SHiFT: An Efficient, Flexible Search Engine for Transfer Learning

Cedric Renggli
ETH Zurich
cedric.renggli@inf.ethz.ch

Xiaozhe Yao
ETH Zurich
xiaozhe.yao@inf.ethz.ch

Luka Kolar
ETH Zurich
luka.kolar@inf.ethz.ch

Luka Rimanic[*]
ETH Zurich
luka.rimanic@inf.ethz.ch

Ana Klimovic
ETH Zurich
aklimovic@inf.ethz.ch

Ce Zhang
ETH Zurich
ce.zhang@inf.ethz.ch

ABSTRACT

Transfer learning can be seen as a data- and compute-efficient alternative to training models from scratch. The emergence of rich model repositories, such as TensorFlow Hub, enables practitioners and researchers to unleash the potential of these models across a wide range of downstream tasks. As these repositories keep growing exponentially, efficiently selecting a good model for the task at hand becomes paramount. However, a single generic search strategy (e.g., taking the model with the highest linear classifier accuracy) does not lead to optimal model selection for diverse downstream tasks. In fact, using hybrid or mixed strategies can often be beneficial. Therefore, we propose SHiFT, the first downstream task-aware, flexible, and efficient model search engine for transfer learning. Users interface with SHiFT using the SHiFT-QL query language, which gives users the flexibility to customize their search criteria. We optimize SHiFT-QL queries using a cost-based decision maker and evaluate them on a wide rang of tasks. Motivated by the iterative nature of machine learning development, we further support efficient incremental executions of our queries, which requires a special implementation when jointly used with our optimizations.

PVLDB Reference Format:
Cedric Renggli, Xiaozhe Yao, Luka Kolar, Luka Rimanic, Ana Klimovic, and Ce Zhang. SHiFT: An Efficient, Flexible Search Engine for Transfer Learning. PVLDB, 16(2): 304 - 316, 2022.
doi:10.14778/3565816.3565831

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DS3Lab/shift.

1 INTRODUCTION

Transfer learning [26, 47] is an emerging paradigm of building machine learning (ML) applications, which can have a profound impact on the architecture of today's machine learning systems and platforms. In a nutshell, transfer learning aims at training ML models with high quality without having to collect enormous datasets or spend a fortune on training from scratch. Instead, models are first *pre-trained* on typically large and possibly private *upstream* datasets, and are then made available via model repositories such as TF-Hub, PyTorch Hub, and HuggingFace. Given a new *downstream* dataset, representative for the ML task, a user picks *some* of these pre-trained models to *fine-tune*, which typically requires adding randomly initialized layers to parts of the pre-trained deep network, and tuning all the parameters using the limited amount of downstream data. Transfer learning has been successfully applied to many domains and tasks [3, 11, 24, 39].

Table 1: Example search queries supported by SHiFT, returning an ordering of registered models by their...

Q1: best reported upstream accuracy [14]
Q2: best nearest neighbor classifier accuracy [25, 34]
Q3: best linear classifier accuracy [5, 29, 43]
Q4: Q1 and Q3 (on models excluding the result of Q1) [37]
Q5: highest (fine-tune) accuracy on the most similar task [1]

This process, despite being cheaper compared to training from scratch (i.e., with a fully randomly initialized network), still requires all parameters to be updated several times, which can be computationally demanding. With the growing number of pre-trained models available in online platforms like TF-Hub, PyTorch Hub, and HuggingFace, it is computationally infeasible to fine-tune *all* models to find the one that performs best for a downstream task. As a result, a key defining component of a transfer learning application is a *model search strategy*, which provides cheaper ways to identify promising models to use. One challenge is that *different tasks might require very different search strategies* [26, 53] and Table 1 illustrates several popular ones. Today, a user of these model repositories conducting transfer learning needs to manage models and implement search strategies all by themselves. Over the years, in the context of building usable machine learning system (e.g., Ease.ML [2, 13, 23]), we have observed several challenges that our users face:

Challenge 1. Our first observation is that the amount of data and computation a user needs to perform quickly exceeds what most users, even competent software engineers, can handle. Some challenges are more on the engineering side — today's pre-trained models are scattered in different repositories, including HuggingFace, TF-Hub, and PyTorch Hub, lacking a unified abstraction. However, many others are technical — HuggingFace, TF-Hub, and PyTorch Hub contain 34K, 1K, and 48 models, respectively, and sum up to more than 100GB in size. Simply downloading all these models can take hours, not to mention running inference of all these models over a given dataset and implementing state-of-the-art search strategies. As a consequence, many users in our experience simply

Proceedings of the VLDB Endowment, Vol. 16, No. 2 ISSN 2150-8097.
doi:10.14778/3565816.3565831

[*] Now at Google.

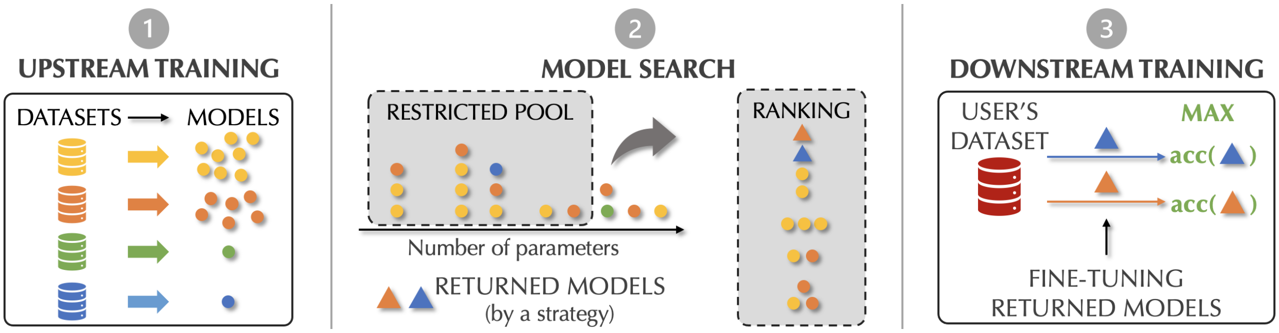


Figure 1: Three stages of a transfer learning setup: (1) Models are trained *upstream* for various architecture and datasets, (2) the models are restricted into a *pool*, and ranked by a search strategy, and (3) a subset of the models based on the ranking are fine-tuned on user's dataset in a downstream process. SHiFT focuses on supporting the middle stage (2) efficiently and in a flexible, generic way.

resort to only using the latest model — *this ignores the vast diversity of available models and, as we show in our previous systematic benchmark [37], can lead to a significant quality gap.*

Challenge 2. The second challenge we observe is that today's data-centric development pattern of ML applications provides unique opportunities to speed up the model search process, which, if left to the users, are quite hard to capture. In many cases, users will conduct iterations on the datasets—data cleaning, acquisition of both labels and features—and execute similar model search queries for each of these datasets over time. Since these datasets are similar to each other, it is possible to save a significant amount of computation if we carefully design incremental maintenance strategies for different search queries — *all these opportunities to speed up model search are not captured by today's model repositories.*

Challenge 3. The third challenge we observe is that model search is an active research area where new strategies are coming out quite frequently, yet easily implementing them in a system and comparing them to existing search strategies is a painful task. Just in the last three years, researchers proposed at least eight new strategies [1, 5, 9, 14, 25, 29, 34, 37]. Having users to catch up with these latest developments can be tedious and potentially a huge waste of resources. *We are in a dire need for a unified framework that can be extended in a flexible way to support, benchmark, and automatically optimize newly proposed search strategies.*

SHiFT: Towards Data Management for Transfer Learning. All three challenges, in our opinion, lead to the same hypothesis:

