ModelKeeper: Accelerating DNN Training via Automated Training Warmup

Fan Lai, Yinwei Dai, Harsha V. Madhyastha, Mosharaf Chowdhury
University of Michigan

Abstract

With growing deployment of machine learning (ML) models, ML developers are training or re-training increasingly more deep neural networks (DNNs). They do so to find the most suitable model that meets their accuracy requirement while satisfying the resource and timeliness constraints of the target environment. In large shared clusters, the growing number of neural architecture search (NAS) and training jobs often result in models sharing architectural similarities with others from the same or a different ML developer. However, existing solutions do not provide a systematic mechanism to identify and leverage such similarities.

We present ModelKeeper, the first automated training warmup system that accelerates DNN training by repurposing previously-trained models in a shared cluster. Our key insight is that initializing a training job's model by transforming an already-trained model's weights can jump-start it and reduce the total amount of training needed. However, models submitted over time can differ in their architectures and accuracy. Given a new model to train, ModelKeeper scalably identifies its architectural similarity with previously trained models, selects a parent model with high similarity and good model accuracy, and performs structure-aware transformation of weights to preserve maximal information from the parent model during the warmup of new model weights. Our evaluations across thousands of CV and NLP models show that ModelKeeper achieves 1.3×–4.3× faster training completion with little overhead and no reduction in model accuracy.

1 Introduction

Modern machine learning (ML) clusters train thousands of deep neural networks (DNNs) every day [37, 67]. For a specific ML task, ML developers often start with exploring various model architectures using Neural Architecture Search (NAS) to find the one with desired accuracy [77]. In preparation for model serving, developers may train tens of models to customize the latency-accuracy trade-off across hardware [21, 35], to organize weak and powerful DNNs into different inference stages for fast feature extraction [20], and/or to dynamically select tens of models and combine their predictions to maximize ensemble accuracy [26, 30, 65]. Overall, from inception to deployment, ML development often requires training hundreds of models across developers [62, 75].

Naturally, many recent advances in ML training optimizations have focused on faster DNN execution, e.g., by increasing parallelism [50, 70], improving communication [40, 55], or increasing GPU utilization [28, 68, 69, 73]. However, little has been done to exploit the natural similarity between models that are trained as part of the same NAS process, models targeting the same ML task in different hardware, or models embedded in different applications. Indeed, our analysis of three large CV and NLP model zoos shows that more than 60% of widely-used models can find an architecturally similar counterpart within the same zoo (§2.2).

In this paper, our key insight is that *one can reduce the amount of training needed for model convergence by leveraging a well-trained model's weights to warm up the training of a new model.* This is because any DNN model is fundamentally a computation graph of tensor weights and operators; transforming weights of trained models with similar architectures to a new model can accelerate model convergence (similar to transfer learning [60, 71] but across architectures).

Despite the potential for large benefits, there exists little systematic support for automated repurposing of weights. Today's frameworks may provide pre-trained models, but are limited to a few models and specific datasets, and/or require domain knowledge to manually search, transfer and contribute a trained model's weights [6]. As such, ML developers have to train models from scratch more often [56]. A few recent AutoML frameworks (e.g., Retiarii [77]) repurpose trained models. However, they are limited to individual jobs within a NAS task because they rely on the lineage of model mutation to enable the transfer. When models are submitted by various developers and/or frameworks with distinct architectures and performance requirements, these solutions do not apply.

We introduce ModelKeeper, a cluster-wide training warmup system, to reduce the training execution needed for model convergence via automated model weight transformation (§3). ModelKeeper adaptively manages a collection of trained models (i.e., *model zoo*) from prior training jobs corresponding to different ML tasks. For a new training job, ModelKeeper selects and transforms a trained model's weights (i.e., *parent model*) to the training model (i.e., *query model*) before training takes place. It can benefit various ML applications, including exploratory training (e.g., improving Retiarii [77] further) and general training (e.g., using PyTorch [9]) of CV/NLP models, with few-lines-of-code change.

ModelKeeper addresses two primary challenges toward selecting a suitable parent model and repurposing its weights. First, ModelKeeper must determine similarity between two models (§4.1). Intuitively, we can treat each DNN model as a directed graph, where nodes represent tensors (layers) and edges represent data flows, and use heuristics for the classic NP-hard graph edit distance problem [31] to find the matching similarity. However, maximizing matching by skipping nodes can be harmful because the computation of each tensor affects that of the subsequent ones in a trained parent model. To this end, we present a structure-aware dynamic programming approach to capture the similarity (transformable tensor weights) between two models. To scale to real-world zoos with thousands of models, we then introduce a two-stage hierarchical search algorithm to identify similar models efficiently.

Second, perfect matching is unlikely as two models are seldom identical. Therefore, given many candidate parent models with different similarity scores and each with different accuracy, which one to pick and then how to transform its weights to the query model (§4.2)? A more similar parent model enables transforming more weights, while a more accurate one implies a better training jump start after the transformation. When the two are at odds, we adopt a bucketing heuristic: potential parent models are put into different buckets in terms of their similarity to the query model, grouping comparable parent models together. We then pick the most accurate parent from the bucket containing the most similar parent models. Nevertheless, tensor mappings from the parent to the query model can be incomplete (e.g., due to non-identical architectures). To preserve maximal parent model information, we introduce width and depth operators to transform parent model weights into the query model with negligible overhead.

We have integrated ModelKeeper with four popular ML frameworks (§5): Ray [49], AutoKeras [41], MLFlow [75], and Microsoft NNI with Retiarii backend [77].[1] Our evaluations across thousands of DNN training jobs in CV and NLP applications (§6) show that ModelKeeper can save 23%–77% total amount of training needed (i.e., 1.3×–4.3× faster training) than the state-of-the-art without model accuracy drop, while efficiently serving cluster-scale warmup requests.

Overall, we make the following contributions in this paper:

1. We present ModelKeeper, a system to enable automated training warmup for faster DNN training in clusters;
2. In order to maximize training speedup, we demonstrate how to scalably compute similarities between models and how to transform an already-trained model's weights to a yet-to-be trained model with little overhead;
3. We integrate ModelKeeper with multiple advanced ML frameworks, and evaluate it across thousands of CV and NLP models to show large improvements.

[1] ModelKeeper is available at https://github.com/SymbioticLab/ModelKeeper.

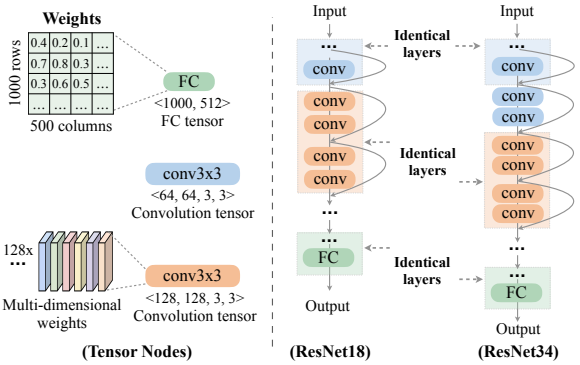


Figure 1: ***A DNN model is essentially a graph of tensors. Model outputs are determined by tensor weights and their control flow.***

2 Background and Motivation

2.1 DNN Model Training

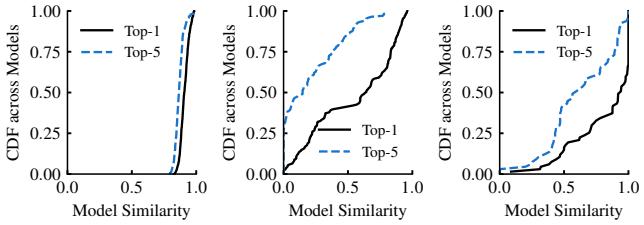
Modern DNN frameworks represent DNN computations as a directed computation graph with tens to thousands of nodes across branches (Figure 1) [9,38]. Each node implies a mathematical tensor operation (e.g., matrix multiplication or convolution) along with its tensor weights and input, where weights are n-dimensional arrays typically consisting of floats. DNN training often covers thousands of iterations across mini-batches of data to minimize the training loss. In each iteration, the computation graph takes a data mini-batch as the input, and performs a (1) *forward pass*, where each node conducts the tensor operation on the output of parent nodes to get the training loss regarding the model output and ground truth; and a (2) *backward pass*, which updates the weight values, from the last to front tensors, using the gradients derived by the training loss with respect to the current weight. Therefore, the DNN model is essentially a graph of weights orchestrated by tensor operators, and training searches the best weight values.

2.2 Opportunities for Repurposing Models

In this paper, we focus on *reducing the amount of training needed* to train a new model by automatically repurposing the weights of previously trained models. Our approach of warming up the weights of a new model before its training starts is based on the following observations.

Pervasive model similarity. With the rapid increase in the number of ML training jobs in datacenters [28,37], similarities between training jobs are increasing too [67]:

- First, for a specific ML task, ML developers often explore various model architectures using Neural Architecture Search (NAS) to find the preferred model architecture (e.g., better capacity-accuracy frontier [77]), or to investigate the performance consistency of new optimizations across models (e.g., ML ablation study) [48]. For example, Microsoft tuning clusters perform as many as 75 exploratory training jobs in median for user apps [46].
- Second, in preparation for ML deployment, developers can train dozens of models to either customize the latency-



(a) NASBench (b) Imgclsmob (c) HuggingFace

Figure 2: ***Pervasive model similarity in today's model zoos. We measure the top-1 and top-5 architectural similarities of each model to other models, and report the distribution across models. 1 indicates identical model architectures.***

accuracy tradeoff across hardware (e.g., in video analysis systems [35, 39]), or to dynamically select tens of models and combine their predictions in order to maximize accuracy under changing loads in today's ensemble-serving systems [30, 65] (e.g., AWS Autogluon [26]).

