Nautilus: An Optimized System for Deep Transfer Learning over Evolving Training Datasets

ABSTRACT

Deep learning (DL) has revolutionized unstructured data analytics. But in most cases, DL needs massive labeled datasets and large compute clusters, which hinders its adoption. These limitations can be overcome using a popular paradigm called *deep transfer learning* (DTL). With DTL, one adapts a pre-trained DL model instead of training a model from scratch. Thus, DTL reduces the massive training data and compute requirements to train a model. During adaptation, a common practice is to *freeze* most pre-trained model parts and adapt only the remaining. Since no single adaptation scheme is universally the best, one often evaluates several schemes, which is also called *model selection*.

We also observed that data labeling for DTL is seldom a one-off process. One often updates their labeled data intermittently by adding new labeled data and performs model selection to evaluate the accuracy of the trained models. Today, one executes this workload by performing computations for the entire pre-trained model and repeats it for every model selection cycle. This approach results in redundant computations in frozen model parts and causes usability and system inefficiency issues.

In this work, we cast the DTL model selection in the presence of frozen layers as an instance of the *multi-query optimization* and propose two optimizations that reduce redundant computations and training overheads. We implement our optimizations into a data system called Nautilus. Experiments with end-to-end workloads on benchmark datasets show that Nautilus reduces DTL model selection runtimes by up to 5X compared to the current practice.

1 INTRODUCTION

Deep Learning (DL) achieves near-human accuracy for many image and language analytic tasks. Its success is mainly driven by how it extracts a hierarchy of relevant parameterized features from raw data, with the parameters learned automatically during training [21]. These parameters are grouped into *layers*, and each layer captures a different level of abstraction about the data, from generic ones to specific ones [32, 40, 60, 75]. But, DL has a major bottleneck: model training is expensive. In many cases, it needs large training datasets (e.g., millions of records) and incurs high compute costs (e.g., days to weeks of GPU time). This hinders DL adoption, especially in low-resource settings. These bottlenecks can be overcome using a popular paradigm called *deep transfer learning* (DTL).

Example Use Case: Consider a data scientist tasked to develop a named entity recognition model to identify disease entities from clinical text. She is provided a large unlabeled clinical text dataset. For this task, she decides to adopt the DTL paradigm. She downloads a pre-trained model (e.g., BERT [17]) from a model hub [1], removes the last few layers in the model, and adds few new layers on top of it. She also *freezes* most of the pre-trained layers and trains only the new layers and the final few layers of the pre-trained model. She explores several freezing schemes and training hyperparameter values (e.g., learning rates) to find the model with the best accuracy.

DTL leverages the fact that most of the features learned by a DL model when trained on a large dataset like Wikipedia are general enough to be reused in other similar settings. While both pre-trained and newly added layer parameters can be trained, one can freeze the parameters from most pre-trained layers as they are generic enough to be directly reused [17, 36, 51, 62]. This approach also reduces the chances of model overfitting [43, 62] and the compute costs as computations needed to update the frozen layers can be avoided. However, different freezing schemes and training hyperparameters can lead to different model accuracies. Hence, model selection is unavoidable for DTL. Overall, DTL significantly reduces the labeled data requirements (e.g., from millions to few thousand).

Even though DTL significantly reduces the large training data requirement, it does not eliminate it. Often, practitioners create training datasets by manually labeling the data. We also observed that in many domain applications, data labeling is seldom a one-off process [6, 9, 45, 66]. Practitioners often update the training dataset intermittently by labeling new data and evaluate the model accuracy to ensure that they have labeled a sufficient amount of data to train a model that meets their target accuracy.

Example Use Case (Continued): To create the labeled dataset for training, our data scientist adopts the *active learning paradigm* (AL) [45, 66]. AL operates in cycles. Each cycle, she labels a new batch of data and trains the model on all the labeled data up to that cycle. The model trained in the current cycle is used to sample the most informative data for the next cycle using some sampling technique (e.g., uncertainty, diversity [66]). Figure 1 (A, B) presents an illustration of her workflow.

AL is an emerging paradigm that focuses on reducing data labeling efforts, which also requires periodic model selection. However, AL is not the only paradigm that requires periodic model selection during data labeling. For example, other popular data labeling approaches–such as simple manual annotation, crowd workers (e.g., SageMaker Ground Truth [65]), and programmatic supervision [57]–may also require periodic model selection to evaluate the benefit of labeling more data on model accuracy.

1.1 Current Practice and Inefficiencies

Today, one executes the DTL workload by training a DL model with frozen layers as it is and repeats the process for all model selection cycles [14, 45]. This leads to incurring redundant computations in frozen layers as they are repeatedly invoked with the same inputs to generate the same output. We identify three types of redundancies:

- **Redundancies across training epochs**: DL model training is iterative. Each iteration, also called an epoch, reads the full dataset and feeds it through the layers. This leads to redundant computations across training epochs.

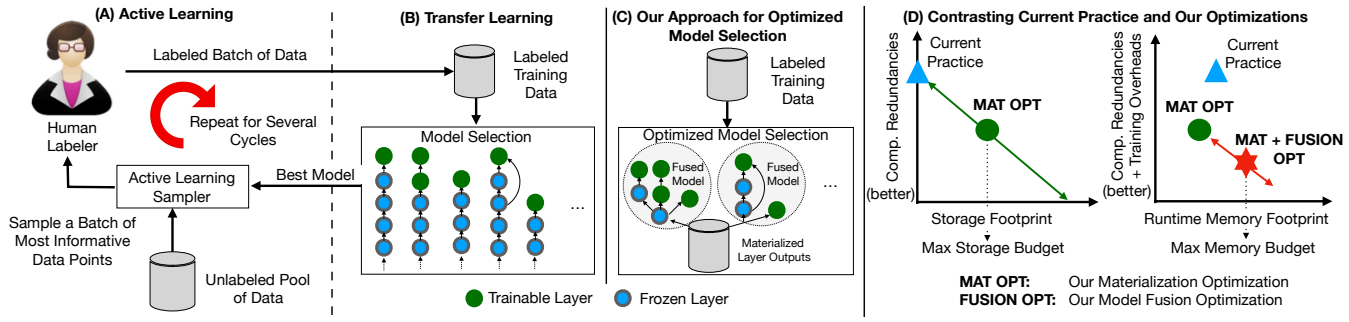

Figure 1: (A) Human labeler labels batches of most informative data. (B) A pre-trained model is adapted for a target task. (C) Our approach for optimized DTL model selection performs materialization and model fusion optimizations. (D) Contrasting the current practice and our approach on different trade-off spaces.

- **Redundancies across models**: Practitioners have to perform model selection where they explore several different layer freezing schemes and training hyperparameters (e.g., learning rate, batch size). Thus, a model selection workload can contain models that share frozen layers. Independently training them leads to redundant computations across models.
- **Redundancies across model selection cycles**: Model selection is repeated for every new snapshot of training data. This leads to redundant computations across model selection cycles.

Overall, these redundancies are problematic, at least for three main reasons. First, they increase the model selection runtimes and impede human productivity. Human labelers may have to wait longer until model selection completes to proceed to the next data labeling cycle. Second, they lead to higher monetary costs, especially in pay-as-you-go environments in the cloud. Third, they also lead to significantly higher energy consumption and associated environmental issues, which are expected to further amplify by the wide adoption of DL in many domains [7].

1.2 Our Proposed Approach

In this work, we use a database-inspired lens to formalize, optimize, and accelerate the DTL model selection in the presence of frozen layers. We cast the DTL model selection as a novel instance of *multi-query optimization* (MQO) [64] and perform two data management-inspired optimizations:

- **Materialization Optimization:** We materialize intermediate layer outputs from a chosen set of frozen layers on the first time they are computed and avoid repeated recomputations. However, the size of the intermediate layer outputs can be orders of magnitude (even up to 100X) larger than the input data, and it may not be possible to materialize all frozen layers due to storage constraints [22]. Even if it is possible to materialize all frozen layers, it may be the case that some outputs can be computed faster using others instead of loading them. Therefore, the challenge is to pick an optimal set of frozen layers that can reduce model selection runtimes subject to a storage budget as shown in Figure 1 (D). This optimization is an instance of view selection being combined with MQO to optimize DL workloads [13]. By doing so, we reduce all three types of computational redundancies.
- **Model Fusion Optimization:** Even after the materialization optimization, there can be remaining frozen layers shared among models in the workload. Thus, we propose model fusion optimization, which is inspired by the pipelined multi-query execution in relational query processing [16]. It builds on top of our materialization optimization and reduces the redundant computations by fusing multiple models and eliminating frozen common subexpressions in the models. It also amortizes model training overheads and I/O overheads [44]. However, excessive model fusion can increase the runtime memory footprint and cause workload crashes. The challenge is to pick an optimal set of models to be fused, such that it reduces model selection runtimes subject to a runtime memory budget as shown in Figure 1 (D).

Our optimization techniques are orthogonal to the data labeling scheme used. Thus, we can support all kinds of data labeling schemes–such as active learning, simple manual labeling, crowd workers, and programmatic supervision–in a unified manner.

We implement our techniques into a system we call Nautilus. It runs on top of the popular DL libraries Keras and TensorFlow [4]. Nautilus is tailored to limited-resource settings such as workstations and personal computers, which cover a vast majority of DTL use cases [45]. Nautilus provides easy-to-use APIs to specify the DTL workload over evolving training data and optimizes DTL model selection. We evaluate Nautilus empirically on five workloads, including one from an influential NLP publication [17], on two benchmark ML datasets: CoNLL [71] and Malaria [56]. Nautilus avoids many compute redundancies and significantly reduces training and I/O overheads enabling up to 5X model selection time reductions. Thus, Nautilus significantly reduces system resource costs and also improves human productivity (i.e., less waiting time between model selection cycles) of DTL workloads. Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to formalize and optimize deep transfer learning (DTL) workloads over evolving training data from a data management standpoint.
- We cast the iterative training of DL models with frozen layers as a new instance of MQO and present a materialization optimization technique to reduce redundant computations of DTL workloads.
- We present model fusion optimization, which builds on top our materialization optimization to further reduce redundant computations and model training overheads of DTL workloads.
- We implement our ideas into a data system called Nautilus and perform an extensive empirical evaluation using five end-to-end

workloads on two benchmark ML datasets. NAUTILUS reduces DTL model selection runtimes by up to 80% in some cases.