> *We should shift the responsibility of querying and conducting model search over today's model repositories from individual users to a data management system, which can (1) provide a unified abstraction to connect all major model repositories, (2) provide a flexible, extendable way of specifying search strategies, (3) automatically optimize its execution, and (4) support efficient incremental execution.*

In this paper, we present SHiFT, one of the first data management systems for transfer learning — SHiFT is definitely *not* the first "model management system", an area that has attracted a lot of interest from the data management community [18, 19, 31, 40, 44, 51]; however, to our best knowledge, it is the first one that focuses on enabling an efficient and flexible model search functionality for transfer learning. This is achieved by supporting diverse model search queries to find (near-)optimal pre-trained models for fine-tuning them on a downstream task. Our technical contributions are as follows:

- **C1 System Design.** We propose a unified abstraction in SHiFT that can model all existing model search strategies that we are aware of, and is extendable with respect to new models, model repositories, as well as potential new search strategies. We then present SHiFT-QL, a novel query language used to interract with SHiFT. To compare different search strategies, we furthermore design an easy-to-use benchmark module.
- **C2 System Optimization.** We carefully studied several system optimization opportunities, both for a single query run and for incremental maintenance. **(C2.1)** For a single query run, we propose several optimizations to speed up the model search process, including a novel strategy based on successive halving [12] to use resources more efficiently. **(C2.2)** Furthermore, we develop a cost model to *automatically* decide whether to apply different system optimizations, given a search query, as well as systems and model properties. **(C2.3)** SHiFT also efficiently supports incremental query execution, as required by the iterative nature of the model development process. This functionality is non-trivial, and we carefully design an incremental maintenance strategy for successive-halving.
- **C3 Evaluation.** We evaluate SHiFT over a diverse set of queries across three computer vision and two NLP datasets and more than 150 diverse models. We show that SHiFT outperforms a baseline implementation of the same search strategy by up to 1.57×, and in the incremental scenario up to 4.0× for 10% feature changes for reasonably large test sets. Compared with fine-tuning all models without search strategy, SHiFT can be up to 22.6×-45.0× faster on vision tasks. We furthermore provide an extensive study using our benchmark module.
- **C4 Open Source.** We make SHiFT publicly available by open-sourcing it under https://github.com/DS3Lab/shift. We provide a clean documentation on how to implement new search strategies as well as guidelines and best-practices for choosing the right SHiFT-QL query under: http://docs.shift.ml

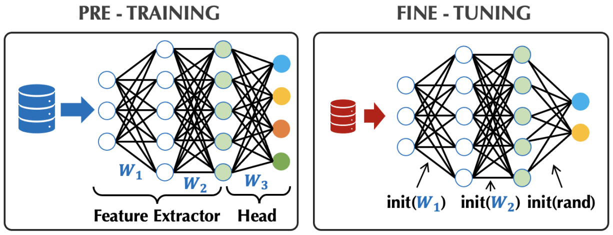


Figure 2: The difference between pre-training (left) and fine-tuning (right). The features stem from the last layer.

2 BACKGROUND AND MOTIVATION

2.1 Transfer Learning

Transfer learning has been applied very successfully on popular deep learning modalities such as computer vision (CV) and natural language process (NLP) over the last few years [32, 42, 46, 47]. Transfer learning is typically divided into three steps as illustrated in Figure 1: (1) An upstream, or pre-training part, where a machine learning model is trained using a well-established approach (e.g., randomly initialized weights and using mini-batch stochastic gradient descent as an optimizer) and an upstream dataset. (2) Having access to multiple such models trained on various datasets or model architectures, users pick a single or a subset of the available models, through some model search strategy, for the subsequent part. Note that users typically do not have access to the upstream datasets for this model search part. (3) The chosen models are fine-tuned in a downstream part on users' datasets. There are multiple strategies on how to fine-tune a pre-trained model, the most prominent being illustrated in Figure 2. Transfer learning and searching for models is mainly studied for classification tasks. We focus on this setting, hoping that our findings and proposed system will enable progress beyond classification. The pre-trained model is split into two parts: a *feature extractor*, typically the entire network until the last layer, and the classification *head* consisting of the final linear classification layer. The new model is then a copy of the feature extractor (i.e., weights and architecture), and a new randomly initialized head. Replacing the pre-trained linear head by a new one is usually unavoidable, as the number of classes often changes from the upstream task to the downstream one. All the weights of the network are then refined for multiple iterations using an iterative optimization algorithm like stochastic gradient descent (SGD).

The benefits of this three-step transfer learning process over training from scratch (i.e., training a randomly initialized network using the downstream dataset only) are twofold: (A) Using transfer learning, one can fine-tune a very large and complex network (i.e., many parameters forming a highly non-convex space to optimize in) even for limited amount of downstream data. (B) The fine-tuning process requires a much smaller number of steps in the iterative optimization process. Both (A) and (B) ensure a good initial condition to start the fine-tune optimization process. They are enabled by the fact that much larger, sometimes private upstream datasets are used to train for many iterations. The choice of which pre-trained model to fine-tune has a high impact on the final accuracy one can expect (e.g., a sub-optimal pick in our experiments can lead to a 43% downstream accuracy gap). If we are not concerned with optimization compute and power, we could fine-tune all models and pick the best one afterwards. This brute-force approach is usually infeasible in practice given the amount of currently available pre-trained models, along with the fact that this will not scale to more models in the future. Therefore, we require a more efficient search strategy, which limits the number of models we need to fine-tune.

2.2 Existing Pre-Trained Model Hubs

There exist multiple prominent online pre-trained model repositories, most notably Tensorflow Hub, PyTorch Hub and HuggingFace.[1] Each repository has its own interface for accessing the fully trained model, using the corresponding deep learning framework, with optional direct access to the (last-layer) feature extractor in the case of TF-Hub. HuggingFace has an interface for both Tensorflow and PyTorch. These existing online repositories enable easy access to pre-trained models along with their weights and some additional meta-data such as the domain and tasks this model is designed for (e.g., vision and classification), number of parameters, name of the dataset used to pre-train, or performance on this upstream dataset.

Limitations. Whilst accessing the models via existing online repositories is simple, they all share a common limitation when searching for the right model. That is, all platforms offer only plain search fields or filters[2], which retrieve models by their name or some meta-data properties. As we will see in the next section, this allows to restrict the model pool and run (some) task-agnostic search queries. The implementation and support of any other search strategy is currently left to the user.

2.3 Model Search Strategies

A search query is a function $m(\mathcal{M}, \mathcal{D}, B)$ with a budget B, a set of models $\mathcal{M}$, and a downstream task represented by a dataset $\mathcal{D}$ as its input. The function outputs a set $\mathcal{S}_m \subset \mathcal{M}$ with $|\mathcal{S}_m| \leq B$. The input set $\mathcal{M}$ either represents all models registered in `SHiFT`, or a restricted subset, which we call a *model pool*. The quality of a query is not measured by the metric that the query itself uses (e.g., proxy accuracy or task similarity), but rather by the maximal accuracy attained when fine-tuning the resulting models, those in $\mathcal{S}_m$. The difference between this accuracy and the maximal achievable accuracy if one would fine-tune all available models to the user is called *regret*, formally:

$$\max_{m_i \in \mathcal{M}} \mathbf{E}[t(m_i, \mathcal{D})] - \mathbf{E}\left[\max_{s_i \in \mathcal{S}_m} t(s_i, \mathcal{D})\right], \tag{1}$$

where $t(m, \mathcal{D})$ is the test accuracy achieved when fine-tuning model m on dataset $\mathcal{D}$. Clearly, the budget B, which restricts the size of the subset $\mathcal{S}_m$, has a direct impact on the regret. A large and diverse set $\mathcal{S}_m$, with any diversity measure, is much more likely to result in small regret, while having a budget of $B = 1$ makes the task of finding exactly the best model challenging.