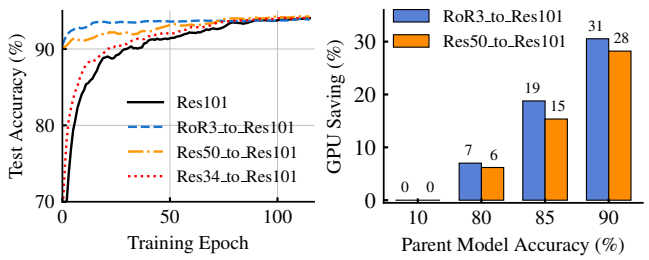
- Third, the potential for similar models increases with increasing users. For example, over 100K ML models are submitted to Kaggle competitions each month [3]. Each competition can have thousands of participants developing their models independently, and participants are reported to have trained many similar models [22].

Indeed, our analysis of three large public model zoos – Imgclsmob [10] for ImageNet classification (435 models), HuggingFace [2] for English text generation (2.5K models), and NASBench [24] for NAS task (16K models) – reinforces these observations. Figure 2 reports the pairwise architectural similarity across models in each model zoo. We measure the similarity of each model to other zoo models[2] in terms of the normalized graph edit distance of two directed model computation graphs [27] ($\in$ [0, 1]), where 1 indicates identical graphs. We observe that more than 60% of the models have at least one other model (top-1) in the zoo with a similarity over 0.6. Pervasive similarity is prominent in all these model zoos because modern models often rely on similar architecture designs but with wider/deeper layers or branches. For example, convolution layers are widely-used in CV models [33, 36], while NLP models are often stacked by attention layers [63].

Similar models can warm-start training. Recent theoretical [60] and empirical [71] efforts from the transfer learning community show that inheriting well-trained parent model weights can speed up the training of a new model, because this warm start enables an informed weight initialization (e.g., training from the basin of loss curvature). Yet, different from their focus that manually transfers the same model across datasets for better model generalization accuracy [53, 78], we notice that transforming a trained model's weights to a new model (i.e., across architectures) can accelerate its training.

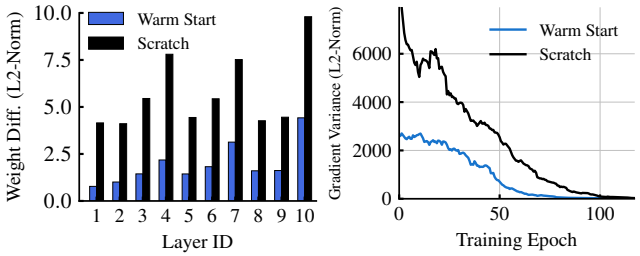
Consider the training of ResNet101 on CIFAR-10 dataset

[2]To avoid over-optimistic identification of model similarity, we removed identical models in each zoo and focus on different model architectures.



(a) Warm start accelerates training. (b) Parent model accuracy matters.

Figure 3: ***Transferring model weights from well-trained models with similar architectures can accelerate new model training.***



(a) Smaller divergence to the optimal. (b) Smaller gradient variance.

Figure 4: ***Warm start provides better initial weights search space. We use RoR3 to warm start ResNet101.***

as an example. We copy the tensor weights of a well-trained parent model (e.g., ResNet50 or RoR3 [76]) to the ResNet101 tensor if two tensors have identical properties (e.g., same operator and weight dimensions), while the rest of the training proceeds as normal. We notice that (1) warmup training can reduce the amount of training needed, while obtaining the same final accuracy to that of training from scratch with random weight initialization (Figure 3(a)); and (2) the savings are more encouraging when inheriting from more similar models – similarity of ResNet34, ResNet50, and RoR3 to ResNet101 is 0.19, 0.48, and 0.85, respectively – and better performing models (Figure 3(b)), which respectively determine whether it is possible and beneficial to transform the weights.

These improvements are because they speed up the search in the space of weight values. If we consider ResNet101 as an example, (1) warming it up using RoR3's weights before training starts results in a smaller distance to the final weights achieved when the model converges (Figure 4(a)), and (2) during the training, this informed weight initialization enables smaller gradient variance (i.e., more consistent gradient directions) towards the basin of loss curvature (Figure 4(b))), thus requiring fewer iterations to convergence in theory [12, 53].

3 ModelKeeper Overview

ModelKeeper is an automated training warmup system for various ML tasks that accelerates DNN training by warm-starting models with weights from already-trained models.

Design Space Large training clusters are shared between users with varying expertise, and they can train a large num-

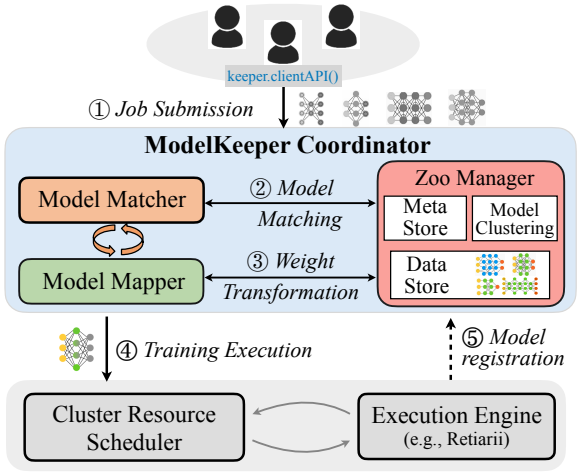


Figure 5: ***ModelKeeper architecture. It can run as a cluster-wide service to serve different users and/or frameworks.***

ber of jobs with different model architectures. Consequently, ModelKeeper must minimize the information needed and overhead incurred for each training model (i.e., *query model*), while offering users the flexibility in their request (e.g., using ImageNet model zoo to warm start models on other image datasets). In fact, determining which dataset (model zoo) as the source to transfer is as yet an open problem in the transfer learning community [45, 71, 78]. ModelKeeper is complementary to and benefits existing ML efforts as it automates training warmup (e.g., searching, transforming, and contributing a trained parent model's weights) for a given model zoo, instead of making the developer keep tracking all models and handcraft which model to repurpose [6]. We empirically show that ModelKeeper can benefit the model training across datasets too (§6.4).

For a given model zoo, the effectiveness of transforming parent model weights relies on two key aspects: (i) *Model similarity*: it dictates the similarity of two model architectures, including the weights shape and operation type of a tensor; and (ii) *Parent model accuracy*: it determines the value of transformation. Having architectural similarity is the prerequisite to transforming more weights information of a parent model, while better parent model accuracy implies potentially better warm start after transformation.

As such, ModelKeeper should repurpose a parent model with large similarity and better accuracy. We provide the theoretical analysis to support why ModelKeeper can benefit model convergence following this principle in Appendix A.

System Components ModelKeeper is a complementary system to existing ML training (Figure 5), and has integrations with various frameworks (e.g., Microsoft NNI [4] and Ray [49]). It consists of the remote coordinator, which serves user query models before their training executes, and the client agent that allows users to submit model warmup requests. ModelKeeper coordinator employs three key components to warm up models by transforming a trained model's weights:

- *Model Matcher*: to identify architecturally similar models in the zoo of trained models;
- *Model Mapper*: to select a zoo model with good architectural similarity and accuracy as the parent model, and transforms the parent model weights to the query model;
- *Zoo Manager*: to adaptively manage zoo models that can be submitted from users to support transformation at scale.

Figure 6 reports the example interface on the client agent, where the user benefits from ModelKeeper with a few lines of code in training (Coordinator interfaces are in Section 5).

```
from modelkeeper import ModelKeeperClient

def training_with_keeper(model, dataset):
  # Create client session to keeper coordinator
  keeper_client = ModelKeeperClient(coordinator_ip)
  warmed_model, meta = keeper_client.query_for_model(
              model, meta={'data': 'Flowers102',
              'task':'classification', 'tags': None})

  acc = train(warmed_model, dataset) # Training starts

  # Register model to ModelKeeper when training ends
  keeper_client.register_model(warmed_model,
          meta={'data': 'Flowers102', 'accuracy': acc,
          'task': 'classification', 'tags': None})
  keeper_client.stop()
```

Figure 6: ***Code snippet of ModelKeeper client service APIs.***

Training Lifecycle When the developer creates a new training job, ① she first initiates a client connection to the remote ModelKeeper coordinator, and then issues a query with the specified job meta information. ModelKeeper client agent will automatically extract the model information needed (e.g., model computation graph) and issue a request (mostly size < 1 MB) to the coordinator. ② Upon receiving the request, Matcher consults its metadata store, identifies zoo models that the user can access and meet the specified tag (e.g., name of the preferred parent models), and measures their architectural similarity to the query model. ③ Mapper selects a parent model with large architectural similarity and good accuracy out of these zoo models. Thereafter, it loads model weights of this parent model from the data store, and transforms parent model weights, based on pairwise tensor mapping from the Matcher, to the query model. Note that this transformation only updates tensor weight values, while others (e.g., model architecture) remain the same. ④ The coordinator responds to the developer with warmup model weights, and the rest of the training remains as usual. ⑤ When the training completes, ModelKeeper can automatically register the trained model to the Zoo Manager to benefit future jobs.