Outline: Section 2 presents some background, formalizes the workload, and explains our assumptions. Section 3 provides an overview of NAUTILUS and implementation details. Section 4 dives into the system optimizations. Section 5 presents the experiments. We discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND AND PRELIMINARIES

We start with a brief background on DL model training, which is needed to understand our system optimizations. We then present the formal problem description and list our assumptions. The notation used is explained in Table 1.

2.1 Background on DL Model Training

DL models are DAG structured graphs. A node in the graph is also called a *layer* and a layer takes in one or more input tensors and produces an output tensor. Most layers also have trainable parameters, and they are trained using a combination of two techniques: mini-batch stochastic gradient descent (SGD) and back-propagation.

Mini-batch SGD: Mini-batch SGD is an iterative numerical optimization method, which performs multiple passes over the data using small data batches called mini-batches. A single pass is also called an *epoch* of training. In an epoch, SGD randomly samples mini-batches and estimates loss gradients (e.g., cross-entropy loss [21]) with respect to all the trainable model parameters. During training, the parameters are iteratively updated using the gradients.

Back-propagation: Loss gradients in SGD are calculated using the *back-propagation* technique. It leverages the DAG structure of the model and the chain rule of differentiation to efficiently calculate the gradients. For every mini-batch, it first calculates all intermediate layer and the model outputs. It then calculates the mini-batch training loss using the model outputs and the target labels/values. These two steps are collectively called the *forward-pass* of training. After the forward-pass, back-propagation calculates the loss gradient with respect to the model outputs and traverses backward from the model output layers towards the model input layers. Along its path, it propagates the loss gradient through the layers. The loss gradient with respect to the inputs of a layer, also called the *input gradients*, is calculated using the loss gradient with respect to the output of that layer. Back-propagation also calculates the loss gradients with respect to the unfrozen layer parameters, also called the *parameter gradients*, and uses them to update the trainable parameters. This second pass of going backward is called the *backward-pass* of training.

For more technical details on SGD and back-propagation techniques, we refer the reader to [21, 23].

2.2 Definitions and Data Model

We start by defining some terms and notation to formalize the DTL workload. We will use these terms in the rest of this paper.

DEFINITION 2.1. *A **layer** is a function l that takes a list of input tensors $t_1, t_2, \ldots, t_m$ ($m \geq 1$) of fixed shape and outputs a tensor $t' = l(t_1, t_2, \ldots, t_m)$ of potentially different, but fixed shape. A list of tensors $t_1, t_2, \ldots, t_m$ are said to be shape-compatible with l iff their shapes conform to what l expects for its inputs.*

Table 1: Notation used in Section 2

Symbol	Description
$M = (L, E)$	A DL model with L layers and E edges.
$f(l)$	$f(l) = True/False$. Function indicating layer l is frozen during model training.
$m(l)$	$m(l) = True/False$. Function indicating layer l is materializable.
ϕ	Set of training hyperparameters.
Q	$Q = \{(M_1, \phi_i), \ldots, (M_n, \phi_n)\}$. Set of model and training hyperparameter pairs.
D	Labeled dataset. D_k corresponds to the dataset snapshot at time k. D_k^{train} and D_k^{valid} are training and validation splits of D_k, respectively.
$g(M, \phi, D)$	$g(M, \phi, D)$. Training function that takes in a M, ϕ, and D. After training M using ϕ on D^{train}, returns the model accuracy on D^{valid}.
ΔL^+,ΔL^-	Added (ΔL^+) or removed (ΔL^-) layers.
ΔE^+,ΔE^-	Added (ΔE^+) or removed (ΔE^-) edges.
ΔD_k^+	New labeled data for the model selection cycle k.

DEFINITION 2.2. *A **model** $M = (L, E)$ is a directed acyclic graph (DAG) of layers $L = l_1, \ldots, l_n$ and edges E between the layers.*

DEFINITION 2.3. *A layer l is **frozen** if its learnable parameters are not updated during training. A layer with no learnable parameters is also frozen. $f(l)$ is a function that indicates a layer l is frozen or not.*

Frozen layers incur redundant computations. However, a frozen layer that has a non-frozen ancestor doesn't incur redundant computations. Thus, we introduce the notion of a *materializable* layer to identify layers that will contribute to redundant computations.

DEFINITION 2.4. *A layer l is **materializable** if it's a model input layer (i.e., $l \in I$) or is a frozen layer with all its parent layers being materializable. $m(l)$ is a function that indicates a layer l is materializable or not.*

2.3 Workload Formalization

We are given a candidate set of model and training hyperparameter pairs $Q = \{(M_i, \phi_i) : i \in 1 \ldots n\}$. Each candidate model M_i is adapted from a source pre-trained model $M_{src} = (L_{src}, E_{src})$ and its pre-trained layers are frozen according to some scheme. We assume that we have access to a model training function that trains a candidate model M_i using hyperparameters ϕ_i on a training split of D_k^{train} and returns validation accuracy on a validation split D_k^{valid}. We represent this model training function as $g(M_i, \phi_i, D_k)$. We then perform model selection to find the best candidate model based on validation accuracy and repeat it whenever the dataset snapshot changes from D_k to D_{k+1}. More precisely, we describe the workload as follows:

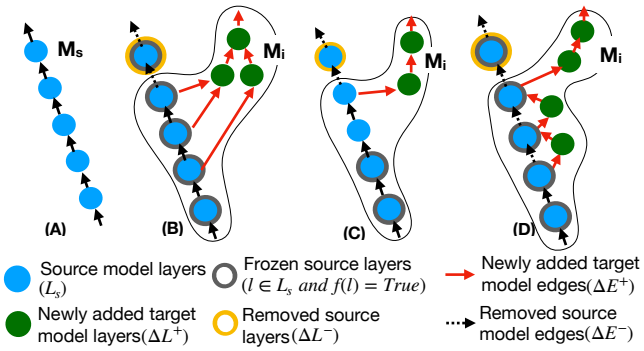

Figure 2: Transfer learning approaches. (A) Source model M_S. (B) Feature transfer. (C) Fine-tuning. (D) Adapter training.

$\forall D_k \in \{D_0, D_1, \dots\}$:

$$\underset{(M_i,\phi_i)\in Q}{\arg\max}\ g(M_i, \phi_i, D_k) \tag{1}$$

$$M_i = (L_i, E_i) \tag{2}$$

$$L_i = \Delta L^+ \cup (L_{src} - \Delta L^-),\ E_i = \Delta E^+ \cup (E_{src} - \Delta E^-)$$

$$\Delta E^+ \subseteq (\Delta L^+ \times \Delta L^+) \cup (L_{src} \times \Delta L^+) \cup (\Delta L^+ \times L_{src}) \cup (L_{src} \times L_{src}) \tag{3}$$

$$D_{k+1} = D_k \cup \Delta D_k^+ \tag{4}$$

Equation 1 captures the model selection step. Equation 2 captures the structure of a candidate model M_i, which is obtained by adding a new set of layers ΔL^+ and edges ΔE^+ to M_{src} and removing a set of existing layers $\Delta L^- (\subset L_{src})$ and edges $\Delta E^- (\subseteq E_{src})$ from M_{src}, while ensuring the DAG structure and the shape compatibility of all layer inputs. Equation 3 captures the structure of ΔE^+, which has four different types of edges based on the originating and terminating layer type. Finally, Equation 4 captures how the next labeled data snapshot D^{k+1} is obtained by adding a new set of labeled data records ΔD_k^+ to the current snapshot D_k.

2.4 Popular Transfer Learning Approaches

We found three transfer learning approaches that are popular among practitioners: features transfer, adapter training, and fine-tuning. They can be treated as special cases of our general workload formalization that impose specific structural properties on the newly added edges ΔE^+ and the layer freezing scheme.

- **Feature Transfer:** In this approach, one freezes all layers in L_{src} (i.e., $f(l) = True,\ \forall l \in L_{src}$) and restricts ΔE^+ to only contain edges between newly added layers or edges from a source model layer to a newly added layer (i.e., $\Delta E^+ \subseteq (\Delta L^+ \times \Delta L^+) \cup (L_{src} \times \Delta L^+)$). Common practice is to add new layers on top of the penultimate layer, any other top-level layer, or a collection of top-level layers in M_{src} [17]. The structure of an example M_i that uses the feature transfer approach is shown in Figure 2 (B).
- **Fine-Tuning:** This approach is similar to feature transfer but there is at least one pre-trained layer l in the adapted model that is unfrozen (i.e., $f(l) = False$). Parameters of all such layers along with the parameters of the newly added layers ΔL^+ are learned during training [17]. While one can unfreeze all layers in M_{src}, researchers have shown that freezing most of the lower-level layers in M_{src} and fine-tuning only the top few layers can achieve similar results to fine-tuning all layers [36]. This also avoids the risk of pre-trained information in M_{src} getting overwritten due to overfitting, which can easily happen in transfer learning settings with limited training data [43, 62]. The structure of an example M_i which uses the fine-tuning approach is shown in Figure 2 (C).
- **Adapter Training:** In this approach, one freezes most of the layers in L_{src}, but not necessarily all (i.e., $f(l) = True,\ \exists l \in L_{src}$). However, ΔE^+ can be more general with having edges between newly added layers, edges going from source model layers to newly added layers, and also edges going from newly added layers to source model layers (i.e., $E^+ \cap (L^+ \times L_{src}) \neq \phi$). The common practice is to add small bottleneck layers called *adapters* between the layers of M_{src} [29, 53, 58]. While one can add adapters to all layers in M_{src}, researchers have shown that adding adapters only to the top-level layers can be as effective as adding adapters to all the layers [61]. The structure of an example M_i which uses the adapter training approach is shown in Figure 2 (D).