Pool Restriction. The downstream task itself defines the input modality (e.g., image or text) that needs to be supported by the models. Users then usually have use-case specific constraints, such as the framework (e.g., TensorFlow or Pytorch) one is restricted to, or the total number of parameters.

[1] https://tfhub.dev, https://pytorch.org/hub, and https://huggingface.co/models
[2] HuggingFace released an evaluation library (https://huggingface.co/docs/evaluate) after the submission of this paper. The functionality corresponds to simple task-aware search queries, yet not more complex hybrid or nested queries.

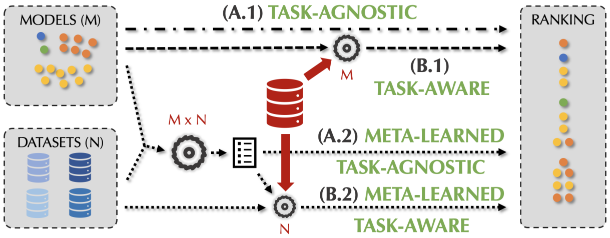


Figure 3: Categorization of model search strategies with their computational complexity. The N benchmark datasets are different from the ones used to train upstream, and the computation of the cross-product fine-tune table ($M \times N$) is decoupled from the search complexity.

We do not consider the pool restriction itself as a search strategy but rather as an integral part of any other search strategy that we outline next. Note that in the illustration in Figure 3, the search strategies do not return a subset of the model pool $\mathcal{M}$, but rather rank models in $\mathcal{M}$. We can map this ranking to our formal description of a search query $m(\mathcal{M}, \mathcal{D}, B)$, by selecting the *top-B* models according to the ranking, randomly breaking ties.

Strategies. Figure 3 illustrates the different *model search strategies* along with their computational requirements. We remark that model search strategies were extensively studied in our work [37], whereas here we present an overview of these methods and facts that are important from the perspective of SHiFT. As in [37], we divide model search strategies into two main categories: (A) task-agnostic strategies, which are those that ignore the downstream dataset, and (B) task-aware strategies, those that do take the downstream dataset into consideration.

(A.1) Task-Agnostic Search. The first category ranks the models in the pool by completely ignoring the downstream dataset. This can range from naively ordering the models by their name, size, or date of appearance, to more prominent strategies suggested by related work: (a) ranking models trained on ImageNet based on the upstream accuracy [14] (for images), or the average GLUE [45] performance (for text), and (b) favoring more robust models [8].

(A.2) Meta-Learned Task-Agnostic Search. Instead of restricting to meta-data reported by the model publishers, one could fine-tune every model registered in the system on a fixed set of N benchmark datasets (e.g., the 19 from VTAB [53]), or on a subset (e.g., only natural datasets in VTAB) and the aggregation metric used to rank the models (e.g., maximum over this subset) chosen by the user. Supporting such meta-learned search strategies requires SHiFT to run some computation upon registering a new model by fine-tuning it using all suitable benchmark datasets, whereas the retrieval phase remains independent of the downstream task.

(B.1) Task-Aware Search. Using linear classifier accuracies as a proxy to rank the models is often regarded as the standard when incorporating the downstream task into the search process [9, 14]. Instead of fine-tuning the weights of the pre-trained network as described previously, one *freezes* them and only learns the weights of a newly initialized linear head. In a large empirical study we have shown that such a linear proxy can suffer from a relatively high regret when trusting this search strategy over exhaustively fine-tuning all models and then picking the best one [37]. Nevertheless, this approach still represents one of the most powerful known search strategy. To improve computation time, researchers have proposed faster proxy methods by approximating the linear classifier accuracy [5, 29, 43], or relying on a cheaper classifier like the k-nearest neighbor [34, 37] and its approximations [25].

(B.2) Meta-Learned Task-Aware Search. The goal is to favor models that perform well on benchmark datasets *similar* to the downstream one. A prominent way of determining the similarity between datasets representing an ML task was introduced via learned task representations by Achille et al. [1]. To return a ranking of the models, one has to compute a Task2Vec representation for the new dataset and then find the nearest task (i.e., via the distance metric introduced by Achille et al. [1]). Registering a new model can be computationally demanding as it requires fine-tune accuracies of this new model on all benchmark dataset, but is decoupled from the model search performed by the user.

Hybrid Search. Empirically, with currently publicly available pre-trained models, for every single method there exists a case in which it suffers from a relatively high regret (i.e., returning a suboptimal model to fine-tune) [37]. In our prior work, we proposed to extend the returned set to two or more models, where one can start mixing strategies (e.g., best task-agnostic and best task-aware model) [37]. In particular, we showed that a simple hybrid search strategy that suggests fine-tuning the top-1 task-agnostic and top-($B-1$) task-aware model, often leads to superior results compared to fine-tuning the best B models based on a single strategy.

2.4 Need For SHiFT

The hybrid strategy outlined above represents the most robust choice for searching models *independently* from the model pool or possible relations between downstream task and benchmark datasets. Users may nonetheless want to break this independence assumption by incorporating specific knowledge about the filtered models or the relation to benchmark datasets. As an example, if users restrict the pool to models trained on the same upstream dataset, ranking the models based on upstream accuracy often represents a cheap and near-optimal search strategy. Hence, we require a system that gives users the *flexibility* to express a model search query that reflects their most important criteria. The system should also *efficiently* execute search queries, automatically applying query execution optimizations under the hood. In order to support inexperienced users, we provide detailed guidelines for using the best queries in our system and mimicking best-practices under https://docs.shift.ml/guidelines. Finally, bearing in mind that model development is typically an iterative process, if a user is not happy with the returned models or fine-tuning results, she may iterate by changing the data or the query, and then use the newly returned models. Thus, we need a system that can support various search strategies with efficient initial and iterative executions.

Benchmarking. Comparing new search strategies is a demanding task. Firstly, as explained before, existing search strategies are not supported by the existing model repositories and therefore have to be re-implemented by every researcher. Secondly, when evaluating strategies, users need to have access to a large set of (diverse) models, each of them being fine-tuned on a large set of benchmark

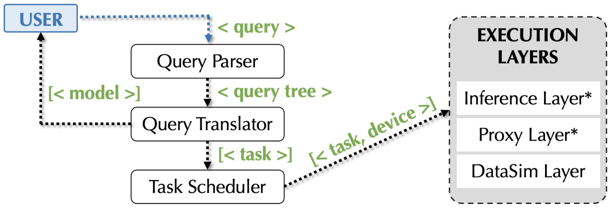


Figure 4: The logical components and different types that connect them that compose SHiFT, where (*) represents components that we optimize in Section 4 and 5.

datasets. Researchers need support in both aspects to be able to derive new theoretical understanding for existing approaches, or to test new search strategies against existing ones.

3 SYSTEMS ARCHITECTURE

We now present the architecture of SHiFT, allowing us to efficiently and flexibly execute the search queries outlined in the previous section. We abstract our system into multiple logical components, visualized in Figure 4. The labels on the arrows indicate *types* of the input for each component. We start by describing our novel logical model for transfer learning in Section 3.1. We then present our query language and its parser in Section 3.2, and use Section 3.3 to describe the scheduler and execution layer. Figure 13 in Appendix A[3] provides a complete overview of SHiFT. SHiFT is designed as a server-client architecture connected by standard HTTP requests. From the client perspective, the input to the system is a SHiFT-QL query, and the output is the corresponding result, i.e., a list of models, together with the information whether the query was executed successfully or not. On the server side, SHiFT takes the parsed SHiFT-QL query tree as the input, and determines how to provide the results. SHiFT internally caches intermediate results within and across queries to optimize execution time. We elaborate on caching and other system optimizations in Section 4.