4 ModelKeeper Design

In large shared clusters, models are often submitted by various developers and/or frameworks with diverse architectures at different points in time. The large variety in model architectures and accuracy characteristics lead to novel system challenges in automating weight transformation from a parent model with high similarity and better accuracy:

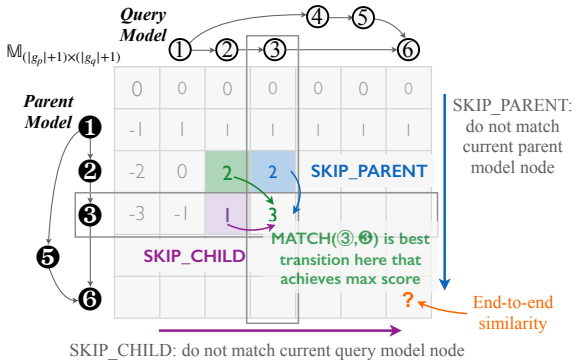


Figure 7: ***ModelKeeper relies on dynamic programming-like heuristics to measure graph-level model architectural similarity.***

- Having similar model architectures is the prerequisite to transforming weights across architectures. How to identify more architecturally similar zoo models (§4.1)?
- As the similarity and accuracy of many potential parent models can come at odds, which one to pick and then how to transform its weights to the query model even in the presence of non-identical architectures (§4.2)?
- New training jobs and trained models can join the cluster on the fly. How to serve user warmup requests at scale for high throughput clusters in the wild (§4.3)?

4.1 Matcher: Identify Similar Models

The similarity between two model architectures determines the number of tensor weights that we can transform. Hence, we need to identify the graph-level architectural similarity of each parent and query model pair (Figure 7) and their pairwise tensor mappings. It is tempting to model it as a classic NP-hard graph edit distance (GED) problem [15] by treating tensors as nodes and data flows as edges with the goal to morph one graph to the other with minimum edits (e.g., add/delete a node). However, model matching encounters new challenges: (i) *Prefix Preference*: we prefer to match the prefix over the suffix of model graphs. Because prefix tensors are more transferable since they capture general input features (e.g., image color blobs) [53, 71]. Moreover, model weights are trained systematically over tensors, so any edit on prefixes can result in information loss to subsequent tensors [25]; (ii) *Partial Matching*: we can partially transform the weights of a smaller dimensional tensor to a wider one to match more tensors, or postpone its matching to preserve its exact weights information; and (iii) *Scalability*: as each model can consist of thousands of nodes across branches, capturing the similarity to thousands of zoo models is challenging.

ModelKeeper Matcher measures the graph-level similarity of models, in terms of the total number of transformable weights after mapping tensor pairs from the parent to the query model. It uses the widely-used ONNX tool [7] to extract the computation graph. ONNX supports various model formats (e.g., Tensorflow and PyTorch), which allows us to perform the cross-framework transformation.

Structure-Aware Pairwise Model Matching We introduce a dynamic programming-based heuristic to measure the end-to-end similarity (i.e., number of weights to transform) of two models. It relies on a similarity table $\mathbb{M}_{(|g_p|+1)\times(|g_q|+1)}(i,j)$ to record the best similarity after matching the prefixes of the parent and the query model. Here, $|g_p|$ and $|g_q|$ respectively denote the number of tensors of the parent model and the query model. Then, it enumerates plausible matching operations from previous states (e.g., $\mathbb{M}(i-1,j-1)$), and takes the operation that can acquire the maximum similarity to enter the next state (i.e., $\mathbb{M}(i,j)$).

Figure 7 shows the execution of our structure-aware matching algorithm. It traverses the similarity table in the topological order of graph tensors. This allows us to embed graph-level information while prioritizing the match of prefixes. To advance to the current tensor pair (i, j), it enumerates three plausible operations:

1. *MATCH*: transform weights of i's parent to j's parents.
2. *SKIP_PARENT*: give up transforming tensor i's parent;
3. *SKIP_CHILD*: give up transforming to tensor j's parent;

Then, it updates the table to obtain the maximum similarity after each step based on previous states as follows:

$$\mathbb{M}(i,j) = \max_{k \in parent(i)} \begin{cases} \mathbb{M}(k, j_{parent}) + MATCH(k, j_{parent}) & (1) \\ \mathbb{M}(k,j) + SKIP\_PARENT & (2) \\ \mathbb{M}(i, j_{parent}) + SKIP\_CHILD & (3) \end{cases}$$

To get the overall transformable weights, we can reward each operation based on the number of tensor weights transformed. When tensor i and j belong to the same operator (e.g., convolution), the fraction of transformed weights along each weights dimension in *MATCH* operation (1) is:

$$MATCH(i,j) = \frac{\prod_{dim=1} \min(dim(i), dim(j))}{\prod_{dim=1} \max(dim(i), dim(j))} \quad (\in [0,1]) \quad (4)$$

Otherwise, we assign *MATCH(i, j)* to -1, as this transformation is useless and even loses the weights of that parent model tensor. Similarly, SKIP_PARENT is set to -1 as it loses the parent model tensor, and SKIP_CHILD is 0, since it does not transform the parent model tensor.

Capturing the graph-level similarity is more challenging when tensor j of the query model is the intersection of multiple upstream branches. Because different upstream branches to j may follow the same branch of the parent model during their matching, leading to repetitive (conflicting) matching. As shown in Figure 7, when we reach (❻, ⑥), branch (②→③) and (④→⑤) may both be matched to (❷→❸) that maximizes their own similarity. To avoid conflicting matching, the similarity to j is the sum of upstream branches ($\mathbb{M}(i,j) = \sum_{k \in parent(j)} \mathbb{M}(i,k)$), and we greedily adopt the matching of a branch to tensor j, whose trajectory achieves the largest similarity (i.e., match ②→③ to ❷→❸), to maximize their sum. Meanwhile, we discard the trajectory of other

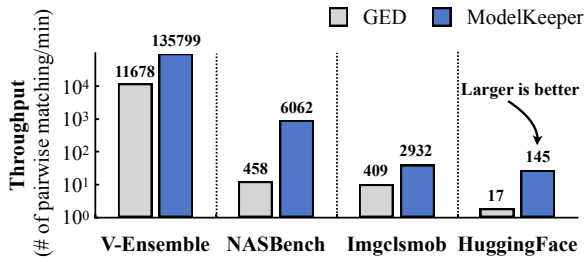


Figure 8: ***Keeper is order-of-magnitude more scalable than existing GED. V-Ensemble is a model zoo for ensemble training (§6.1).***

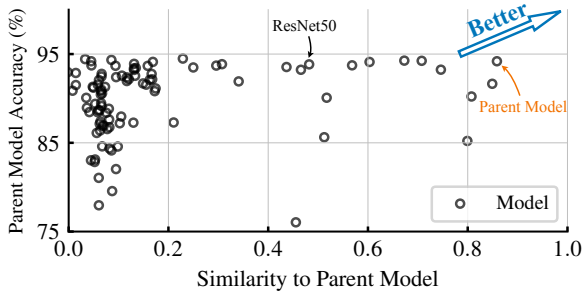


Figure 9: ***Models vary in accuracy and architecture (Imgclsmob zoo). We measure their similarity w.r.t. ResNet101, and prefer to transform a parent model with better similarity and accuracy.***

branches where conflict exists. As such, branch ④→⑤ takes the inferior match (❺), where ④ is skipped.

The last entry of the table, i.e., $\mathbb{M}(|g_p|,|g_q|)$, gives the end-to-end similarity. Note that we can learn the pairwise tensor mappings by backtracking operations taken to reach $\mathbb{M}(|g_p|,|g_q|)$ over the table in linear time. For a specific model, our heuristic will naturally treat the model itself as the most similar model, because matching will always take operation $MATCH(i,i)$ in each step to maximize the similarity.

Figure 8 reports that our pairwise matching can match thousands of model pairs in a second, and achieves higher throughput than the state-of-the-art GED solution [57] in this model matching scenario. More importantly, our empirical results report that our structure-aware matching achieves better training warmup than the GED solution (§6.3).

4.2 Mapper: Transform Maximal Parent Information

The effectiveness of weight transformation is determined by the similarity (transfer more weights) and accuracy (transfer better weights) of parent models. Unfortunately, it is impractical to pick the optimal parent model, since the performance of transformation can only be known after training each derived warmup model to converge. Worse, the variety of model similarity and performance leads to the tussle in selecting the parent model. As shown in Figure 9, while some models (e.g., ResNet34) possess high accuracy, their low similarity to ResNet101 can cap the number of weights that can transform. Next, we introduce Mapper to exploit the sweet spot of both aspects, and then to transform maximal parent model weights in the presence of partial matching.