Model Selection for Popular Approaches: Model selection is unavoidable for any DL model training as one has to tune the training hyperparameters like batch size, regularization, learning rate, and number of training epochs. In addition to the above common training hyperparameters, popular transfer learning approaches also need additional architectural tuning. For example, feature transfer needs exploring features from several layers or layer combinations. Fine-tuning needs exploring different layer freezing schemes (e.g., up to which layer to freeze?). Adapter training also needs exploring different adapters and adapter placement schemes (e.g., to which layers to add adapters?). It has been shown that all these approaches can be equally competitive for a wide variety of transfer learning tasks [51, 62]. Thus, practitioners need to explore multiple approaches before picking the best one.

2.5 Assumptions and Limitations

For tractability, we make the following simplifying assumptions.

- We assume that the model selection workload specification is fixed and is repeated for all model selection cycles. This approach can support both grid and random search-based model selection, which covers an overwhelming majority of practical and research DTL workloads [10]. We leave adding support for more complex model selection methods for future work.
- We also assume that the layer freezing scheme used for adapting the source model M_{src} is also fixed throughout the entire model training. However, few dynamic layer freezing schemes have also been proposed in the literature [15, 18, 30]. We leave adding support for these layer freezing schemes to future work.
- Our workload formalization is tailored to DAG-structured models (i.e., model graphs with no cycles). However, there is a family of DL models called recurrent DL models that does have cycles. Nautilus can support recurrent models by unraveling them in time and transforming them into a non-recurrent DL model.
- Some DL models, especially in computer vision applications, contain random data augmentations such as random cropping and flipping in their pre-processing steps [50, 67]. In such settings, our optimizations will not yield benefits as each training record is unique. One can overcome this by materializing a larger augmented training dataset first and then using it to train the model.

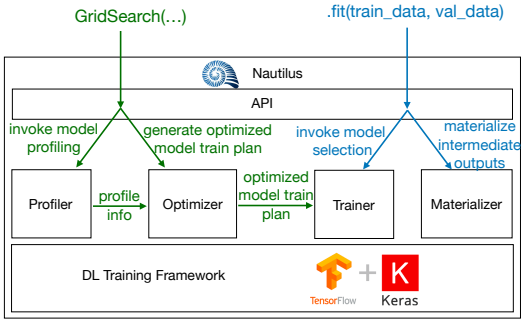

Figure 3: High-level architecture of NAUTILUS and the interactions between system components. fit(...) method is called for every model selection cycle.

It should be noted that even in some computer vision applications such as in medical imaging, random data augmentation is not always useful as it can generate implausible training data [49, 52].

3 SYSTEM OVERVIEW

We implement NAUTILUS on top of TensorFlow and Keras libraries. NAUTILUS optimizes DTL model selection. It has 5 main components: API, Profiler, Optimizer, Materializer, and Trainer. Figure 3 presents the architecture of NAUTILUS. Next, we provide details on NAUTILUS's components.

- **API:** NAUTILUS's API is inspired by libraries like Scikit-Learn and Keras. Users create a model selection object by specifying a parameter search space and a user-defined model initialization function. The model initialization function encapsulates the logic to interpret the search parameter values. It takes in an instance of parameter values ϕ_i and returns a Keras model ready to be trained. Thus, NAUTILUS can support both architectural tuning parameters (e.g., which layers to add, prune, or freeze) and training hyperparameters (e.g., learning rate) in a unified manner. Users can also override the default system config values used by the optimizer. These include storage and runtime memory budget, expected maximum number of training records, disk throughput, and compute throughput values. Users initiate model selection by calling the *fit(...)* method and passing a batch of training and validation data. It is called for every model selection cycle.
- **Profiler:** When a user initializes a workload, NAUTILUS internally invokes its Profiler. Profiler invokes the user-defined model initialization function to initialize all models, profiles them, and finally stores the initialized model checkpoints on disk. For profiling, it uses the features available in TensorFlow. The profiling information includes shapes of all intermediate output tensors and the forward-pass layer compute costs in FLOPs.
- **Optimizer:** The optimizer takes in the profiling information and system configuration values and generates an optimized model training plan. It then generates the model checkpoints for the new model training plan by reading the original model checkpoints and stores the new model checkpoints on disk. It also creates a model checkpoint that is used to generate the outputs of the chosen materialized layers. We discuss system optimizations in more detail in Section 4.
- **Materializer:** When a user initiates a model selection step by passing a new batch of labeled data, the API calls the Materializer to update both the labeled dataset and outputs of chosen materialized layers. The Materializer reads the output materialization model checkpoint, generates the intermediate outputs, and stores them on disk. One could also store the outputs in DRAM. However, their size can be significant (e.g., 10s of GBs) and can exhaust DRAM. Also, DL models are often compute-intensive, and I/O overheads can be mitigated by prefetching. Thus, if there is excess DRAM available, we rely on the OS disk cache to cache the intermediate outputs.
- **Trainer:** The Trainer trains the models on the labeled training dataset according to the optimizer-generated training plan and saves the trained model parameters on disk. It extends the model training feature in Keras to support training a model with multiple optimizers with each optimizer operating on a separate trainable branch of a model. This feature is needed for our model fusion optimization, which we discuss in more detail in Section 4.3. Finally, the Trainer returns the model that has the best validation accuracy back to the user.

  In the current version, the Trainer supports single-node model training with or without GPU support. If multiple GPUs are available, the Trainer can also train models in data-parallel manner. We have focused on supporting DL models that fit in single-node/device memory during training (i.e., DRAM for CPU training and GPU's memory for GPU training) as many practical DTL applications operate in low-resource settings. Furthermore, the runtime memory usage of DL models is significantly reduced by pre-trained layer freezing, making most DTL models trainable on single-node/device memory. We discuss more details about runtime memory usage in Section 4.3.3.

4 SYSTEM OPTIMIZATIONS

We first introduce the notion of *multi-model graph*, the core data structure used by our optimizations. We then dive into more details of our optimizations. We conclude the section by characterizing the attainable theoretical speedups. Table 2 presents the additional notation used in this section.

4.1 Multi-Model Graph

We create an information graph composed of all candidate models in a model selection workload, which we call a *multi-model graph*. It is inspired by the AND view graph in relational multi-query optimization [25]. But we adapt it to the DL model selection context by leveraging the properties of DL models and training. Next, we define some helper terms and formalize the multi-model graph.

DEFINITION 4.1. *An **expression** for a layer l in a model M is a DAG of layers with model input layers I as sources and l as the sink.*

DEFINITION 4.2. *An expression is a **materializable expression**, iff the sink layer of the corresponding DAG is materializable.*

DEFINITION 4.3. *Two layers l_i and l_j are said to be **identical**, i.e., $l_i \equiv l_j$, if both of them are of same type, have identical configuration values, and identical trainable parameter values. Two expressions e_i and e_j are said to be identical, i.e., $e_i \equiv e_j$, if both of them have the same DAG structure and all corresponding layer pairs are identical.*

Table 2: Additional Notation used in Section 4

Symbol	Description
I, O	Input (I) and output (O) layers in model M.
U, V	Materializable (U) and materialized (V) layers in model M.
B_{disk}, B_{mem}	Disk storage budget (B_{disk}) and runtime memory budget (B_{mem}).
M^{opt}	Optimal reuse plan model for M.
$c_{comp}(l)$, $c_{load}(l)$	Functions for estimating the computation cost and data load cost of layer l.
$s_{disk}(l)$, $s_{mem}(l)$	Functions for estimating the storage usage and runtime memory usage of layer l.
$C(M)$	Training cost of model M.
$s_{mem}(M)$	Peak memory usage for training model M.
$q(l, M)$	$q(l, M)$ = is a function indicating the layer l is *pruned*, present and *computed*, or present and *loaded* (i.e., $l \in I$) in M.
r	Expected maximum training data records.

Definition 4.4. *A model $M = (L, E)$ is called a **multi-model graph** for the models $M_1, M_2, \ldots, M_n$, iff for every output layer of every M_i, there is an expression in M that is identical to the expression of M_i's corresponding output layer.*

Constructing the Multi-Model Graph: For a model selection workload with a set of models $M_1, M_2, \ldots, M_n$, we construct the multi-model graph M by merging all the materializable identical sub-expressions in them. If layer l in model M_i is materializable (i.e., $l \in U_i$), the corresponding layer l in the multi-model M is also materializable (i.e., $l \in U$), and vice-versa. Only the materializable layers in the multi-model need to be considered for materialization (i.e., $V \subseteq U$). We use 4 metrics to capture the runtime and layer output characteristics of all layers in M. They can be obtained by profiling the original models in the workload. We represent these values normalized for a single training record. They include:

- $c_{comp}(l)$, which captures the layer **computation cost** in terms of floating-point operations (FLOPS). It includes both the forward- and backward-pass computation costs. The forward cost can be directly obtained from the profiling information. However, profiling information provided by DL frameworks often does not include backward-pass cost. Hence, as per the standard practice [5, 23], we use the forward cost to estimate the backward cost. For a trainable (i.e., not frozen) layer, we set it to thrice the number of forward-pass FLOPs to account for forward-pass, input gradient, and parameter gradient computations. For a frozen but not materializable layer, we set it to twice the number of forward-pass FLOPs to account for both forward and input gradient computations. For a materializable layer, we set it to the forward-pass FLOPs as there is no back-propagation happening.
- $s_{disk}(l)$, which captures the layer **output size on disk** in Bytes. We estimate it using the output tensor dimensions and data type.
- $c_{load}(l)$, which captures the layer **output loading cost from disk** in terms of missed compute FLOPs. We calculate it by first estimating the disk read time and multiplying it by the FLOPs throughput of the system. We ignore the data transfer time from DRAM to GPU memory as disk load time dominates the total time. Both compute throughput and disk read speed affect $c_{load}(l)$. We use pre-configured values for them, which match the characteristics of the available hardware.
- $s_{mem}(l)$, which captures the layer **output size in memory** in Bytes. We estimate it using the output tensor dimensions and data type, similar to the on-disk output size. However, for a composite layer that consists of several basic layers (e.g., a transformer layer composed of several dense, addition, and layer normalization layers [73]), we estimate it by summing the size of all child layer output tensors. We treat composite layers differently to account for all the intermediate output tensors that the backward-pass may need. We explain this in more detail in Section 4.3.3.

4.2 Materialization Optimization

We formally present our materialization optimization problem and present a mixed-integer linear programming-based solution.