3.1 Logical Model of Transfer Learning

One of the most important challenges in building SHiFT is to provide a clean abstraction for the search process of transfer learning, which needs to be flexible enough to model most popular search processes that users are using in practice, but in the meantime, needs to be high-level enough for us to capture the opportunities of system optimizations and incremental maintenance. In the following, we describe the SHiFT's logical model, which is based not only on our own experience in model search [37] but also a comprehensive survey of existing popular search strategies [1, 5, 9, 14, 25, 26, 29, 34].

Unified Logical Views for Model Repositories and Datasets. During the model search process, there are two key players: (1) a diverse collection of model repositories (e.g., TF-Hub, PyTorch Hub, and HuggingFace) and (2) a collection of datasets stored in a diverse set of formats (e.g., numpy, CSV, TFDS, etc.). The starting point of SHiFT is to provide a unified view for both models and datasets. The 'Models' view is a relational view containing information of models across various sources. As illustrated in Figure 5, each model corresponds to a single row in the 'Models' view, which contains "meta-data" about its source, modality, number of parameters, etc.

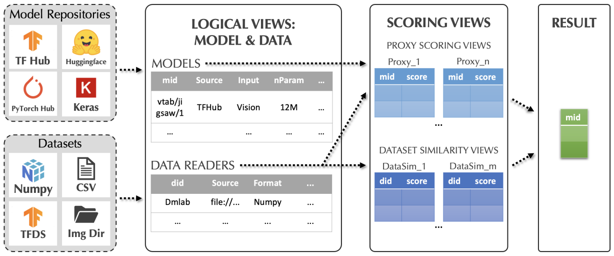


Figure 5: SHiFT's Logical Model of Transfer Learning

Each model ID also is associated with various functions to deal with tasks such as inference and fine-tuning, all of which are virtualized in dockerized environments to unify the API differences of different model repositories (see Section 3.3). In practice, we observe that having a relational view for 'Models' is particularly useful as users often conduct specific filtering queries over all models (e.g., to only keep models with # parameters smaller than a given constant to ensure inference latency). These can be done via standard SQL queries over the 'Models' view. The 'DataReaders' view is a relational view containing information of datasets stored in different formats. Each dataset corresponds to a single row in the 'DataReaders' view, which contains its meta-data. Each unique DataReaders is also associated with an *iterator* that enumerates (x, y) pairs where x is a Numpy array for a single feature vector and y is a Numpy array for a single label.

Benchmark Results View. In addition to the two views outlined before, SHiFT gives access to a view called 'BenchmarkResults', acting as a many-to-many join between models and datasets. The view is populated by the administrator of SHiFT with the accuracy reported when fine-tuning a fixed model on a fixed benchmark dataset. The system does not distinguish between benchmark and non-benchmark datasets, and treats both as tuples in the 'DataReaders' view. The 'BenchmarkResults' is useful for two distinct use-cases. Firstly, when designing meta-learned search queries, users can join and use the results between models and datasets via this view. Secondly, in order to easily compare different search strategies, given a set of M models and N benchmark datasets, for which the post fine-tune accuracies are present in the view, the benchmark module of SHiFT can run a search strategy for any of those N benchmark datasets by simulating a state of the system where the corresponding fine-tune results are not present in the 'BenchmarkResults' view. The accuracy of the returned model can then be compared to the best model accuracy out of the M ignored fine-tune results.

SHiFT-QL *Query.* A SHiFT-QL query defines a unique search strategy for transfer learning. In our design, a SHiFT-QL query consists of two components: (1) a collection of "scoring views" and (2) a generic SQL query over these scoring views. Note that all scoring views are *lazily evaluated up to various fidelity and precision*, which renders the query execution and optimization non-trivial and unique for SHiFT.

Lazily Materialized top-K Scoring Views. Given the 'Models' view and the 'DataReaders' view, a user can define scoring views of two types. The first type, which we call *proxy scoring views* aims to capture proxy tasks that are used to assess a model's transferability. A proxy scoring view extends the SQL syntax and can be defined as

[3] Accessible via https://github.com/DS3Lab/shift

```
CREATE PROXY SCORING VIEW name
SQLQUERY         -- e.g., SELECT * FROM Models
                 --       WHERE nParam < 12M
<SHIFT>
  ORDER BY ScoringAlgorithm [DESC | ASC] LIMIT K
  [TESTED ON DataReader1]
  [TRAINED ON DataReader2]
  [WITH DataReader3 ...]
</SHIFT>
```

where SQLQUERY is a standard SQL query whose output has the same schema as the Models view. In this way, a user can use arbitrary SQL queries to conduct different filtering strategies or join with auxiliary information about models to select models. Given all models that SQLQUERY returns, a ScoringAlgorithm is associated with a function (see Section 3.3) that maps a single Model, and a series of DataReader into a real-valued score:

$$\texttt{ScoringAlgorithm} : \texttt{Model}[\times\texttt{DataReader}\times\ldots\times\texttt{DataReader}] \rightarrow \mathbb{R}$$

By default, a proxy scoring view will only return the top-K models according to the output of the ScoringAlgorithm. This is often the case in most search strategies that we see in practice, and as we will see later, will open up novel opportunities for system optimizations.

The second type of scoring views are what we call *dataset similarity views*, which are used to compute similarities between different datasets, an important signal in many model search strategies — a model that performs well on a *similar* dataset is likely to perform well on the target dataset, if we are able to compute datasets similarities reliably. A dataset similarity view also extends the SQL syntax and can be defined as

```
CREATE DATASET SIMILARITY VIEW name
SQLQUERY         -- e.g., SELECT * FROM DataReaders
                 --       WHERE Modality = Image
<SHIFT>
  ORDER BY DataSimMetric [DESC | ASC] LIMIT K
  TESTED AGAINST TargetDataReader
</SHIFT>
```

where SQLQUERY is a standard SQL query whose output has the same schema as the DataReaders view. In this way, a user can use arbitrary SQL queries to conduct different filtering strategies or join with auxiliary information about datasets similar to the selected dataset. DataSimMetric is associated with a function that computes the similarity between a pair of datasets:

$$\texttt{DataSimMetric} : \texttt{DataSet} \times \texttt{DataSet} \mapsto \mathbb{R}$$

Given all datasets returned by SQLQUERY, SHiFT computes its similarity with the target TargetDataReader. Similar to a proxy scoring view, a dataset similarity view also by default keeps the top-K datasets according to its similarity with the TargetDataReader.

SHiFT-QL *Query: Putting Things Together.* Given a collection of scoring views, a SHiFT-QL query is a generic SQL statement querying these views. This allows flexible aggregation and voting strategies, and are crucial for many search strategies [37]. We provide several syntax sugars to make the query more succinct. When there is no ambiguity, we often ignore the <SHIFT> and </SHIFT> tags. Moreover, we also support creating scoring views implicitly if such a query is nested in another SQL query. As an example, to specify one hybrid search strategy developed in [37]: *Return the vision model with fewer than 10M parameters and the best upstream accuracy and another vision model that has the best linear classifier accuracy,* a user can write the following SHiFT-QL query:

```
(SELECT ModelId FROM Models
WHERE Input = 'Vision' AND nParam <= 10M
ORDER BY UpstreamAccuracy DESC LIMIT 1) Q1
         UNION
(SELECT ModelId FROM Models
WHERE Input = 'Vision' AND ModelId NOT IN Q1
ORDER BY Linear(lr=0.1) ASC LIMIT 1
TESTED ON TestReader TRAINED ON TrainReader) Q2
```

We provide the SHiFT-QL for the popular search queries Q1-Q5 from Table 1, as well as a more complex nested query in Appendix B.