As shown in Algorithm 1, Mapper relies on Matcher to

Input: Query model q, Model Zoo M
Output: Warmup Model Weights

```
1  NumOfBuckets B = 10                          ▷ Model similarity ∈ [0, 1]
2  Function GetModelSim(Query q, Models M)
       /* Structure-aware matching for model similarity. */
3      topo_query_tensors = SortByTopo(q)
4      model_similarity = {}
5      for model m ∈ M do
6          similarity_table = zeros(|g_m|+1, |g_q|+1)
7          for tensor i ∈ CachedModelTopo(m) do
8              for tensor j ∈ topo_query_tensors do
                   /* Enumerate and merge intersection. */
9                  similarity_table[i][j] = Equation (1-3)
10         model_similarity[m] = similarity_table[|g_m|][|g_q|]
11     return model_similarity
12 Function QueryForModel(Query q, Model Zoo M)
       /* Bucket models in terms of similarities. Pick the model in
          the top-similar bucket with the best performance. */
13     model_similarity = GetModelSim(q, M)
14     top_similar_bucket =
         BucketBySimilarity(model_similarity, B).first
15     for model ∈ top_similar_bucket do
16         if model.perf > best_parent.perf then
17             best_parent = model
       /* Perform width and depth weight transformation */
18     warmup_weights = TransWeight(best_parent, q)
19     return warmup_weights
```

Algorithm 1: Select the parent model to transform.

identify similar models (Line 2). As having a good similarity is the prerequisite for transformation, we need to first ensure picking similar models. To this end, Mapper takes the popular bucketing strategy to allocate models into B buckets in terms of their similarity (Line 14). Taking Figure 9 as an example, with $B = 10$ by default, *bucket* 10 will accommodate models with similarities between 0.9 and 1.0, so models in the same bucket have comparable similarities. Then, Mapper traverses from the last bucket (*bucket 10*) to the first until reaching the first one with nonempty models (*bucket 9*), from which it selects the model with the best performance as the parent model (Line 15). As such, the parent model approaches the boundary of better model similarity and accuracy. Later, Mapper performs structure-aware weight transformation to initialize the query model weights (Line 18).

Information-Preserving Weight Transformation To maximize the end-to-end number of weights to transform, Matcher allows partial matching: it may map a small tensor of the parent to a wider one of the query model, or skip the

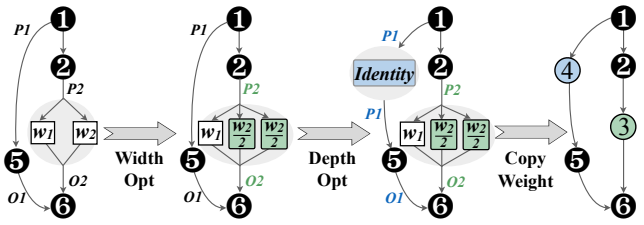


Figure 10: ***Width and depth operator to transform parent model.***

mapping of some tensors in the parent or the query model. Here, the straw-man solution (e.g., in Retiarii [77]), which transfers the weights of parent model tensors if and only if two tensors are identical, can be suboptimal (§6.3), since losing the parent model tensor can make the transfer of its subsequent tensors useless.

To preserve maximal parent model information under partial mappings, Mapper employs a width operator and a depth operator, which extend the well-known ML technique for function-preserving model transformation (e.g., Net2Net [17] and Network Morphism [66]). But unlike existing model transformation techniques [41], which are limited to *expand* the depth and width of a pre-determined model, or complicated transfer learning (e.g., knowledge distillation [34]) that requires additional computation (e.g., pre-training) and/or intrusive implementation, our operators transform the parent model weights into the query model with little overhead.

Our graph-level transformation proceeds in the topological order of tensors. Mapper handles the expanding case similar to today's function-preserving transformation (Figure 10): (i) to transform a parent model tensor to a wider query model tensor, the width operator copies the parent model weights to its mapping tensor of the query model, and pads the rest of the columns via weighted replication from other columns; and (ii) when the mapping requires inserting a new tensor into the parent model (i.e., SKIP_CHILD), the depth operator will initialize the weights of this mapping tensor to be an identity tensor. i.e., this tensor will directly pass the output of its parent tensors to the child tensors, in order to keep the same parent model's output. Readers can refer to Net2Net [17] for more details. We note that both expanding operations, in theory, can preserve the parent model information (i.e., with the same tensor output) for many tensor operators (e.g., the wide-used full connection and convolution layers).

The pruning case, however, cannot preserve full parent model information, because we lose some tensor weights of the parent model in transformation. Our solution is inspired by today's ML model pruning criteria [32]. Specifically, when we need to fit wider tensor weights (i.e., with larger array dimensions) to a smaller dimensional tensor of the query model, the width operator will progressively pick and copy the largest weight values of the parent model tensor to the mapping tensor of the query model. This is because, intuitively and empirically, larger magnitude values often have more impact on the model output [14]. From the depth perspective, when we skip transforming (i.e., SKIP_PARENT) a parent model tensor, the depth operator will add noise to the weight values of that tensor's neighbors. It disturbs the affinity of trained parent model weights so that neighboring tensors can still keep most information while being able to learn new weights [51].

Our transformation can be applied to various models for informed weight initialization. Thereafter, training proceeds as normal, and the warmup model will gradually converge the weight values that fit its architecture the best. As a generic system, ModelKeeper can accommodate other transformation techniques too as they become available. We provide a theoretical analysis of our transformation in Appendix A.2, and empirically show performance improvements using our transformation over its counterparts (§6.3).

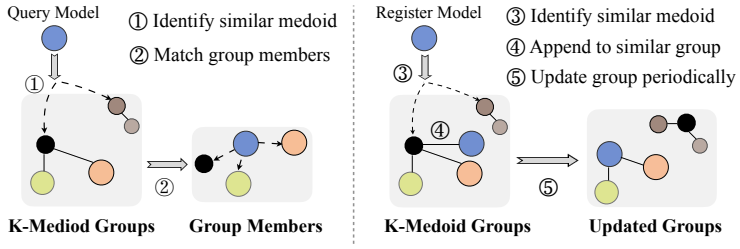
4.3 Zoo Manager: Transform Effectively At Scale

In reality, cluster users register their trained models to the model zoo on the fly, leading to scalability and performance challenges. First, while gathering more models increases the opportunity to transform better parents, the ever-growing number of models (e.g., > 70K models in the HuggingFace model hub of all tasks [2]) and model size (e.g., NLP models can be tens of GBs [16]) can lead to large matching overhead and storage cost. Moreover, models registering to the zoo may have low accuracy (e.g., due to insufficient training), which can harm the effectiveness of weight transformation. As such, ModelKeeper employs a Zoo Manager to support effective transformation at scale under dynamics.

Two-Stage Hierarchical Model Matching Despite being able to match thousands of lightweight CV models every minute (Figure 8), our pairwise matching heuristic can still be insufficient for model zoos with tens of thousands of models or complicated model architectures (e.g., NLP models). For example, to serve a query model using the HuggingFace model zoo for English text generation (2.5K models), performing pairwise matching on these zoo models can take ∼17 minutes, namely, 2.5K models over the throughput (145 matching/minute). This long search time is further exacerbated in today's large cluster with sub-minute job arrivals [37], eventually hurting the user experience.

To ensure an *interactive* service, Zoo Manager adaptively clusters zoo models into a well-defined number of groups, whereby Matcher can perform *two-stage* matching to reduce the number of matching pairs needed to identify more similar models. Intuitively, models with similar architectures would have comparable model similarity to the same query model, so we may be able to cluster zoo models into multiple groups, and then perform pairwise matching on the group members of top similar model groups. However, it is non-trivial to decide what features to use for clustering models, and how many groups are needed. Clustering too few groups does not scale down the problem enough, while too many can lead to a large overhead in identifying which group to prioritize.

We deploy K-medoids clustering [52] to combine pair-



(a) When query model arrives (b) When new zoo model joins

Figure 11: ***Matcher clusters models into groups to reduce the search space, and then performs model matching within groups.***

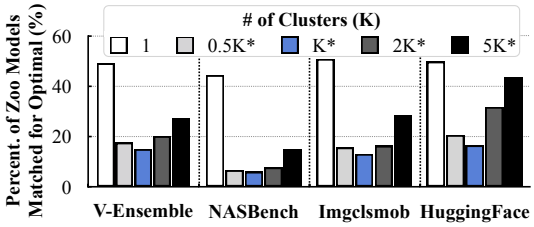
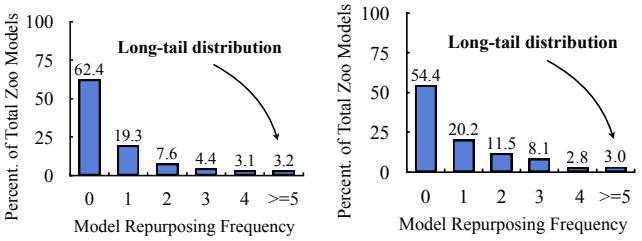


Figure 12: ***ModelKeeper can find the optimal number of clusters K^* in hierarchical matching, and can identify the most similar models with fewer zoo models (e.g., 5% in NASBench) needed to explore.***



(a) Imgclsmob model zoo. (b) NASBench model zoo.

Figure 13: ***A few zoo models are more frequently repurposed as the parent by Keeper. Numbers are from our evaluations (§6.2).***

wise model matching and clustering to find a sweet spot. K-medoids can directly take the distance of two points as input to minimize the distance between data points and their cluster center. Here, models can be taken as different points, and the distance is the reciprocal of their similarity. Compared to other clustering methods (e.g., K-means), K-medoids circumvents the need for embedding complicated model graphs, and it is more compatible with pairwise model matching.

As shown in Figure 11, when a query model arrives, Matcher identifies its similarity to each group medoid, and then conducts pairwise matching on the members of top similar groups. Similarly, when a new model registers, Matcher measures its similarity to group medoids, and assigns this new model to the group whose medoid is the most similar. This enables interactive queries to the latest models. Later, Matcher periodically triggers K-medoids to update the clustering.