4.2.1 Materialization Optimization Problem. Our goal is to find an optimal set of intermediate layer outputs to materialize subject to a storage constraint. Given a set of such layers, we rewrite the model graphs to reuse these intermediate outputs during training. We assume that the model selection workload is fixed and repeated for all model selection cycles. Thus, we focus on materialization optimization for a single model selection cycle. Intermediate layer materialization incurs compute and I/O costs. However, it is amortized by the iterative DL model training costs, which get further amplified by multiple candidate models and multiple model selection cycles. Therefore, we ignore the computation and I/O costs for materializing the chosen layers and optimize for minimizing the total model training time. We estimate the storage footprint using a pre-configured maximum number of records r and reuse the obtained optimal materialization plan for all model selection cycles until the training dataset size reaches that limit. We explain how we relax the max training records constraint in Section 4.2.3.

Let $M = (L, E)$ be the multi-model graph for the set of models $\{M_i = (L_i, E_i) : \forall i \in 1, \ldots, n\}$. U is the set of materializable layers in M and V is the optimal set of materialized layers ($V \subseteq U$). I_i and O_i are the input and output layers of M_i, respectively. $C(M)$ is a function that estimates the training cost of model M (for one input record in FLOPs) and $q(l, M)$ is a function that indicates the presence of layer l in model M. We first introduce the notion of an optimal reuse plan model that captures how we should rewrite a model graph to reuse the materialized layer outputs in V and then explain how we find V.

Definition 4.5. *M_i^{opt} is called the **optimal reuse plan model** for M_i, iff*

(1) It has the same output layers as M_i (i.e., $O_i^{opt} = O_i$).
(2) Every layer l in M_i^{opt} is also in M_i (i.e., $L_i^{opt} \subseteq L_i$).
(3) For every layer l in M_i^{opt}, parents of l in M_i^{opt} are same as its parents in M_i; or l is in V (i.e., $l \in V$).
(4) Has the lowest training cost $C(M_i^{opt})$ out of all such candidates.

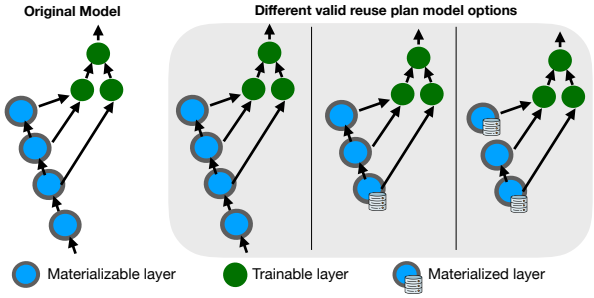

Figure 4: Different valid reuse plan model options for a model with materializable layers.

Training M_i^{opt} is equivalent to training M_i as both perform logically equivalent operations. M_i^{opt} can be obtained from $M_i = (L_i, E_i)$ by taking one of the following three actions for every layer $l \in L_i$: (1) pruning i.e., $q(l, M_i^{opt}) = pruned$, (2) retaining and computing i.e., $q(l, M_i^{opt}) = computed$, and (3) retaining and loading as an input i.e., $q(l, M_i^{opt}) = loaded$. Figure 4 shows an example model graph and several valid reuse plan models. We estimate the training cost of an optimal reuse plan model $C(M_i^{opt})$ by summing all layer compute costs and input loading costs as follows:

$$C(M_i^{opt}) = \sum_{l \in L_i} \mathbb{1}\{q(l, M_i^{opt}) = computed\} \cdot c_{comp}(l) + \mathbb{1}\{q(l, M_i^{opt}) = loaded\} \cdot c_{load}(l) \tag{5}$$

Equation 5 makes the simplifying assumption that layer computations and input loadings do not overlap during training. However, DL model training operates in pipelined fashion on mini-batches of training data, and it may be possible to hide some of the data load costs by pre-fetching the data. Nevertheless, our formulation provides a reasonable upper bound for the total model training cost that is sufficient for our purpose.

With the optimal reuse plan model M_i^{opt} and its training cost $C(M_i^{opt})$ defined, the materialization optimization problem can be formally expressed as follows:

$$\underset{V, M_i^{opt} \forall i \in \{1,\ldots,n\}}{\arg\min} \sum_{i=1}^{n} C(M_i^{opt}) \cdot r \cdot \text{epochs}(\phi_i) \tag{6}$$

subject to:

$$\sum_{l \in V} s(l) \cdot r \leq B_{disk} \tag{7}$$

Equation 6 minimizes the total model training time. The model training time of a single model M_i is estimated by multiplying the training cost of the corresponding optimal reuse plan model M_i^{opt}, the maximum number of training records r, and the number of training epochs $\text{epochs}(\phi_i)$. $\text{epochs}(\phi_i)$ is a training hyperparameter provided by the user. Equation 7 ensures that the materialized layer outputs do not exhaust the disk storage budget B_{disk}.

4.2.2 Mixed Integer Linear Programming Formulation. We present a mixed-integer linear programming (MILP) formulation of the materialization optimization problem. Let $l_{i,j}$ be the j^{th} layer of i^{th} model $M_i = (L_i, E_i)$ in the multi-model M. u_k is the k^{th} layer of the set of materializable layers U in M. We introduce three sets of binary indicator variables X, Y, and Z as follows:

$$\text{for all } i \in \{1,\ldots,n\}, j \in \{1,\ldots,|L_i|\}, k \in \{1,\ldots,|U|\}$$
$$(a)\ X_{i,j} = \mathbb{1}\{q(l_j, M_i^{opt}) \neq pruned\}$$
$$(b)\ Y_{i,j} = \mathbb{1}\{q(l_j, M_i^{opt}) = computed\} \tag{8}$$
$$(c)\ Z_k = \mathbb{1}\{u_k \in V\}$$

$X_{i,j}$ indicates whether the j^{th} layer of M_i is present in M_i^{opt}. $Y_{i,j}$ indicates whether the j^{th} layer of M_i is computed in M_i^{opt}. $X_{i,j}$ and $Y_{i,j}$ collectively determine $q(l_{i,j}, M_i^{opt})$. Z_k indicates whether the materializable layer u_k of M is materialized. With these indicator variables defined, the MILP-based approach for the materialization optimization problem can be expressed as follows:

$$\underset{X,Y,Z}{\arg\min} \sum_{\forall\ i,j} \Big(X_{i,j} \cdot c_{load}(l_{i,j}) + Y_{i,j} \cdot (c_{comp}(l_{i,j}) - c_{load}(l_{i,j})) \Big) \cdot r \cdot \text{epochs}(\phi_i) \tag{9}$$

subject to:

$$(a)\ l_{i,j} \in O \implies X_{i,j} \geq 1,\ \forall i,j$$
$$(b)\ X_{i,j} - Y_{i,j} \geq 0,\ \forall i,j$$
$$(c) \sum_{l_{i,k} \in \text{parents}(l_{i,j})} X_{i,k} - Y_{i,j} \geq 0,\ \forall i,j \tag{10}$$
$$(d)\ u_k \equiv l_{i,j} \implies X_{i,j} - Y_{i,j} \leq Z_k,\ \forall i,j,k$$
$$(e) \sum_{u_k \in U} Z_k \cdot s_{disk}(u_k) \cdot r \leq B_{disk}$$

Equation 9 is equivalent to the optimization objective presented in Equation 6. Equation 10 (a) ensures that output layers of all models are not pruned and avoids the trivial solution where all layers are pruned. Equation 10 (b) ensures a computed layer is not pruned and Equation 10 (c) ensures parents of a computed layer are also not pruned. Equation 10 (d) ensures that only the materialized layers are loaded from the disk and Equation 10 (e) ensures that the cumulative size of the materialized layers does not exhaust the storage budget, which is equivalent to Equation 7.

Given the above, a straightforward approach to optimization is to use an MILP solver like Gurobi [26]. Z indicates the materialized layers and X and Y can be used to construct the optimal reuse plan models. If disk storage budget is not a critical resource, Z may contain materialized layers that do not get used in reuse plan models. Such layers can be discarded using a post-processing step. It can be shown that the above materialization optimization problem is NP-hard using a reduction from the known NP-hard Knapsack problem [34, 74]. However, we found that an MILP solver-based approach finds the optimal solution within a short execution time (e.g., few 10s of seconds) at the scale of practical DTL model selection workload sizes. We provide more details on MILP execution time in Section 5.3.

4.2.3 Incremental Feature Materialization. For every new batch of labeled data, we materialize it and also incrementally update the outputs of the chosen materialized layers. We repeat this until the pre-configured maximum number of training records r is reached. When we reach the maximum number of training records, we use

Algorithm 1 : FuseModels(Q, B_{mem}, V)

$Q' = \{(M_i, M_i^{opt}, \phi_i) | \forall i \in [1, \ldots, |Q|]\}$
while there are not-considered fusible model pairs in Q' **do**
  $P = \{(i,j) | \text{all not-considered fusible model pair indices}\}$
  $M_{i,j} \leftarrow$multi-model for M_i and $M_j, \forall (i,j) \in P$
  $M_{i,j}^{opt} \leftarrow$optimal reuse plan model for $M_{i,j}, \forall (i,j) \in P$
  $c_{i,j} \leftarrow C(M_i^{opt}) + C(M_j^{opt}) - C(M_{i,j}^{opt}), \forall (i,j) \in P$
  $i_*, j_* \leftarrow \arg\max_{(i,j) \in P} c_{i,j}$, such that $s_{mem}(M_{i,j}^{opt}) \leq B_{mem}$
  $Q' \leftarrow Q' \cup \{(M_{i_*,j_*}, M_{i_*,j_*}^{opt}, \phi_{i_*} \cup \phi_{j_*})\}$
  $Q' \leftarrow Q' \setminus \{(M_{i_*}, M_{i_*}^{opt}, \phi_{i_*}), (M_{j_*}, M_{j_*}^{opt}, \phi_{j_*})\}$
end while
return $\{(M_i, \phi_i) | \forall i \in 1, \ldots, |Q'|\}$

an exponential backoff scheme with a factor of 2 to update r (i,e., $r \leftarrow 2 \times r$). We then rerun the materialization optimization to find a new set of materialized layers and materialize them. Overall, our exponential backoff scheme to update r provides a good balance between materialization overheads and storage wastage.