Tracking Data Changes via Change- and Add-Readers. To support efficient incremental executions over data changes, we extend this simple concept of a data reader to a *mutable* reader, allowing a reader to be a composite of an initial data reader and a list of *change-* or *add-readers*. Every change-reader is accompanied by a list of indices of the same length as the initial reader, indicating which samples to replace. The change- and add-readers are then processed in a linear order to build the final mutable data reader. The two advantages of representing our data as such are (a) extensibility to other data sources and (b) the ability to cache inferred features on a per-reader (initial, change, or add) level. On the flip side, removing samples from a data reader requires users of SHiFT to define a new reader, resulting in a new execution from scratch. We plan to support deletions in the future. It is also important to note that our current data provenance system tracks the changes in a rather naive way. In the future, we should provide the support of modern data provenance systems for ML (e.g., [10] and [41]).

Supporting New Search Strategies. One design goal of SHiFT and SHiFT-QL is to make it easier for researchers to provide new search strategies in the future. In our current system, this can be done by registering a new ScoringAlgorithm or a new DataSimMetric. We are optimistic that this will support many new search strategies in the near future (e.g., the "more robust model is more transferable" strategy that just comes up months ago [8], which requires registration of robustness metrics as additional attributes in the dataset). We provide simple tutorials on how to implement new simple strategies online, and evaluate additional search strategies beyond the ones present in the paper (e.g., random sampling or LEEP [29]) in Appendix F using our benchmark module.

3.2 Query Parser and Translator

Our query parser takes a possibly complex SHiFT-QL query as an input, and generates a parsed query tree, where every node is either a proxy scoring or dataset similarity view. The tree is then traversed in a bottom-up approach, by evaluating the leaf nodes until completion before evaluating the parent nodes. We leave query tree optimizations such as push-down operations or balancing compute across different nodes for future work. Every SQLQUERY is evaluated directly against our database. Task-agnostic search strategies represent SHiFT-QL queries using neither proxy scoring nor dataset similarity views. Meta-learned task-agnostic queries can be defined by the user by using dataset similarity views

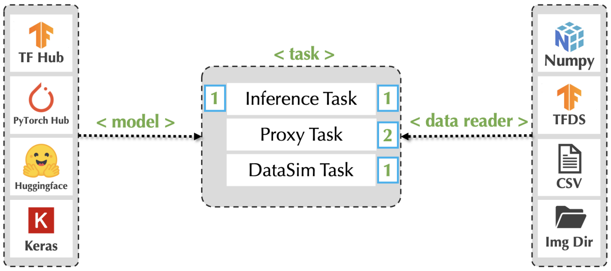


Figure 6: SHiFT tasks and their dependencies to model or data sources. The numbers specify on how many objects a specific task depends on (e.g. Proxy tasks depends on 2 data readers).

and filtering models using populated benchmark fine-tune results in it. For queries relying on proxy scoring views, the query translator will check if the result for the specified method name is known in the corresponding view (i.e., a tuple for each model and the specified readers exists in the database). If so, the system will directly return the results, or pass them to a parent node in the parsed query tree. Otherwise, the system will dispatch a list of SHiFT tasks for missing values to the task scheduler. Task-aware search queries are split into two inference tasks per model, one for the test and another for the train data source, and an additional single proxy task per model. The outputs of the inference tasks are used as input to the corresponding proxy task. This allows us to possibly reuse the cached feature representations per pre-trained model. To support better load balancing, we split both inference tasks for partitions of the datasets (c.f., Section 4). Meta-learned task-aware search queries rely on dataset similarities. If the embedding for the downstream task reader is not in the database, the system will dispatch a DataSim task to compute it. The embeddings are then used to compute the distance between tasks and rank the readers.

3.3 Task Scheduler and Execution

A SHiFT task represents the smallest computational unit of our system. The scheduler assigns every task to a single hardware device. We define three different tasks SHiFT supports: (1) inference tasks, (2) proxy tasks, and (3) DataSim tasks. The number and types of tasks executed depend on the query provided by the user, then parsed and translated by SHiFT (e.g., Q2 will only create inference and proxy tasks). Each of them has dependencies on models or data readers, or both (c.f., Figure 6). The scheduler of SHiFT is fairly simple. Every task gets assigned, in order of entering the queue, to the next free device, as soon as its dependency tasks are successfully terminated. Every GPU on a single machine represents a device, and a subset of the CPU cores forms another one. Costly inference tasks are assigned to GPU devices, if any are available. Proxy and DataSim tasks are also handled by the CPU.

Inference Task. To support various pre-trained model sources and frameworks, we define a minimal interface used by inference tasks. All supported model sources require a simple forward function for a batch of samples originating from every data source visualized in Figure 6. This function returns a 2D Numpy array with each sample (out of n) in the data reader representing a row. The feature dimension is determined by the pre-trained model. The resulting extracted features are stored on the disk, and references to the corresponding reader and model combination, using a hash of the earlier, and the model name as a unique identifier of the latter are saved. Formally, the following interface needs to be specified for each combination of models and data sources.

```
def extract(pre_trained_model: model,
            source: data_reader) ->
            np.array(shape=[n,dim])
```

Most of these 16 possible combinations are natively supported by the frameworks and API (e.g., using the Keras `fit` function for parts of a Keras model, or `KerasLayer` with TF-Hub models), or by casting the data sources into a supported format (e.g, using tensorflow data sources for both Keras and TF Hub). PyTorch Hub models typically only store the models with their PyTorch code and no standardize interface. Every model registered in SHiFT therefore needs to specify a hook function to extract the (last-layer) features, which requires us to know the internal structure of the models (i.e., the layer names). Custom trained or fine-tuned models can be exported as Keras models to disk, and then used for subsequent search queries upon registration into the database.

Proxy Task. By splitting task-aware search queries into inference and proxy tasks, we bypass the requirement of implementing the proxy computation for every combination of model and data source. The proxies are defined over Numpy arrays, where the extracted n train and m test features (ending with `_X`) stem from a model and data source combination after having performed an inference task, and the labels (ending with `_y`) are independent of the models.

```
def compute_proxy(train_X: np.array(shape=[n,dim]),
                  test_X: np.array(shape=[m,dim]),
                  train_y: np.array(shape=[n,]),
                  test_y: np.array(shape=[m,])) ->
                  proxy_value: float
```

We implement two different proxy estimators: (1) the nearest neighbor (NN) accuracy for two different distance functions (cosine dissimilarity, and Euclidean, L2 distance), and (2) a linear classifier accuracy trained with stochastic gradient descent (SGD) and arbitrary hyper-parameters, such as learning rate, L2 regularizer, mini-batch size and number of epochs.

DataSim Task. A dataset similarity (DataSim) task computed embeddings of a data reader representing a machine learning task (e.g., Task2Vec [1] to compute 8512 dimensional vectors). The embedding is then stored along with the meta-data of the reader in the database. Using any distance function (e.g., the non-symmetric one suggested by Achille et al. [1]), a subset of registered data readers in the database, for which the embeddings are pre-computed, can be ordered and limited in a straightforward fashion. The final meta-learned task-aware search query is then no different from the meta-learned task-agnostic. We use the code provided by Achille et al. [1] for running the DataSim tasks.

