasks take visibly longer to finish, confirming the higher overheads. We studied a breakdown of these overheads and found that up to 18% is caused by higher I/O overheads, and up to 24% is caused by higher metadata overheads. The rest of 58% is caused by the higher variability of the task runtimes (standard deviation of 17.91 vs. 16.15 in the case EvoStore), which results in delays on the controller. Thus we conclude our approach shows another promising advantage over HDF5+PFS: improved stability of techniques that leverage transfer learning.

[1] As the search progresses, we start with 0% and eventually increase to ≈ 50% layers frozen as the search progresses for the reasons outlined in [37]

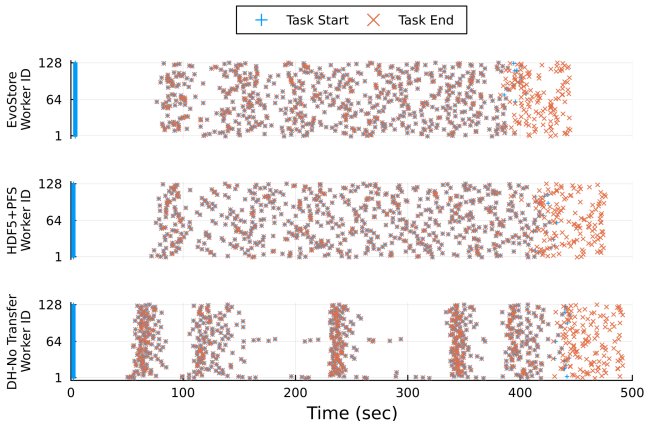

Figure 9: Task evolution expressed as start/finish timestamp (X axis) on each of 128 GPUs (Y axis). DH-No Transfer has greater contention with a stronger wave behavior than EvoStore and HDF5+PFS. In this figure, HDF+PFS uses Redis for a metadata server

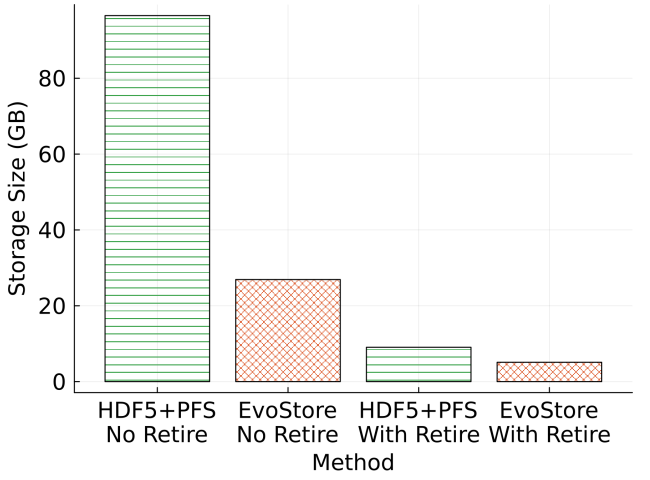

Figure 10: Storage Space Overhead: EvoStore uses 3.5× less space than HDF5+PFS when DL model candidates are not retired by the NAS search strategy. When the candidates are retired, EvoStore uses 5.3× less space that in the previous case and 1.7× less space than HDF5+PFS. Retirement for HDF5+PFS is implmemented using Redis to track metadata

Finally, we discuss the storage space used by EvoStore vs. HDF5+PFS with Redis for metadata. We compare both approaches in two scenarios: when DL model candidates are allowed to accumulate in the repository without being retired, and, respectively, when they are retired by the search strategy as soon as they are eliminated from the active population. As can be observed in Figure 10, there is a large gap between EvoStore and HDF5+PFS, both with and without retirement. This demonstrates that the DL model candidates have a high degree of similarity, which leads to long common prefixes and therefore a large degree of transfer learning. Thanks to incremental storage, EvoStore avoids an explosion of storage space overhead both with and without retirement, which makes it is a viable strategy in both cases to unlock the multiple benefits of transfer learning discussed above (higher quality of DL model candidates, less time to target candidate accuracy and shorter end-to-end runtime).

6 CONCLUSIONS

We introduce EvoStore, a distributed DL model repository designed to store and retrieve DL models organized into a lineage, which is an instrumental capability for transfer learning and fine-tuning. To this end, we contributed with several concepts (incremental storage based on owner maps, specialized metadata query support, and garbage collection, fine-grain tensor storage) and readily available implementation integrated within the AI ecosystem.

We demonstrated the benefits of our approach in comparison with several approaches at scale (up to 512 GPUs), both using micro-benchmarks and a real-life NAS problem, which we extended to take advantage of transfer learning. Results show up to 3x higher DL model write throughput, up to 1000x faster metadata queries, 3x faster NAS discovery of high-quality models, 25% faster end-to-end runtime and up to 5x less storage space utilization compared with state-of-art.