To select the most similar models for each query model, Matcher identifies the best group medoid i by performing K pairwise matching, followed by K_i runs to match the members of group i. Assuming a zoo of M models ($M = \sum_i^k K_i$), to minimize the average matching runs on each group (i.e., $\min\big((K+K_i)/K\big)$), we can get the optimal number of groups $K^* = \sqrt{M}$. Figure 12 reports that, compared to the non-clustering design (i.e., $K = 1$), this two-stage design requires matching only 5%-16% of all zoo models to identify the most similar models, thus reducing the query hang time (§6.3).

Capping Zoo Size Hosting all zoo models can consume noticeable storage space. For example, the HuggingFace model zoo takes tens of TBs of storage [2]. In fact and understandably, we notice that a small portion of zoo models are more frequently repurposed than others (Figure 13). This is because certain models contain more similar blocks to other models (e.g., ResNet50 is more likely to be used to warm up other large ResNet models than ResNet18).

To harvest more warmup opportunities subject to the zoo capacity limit, we can formulate it as a knapsack packing problem, where each item (model) is associated with a weight (model size) and a value (repurposing frequency as the parent model), and our goal is to maximize the total value achieved. Namely, warm up as many jobs as possible (aka maximum total repurposing frequency). As such, solving this packing problem enables us to identify which item (model) to keep in the knapsack (model zoo). But on the other hand, models that are popular to train can change over time. For example, users incline to train more recent and/or advanced models. To account for the temporal variation in the repurposing frequency of each zoo model, we take the moving average of model values (e.g., decaying their repurposing frequency by 0.9 every day), and trigger the packing solver upon reaching the storage limit. We show that ModelKeeper can perform well even under severe storage limit (§6.4).

Avoiding Low-Accuracy Models Low accuracy models registering to the zoo (e.g., due to user error) not only wastes storage but can harm the transformation, so we need to ensure the zoo uses models with decent accuracy. To this end, other than selecting the model with better accuracy as the parent using the bucketing design at the query time, Zoo Manager evicts zoo models with outlier accuracy at runtime. By default, we take the popular Z-score criteria (i.e., model accuracy below the mean by more than two standard deviations) to identify outliers [58]. Moreover, for the same model architecture, it only keeps the model with the best accuracy. We show that ModelKeeper can accelerate training even in the presence of low accuracy models in unfavorable environments (§6.4).

Complexity Analysis The complexity of pairwise model matching is $O(|g_p| \times |g_q|)$,[3] and that of model clustering is $O(M^{2.5})$ for the zoo with M models. Mapper takes linear time to select and transform the parent model. The magnitude of these factors is mostly within O(1K) (§6.1). Our evaluations

[3]We omit the complexity in enumerating tensor parents (i.e., $k \in parent(i)$), since the node degree is orders of magnitude smaller than $|g_p|$.

show that ModelKeeper incurs negligible overhead (§6.3).

5 Implementation

We have implemented a system prototype of ModelKeeper, with around 2K lines of Python code as the frontend library and 1K lines of C++ code as the backend. Our implementation provides user-friendly APIs and supports many popular ML frameworks, such as Microsoft NNI [4], AutoKeras [41], Ray [49], and MLflow [5], with few-lines-of-code plugins.

ModelKeeper Components ModelKeeper coordinator supports distributed deployment across machines. Each coordinator controller processes a single scheduling thread to poll client requests from its queue, and reserves a thread pool for Matcher. Matcher performs pairwise model matching in parallel for each query model, and then Mapper creates a worker thread to transform parent model weights using *numpy* format. Zoo Manager updates the model clustering every 5 minutes, and uses *ortools* library to solve the knapsack problem. The client agent communicates with the coordinator via TCP connections.

Fault Tolerance ModelKeeper uses *Redis* in the coordinator to store the metadata and model weights in a fault-tolerant manner, and this metadata is cached in the memory with small footprints. Changes to the model zoo (e.g., registering new models) follow the write-ahead transaction to the storage. At runtime, the coordinator runs a daemon process to monitor the liveness of the service, which will create a new service process if the existing one crashes. The new process then fetches the latest checkpoint from *Redis* to catch up.

Interfaces We pack interfaces into a Python library. The cluster manager can initiate the coordinator in three lines:

```
from modelkeeper import ModelKeeperCoordinator
keeper_service = ModelKeeperCoordinator(config)
keeper_service.start()
```

Users can initiate the client agent in a few lines (Figure 6).

6 Evaluation

We evaluate the effectiveness of ModelKeeper on three mainstream frameworks for exploratory and general DNN training, using five large-scale CV and NLP model zoos across thousands of models. We summarize the results as follows:

- ModelKeeper saves 23%-77% total amount of training execution needed (i.e., 1.3×-4.3× faster training) than the state-of-the-art without accuracy drop of models (§6.2).
- ModelKeeper outperforms its counterparts by exploiting the parent model with high similarity and better accuracy using different design components (§6.3).
- ModelKeeper improves performance over a wide range of parameters and practical cluster setups in the wild (§6.4).

6.1 Methodology

Cluster setup. We evaluate ModelKeeper on an 80-node cluster (40 GPU nodes and 40 CPU nodes). Each GPU node has a Tesla P100 GPU with 16 GB GPU memory and 16-core CPUs. Since most HuggingFace NLP models exceed our GPU memory capacity, we resort to CPU nodes. Each node has 32-core CPUs and 384 GB of memory. ModelKeeper coordinator runs on a 32-CPU server with 10 Gbps bandwidth.

Workloads. We evaluate ModelKeeper using five widely-used CV/NLP model zoos and realistic workloads (Table 1):

- *NASBench* [24]: an image classification model zoo with thousands of lightweight models for NAS task.
- *AutoKeras Zoo* [41]: a CNN model zoo generated by AutoKeras during the bayesian NAS searching.
- *Imgclsmob* [10]: a popular zoo of state-of-the-art CV models (e.g., DenseNet [36]). Most models are heavyweight.
- *V-Ensemble* [65]: a benchmarking workload for ensemble training, which has hundreds of variants of VGG models.
- *HuggingFace* [2]: a collection of advanced HuggingFace NLP models (e.g., Bert [23]) for next word prediction.

We train Imgclsmob-Small models on CIFAR dataset and ImageNet32 dataset for 32×32 small image inputs, Imgclsmob models on Flowers102 dataset for 224×224 large images, and HuggingFace models on the large WikiText dataset. ImageNet32 is a downsampled 120-category ImageNet dataset (e.g., smaller input size) for efficient computation.

To emulate practical cluster setups, NAS models are generated by the searching algorithm on the fly, and training jobs are submitted following the arrival of Microsoft Trace [37]. The same workload does not contain identical model architectures. ModelKeeper model zoo starts empty for each workload, and jobs contribute (upload) their trained models to the zoo as they complete over time.

Parameters. We follow the default setting specified in each model zoo: (1) *CV models*: the SGD optimizer with minibatch size 64 and initial learning rate 0.01; and (2) *NLP models*: the AdamW optimizer with minibatch size 32 and initial learning rate 8e-5. We use the ReduceLROnPlateau scheduler to decay the learning rate by 0.5 once the training loss stagnates.

Baselines. We compare ModelKeeper to the following:

- *Retiarii [77]*: Microsoft's training framework that relies on the lineage of graph mutation to warm up NAS models.
- *AutoKeras [41]*: An advanced AutoML system based on Keras that applies lineage-based warmup for NAS models.
- *MotherNet [65]*: An ad-hoc ensemble training algorithm that trains a model subnet, which introduces intrusive overhead and implementation, to warm start models.

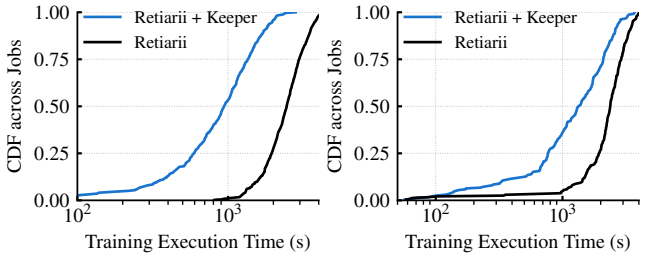
Existing efforts limit to individual NAS/ensemble jobs, while ModelKeeper can support various tasks across jobs and users.

Metrics. We care about the *training execution time* needed to train to converge and the *model convergence accuracy*.

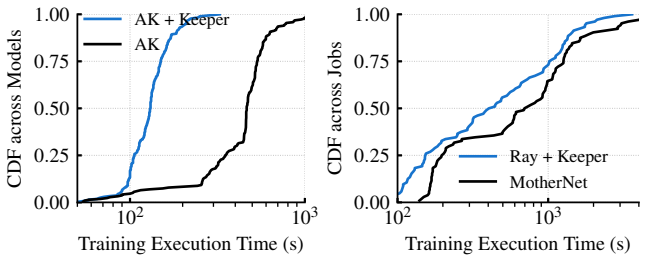
We run with five realistic Microsoft Traces [56], and report the average over 5 runs.