4 SYSTEM OPTIMIZATIONS

4.1 Successive-Halving (SH)

In SHiFT, all proxy scoring views consist of a top-K query over a list of scores; furthermore, each score is computed as a function over `DataReaders` which consist of a set of data examples. This structure opens up unique opportunities for system optimizations — since many of these scoring functions are relatively stable with respect to sub-sampled datasets, we can *approximate* this top-K view with a scoring function evaluated over only a *subset* of data examples. One key optimization is to estimate the proxy value only for a small subset of the (training) data on most models, and a large fraction of the data only on a small subset of the models. There can be various ways for this. Currently, SHiFT uses successive-halving (SH) [12], which is invoked as a subroutine inside the popular Hyperband algorithm [20]. Algorithm 1 in Appendix C outlines the algorithm, noting that an *arm* in our context represents a model, and *pulling an arm* corresponds to running inference on more data and estimating the proxy using the extracted features for *all* the data seen by the model so far. In a nutshell, we can summarize the idea of SH as follows: Start by uniformly allocating a fixed initial budget ($B/\log_2(M)$) to all M models and then evaluating their performance. Keep only the better half of the models and repeat this until a single model remains. The algorithm has two different hyper-parameters: a chunk size (how many samples represent an arm pull) and the overall budget B. Both can be specified by the user and have an impact on the accuracy of the results and compute time. We propose a chunk size guaranteeing that the last model has processed the entire dataset as a default for SHiFT, and use the minimal budget required to return a fixed number of models.

SH Minimal Budget and Chunk Size. In order to preserve the semantic of the queries whilst performing successive halving (i.e., not sub-sampling the data), we need to guarantee $r_k > 0, \forall k \iff \frac{B}{L\lceil\log_2(|M|)\rceil} > 1$. Let us assume a top-q queries with $q = 1$ (the derivation can simply be extended to arbitrary values of q). We need $B \geq L\lceil\log_2(|M|)\rceil, \forall L$. The largest L is reached at the first step where $L = |M|$. Hence, we need to have $B \geq |M|\lceil\log_2(|M|)\rceil$. Conversely, at kth step, each remaining model is given $r_k \times C$ additional training samples, in total each remaining model has processed $\sum_{j=0}^{k} r_j \times C$ training samples. When $k = \lceil\log_2(|M|)\rceil - 1$, the remaining models have processed $C \times \sum_{j=0}^{\lceil\log_2(|M|)\rceil-1} r_j$ training samples. Hence, the minimal chunk size C_{min} such that the remaining models have processed all training samples is given by

$$C_{min} = \frac{N}{\sum_{j=0}^{\lceil\log_2(|M|)\rceil-1} r_j}$$

where $r_j = \lfloor \frac{B}{\lceil M/2^j\rceil\lfloor\log_2(|M|)\rfloor} \rfloor$.

4.2 Cost Model for Successive Halving

Successive-halving, while always being able to decrease the amount of examples processed, does not always outperform the baseline strategies in wall clock time. Moreover, as we show in the experiments, it can sometimes even be slower. This might seem counter-intuitive, but the main reason lies in hardware accelerators, such as GPUs, which offer the ability to massively parallelize tasks up to a fixed number of samples. For instance, running inference for one sample through a deep neural network requires roughly the same time as a mini-batch of multiple samples. The maximum mini-batch size is often limited by the device memory. Therefore, one should not split very small readers into multiple chunks to speed up a task, rendering SH inappropriate for small datasets. Additionally, SH introduces sequential dependencies between tasks, which can render the algorithm inefficient or unable to scale to multiple GPUs. One such a cause lies in the repeated model loading, or access to the extracted test features, noticing that we always use the entire test set to estimate a proxy value after an arm pull.

We therefore introduce another key component into SHiFT: a cost-based decision making process that automatically decides whether to use successive halving. To this end, we derive a cost model for SHiFT with and without SH. Our cost model requires a few variables, either pre-computed or available based on the query. Let N be the size of our training dataset, O the size of the test dataset, and M the number of models. Furthermore, let P represent the number of equal devices (e.g., GPUs). The time required to load a model i onto a device is given by L_i. Furthermore, the time to load the (training) dataset is represented by T_N, whereas the time required to load the inferred test representations for the model i is given by T_i. We simplify this by assuming a global T_O, since the representations only differ in their dimensions. We neglect the time to load the raw test dataset as this is equal regardless of the optimization. The time to run inference for k samples and model i on the device is given by $I_i(k)$, which we assume to be linear, hence $I_i(k) = I_i k$, for some constant I_i. The time to compute a proxy $E^i_{Proxy}(k)$ follows the same principle, although we assume that it is independent of the model (neglecting the dimension of the representations), hence $E^i_{Proxy}(k) = E_{Proxy}k$. The cost for running a top-1 query without SH on multiple GPUs is computed with

$$T_{\text{w/o}} := \frac{1}{P}\sum_{i=1}^{M}\Big(\underbrace{T_N + T_O}_{\text{Load data}} + \underbrace{2L_i}_{\text{Load model}} + \underbrace{I_i O}_{\text{Test inference}} + \underbrace{I_i N}_{\text{Train inference}} + \underbrace{E_{Proxy}N}_{\text{Proxy estimation}}\Big),$$

Notice that we need to load the train dataset again every time when there is a new model (request), as every task is executed independently. The counterpart, running a top-1 query with SH on multiple GPUs, where we assume perfect parallelization which is harder to achieve for heterogeneous models and small chunks C, is given by

$$T_{\text{w/}} := \underbrace{\frac{1}{P}\sum_{i=1}^{M}(L_i + I_i O)}_{\text{Test inference}} + \underbrace{\sum_{k=1}^{\lceil\log_2(M)\rceil}\frac{1}{\min(P, |S_k|)} \times}_{\text{SH iterations}}$$
$$\Big(\underbrace{\sum_{j\in S_k}(L_j + T_N + I_j C r_k)}_{\text{Train inference}} + \underbrace{\sum_{j\in S_k}\Big(T_O + E_{Proxy}\sum_{l=1}^{k}(Cr_l)\Big)}_{\text{Test load and proxy estimation}}\Big),$$

where C is specified by the user or taken as C_{min} (c.f., Section 4), S_k and r_k are taken from Algorithm 1. Clearly, the sets S_k of models *surviving* during the SH algorithm have an impact on the runtime. Following the trend of larger and slower models surviving the longest, we define S_k to be the set of k models with the largest inference time I_j for all $j \in [M]$. SHiFT will use this cost model (i.e., the minimum of $T_{\text{w/o}}$ and $T_{\text{w/}}$) to automatically decide, based on system's specifications (e.g., hardware devices and model inference times), number of samples, and models in the restricted pool, whether to use the SH algorithm.

SH for Other Queries. We only use the SH optimization for task-aware queries for which a fraction of the samples can be used for *ranking* models with high confidence. Classifier accuracies (e.g., NN or linear classifier) are known to satisfy this property [38]. Furthermore, the overhead of running multiple sequential tasks when using the SH algorithm is kept small for task-aware queries, thanks to the two-stage approach (i.e., the inference and proxy estimation phase), which is not the case for the other search queries. Inference tasks are typically much more time-consuming compared to the proxy estimation. Nonetheless, when *pulling* an arm an additional time, the data from the previous arm pulls are not required to be run through the network again as they can be fetched from the disk for the subsequent proxy estimation tasks.