Encouraged by these promising results, in future work we plan to explore several avenues: (1) different NAS strategies with more frequent transfer learning (e.g., zero-cost proxies with a few iterations instead of a full epoch); (2) leverage ancestry and provenance queries to explain the quality of the DL model candidates; (3)apply our approach to other transfer learning scenarios different from NAS (notably continual learning).

Zero cost proxies offer the opportunity to reduce the training costs. With reduced training costs, the percentage of the workflow dominated by I/O increases potentially requiring further improvements over the approach described here, but also possibly present new quality trade offs than what is presented above.

We also want to explore if the lineage of the models can be used to explain or debug model performance on a particular type of data similar to how git does for source code. This kind of workflow poses the opportunity to do aggressive pre-fetching of models to workers given known access pattern.

Lastly, applying the approaches outlined above to continual learning presents a different set of challenges and opportunities. In continual learning, there may be additional factors to consider when choosing which model to transfer from such as the age of the model.

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. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

REFERENCES

[1] The TensorFlow Authors. 2022. Tensorflow SavedModel. https://www.tensorflow.org/guide/saved_model.
[2] Prasanna Balaprakash, Romain Egele, Misha Salim, Stefan Wild, Venkatram Vishwanath, Fangfang Xia, Tom Brettin, and Rick Stevens. 2019. 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* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 37, 33 pages. https://doi.org/10.1145/3295500.3356202

[3] Prasanna Balaprakash, Michael Salim, Thomas D. Uram, Venkat Vishwanath, and Stefan M. Wild. 2018. DeepHyper: Asynchronous Hyperparameter Search for Deep Neural Networks. In *HiPC'18: The IEEE 25th International Conference on High Performance Computing*. IEEE, Bengaluru, India, 42–51.

[4] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Comput. Surv.* 52, 4, Article 65 (aug 2019), 43 pages. https://doi.org/10.1145/3320060

[5] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* 13, null (feb 2012), 281–305.

[6] Francois Chollet. 2018. *Deep learning with Python*. Manning Publications, Shelter Island, NY.

[7] Georgi Dikov and Justin Bayer. 2019. Bayesian Learning of Neural Network Architectures. In *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 89)*, Kamalika Chaudhuri and Masashi Sugiyama (Eds.). PMLR, 730–738. https://proceedings.mlr.press/v89/dikov19a.html

[8] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Yang Zonghan, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfei Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence* 5 (2023), 1–16. https://doi.org/10.1038/s42256-023-00626-4

[9] Xuanyi Dong and Yi Yang. 2020. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=HJxyZkBKDr

[10] Romain Égelé, Prasanna Balaprakash, Isabelle Guyon, Venkatram Vishwanath, Fangfang Xia, Rick Stevens, and Zhengying Liu. 2021. AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data. In *SC '21: The 2021 International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis, USA, Article 30, 14 pages.

[11] Romain Égelé, Prasanna Balaprakash, Isabelle Guyon, Venkatram Vishwanath, Fangfang Xia, Rick Stevens, and Zhengying Liu. 2021. AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3458817.3476203

[12] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. {Check-N-Run}: A checkpointing system for training deep learning recommendation models. In *NSDI'22: The 19th USENIX Symposium on Networked Systems Design and Implementation*. Renton, USA, 929–943.

[13] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *Journal of Machine Learning Research* 20, 1 (2019), 1997–2017.

[14] Teven Le Scao et al. 2023. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. arXiv:2211.05100 [cs.CL]

[15] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. 2016. Using Storage Class Memory Efficiently for an In-Memory Database. In *SYSTOR '16: The 9th ACM International Systems and Storage Conference*. Haifa, Israel, Article 21, 1 pages.

[16] Antonio Gulli and Sujit Pal. 2017. *Deep Learning with Keras*. Packt Publishing.

[17] HDF5. 2023. Hierarchical Data Format. https://www.hdfgroup.org/HDF5/.

[18] Jia, Yngqing and Shelhamer, Evan. 2014. Model Zoo. https://caffe.berkeleyvision.org/model_zoo.html.

[19] Kirthevasan 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*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf

[20] Ruslan Kuprieiev, skshetry, Dmitry Petrov, Paweł Redzyński, Peter Rowlands, Casper da Costa-Luis, Alexander Schepanovski, Ivan Shcheklein, Gao, Batuhan Taskaya, David de la Iglesia Castro, Jorge Orpinel, Fábio Santos, Ronan Lamy, Aman Sharma, Dave Berenbaum, daniele, Zhanibek, Dani Hodovic, Nikita Kodenko, Andrew Grigorev, Earl, Nabanita Dash, George Vyshnya, maykulkarni, Max Hora, Vera, and Sanidhya Mangal. 2022. DVC: Data Version Control - Git for Data & Models. https://doi.org/10.5281/zenodo.7387773

[21] Liam Li and Ameet Talwalkar. 2020. Random Search and Reproducibility for Neural Architecture Search. In *PMLR'20: The 35th Uncertainty in Artificial Intelligence Conference*. Tel Aviv, Israel, 367–377.