Category	Task	Workload	# of Models	Dataset	Avg. Time Improvement	Avg. Acc. Difference
Exploratory Training	Grid Search NAS	NASBench [24]	1,000	CIFAR-100	2.9×	0.39%
	Evolution NAS				2.4×	0.38%
	AK-Bayesian NAS [41]	AutoKeras Zoo [41]	500		4.3×	0.31%
General Training	Image Classification	Imgclsmob [10]	389	Flowers102 [54]	2.8×	0.23%
		Imgclsmob-Small	179	CIFAR-10	2.1×	0.02%
				CIFAR-100	1.6×	0.18%
				ImageNet32 [19]	1.3×	0.03%
	Ensemble Training	V-Ensemble [65]	104	CIFAR-100	1.7×	0.08%
	Language Modeling	HuggingFace [2]	69	WikiText-103 [47]	1.8×	-0.13 perplexity

Table 1: ***Summary of improvements. ModelKeeper improves training execution time without accuracy drop, by reducing the amount of training needed (i.e., GPU Saving). Accuracy difference is defined by Acc.(Keeper) - Acc.(Baseline), and smaller perplexity is better.***

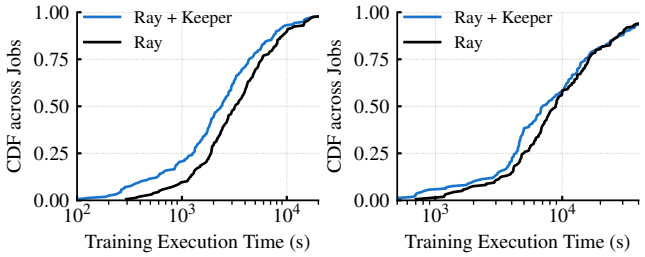


(a) Grid Search NASBench (b) Evolution NASBench

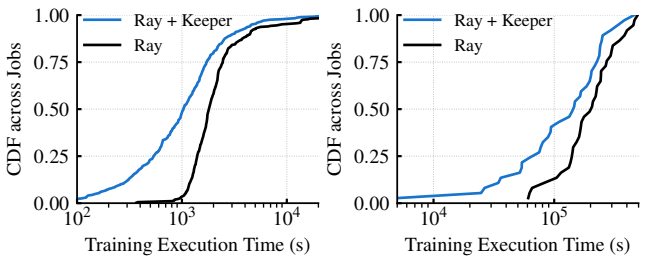


(c) Bayesian Optim. (AutoKeras) (d) Ensemble training

Figure 14: ***ModelKeeper outperforms existing warmup training.***



(a) CIFAR-100 (Imgclsmob-Small) (b) ImageNet (Imgclsmob-Small)



(c) Flowers (Imgclsmob) (d) NLP (HuggingFace)

Figure 15: ***ModelKeeper improves general training tasks.***

6.2 End-to-End Performance

In this section, we evaluate how ModelKeeper (Keeper) is complementary to and benefits today's ML frameworks. Here, we run the NAS task using Microsoft NNI (with Retiarii backend [77]) and AutoKeras, and other training tasks on Ray [49]. Table 1 summarizes the average improvement on each training workload after applying ModelKeeper.

ModelKeeper outperforms existing warmup solutions. ModelKeeper outperforms existing training warmup solutions in Retiarii, AutoKeras, and MotherNet by 1.7×-4.3× (Figure 14), by saving 43.1%-76.7% total amount of training needed. Their inefficiency is due to two primary reasons:

(i) Suboptimal parent model selection: Retiarii and AutoKeras track the lineage of graph mutation and treat the base model in evolution as the parent model. However, as multiple layers can be modified on the base model in searching new models, such rigid parent selection can miss better parent models out of other explored NAS models. Similarly, MotherNet not only requires additional training of the model subnet, but can not repurpose better-trained models on the fly.

(ii) Insufficient weight transformation: Their design, which simply copies the weights from the parent model when two tensors are identical, is lossier than ModelKeeper. For example, inserting randomly initialized prefix tensors can make the copy of subsequent tensors useless.

Meanwhile, they are limited to specific NAS or ensemble training tasks and cannot serve various DNN training jobs on the fly in the cluster wide.

ModelKeeper accelerates ML training for various tasks. Figure 15 and Table 2 report the performance of individual jobs. Compared to training from scratch, we observe that: (i) ModelKeeper achieves 1.3×-4.3× faster training, saving 23%-77% training execution, across a wide range of work-

Workload	Time Improvement			Acc.(Keeper) - Acc.(Baseline)		
	25th	50th	75th	25th	50th	75th
NAS-Grid	1.5×	2.0×	3.1×	0.01%	0.25%	0.42%
NAS-Evol	1.2×	1.6×	3.0×	0.03%	0.19%	0.48%
Flowers102	1.2×	2.1×	3.3×	0.0%	0.16%	0.37%
CIFAR-100	1.1×	1.5×	2.0×	-0.04%	0.08%	0.32%
ImageNet32	1.0×	1.2×	1.6×	-0.07%	0.0%	0.11%
V-Ensemble	1.1×	1.5×	1.9×	0.02%	0.07%	0.65%
HuggingFace	1.2×	1.4×	2.1×	0.2 ppl	-1.3 ppl	-3.87 ppl

Table 2: ***Keeper saves training execution time of individual jobs without accuracy drop. Smaller perplexity (ppl) is better.***

loads. This improvement is more pronounced in a larger zoo because of having more trained models to repurpose. (ii) Improvements on different workloads report a positive correlation with the prevalence of model similarity in that model zoo. Here, ModelKeeper achieves larger improvement on NAS-Bench, which is consistent with the fact that this model zoo owns higher inter-model similarities (Figure 2). (iii) Although ModelKeeper starts from an empty zoo and jobs arrive on the fly, we can still save the training execution for 70%-95% individual jobs (Table 2). We note that the 25th percentile improvement of small-scale model zoos (e.g., Imgclsmob) is inferior to others. This is because not all training models are warmed up due to the cold start of this online setting, and the fact that ModelKeeper will not warm start the model that does not have a similar parent (similarity > 0).

ModelKeeper speeds up training without accuracy drop. Table 1 and Table 2 report that, on average, ModelKeeper can achieve similar (or even slightly better) final model accuracy. Intuitively, ModelKeeper should perform no worse than baseline accuracy, since the rest of the training (e.g., data) remains the same. However, we note that this slightly better model performance is consistent with the observations in ML network morphism [66], which interprets it as the internal regularization ability. Specifically, by transferring from well-trained models, model weights have been placed in a good position in the space, resulting in a more regularized network to reach a better basin of the loss curvature [25, 66]. In contrast, training from scratch can get stuck in local minima.

6.3 Performance Breakdown

In the rest of the evaluations, we refer to the improvement on V-Ensemble as that over training from scratch for brevity.

Breakdown of Components We break down ModelKeeper by disabling Matcher and Mapper respectively: (1) *Keeper w/o Matcher*: remove our Matcher design, and instead resort to a state-of-the-art graph matching strategy [57] to select a parent model with the most pairwise tensor mappings; and (2) *Keeper w/o Mapper*: disable our Mapper design, so only transform the parent model weight if and only if two tensors are identical. Figure 16 reports the improvement of these

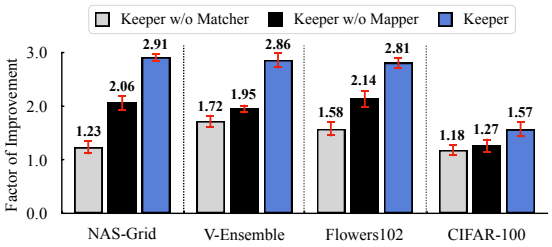


Figure 16: ***Breakdown of Keeper components.***

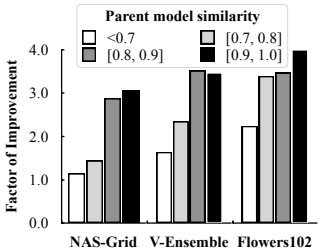


Figure 17: ***Faster training with higher model similarity.***

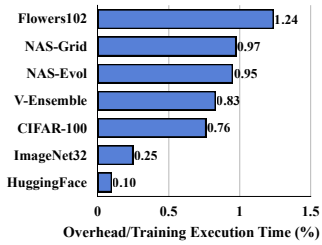


Figure 18: ***Keeper introduces negligible overhead.***

variants. We notice: (i) the classic GED solution, in *Keeper w/o Matcher*, achieves suboptimal performance, since model matching prefers to match prefixes, and partial matching allows better overall similarity. (ii) transforming weights only for identical tensors, in *Keeper w/o Mapper*, is inferior to Keeper information-preserving transformation. (iii) Matcher and Mapper contribute comparable improvements.

Breakdown of Improvement Characteristics Figure 17 reports the average improvement after categorizing training models by their similarity to the parent model. We note that: (1) Keeper tends to achieve better execution saving for models with a higher parent model similarity. This again supports our parent model selection criteria that prioritize models with higher architectural similarity. (2) Improvements of different similarity regimes (e.g., [0.7, 0.8] vs. [0.8, 0.9]) are often distinct, and this becomes vague as similarity over 0.8. Because most layers have been largely warmed up, and deeper layers are too specific to the parent model to be transferable [71].

Overhead Analysis Figure 18 reports Keeper's overhead, i.e., the time taken between initiating the query and starting to train, over the training execution time. We report the average of all jobs, and notice that Keeper introduces less than 1.5% overhead (< 43 s) across all workloads.