4.3 Other Optimizations

Caching. Caching is crucial for rapid incremental query executions. SHiFT internally caches dataset similarities, feature vectors, and proxy values in order to reuse intermediate results within and across queries as much as possible. When dispatching task requests, SHiFT ensures that only necessary requests are executed. For example, if a user sends a request for a proxy estimator (e.g., the nearest neighbor accuracy) and a single model, SHiFT will create two inference requests, one for the test and train reader, and a proxy estimator request. The former requests turn the input data into feature vectors and store them on the disk. If, at a later stage, the user requests another proxy estimator for the same model (e.g., the linear classifier accuracy), SHiFT will notice that it can reuse the cached feature vectors. Therefore, SHiFT will only dispatch a proxy estimation request, which uses the feature vectors to calculate the proxy value. The dispatched requests are then handled by the task scheduler asynchronously. Once a certain request is done, the results will be written into the database and ultimately returned to the user upon running the same query.

Load-Balancing. To automatically load-balance heterogeneous workloads, mainly stemming from large discrepancies in inference times between models, to multiple hardware devices, we automatically split readers for every inference task. The number of partitions is equal to the number of GPU devices, unless a reader is smaller than a fixed threshold. This guarantees fast executions especially when running inference on small readers (i.e., change-readers).

5 INCREMENTAL EXECUTIONS

If a user is not satisfied with the results of the query, either in terms of the proxy values or other properties of the models returned (e.g., diversity or downstream performance), she typically iterates by incrementally executing another, often similar query, or re-run the same query performing one of the following changes:

(1) add a model to the database,
(2) change the data (features or labels) to test or train readers by using change-readers,
(3) add data to the test or train reader using an add-reader.

Incremental Scenario 1: Changing Data or Queries, or Adding Models. Changing data (features or labels), a query, or adding a model to the database will naturally benefit for the caching mechanisms we introduced in the previous section. For example, a task-aware search query will only require SHiFT to dispatch two inference tasks and a single proxy task per additional model, retrieving the other results directly from the database. When using the SH optimization, the same idea of reusing the caches for intermediate results (features and proxy estimation values) applies. Data manipulations are slightly more involved. Based on the defined mutable data reader concept in Section 3.1, costly inference operations only need to be executed for the changed features. The cheaper proxy task is then evaluated on the final mutable reader (i.e., after having iterated over all change- and add-readers) only once, or for every subsequent iteration in the SH algorithm.

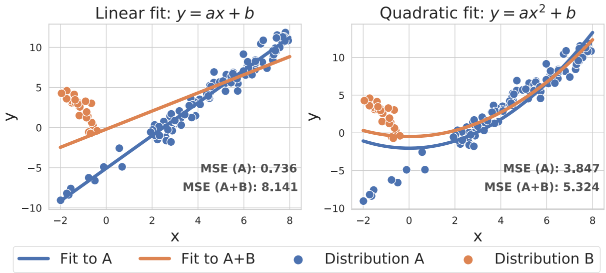


Figure 7: Two models (linear and quadratic), for which the order (i.e., based on the minimal MSE) changes if they have access to data only from the first distribution (A), compared to both distributions (A+B).

Incremental Scenario 2: Adding Data. Adding data to the test reader requires us, similarly to changing data, to run inference on all models of the query using these additional data samples, and then rerun the proxy estimation value for every intermediate step in the SH algorithm. However, blindly appending *training* data at the end of the data reader can result in an undesired behavior. Concretely, all models that did not process the entire dataset (i.e., were eliminated in the SH algorithm based on a subset the head of the data reader) will not benefit from the appended samples. This can be problematic if the added data stems from a different distribution, thus possibly *eliminating* better arms (i.e., models) while running the SH algorithm. We construct and illustrate one such synthetic case in Figure 7, where we assume to have two models (a linear one on the left and a quadratic one on the right), and data coming from two distributions (A in blue and B in orange). If the models were ranked on the basis of the errors of distribution A alone, the quadratic model would be eliminated. If then data from distribution B is appended to the reader (via an add-reader), the winning linear model from before would actually be inferior to the quadratic one. We show another real-world example in Section 6.5.

To address this issue, ideally, one would randomly reshuffle the full data reader with appended data. Despite being favorable from a statistical point of view, managing the cache and preventing a complete re-execution is far from trivial.

In SHiFT we *uniformly distribute* the new samples as an alternative strategy. This enables high performance and results in a small difference compared to the fully shuffle approach from a statistical point of view. As a justification of its statistical property, let us assume that we have an initial data reader of size N together with an add-reader that contains αN samples, with $\alpha \geq 0$. Moreover, assume that the chunk sizes used by the SH algorithm on the initial data reader is of size βN, with $0 \leq \beta \leq 1$. We want to compare the two strategies: (1) randomly inserting the new samples anywhere between the existing ones and re-partitioning the samples into buckets afterwards, and (2) uniformly distributing the new samples amongst all existing buckets. Notice that the number of buckets remains constant after handling the new samples, thus increasing the size of the buckets to $\beta N(1+\alpha)$. Furthermore, it is obvious that from an implementation perspective, the second strategy is superior to the first, whereas the first strategy introduces less bias into the sampling process. However, both approaches yield the same number of samples from both distributions in expectation. We define a random variable C, which represents the number of initial samples in the first bucket when the first strategy is run. Coincidentally, C follows a hyper-geometric distribution with an expectation of $\mathbb{E}[C] = \beta N$, which is exactly the number of samples in the same first bucket we get when applying the second strategy.

6 EVALUATION

6.1 Experimental Setup

We conduct our experimental study next on computer vision and NLP classification tasks, representing the most prominent applications of transfer learning [53]. Nonetheless, SHiFT is flexible and the code-base supports workload beyond these modalities.

Models. We compile a diverse list of 100 computer vision and 60 NLP models, whose details, including additional configuration such as inference time for the GPU type needed in the next paragraph, are given in Appendix D. To restrict the search space for fine-tuning, we follow Zhai et al. [53] and train the models for 20 epochs, using a mini-batch size of 16, momentum of 0.9, and learning rate of 0.01, with the Adam optimizer.

Datasets. We conduct our experiments with 3 vision datasets representing different downstream tasks: (1) Oxford Flowers 102 [30] (Flowers) with 1K training and 6K test samples, (2) CIFAR-100 [17] (CIFAR), with 50K training and 10K test samples, and (3) Dmlab [53] with 65K training and 23K test samples. We furthermore use two NLP datasets from the glue benchmark [45]: (1) cola, with 8.5K training and 1K test samples, and (2) sst2, with 67K training and 1.8K test samples.

Hardware. We use a GPU cluster (single machine) with 8 NVIDIA TITAN Xp for SHiFT. The system is configured to either use a single or all eight GPUs. For fine-tuning models, we use a different cluster with slightly more performant NVIDIA GeForce RTX 2080 Ti GPUs.

Queries. We evaluate the performance of the 3 queries Q2-Q4 from Table 1, with a focus on computational efficiency. Q1 is included in Q4 and directly evaluated against the database and therefore omitted in the experiments. Q5 uses the Task2Vec code to find the nearest benchmark task. To successfully apply Q5, or any meta-learned query, we require a large set of benchmark datasets, all fine-tuned on all modules. We provide results for both meta-learned queries Q5 and Q7 using our benchmark module in Appendix F. Whenever the SH algorithm is used, depending on the cost model and specified in the experiments, we set the budget and chunk size to be minimal according to Section 4. This ensures that the semantics of the queries are kept intact, i.e. the data is not sub-sampled.

Table 2: Execution time for fine-tuning (FT) all the models via enumeration compared to running Q2-Q4 using SHiFT with and without automatic optimization (AO).