[22] Zhuozhao Li, Ryan Chard, Logan Ward, Kyle Chard, Tyler J. Skluzacek, Yadu Babuji, Anna Woodard, Steven Tuecke, Ben Blaiszik, Michael J. Franklin, and Ian Foster. 2021. DLHub: Simplifying publication, discovery, and use of machine learning models in science. *J. Parallel and Distrib. Comput.* 147 (2021), 64–76.

[23] Jason Liang, Elliot Meyerson, Babak Hodjat, Dan Fink, Karl Mutch, and Risto Miikkulainen. 2019. Evolutionary Neural AutoML for Deep Learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

[24] Hongyuan Liu, Bogdan Nicolae, Sheng Di, Franck Cappello, and Adwait Jog. 2021. Accelerating DNN Architecture Search at Scale Using Selective Weight Transfer. In *CLUSTER'21: The 2021 IEEE International Conference on Cluster Computing*. Portland, USA, 82–93.

[25] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations (ICLR)*.

[26] Pablo Ribalta Lorenzo, Jakub Nalepa, Luciano Sanchez Ramos, and José Ranilla Pastor. 2017. Hyper-Parameter Selection in Deep Neural Networks Using Parallel Particle Swarm Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*.

[27] Meghana Madhyastha, Robert Underwood, Randal Burns, and Bogdan Nicolae. 2023. DStore: A Lightweight Scalable Learning Model Repository with Fine-Grained Tensor-Level Access. In *ICS'23: The 2023 International Conference on Supercomputing*. Orlando, USA, 133–143. https://hal.inria.fr/hal-04119926

[28] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *FAST'21: 19th USENIX Conference on File and Storage Technologies*. 203–216.

[29] Bogdan Nicolae, Jiali Li, Justin Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *CGrid'20: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. Melbourne, Australia, 172–181.

[30] Keren Ouaknine, Oran Agra, and Zvika Guz. 2017. Optimization of RocksDB for Redis on Flash. In *ICCDA '17: The 2017 International Conference on Compute and Data Analysis*. Association for Computing Machinery, Lakeland, USA, 155–161.

[31] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameters Sharing. In *ICML'18: The 2018 International Conference on Machine Learning*.

[32] Pytorch. 2018. PyTorch Hub. https://pytorch.org/hub/.

[33] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized Evolution for Image Classifier Architecture Search. In *AAAI'19: The 2019 AAAI Conference on Artificial Intelligence*.

[34] Robert B. Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Qing Zheng. 2020. Mochi: Composing Data Services for High-Performance Computing Environments. *Journal of Computer Science and Technology* 35, 1 (2020), 121–144.

[35] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. 2018. A Survey on Deep Transfer Learning. In *ICANN'18: 27th International Conference on Artificial Neural Networks and Machine Learning*, Vol. 11141. Rhodes, Greece, 270–279.

[36] TensorFlow. 2023. TensorFlow Hub. https://www.tensorflow.org/hub/overview.

[37] Robert Underwood, Meghana Madhyastha, Randal Burns, and Bogdan Nicolae. 2023. Understanding Patterns of Deep Learning Model Evolution in Network Architecture Search. In *HiPC'23: The 2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics*. Goa, India, 97–106.

[38] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. 2023. Neural Architecture Search: Insights from 1000 Papers. arXiv:2301.08727 [cs.LG]

[39] Justin M. Wozniak, Rajeev Jain, Prasanna Balaprakash, Jonathan Ozik, Nicholson T. Collier, John Bauer, Fangfang Xia, Thomas Brettin, Rick Stevens, Jamaludin Mohd-Yusof, Cristina Garcia Cardona, Brian Van Essen, and Matthew Baughman. 2018. CANDLE/Supervisor: a workflow framework for machine learning applied to cancer research. *BMC Bioinformatics* 19, 18 (2018).

[40] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. 2023. A brief overview of ChatGPT: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica* 10, 5 (2023), 1122–1136.

[41] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. 2019. NAS-Bench-101: Towards Reproducible Neural Architecture Search. In *ICML'19: The 2019 International Conference on Machine Learning*.

[42] Yiyang Zhao, Linnan Wang, Yuandong Tian, Rodrigo Fonseca, and Tian Guo. 2021. Few-shot Neural Architecture Search. In *ICML'21: The 2021 International Conference on Machine Learning*.

[43] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *ICLR'17: The 2017 International Conference on Learning Representations*.