4.3 Model Fusion Optimization

In model fusion, we partition the set of models such that the fused models corresponding to the partitions reduce the redundant computations with the highest margin while ensuring the runtime memory budget B_{mem} is not exhausted. We first explain our approach for finding such a partitioning. We then explain how to adapt the optimal reuse plan model for the fused model setting and also explain how we estimate fused model memory footprint.

4.3.1 Partitioning the Set of Models. We leverage the pipelining nature of the mini-batch SGD training method to train a fused model, which operates on one mini-batch at a time. We also ensure that all models in a partition have the same training batch size. Otherwise, they cannot be fused during training. Given such a partition, we create the multi-model for the partition and find the optimal reuse plan model. Multi-model creation will fuse only the materializable layers that do not require any training. Therefore, the training optimizer for the fused model's reuse plan model can be represented as the set of optimizers from the source models where each optimizer operates on the corresponding trainable branch.

However, finding the optimal partitioning requires considering all candidate partitions, which is exponential in the number of models in the model selection workload. Furthermore, the training cost ($C(M^{opt})$) and the peak runtime memory usage ($s_{mem}(M^{opt})$) of the optimal reuse plan model for each partition's multi-model is not available in constant time. Thus, we use a greedy heuristic that only considers pair of models to be fused at a time. The high-level approach is presented in the `FuseModels` procedure of Algorithm 1.

`FuseModels` takes in a set of model and training hyperparameter pairs Q, runtime memory budget B_{mem}, set of materialized layers V, and returns a set of fused model and training hyperparameter pairs. For every model M_i in Q, we first find the optimal reuse plan model M_i^{opt} that reuses materialized intermediate layers in V and create a new set Q' containing (M_i, M_i^{opt}, ϕ_i) triples. We then create a set P of candidate model pair indices to be fused that has the same training batch size. For every candidate model pair indices (i,j) in P, we create the multi-model $M_{i,j}$ and find the optimal reuse plan model $M_{i,j}^{opt}$. We explain how we find the optimal reuse plan model $M_{i,j}^{opt}$ given V in Section 4.3.2. We also estimate the runtime memory usage $s_{mem}(M_{i,j}^{opt})$ and ensure that it does not exceed the runtime memory budget B_{mem}. Details on how we estimate $s_{mem}(M_{i,j}^{opt})$ are provided in Section 4.3.3. From the fusible model pairs, we pick the pair that will result in the highest cost reduction. We add the fused model back to Q' and remove the source models M_i and M_j from Q'. The training hyperparameters for a fused model are derived by combining source hyperparameters ϕ_i and ϕ_j. We repeat this process until there are no more fusible models.

4.3.2 Optimal Reuse Plan Given a Set of Materialized Layers. The optimal reuse plan model $M_{i,j}^{opt}$ for a fused model $M_{i,j}$ that reuses materialized layers V can be found by adapting the MILP presented in Section 4.2.2. The main difference here is that the set of materialized layers is already determined. Thus, we no longer need the indicator variable Z (Equation 8 (c)) and also remove the constraints (d) and (e) in Equation 10. We use $\{M_{i,j}\}$ as the set of input models. In this case, the multi-model M corresponding to the set of input models will be the same as the input model $M_{i,j}$. After the optimization, $M_{i,j}^{opt}$ can be obtained from the resulting indicator variable values X and Y. While most MILP problems are NP-hard, it has been shown that the resulting MILP problem can be solved in PTIME via a reduction to the Max-Flow problem [74].

4.3.3 Estimating Peak Runtime Memory Usage. Estimating the peak runtime memory usage of training a DL model is a challenging task as it depends on various factors including both workload- and system-specific. However, we can identify three main types of memory usage that dominates the overall usage: (1) memory to store the parameter tensors, (2) memory for the workspace for performing layer operations, and (3) memory to store the layer outputs needed for back-propagation.

The first type can be calculated by inspecting data types and dimensions of the parameter tensors. The second type doesn't vary based on model architecture and thus can be estimated by a fixed value (e.g., 1GB). The third type, which often dominates the total memory usage, depends on both the model architecture and the DL system. Next, we discuss more details on how we estimate it.

The back-propagation technique used to train DL models operates in two phases: forward-pass and backward-pass. The backward-pass calculates the gradients and needs access to the layer output tensors generated during the forward-pass. For example, the backward-pass operation of linear algebra-based layers such as Dense and Convolutional layers need access to the forward-pass layer input to calculate the parameter gradient. Some non-linear transformation layers like ReLU need access to layer output to calculate the input gradient. And some other non-linear transformation layers like MaxPooling need access to both layer input and output to calculate the input gradient. Thus, the DL system will accumulate layer output tensors during the forward-pass and gradually release them during the backward-pass. However, the exact order by which output tensors are accumulated and released is determined by the

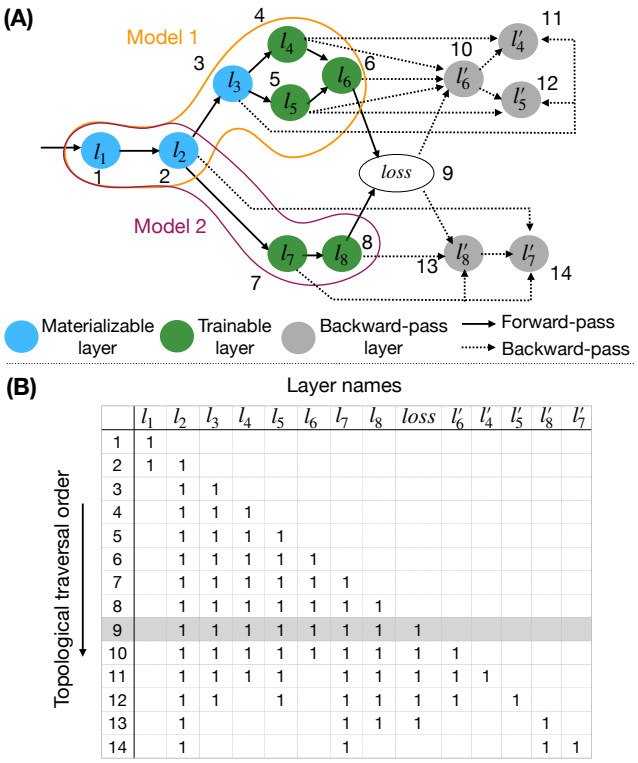

Figure 5: (A) Graph constructed by augmenting a multi-model graph with nodes to represent both the forward- and backward-passes of training. (B) Topological traversal-based live tensor analysis for the model shown in (A).

specific order by which layer operations are performed and also on how aggressively memory is allocated and deallocated by the DL system. Popular DL frameworks like TensorFlow and PyTorch execute operations in a topological order [55, 70].

We estimate the memory needed for storing layer output tensors by performing a topological traversal-based *live tensor analysis*. We augment the optimal reuse plan of the fused model by adding nodes needed to represent the backward-pass. We also add a node to represent the loss computation and add edges between every output layer and the new loss node. This loss node is responsible for calculating the loss for all trainable model branches using the corresponding optimizer. It also ensures that our fused model adheres to the two-phase (i.e., forward and backward) training template supported by DL systems. For every non-materializable layer l_i, we add a node l'_i to represent the backward-pass computation of that layer. We set $s_{mem}(l'_i)$ to be same as $s_{mem}(l_i)$. We treat all backward-pass computations uniformly and add edge dependencies as follows:

- Output from the forward-pass layer by adding (l_i, l'_i) edge.
- Input(s) to the forward-pass layer by adding $\{(l_p, l'_i) : \forall l_p \in parent(l_i)\}$ edges.
- Backward-pass output gradient(s) from the child layers by adding $\{(l'_s, l'_i) : \forall l_s \in child(l_i)\}$ edges.

Figure 5 (A) presents an optimal reuse plan model corresponding to a fused model composed of two source models. Notice that each model's trainable layers are in a separate branch. To estimate peak runtime memory for storing layer outputs, we perform a topological traversal over the created graph structure while keeping track of the live output tensors. The output tensor size of layer l for a single training record is given by $s_{mem}(l)$. The order by which each node is visited during the topological traversal is denoted in the figure. Figure 5 (B) presents the live tensor analysis. When processing a node, we assume all its inputs and output tensors are live. We release a tensor if it is not needed for the current or future nodes. For example, as highlighted in Figure 5 (B), when processing the loss node, the l_1 output tensor is not live as it is not used by any node that is yet to be visited. We estimate the peak memory usage by finding the maximum cumulative memory usage of all live tensors at any traversal step and multiplying it by the training batch size.

Notice that there might be more than one possible topological order and as a result, the order used by our analysis may differ from the order used by the training framework. However, for any topological traversal order, we can claim that the maximum number of live tensors is only one more than the number of live tensors needed when processing the loss node. This is because the loss node acts as a barrier needing the entire forward-pass to be completed before starting the backward-pass. Overall, we found that our approach can provide reasonable memory usage upper bounds that are sufficient for our model fusion optimization.

4.4 Theoretical Speedups

We characterize the attainable theoretical speedup assuming full redundancy avoidance. We ignore the cost of layer materialization, as it gets amortized over many training epochs, models, and model selection cycles. Thus, the speedup can be represented as the ratio between the total training cost of all model layers and the total training cost of only the non-materializable (i.e., $m(l) = False$) model layers, as per Equation 11.

$$\frac{\sum_{i=1}^{|Q|} \sum_{l \in L_i} c_{comp}(l) \cdot \text{epochs}(\phi_i)}{\sum_{i=1}^{|Q|} \sum_{l \in L_i} \mathbb{1}[m(l) = False] \cdot c_{comp}(l) \cdot \text{epochs}(\phi_i)} \tag{11}$$

5 EXPERIMENTAL EVALUATION

We now present an extensive empirical evaluation seeking to answer the following questions. (1) How does Nautilus compare with current practice and other baselines on runtimes, accuracy, and resource utilization? (2) How much Nautilus's optimizations contribute to the overall runtime reductions?