6.4 Sensitivity and Ablation Studies

Impact of Low-Accuracy Models As a cluster-wide service, ModelKeeper should be robust to unfavorable settings where the accuracy of user-registered zoo models can be low (e.g., due to insufficient training). We follow the popular early-stop design in ML [44] to simulate unfavorable setups, where model registration takes place when jobs run to at most X minutes. Figure 19 reports the improvement of execution time across different degrees of unfavorable settings. Here, the x-axis value 40% indicates X is set to be the 60th percentile

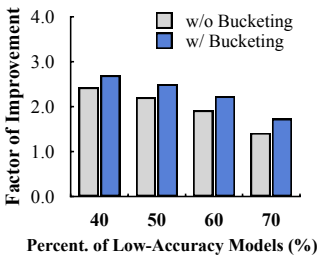


Figure 19: ***Keeper is robust in the presence of poor performance models (NAS-Grid).***

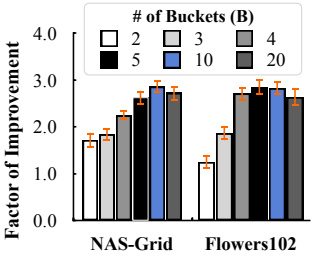


Figure 20: ***Keeper improves training execution time across the different numbers of buckets.***

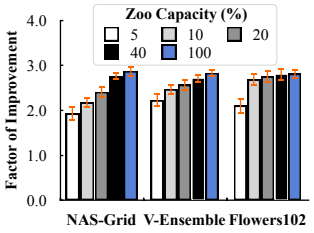


Figure 21: ***Impact of zoo capacity on execution time. Error bars report standard deviation.***

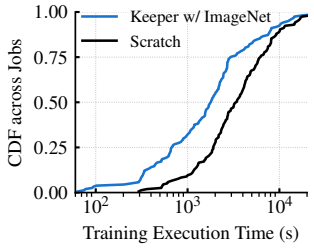


Figure 22: ***Keeper accelerates model training on CIFAR-100 using ImageNet32 model zoo.***

value in execution time distributions, so only 40% zoo models are trained to converge. We observe that: (i) improvement decreases as more zoo models have low accuracy. (ii) Keeper is more robust with our bucketing design as it exploits the similarity-accuracy sweet spot.

Impact of Bucketing Figure 20 reports that ModelKeeper delivers consistent improvement across a wide range of number of buckets B. Meanwhile, we notice that using the most accurate parent model (i.e., $\sim B$=2) or the most architecturally similar parent (i.e., $\sim B$=20) achieves suboptimal improvement, since it respectively undervalues the model similarity and accuracy in selecting the parent model.

Impact of Zoo Capacity Figure 21 reports the average improvement under different zoo capacities. The total size (i.e., 100%) of model zoos in NAS-Grid, V-Ensemble, and Flowers102 are 1.6GB, 17GB, and 31 GB, respectively. We observe that: (i) as expected, the improvement is more pronounced as we allocate more storage to ModelKeeper's model zoo, but (ii) we can still achieve $\sim$2$\times$ improvement under severe capacity limits (e.g., 5% capacity aka < 2GB storage), since Keeper adaptively evicts suboptimal zoo models.

Cross-Dataset Training Warmup Figure 22 reports that ModelKeeper can benefit DNN training across datasets. Here, we warm start the training of Imgclsmob-small models on CIFAR-100 using zoo models from the ImageNet32 workload, and notice 2.5$\times$ faster training on average. This is because front DNN layers capture general input features (e.g., color blobs of images), which are transferable to similar datasets [71]. While picking which dataset as the source for warmup is still an open ML problem [45, 78], ModelKeeper provides systems support for automated warmup transformation across ML tasks and datasets using the given model zoo.

7 Discussion and Future Work

Support for Hyperparameter Tuning ModelKeeper by default automatically searches and transforms the parent model for various training tasks. Meanwhile, the developer can specify which parent model to repurpose using the tag configuration in their request too (Figure 6), while enjoying the automated weights transformation. For example, we may want the same parent model for hyperparameter tuning jobs to eliminate the comparison bias and/or to ensure reproducibility. Moreover, as the training of the query model will be jumpstarted, it would be interesting to investigate how to adapt to better job configurations (e.g., scaling the learning rate in terms of the number of transformed layers [56, 66]) to further improve the training convergence.

Model Sharing in the Wild ModelKeeper repurposes a zoo of trained models to warm start the new training job. These zoo models can be maintained by the cluster provider, and/or contributed by users. For example, AWS SageMaker offers hundreds of pre-trained models for tasks like object detection and natural language processing [8]; HuggingFace Model Hub has gathered $\sim$70K models shared by the community [2]. The former is more managed but expensive to include extensive models and tasks, while the latter has good extensibility but can exhibit great uncertainties (e.g., low-accuracy models). To the best of our knowledge, ModelKeeper moves the first step to *automatically* warm start the cluster-wide model training. However, further investigations on how to democratize it in the wild, such as for privacy and security concerns, are needed. To this end, one possible approach is to develop differential privacy-like solutions [11] (e.g., adding noise to the weights of the contributed models), which naturally leads to an interesting trade-off between privacy and the model quality.

8 Related Work

Deep Learning Frameworks Recent ML efforts have made considerable progress toward efficient inter-job scheduling [28, 49, 56, 74], intra-job computation placement [43, 50], communication optimization [40, 55], specialized execution backend [9, 18, 42], and timely inference [29]. However, they are mostly in-execution optimizations, and/or the total amount of training remains the same. Different from transforming tensors for faster computation (e.g., TASO [38] and PET [64]), ModelKeeper operates on model weights, and acts as a complementary service to accelerate cluster-wide DNN training.

AutoML Systems Retiarii [77] and AutoKeras [41] rely on the lineage of graph mutation to repurpose trained models, whereas they are limited to NAS tasks within individual jobs. Experiment Graph [22] identifies the reusable ML scripts and artifacts in platforms to speed up repeated executions,

so it focuses on the same job execution. As recent AutoML platforms, such as AzureML [59], Amazon SageMaker [1] and MLflow [75], provide a collaborative environment to simplify ML deployments, reusing artifacts can greatly speed up repeated executions (e.g., reuse scripts [22]). ModelKeeper is the first automated training warmup system to accelerate cluster-wide DNN training jobs across users, and improves Retiarii and AutoKeras further (§6.2).

Transfer Learning Transfer learning today mostly transfers the weight of the same model [71], from one source task to another target to alleviate the need for large training data. For a given parent model, network morphism [17, 66] introduces function-preserving transformation to construct child models while preserving the parent information. MotherNet [65] further applies the network morphism to warm start model training, but is limited to ensemble training tasks. ModelKeeper tackles a more challenging scenario for various tasks in the wild, and achieves better performance (§6.2).

Graph Matching Graph matching is one of the NP-hard fundamental problems in graph analysis [61]. To speed up the matching, DAF [31] decomposes the graph into forests. Similarly, AED [57] divides global matching into local matching, and then aggregates the local matching decisions. However, they are insufficient due to the novel properties of DNN graphs, where pairwise matching prefers ordered alignment and allows partial weights transformation. ModelKeeper outperforms them in training speedup and throughput (§6.3).

9 Conclusion

In this paper, we introduce ModelKeeper to enable automated warmup of DNN training jobs at the cluster scale. ModelKeeper manages a collection of already-trained models from different developers and/or frameworks. Before training a model, it selects a high-quality trained parent model and performs structure-aware transformation of parent model weights to warm up the weights of new training models. Our evaluations across thousands of CV/NLP models show that ModelKeeper achieves 1.3×-4.3× faster training completion.

Acknowledgments

Special thanks go to the entire CloudLab team for making ModelKeeper experiments possible. We would also like to thank the anonymous reviewers, our shepherd, Neeraja J. Yadwadkar, and SymbioticLab members for their insightful feedback. This work was supported in part by NSF grants CNS-1909067, CNS-1900665, and CNS-2106184.

References

[1] Amazon SageMaker. https://aws.amazon.com/sagemaker/.

[2] HuggingFace Model Hub. https://huggingface.co/models?sort=downloads.

[3] Kaggle Competition. https://www.kaggle.com/docs/competitions.

[4] Microsoft NNI. https://github.com/microsoft/nni.

[5] MLflow. https://mlflow.org/.

[6] Model Zoo: Discover open source deep learning code and pretrained models. https://modelzoo.co/.

[7] Open Neural Network Exchange (ONNX). https://github.com/onnx/onnx.

[8] Pre-trained machine learning models available in AWS Marketplace. https://aws.amazon.com/marketplace/solutions/machine-learning/pre-trained-models/.

[9] PyTorch. https://pytorch.org/.

[10] Sandbox for training deep learning networks. https://github.com/osmr/imgclsmob.

[11] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, 2016.

[12] Zeyuan Allen-Zhu and Elad Hazan. Variance reduction for faster non-convex optimization. In *ICML*, 2016.

[13] Jordan Ash and Ryan P Adams. On warm-starting neural network training. *NeurIPS*, 33, 2020.

[14] Brian R. Bartoldson, Ari S. Morcos, Adrian Barbu, and Gordon Erlebacher. The generalization-stability tradeoff in neural network pruning. In *NeurIPS*, 2020.

[15] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, 2016.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. abs/2005.14165, 2020.

[17] Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *ICLR*, 2016.

[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[19] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the CIFAR datasets. *CoRR*, abs/1707.08819, 2017.

[20] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC*, 2020.

[21] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *NSDI*, 2017.

[22] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. Optimizing machine learning workloads in collaborative environments. In *SIGMOD*, 2020.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

[24] Xuanyi Dong and Yi Yang. NAS-Bench-201: Extending the scope of reproducible neural architecture search. In *ICLR*, 2020.

[25] Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, and Pascal Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. In *AISTAAS*, 2009.