Datasets: We use two benchmark datasets: *CoNLL-2003* [71] and *Malaria* [56]. *CoNLL-2003* is a text dataset and the prediction task is named entity recognition. *Malaria* is an image dataset and the prediction task is identifying Malaria from blood cell images. For *CoNLL-2003* and *Malaria*, we have unlabeled data pools of sizes 10,000 and 8,000 records, respectively.

Workloads: We run 5 end-to-end workloads covering feature transfer, fine-tuning, and adapter training. Table 3 summarizes the transfer learning and the model selection configuration values of the workloads. Feature transfer workloads (*FTR-**) use BERT-base as the source model. *FTR-1* explores 6 feature transfer strategies that are same as the ones reported in [17]. *FTR-2* and *FTR-1* explore 4 and 1 transfer strategies, respectively. The fine-tuning workload

Table 3: Model selection configurations of workloads.

Workload	Tuning Parameters				# Models
	Transfer Learning Approach	Batch Size	Learning Rate	Epochs	
FTR-1	**Feature Transfer** from: {embedding, second last hidden, last hidden, sum last 4 hidden, concat last 4 hidden, sum all hidden}	{16, 32}	$\{5, 3, 2\} \times 10^{-5}$	{5}	36
FTR-2	**Feature Transfer** from: {second last hidden, last hidden, sum last 4 hidden, concat last 4 hidden}	{16, 32}	$\{5, 3, 2\} \times 10^{-5}$	{5}	24
FTR-3	**Feature Transfer** from: {concat last 4 hidden}	{16, 32}	$\{5, 3, 2\} \times 10^{-5}$	{5, 10}	12
ATR	**Adapter Training** for: {last hidden, last 2 hidden, last 3 hidden, last 4 hidden}	{16, 32}	$\{5, 3, 2\} \times 10^{-5}$	{5}	24
FTU	**Fine-tuning** : {last 3 hidden, last 6 hidden, last 9 hidden, last 12 hidden}	{16, 32}	$\{5, 3, 2\} \times 10^{-5}$	{5}	24

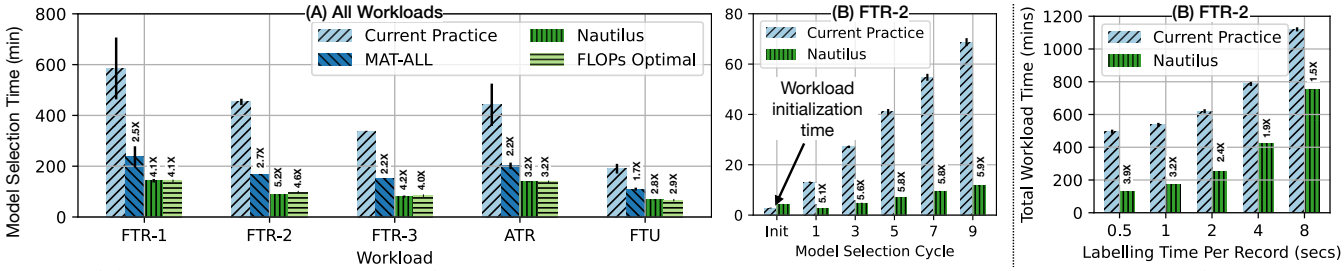

Figure 6: (A) Total model selection time. (B) Model selection time breakdown by model selection cycle for *FTR-2* (only the odd numbered model selection cycles are shown due to space constraints). (C) Total time for *FTR-2* including data labeling time.

FTU uses the popular computer vision model ResNet-50 [28] and we vary the number of fine-tuned residual layer blocks from the top. The adapter training workload *ATR* also uses the BERT-base model. We use Houlsby [29] type adapters and vary the number of layers with adapters from the top. For *FTR-** workloads we add a new transformer layer on top of the extracted features. For all workloads, we add a new Softmax classification layer on top of the last hidden layer. *FTR-** and *ATR* workloads use the *CoNLL-2003* dataset; *FTU* workload uses the *Malaria* dataset.

For all workloads, we generate the labeled dataset iteratively. For each cycle, we label 500 records with a 400/100 train/validation split, perform model selection on all labeled data up to that point, and repeat the process for 10 cycles. We simulate the human labeler by programmatically releasing the labels.

Experimental Setup: We use a machine with 32 GB RAM, Intel i7 3.40GHz CPU, 1TB SSD, and NVIDIA Titan X GPU with 12 GB memory. It runs Ubuntu 18.04 with TensorFlow version 2.4, CUDA version 11.0, and cuDNN version 7.5. For our optimizer, we set the disk read throughput to 500 MB/s and the compute throughput to 6 TFLOP/s, which is 50% of the theoretical FLOPS rate of the Titan X GPU. These hardware settings are configurable by the user. We report average of 3 runtimes with 95% confidence intervals.

5.1 End-to-End Runtimes

Model Selection Time: We first evaluate the total model selection time for 4 different approaches. This enables us to isolate Nautilus's ability to reduce DTL model selection runtimes, which is independent of the data labeling approach and the labeling time. The four approaches that we evaluate are *Current Practice*, *MAT-ALL*, Nautilus, and *FLOPs Optimal*. *Current Practice* is the naive baseline, which trains unmodified models independently and repeats the process for all cycles. It incurs the highest level of redundancies. *MAT-ALL* is a strong baseline that materializes all materializable layers and uses them during training, irrespective of whether it is efficient to compute them rather than loading them. Note that *MAT-ALL* needs parts of our code from Nautilus. Nautilus is our optimizer-picked plan, which performs both our materialization and model fusion optimizations. We execute it with a disk storage budget (B_{disk}) of 25 GBs and a runtime memory budget (B_{mem}) of 10 GBs. *FLOPs Optimal* is calculated by dividing the *Current Practice* time by the theoretical speedup. Figure 6 presents the results.

Nautilus offers significant speedups with the highest speedup of 5.2X seen on *FTR-2*. The highest speedup for *MAT-ALL* is also for *FTR-2*, which is 2.7X. In all cases, Nautilus outperforms the *MAT-ALL* baseline. In the worst case, *MAT-ALL* is 48% slower than Nautilus for the *FTR-2* workload. *MAT-ALL* also consumes more disk storage. The highest storage blowup is for the *FTR-1* workload, which is 12X more than Nautilus (21.6 GB vs. 1.8 GB). This storage blowup may cause *MAT-ALL* not viable for some applications [22].

In all cases, Nautilus achieves slightly better or competitive runtimes to the *FLOPs Optimal* runtime. This is because Nautilus significantly amortizes the training and I/O overheads, which are not accounted for in the FLOPS reduction-based theoretical speedup calculation. Also, Nautilus's speedups vary based on the characteristics of the workload. For example, speedups are generally higher for *FTR-** workloads compared to *ATR* or *FTU*, as the latter workloads have more trainable layers. Also, absolute runtimes are lower for the *FTU* compared to other workloads, as the former uses a less compute-intensive model. Overall, Nautilus reduces DTL model selection runtimes substantially for all workloads.

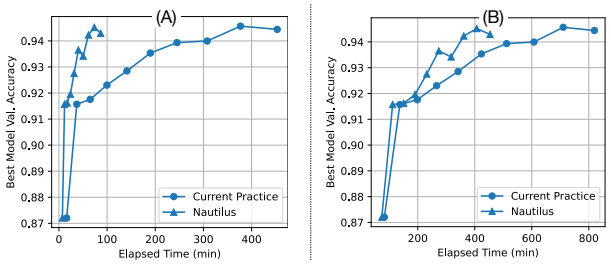

Figure 7: *FTR-2* learning curves with (A) zero and (B) 4 seconds/label data labeling cost values.

Model Selection Time Breakdown: Figure 6 (B) presents the model selection time breakdown by model selection cycle. *Current Practice* and Nautilus take 2.7 and 4.4 minutes to initialize the workload, respectively. By drilling into the workload initialization time, we found that Nautilus spends 63% of time creating the original model checkpoints, which is also performed by the *Current Practice*. Additionally, Nautilus spends 12% time profiling the original models, 3% time generating the optimized plan, and 21% time generating model checkpoints for the optimized plan. Nautilus's speedups are slightly lower in the early cycles compared to the later ones. This is because the later cycles have more training data, and the effect of fixed overheads is less pronounced in them.

Total Workload Time: Finally, we evaluate the total workload time for the *FTR-2* workload for different data labeling runtime values. The *CoNLL-2003* dataset used for the *FTR-2* workload has 20 words per record on average. Hence, we vary the labeling time per data record between 0.5 seconds and 8 seconds. The 0.5 seconds case can be considered as a multi-labeler scenario (e.g., cloud labelers); 8 seconds scenario can be considered as a single-labeler scenario. For the 0.5 seconds scenario, Nautilus achieves a speedup of 3.9X compared to the *Current Practice*. For the 8 seconds scenario, Nautilus's speedup reduces to 1.5X, as higher data labeling time dominates the overall workload time.

5.2 Accuracy

Both *Current Practice* and Nautilus perform logically equivalent SGD training. Thus, they both should achieve the same statistical efficiency. To validate this, we plot the best model's validation accuracy against the elapsed model selection time for the *FTR-2* workload. Figure 7 (A) presents the results. We see that both approaches achieve very similar validation accuracies after every model selection cycle. However, Nautilus achieves them 5X faster. We repeat the experiment with a data labeling time of 4 seconds/label and plot the best validation accuracy against elapsed total time as shown in Figure 7 (B). In this case, Nautilus is 2X faster.

5.3 Drill-Down Analysis

Contribution of Our Optimizations: We run the end-to-end workloads using Nautilus but disable either the materialization (*MAT OPT*) or the model fusion (*FUSE OPT*) optimization. Figure 8 presents the results. For all cases except *ATR*, running Nautilus w/o *FUSE OPT* causes more slowdown than running w/o *MAT OPT*. The highest slowdown for Nautilus w/o *FUSE OPT* is for *FTR-1*, which is 54.7%; for w/o *MAT OPT*, it is for *FTR-3*, which is 31.2%. For *FTU*, Nautilus's runtime does not change w/o *MAT OPT*. This

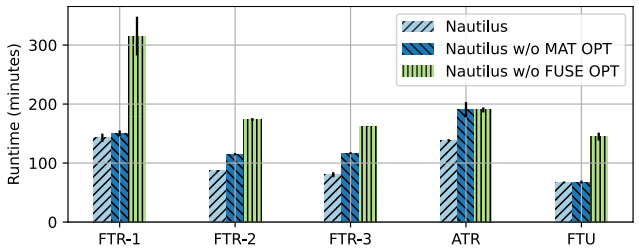

Figure 8: Model selection time with and without *MAT* and *FUSE* optimizations.

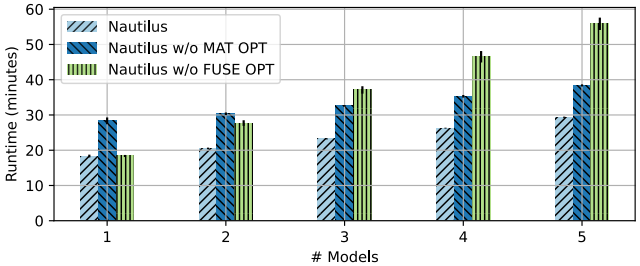

Figure 9: Model selection time for different number of models with and without *MAT* and *FUSE* optimizations.

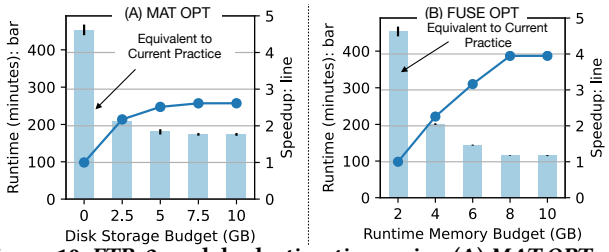

Figure 10: *FTR-2* model selection time using (A) *MAT OPT* vs. storage budget and (B) *FUSE OPT* vs. memory budget.

is because ResNet-50 is a less compute-intensive model and Nautilus computes all materializable layers instead of loading them.

We also run an experiment where we vary the number of models in the model selection workload. For this, we use *FTR-2* and fix the feature transfer strategy to the concatenation of the last four layers, fix the batch size to 16, and vary the number of explored learning rates. Figure 9 presents the results. When the number of models is less than or equal to 2, running Nautilus w/o *MAT OPT* causes more slowdown than running it w/o *FUSE OPT*. However, when the number of models increases, running w/o *FUSE OPT* causes more slowdown. With more models, *FUSE OPT* has more opportunities to avoid redundant computations and amortize training and I/O overheads. Also, with only 1 model, *FUSE OPT* doesn't give any benefits as there are no opportunities for model fusion.

Finally, we run *FTR-2* by only using *MAT OPT* or *FUSE OPT*, and vary the disk storage budget B_{disk} and runtime memory budget B_{mem}, respectively. Running *MAT OPT* with a B_{disk} of 0 GBs is equivalent to the *Current Practice*. Running *FUSE OPT* with a B_{mem} of 2 GBs is also equivalent to the *Current Practice* as it does not fuse any models. For *FUSE OPT* we also ensure that the training process does not consume more than the allocated B_{mem} from the available GPU memory. Thus, this experiment also validates the correctness of our runtime memory estimation and its ability to avoid workload crashes due to out-of-memory errors. As B_{disk} is increased, *MAT*

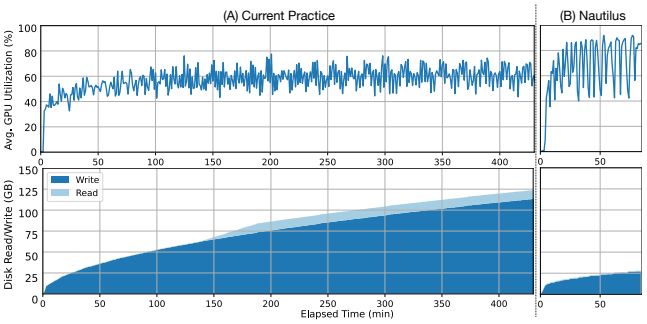

Figure 11: Average GPU utilization and cumulative disk reads and writes for executing the *FTR-2*.

OPT runtime decreases and plateaus after 7.5 GBs where it achieves 2.6X speedup compared to the *Current Practice*. As the B_{mem} is increased, *FUSE OPT* runtime also decreases and plateaus after 8 GBs where it achieves a 4.0X speedup. Nautilus combines the benefits of both optimizations and achieves the lowest runtimes.

System Resources Utilization: We evaluate the GPU utilization and cumulative disk reads/writes for executing the *FTR-2*. Figure 11 presents the results. Nautilus yields a higher average GPU utilization of 66% compared to the 57% of *Current Practice*. It also performs 4.3X fewer disk writes and 11.8X fewer disk reads. This is because *Current Practice* checkpoints the entire original model after every model training, which is around 400-500MBs. But most of the parameters in them are frozen parameters and do not need repeated checkpointing. In contrast, Nautilus checkpoints modified model graphs with most frozen parameters pruned. Writing less amount of data also helps with better page caching for the reads.

6 RELATED WORK

Materialization Optimizations: Our work is inspired by the long line of work on reusing intermediates to optimize ML workloads [42, 46, 54, 69, 72, 74, 76], but ours is the first to apply it to optimize DTL over evolving training data. Prior work in Vista system [46] also uses feature materialization to optimize DL feature transfer-based multi-modal analytics. However, it supports only linear DL model graphs and features extracted from only one layer at a time. Also, Vista's focus is on training classical ML models (e.g., linear regression) on the extracted features from a fixed dataset and not DTL over evolving data. Nautilus generalizes Vista in 3 dimensions. It supports 1) DAG structured DL model graphs with arbitrary feature compositions, 2) evolving labeled datasets, and 3) all 3 popular transfer learning paradigms. Intermediate feature materialization is also used in AutoFreeze [42] to optimize the fine-tuning of a single BERT [17] model. Nautilus supports arbitrary DL models, all popular transfer learning paradigms, and also model selection.

Nautilus's cost model-based materialization optimization extends the optimal reuse plan formulation in Helix system [74]. Specifically, we represent models in a DTL model selection workload using an optimizable graph structure called multi-model graph and jointly solve the materialized intermediate output selection and the reuse plan generation in a single MILP formulation.

Joint Model Training Optimizations: Nautilus's model fusion is a form of common sub-expression elimination (CSE). CSE is also used in several other systems to eliminate redundant data pre-processing steps [24, 41, 48, 77]. Nautilus extends this to also eliminate redundancies in materializable layers. However, existing systems require the user to select the set of models to fuse [48, 77], adopt a trial-and-error approach [41] to find the set of models, or trains each source model in a separate GPU [24]. Nautilus uses profiling information to estimate fused model memory footprint and automatically picks an optimal set of models to fuse.

DL Model Selection: Several systems have been proposed to optimize DL model selection [2, 20, 39, 47, 68]. However, the focus of all these systems is on utilizing the parallelism available in a cluster to scale the DL model selection. In contrast, the focus of Nautilus is on DL model selection in low-resource settings such as workstations or PCs. Also, Nautilus focuses on the human-in-the-loop setting with iteratively generated labeled data.

Nautilus supports two popular model selection procedures: grid and random search, which cover an overwhelming majority of model selection applications [10]. However, there are other more complex model selection procedures proposed in the literature [8, 31, 37, 38]. We leave adding support for them to future work.

Other DL System Optimizations: Various other techniques can be also used to optimize DTL. They include operator fusion [3, 11], hybrid parallel execution [33], layer batching [41, 48, 77], model compression [27], and model distilling [63]. They are complementary to the optimizations performed by Nautilus since they mainly optimize lower-level operator execution. There also exist systems that support training larger than GPU memory models [12, 35, 59]. They are complementary to our work and can be combined with Nautilus to train or fuse larger models.

Nautilus's runtime memory estimation operates at a higher level that is independent of the exact memory allocation/deallocation behavior of the underlying DL system. However, one can also try to mimic the exact behavior and obtain improved memory usage estimates as in [19]. Nevertheless, our approach provides reasonable upper-bounds sufficient for our requirement.

7 CONCLUSIONS AND FUTURE WORK

Deep transfer learning (DTL) is a crucial paradigm for democratizing deep learning. Yet, the current practice of executing DTL workloads faces significant usability and resource inefficiency issues. In this work, we formalize the DTL workload from a data management standpoint and enable two multi-query optimization-inspired optimizations: materialization optimization and model fusion optimization. We implement our optimizations in a data system we call Nautilus. Nautilus reduces DTL model selection runtimes even up to 80% and significantly improves usability and resource usage. As for future work, we plan to expand the model adaptation schemes supported in Nautilus by adding support for more complex model selection and layer freezing schemes.

REFERENCES

[1] [n.d.]. Huggingface Pretrained Models. https://huggingface.co/transformers/pretrained_models.html

[2] [n.d.]. Scaling Out Search with Apache Spark. http://hyperopt.github.io/hyperopt/scaleout/spark/

[3] [n.d.]. XLA: Optimizing Compiler for Machine Learning : TensorFlow. https://www.tensorflow.org/xla

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[5] Dario Amodei and Danny Hernandez. Accessed January 31, 2021. AI and Compute. https://openai.com/blog/ai-and-compute.

[6] AWS. [n.d.]. Automate Data Labeling. https://docs.aws.amazon.com/sagemaker/latest/dg/sms-automated-labeling.html

[7] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *FAccT '21: 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event / Toronto, Canada, March 3-10, 2021*, Madeleine Clare Elish, William Isaac, and Richard S. Zemel (Eds.). ACM, 610–623. https://doi.org/10.1145/3442188.3445922

[8] James Bergstra, Daniel Yamins, and David D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013 (JMLR Workshop and Conference Proceedings, Vol. 28)*. JMLR.org, 115–123. http://proceedings.mlr.press/v28/bergstra13.html

[9] Benjamin Biering. [n.d.]. Getting Started with AI: How Much Data Do You Need? https://2021.ai/getting-started-ai-how-much-data-needed/

[10] Xavier Bouthillier and Gaël Varoquaux. 2020. *Survey of Machine-Learning Experimental Methods at NeurIPS2019 and ICLR2020.* Ph.D. Dissertation. Inria Saclay Ile de France.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[13] Rada Chirkova and Jun Yang. 2012. Materialized Views. *Found. Trends Databases* 4, 4 (2012), 295–405. https://doi.org/10.1561/1900000020

[14] FranÃğois Chollet. [n.d.]. Transfer Learning & Fine-tuning. https://keras.io/guides/transfer_learning/

[15] Alexandra Chronopoulou, Christos Baziotis, and Alexandros Potamianos. 2019. An Embarrassingly Simple Approach for Transfer Learning from Pretrained Language Models. *arXiv preprint arXiv:1902.10547* (2019).