[26] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander J. Smola. Autogluon-tabular: Robust and accurate automl for structured data. *CoRR*, abs/2003.06505, 2020.

[27] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13, 113–129 (2010), 2010.

[28] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *NSDI*, 2019.

[29] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *OSDI*, 2020.

[30] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: A multidimensional optimization for model serving in cloud. In *NSDI*, 2022.

[31] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD*, 2019.

[32] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. NIPS'15, 2015.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[34] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[35] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[36] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[37] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *ATC*, 2019.

[38] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, 2019.

[39] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant video processing. In *ATC*, 2018.

[40] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *OSDI*, 2020.

[41] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *SIGKDD*, 2019.

[42] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Sol: Fast distributed computation over slow networks. In *NSDI*, 2020.

[43] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *OSDI*, 2021.

[44] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *JMLR*, 2018.

[45] Mingsheng Long, Han Zhu, Jianmin Wang, and Michael I. Jordan. Deep transfer learning with joint adaptation networks. In *ICML*, 2017.

[46] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *NSDI*, 2020.

[47] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *CoRR*, abs/1609.07843, 2016.

[48] Richard Meyes, Melanie Lu, Constantin Waubert de Puiseau, and Tobias Meisen. Ablation studies in artificial neural networks. *CoRR*, abs/1901.08644, 2019.

[49] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *OSDI*, 2018.

[50] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *SOSP*, 2019.

[51] Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Structured bayesian pruning via log-normal multiplicative noise. In *NeurIPS*, 2017.

[52] James Newling and François Fleuret. K-medoids for k-means seeding. In *NeurIPS*, 2017.

[53] Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. What is being transferred in transfer learning? *NeurIPS*, 2020.

[54] M-E. Nilsback and A. Zisserman. Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, 2008.

[55] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *SOSP*, 2019.

[56] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *OSDI*, 2021.

[57] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950–959, 2009. 7th IAPR-TC15 Workshop on Graph-based Representations (GbR 2007).

[58] Kevin Roth, Yannic Kilcher, and Thomas Hofmann. The odds are odd: A statistical test for detecting adversarial examples. In *ICML*, 2019.

[59] AzureML Team. Azureml: Anatomy of a machine learning service. In *PAPIs*, 2015.

[60] Nilesh Tripuraneni, Michael I. Jordan, and Chi Jin. On the theory of transfer learning: The importance of task diversity. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *NeurIPS*, 2020.

[61] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[62] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Modeldb: A system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, New York, NY, USA, 2016. Association for Computing Machinery.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[64] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *OSDI*, 2021.

[65] Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. Mothernets: Rapid deep ensemble learning. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *MLSys*, 2020.

[66] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *ICML*, 2016.

[67] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *NSDI*, 2022.

[68] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, 2018.

[69] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *OSDI*, 2020.

[70] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. In *MLSys*, 2021.

[71] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *ArXiv*, abs/1411.1792, 2014.

[72] Hao Yu, Sen Yang, and Shenghuo Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *AAAI*, 2019.

[73] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. In *MLSys*, 2020.

[74] Peifeng Yu, Jiachen Liu, and Mosharaf Chowdhury. Fluid: Resource-aware hyperparameter tuning engine. In *MLSys*, 2021.

[75] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[76] Ke Zhang, Miao Sun, Tony X. Han, Xingfang Yuan, Liru Guo, and Tao Liu. Residual networks of residual networks: Multilevel residual networks. *IEEE Transactions on Circuits and Systems for Video Technology*, 28(6):1303–1314, Jun 2018.

[77] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarii: A deep learning exploratory-training framework. In *OSDI*, 2020.

[78] Han Zhao, Remi Tachet des Combes, Kun Zhang, and Geoffrey J. Gordon. On learning invariant representation for domain adaptation. ICML, 2019.

A ModelKeeper Analysis

A.1 Design Criteria

At its core, ModelKeeper performs informed weight initialization for DNN models by repurposing a well-trained model's weights. Intuitively, we note that

- This can be viewed as an instance of existing transfer learning (TL), where we transform the weight of a model on one dataset to train on "another" dataset (i.e., the warmup model has not viewed that dataset before its training takes place) [13]. More subtly, it is a simplified and complementary TL scenario under homogeneous data distribution and features, so existing TL theories can be applied to validate our effectiveness too.
- ModelKeeper transformation is an informed weight initialization, thus a special case of random initialization. As the rest of the training remains the same, the model should be able to reach similar final accuracy when the model converges.

Why ModelKeeper Can Help Convergence? We next present the theoretical analysis of model convergence to show why ModelKeeper can achieve faster convergence.

Corollary A.1. *(Theorem 1 in [72]). Under widely-used DL assumptions (1) Smoothness: loss function $f(\boldsymbol{w})$ is L-Lipschitz smooth; (2) Bounded gradient variances: with constants $G > 0$, $\sigma > 0$, we assume $\mathbb{E}[\|\nabla f(\boldsymbol{w})\|^2] \leq G^2$ and $\mathbb{E}[\|\nabla F(\boldsymbol{w})\|^2 - \nabla f(\boldsymbol{w})\|^2] \leq \sigma^2$; and (3) Unbiased estimation: on mini-batch ξ, we have $\mathbb{E}_{\xi|\boldsymbol{w}}[\nabla f(\boldsymbol{w})] = \nabla F(\boldsymbol{w})$.*

With learning rate γ to $0 < \gamma \leq \frac{1}{L}$, then for iteration T, the model training convergence rate is:

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla f(\boldsymbol{w}^{t-1})\|^2] \leq \frac{2}{\gamma T}(f(\boldsymbol{w}^0) - f^*) + 4\gamma^2 I^2 G^2 L^2 + \frac{L}{N}\gamma\sigma^2$$

Where N is the number of workers in synchronized data-parallel training, f^* is the optimal training loss.

From Corollary A.1, we can notice that, for the same model, training achieves faster convergence with a smaller initial loss value $f(\mathbf{w}^0)$ in theory (similar to [53, 71]). Indeed, existing gradient variance reduction techniques in the ML community report a similar theory analysis [12]. Here, we empirically show that the initial training loss of the warmup query model, $f(\mathbf{w}^0)$, will start from some basin of loss curvature (e.g., better accuracy in Figure 3 and smaller gradient variance in Figure 4), and theoretically analyze why this enables starting from the loss basin in Appendix A.2.

Admittedly, weight transformation can be lossy (e.g., due to incomplete matching), which breaks the parent model information. We note that capturing the exact convergence comparison herein is extremely challenging, which indeed is a fundamental open problem even in today's transfer learning [25]. Nevertheless, many empirical analyses have reported consistently encouraging improvement [71], and transfer learning is widely-used. Intuitively, for front tensors that enjoy full information-preserving transformation, we can consider them as a prefix subnet, and this subnet holds the same output as the corresponding parent subnet. Therefore, these tensors can still potentially achieve faster convergence according to Corollary A.1.

How to Select Parent Models? In selecting the parent model, ModelKeeper prioritizes the model with (1) *better model accuracy*: this is because parent models with better accuracy enable smaller initial loss $f(\mathbf{w}^0)$, thus allowing better convergence speed (Corollary A.1); and (2) *larger architectural similarity* and *prefix preference*: If we dive to the fundamental of model training, the output activations of a specific model tensor i is $\mathbf{y}^i = f_i(\mathbf{y}^{(i-1)^T}\mathbf{w}_i + \mathbf{b}_i)$. Here, assuming the front $l-1$ tensor are warmup, while $\mathbf{w}_i$ is randomly initialized. The front subnet still enjoys better convergence, so we prefer a model with architectural similarity to maximize this potential. In the forward training propagation, $\mathbf{w}_i$ leads to cascading information loss to subsequent tensors, so we prioritize the match of prefixes to minimize this loss. On the other hand, training front tensors is more difficult but more transferable, because gradient information becomes less informative as it is backpropagated through more subsequent tensors [25], which requires us to match subsequent tensors as many as possible to curb this divergence to the front tensors in backward propagation.

A.2 Information-Preserving Transformation

ModelKeeper employs width and depth operators to perform structure-aware weight transformation, wherein expanding the parent model performs the same to Net2Net [17]. Net2Net theoretically grounds that expanding transformation (e.g., more convolution channels or new convolution tensors) can preserve the parent model information for a wide range of tensors. Specifically, the depth operator tries to deepen a tensor $\mathbf{y}^i = f_i(\mathbf{y}^{(i-1)^T}\mathbf{w}_i + \mathbf{b}_i)$ using two tensors $\mathbf{y}^i = f_i\big(\mathbf{U}^{(i)^T} f_i\big(\mathbf{y}^{(i-1)^T}\mathbf{w}_i + \mathbf{b}_i\big)\big)$, where f_i, $\mathbf{w}_i$, $\mathbf{b}_i$ are the activation function, tensor weights, and bias vectors, respectively. When matrix $\mathbf{U}$ is initialized to an identity matrix, adding $\mathbf{U}$ preserves the same output of its input tensor if f_i is chosen such that $f_i(\mathbf{U}f_i(\mathbf{v})) = f_i(\mathbf{v})$ for all vectors $\mathbf{v}$. This property, f_i, holds for widely-used rectified linear activation in today's DNN models. For example, to insert a new convolution tensor, we should set the convolution kernels to be identity filters. Readers can refer to Net2Net [17] for the theoretical analysis for the width operator. As such, in expanding the parent model, we may preserve the full parent model information.