[16] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. 2001. Pipelining in Multi-Query Optimization. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, Peter Buneman (Ed.). ACM. https://doi.org/10.1145/375551.375561

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[18] Bjarke Felbo, Alan Mislove, Anders Søgaard, Iyad Rahwan, and Sune Lehmann. 2017. Using Millions of Emoji Occurrences to Learn Any-domain Representations for Detecting Sentiment, Emotion and Sarcasm. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Copenhagen, Denmark, 1615–1625. https://doi.org/10.18653/v1/D17-1169

[19] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating GPU Memory Consumption of Deep Learning Models. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1342–1352. https://doi.org/10.1145/3368089.3417050

[20] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 1487–1495. https://doi.org/10.1145/3097983.3098043

[21] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org/

[22] Google-Research. [n.d.]. Huge Embedding Output File Issue #91. https://github.com/google-research/bert/issues/91

[23] Roger Grosse. Accessed January 31, 2021. CSC321 Lecture 6: Backpropagation. http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/slides/lec6.pdf.

[24] Hui Guan, Laxmikant Kishor Mokadam, Xipeng Shen, Seung-Hwan Lim, and Robert M. Patton. 2020. FLEET: Flexible Efficient Ensemble Training for Heterogeneous Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/304.pdf

[25] Himanshu Gupta and Inderpal Singh Mumick. 2005. Selection of Views to Materialize in a Data Warehouse. *IEEE Trans. Knowl. Data Eng.* 17, 1 (2005), 24–43. https://doi.org/10.1109/TKDE.2005.16

[26] Gurobi. Accessed January 31, 2021. Gurobi Optimization. https://www.gurobi.com.

[27] Song Han, Huizi Mao, and William J Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149* (2015).

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[29] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-Efficient Transfer Learning for NLP. In *International Conference on Machine Learning*. 2790–2799.

[30] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. *arXiv preprint arXiv:1801.06146* (2018).

[31] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Population Based Training of Neural Networks. *CoRR* abs/1711.09846 (2017). arXiv:1711.09846 http://arxiv.org/abs/1711.09846

[32] Ganesh Jawahar, Benoît Sagot, and Djamé Seddah. 2019. What Does BERT Learn about the Structure of Language?. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 3651–3657. https://doi.org/10.18653/v1/P19-1356

[33] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, Ameet Talwalkar, Virginia Smith, and Matei Zaharia (Eds.). mlsys.org. https://proceedings.mlsys.org/book/265.pdf

[34] Jon Kleinberg and Eva Tardos. 2006. *Algorithm Design.* Pearson Education India.

[35] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. 2018. TFLMS: Large Model Support in TensorFlow by Graph Rewriting. *CoRR* abs/1807.02037 (2018). arXiv:1807.02037 http://arxiv.org/abs/1807.02037

[36] Jaejun Lee, Raphael Tang, and Jimmy Lin. 2019. What Would Elsa Do? Freezing Layers During Transformer Fine-Tuning. *arXiv preprint arXiv:1911.03090* (2019).

[37] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52. http://jmlr.org/papers/v18/16-558.html

[38] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A System for Massively Parallel Hyperparameter Tuning. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/303.pdf

[39] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *CoRR* abs/1807.05118 (2018). arXiv:1807.05118 http://arxiv.org/abs/1807.05118

[40] Nelson F Liu, Matt Gardner, Yonatan Belinkov, Matthew E Peters, and Noah A Smith. 2019. Linguistic Knowledge and Transferability of Contextual Representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 1073–1094.

[41] Rui Liu, Sanjan Krishnan, Aaron J Elmore, and Michael J Franklin. 2020. Understanding and Optimizing Packed Neural Network Training for Hyper-Parameter Tuning. *arXiv preprint arXiv:2002.02885* (2020).

[42] Yuhan Liu, Saurabh Agarwal, and Shivaram Venkataraman. 2021. AutoFreeze: Automatically Freezing Model Blocks to Accelerate Fine-tuning. *CoRR* abs/2102.01386 (2021). arXiv:2102.01386 https://arxiv.org/abs/2102.01386

[43] Michael McCloskey and Neal J Cohen. 1989. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. In *Psychology of Learning and Motivation*. Vol. 24. Elsevier, 109–165.

[44] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. *Proceedings of the VLDB Endowment* (2021).

[45] Robert Monarch. 2019. *Human-in-the-Loop Machine Learning. Active Learning and Annotation for Human-centered AI.* Manning Publications. https://www.manning.com/books/human-in-the-loop-machine-learning

[46] Supun Nakandala and Arun Kumar. 2020. Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1685–1700. https://doi.org/10.1145/3318464.3389709

[47] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.* 13, 12 (July 2020), 2159–2173. https://doi.org/10.14778/3407790.3407816

[48] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. 2018. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning*. 20.

[49] Kevin P. Nguyen, Cherise Chin Fatt, Alex Treacher, Cooper Mellema, Madhukar H. Trivedi, and Albert Montillo. 2020. Anatomically Informed Data Augmentation for Functional MRI with Applications to Deep Learning. In *Medical Imaging 2020: Image Processing, Houston, TX, USA, February 15-20, 2020 (SPIE Proceedings, Vol. 11313)*, Ivana Isgum and Bennett A. Landman (Eds.). SPIE, 113130T. https://doi.org/10.1117/12.2548630

[50] Luis Perez and Jason Wang. 2017. The Effectiveness of Data Augmentation in Image Classification using Deep Learning. *CoRR* abs/1712.04621 (2017). arXiv:1712.04621 http://arxiv.org/abs/1712.04621

[51] Matthew E Peters, Sebastian Ruder, and Noah A Smith. 2019. To Tune or Not to Tune? Adapting Pretrained Representations to Diverse Tasks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*. 7–14.

[52] Aria Pezeshk, Nicholas Petrick, Weijie Chen, and Berkman Sahiner. 2017. Seamless Lesion Insertion for Data Augmentation in CAD Training. *IEEE Trans. Medical Imaging* 36, 4 (2017), 1005–1015. https://doi.org/10.1109/TMI.2016.2640180

[53] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulic, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020. AdapterHub: A Framework for Adapting Transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP 2020 - Demos, Online, November 16-20, 2020*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, 46–54. https://doi.org/10.18653/v1/2020.emnlp-demos.7

[54] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. SIGMOD.

[55] PyTorch. Accessed January 31, 2021. The Topological Sorting Algorithm for Computation Graphs in PyTorch. https://github.com/pytorch/pytorch/blob/v1.2.0/caffe2/core/nomnigraph/include/nomnigraph/Graph/TopoSort.h#L26.

[56] Sivaramakrishnan Rajaraman, Sameer K Antani, Mahdieh Poostchi, Kamolrat Silamut, Md A Hossain, Richard J Maude, Stefan Jaeger, and George R Thoma. 2018. Pre-trained Convolutional Neural Networks as Feature Extractors Toward Improved Malaria Parasite Detection in Thin Blood Smear Images. *PeerJ* 6 (2018), e4568.

[57] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason Alan Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proc. VLDB Endow.* 11, 3 (2017), 269–282. https://doi.org/10.14778/3157794.3157797

[58] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. 2017. Learning Multiple Visual Domains with Residual Adapters. In *Advances in Neural Information Processing Systems*. 506–516.

[59] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. *CoRR* abs/2101.06840 (2021). arXiv:2101.06840 https://arxiv.org/abs/2101.06840

[60] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2021. A Primer in BERTology: What We Know About How BERT Works. *Transactions of the Association for Computational Linguistics* 8 (2021), 842–866.

[61] Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. 2020. AdapterDrop: On the Efficiency of Adapters in Transformers. *arXiv preprint arXiv:2010.11918* (2020).

[62] Sebastian Ruder, Matthew E. Peters, Swabha Swayamdipta, and Thomas Wolf. 2019. Transfer Learning in Natural Language Processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*. Association for Computational Linguistics, Minneapolis, Minnesota, 15–18. https://doi.org/10.18653/v1/N19-5004

[63] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv preprint arXiv:1910.01108* (2019).

[64] Timos K. Sellis. 1988. Multiple-query Optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52. https://doi.org/10.1145/42201.42203

[65] Amazon Web Services. Accessed January 31, 2021. SageMaker Ground Truth. https://aws.amazon.com/sagemaker/groundtruth/.

[66] Burr Settles. 2009. Active Learning Literature Survey. (2009).

[67] Connor Shorten and Taghi M. Khoshgoftaar. 2019. A Survey on Image Data Augmentation for Deep Learning. *J. Big Data* 6 (2019), 60. https://doi.org/10.1186/s40537-019-0197-0

[68] Scott Sievert, Tom Augspurger, Matthew Rocklin, Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe. 2019. Better and Faster Hyperparameter Optimization with Dask. In *Proceedings of the 18th Python in Science Conference*. 118–125.

[69] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 535–546. https://doi.org/10.1109/ICDE.2017.109

[70] TensorFlow. Accessed January 31, 2021. The Topological Sorting Algorithm for Computation Graphs in TensorFlow. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/grappler/utils/topological_sort.h.

[71] Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*. 142–147. https://www.aclweb.org/anthology/W03-0419

[72] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1285–1300. https://doi.org/10.1145/3183713.3196934

[73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[74] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 446–460. https://doi.org/10.14778/3297753.3297763

[75] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *European conference on computer vision*. Springer, 818–833.

[76] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization Optimizations for Feature Selection Workloads. *ACM Trans. Database Syst.* 41, 1 (2016), 2:1–2:32. https://doi.org/10.1145/2877204

[77] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. 2020. Retiarii: A Deep Learning Exploratory-Training Framework. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 919–936.