			1 GPU		8 GPU	
			Runtime (Hours)	Speedup (vs. FT)	Runtime (Hours)	Speedup (vs. FT)
CIFAR	FT		251.8		31.5	
	Q2	w/o AO	8.7	28.8x	1.6	19.8x
		w/ AO	5.9	**42.9x**	1.3	**24.1x**
	Q3	w/o AO	9.1	27.6x	1.6	19.2x
		w/ AO	5.8	**43.4x**	1.3	**24.3x**
	Q4	w/o AO	7.9	31.7x	1.0	**30.0x**
		w/ AO	5.6	**45.0x**	1.2	25.6x
Dmlab	FT		314.8		39.4	
	Q2	w/o AO	14.3	22.0x	2.4	16.2x
		w/ AO	9.6	**32.7x**	2.0	**19.7x**
	Q3	w/o AO	14.6	21.6x	2.5	16.0x
		w/ AO	9.9	**31.8x**	1.9	**21.2x**
	Q4	w/o AO	13.3	23.7x	1.7	**22.6x**
		w/ AO	9.3	**33.9x**	2.0	20.0x
Flowers	FT		9.6		1.2	
	Q2	w/o AO	2.9	**3.3x**	0.4	**2.8x**
		w/ AO	2.9	**3.3x**	0.4	**2.8x**
	Q3	w/o AO	3.1	**3.1x**	0.4	**2.7x**
		w/ AO	3.1	**3.1x**	0.4	**2.7x**
	Q4	w/o AO	2.8	**3.4x**	0.4	**3.1x**
		w/ AO	2.8	**3.4x**	0.4	**3.1x**

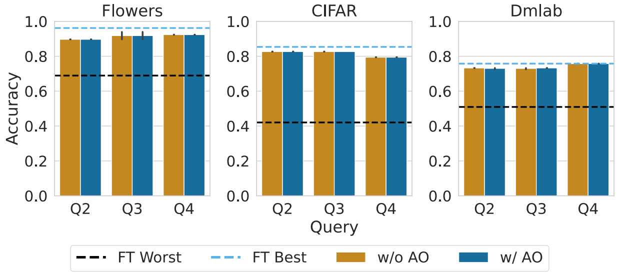


Figure 8: Fine-tune (FT) accuracy of returned model for queries with and without automatic optimization (AO). The variance illustrates the min and max over 4 independent runs, showing some fluctuations for the small Flowers dataset and the linear proxy, which is sensitive to its hyper-parameters.

6.2 End-to-end Performance

We start by validating the end-to-end performance of SHiFT on computer vision task. Due to the space limitation we present the corresponding NLP results to Appendix E. Table 2 compares the runtimes of fine-tuning all the models, the method which we call *enumerate*, to running and using the output of queries Q2-Q4 on SHiFT, with and without automatic optimization (AO). Our cost model, which we validate later in this section, suggests not to use SH for the small Flowers dataset, which is why the SH optimization is enabled only for CIFAR and Dmlab. When it comes to accuracy, Albeit being up to 1.5 orders of magnitude faster (c.f., Table 2), Figure 8 shows that the queries manage to retrieve near-optimal models for all datasets (i.e., suffering from very small regret). Furthermore, the SH optimizations for CIFAR and Dmlab retain the query semantics, not affecting the accuracy over the baseline for

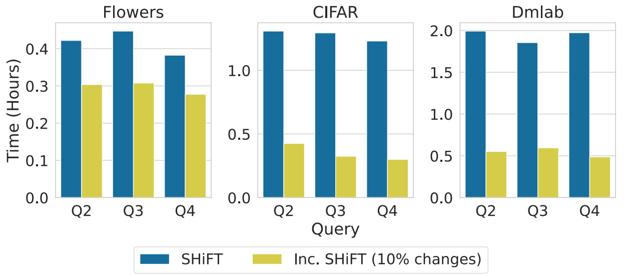
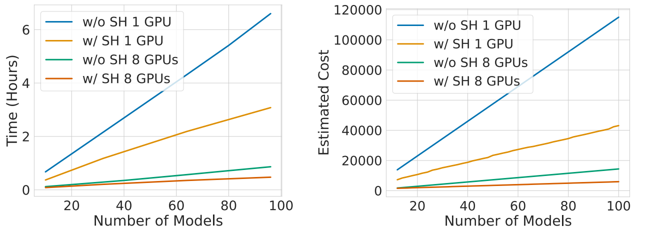


Figure 9: Incremental execution using SHiFT on 8 GPUs. 10% of the samples are randomly replaced.



(a) Execution times of Q2 for CIFAR.

(b) Cost model for Q2 on 50K training samples.

Figure 10: Increasing number of (homogeneous) models. The ResNet-101 V2 model is replicated for the experiments and the cost model.

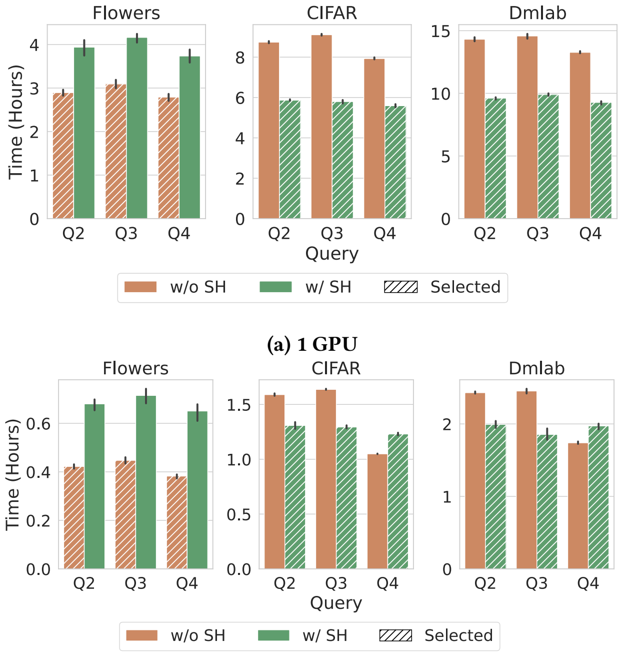
each query. Remember that the main focus of this SHiFT is to support a large set of possibly complex queries independent of the actions on the returned model (e.g., fine-tuning) as efficiently as possible. For a complete empirical study that compares different search queries, we refer to our companion work [37] and Section 6.6, where we benchmark different strategies using SHiFT. Finally, Figure 9 shows the runtime for incrementally running SHiFT on 10% randomly changed samples, leading to significant speedups for larger datasets, where the GPUs are fully utilized.

6.3 Scalability of SHiFT

With the experimental setting described, we implicitly analyze the scaling behavior of SHiFT for an increasing number of GPUs (one and eight), and (training) samples (1K, 50K, and 62K). Increasing the number of models is analyzed in Figure 10 (left), where we deliberately chose a homogeneous setting of replicating the same model architecture multiple times.

6.4 Cost Model: SH Trade-offs

Figure 10 (right) validates the relative performance of our cost model for a set of homogeneous models. In Figure 11 we show the different runtimes with and without SH for 1 and 8 GPU and all queries along with the configuration picked by SHiFT on the 100 diverse models. On the Flowers dataset, our cost model accurately predicts the relative improvements to be expected when not using SH over using SH for a single (1.68x) and multiple GPUs (1.95x). On the larger datasets, CIFAR and Dmlab, the ratio for using 1 GPU (both 1.8x) is validated by our experiments. For multiple GPUs and the larger datasets, SHiFT predicts that SH should outperform non-SH by 1.2x, e.g. on CIFAR for all three queries, while the performance only matches Q2 and Q3, as visible in Figure 11. The hybrid query Q4 removes a very large model, however, our cost model overestimates the benefits of SH in such a case. The reason lies in the heterogeneity of the models (e.g., the largest model takes up almost 10% of the overall inference time) and the order of execution currently neglected in the cost model for both with and without SH on multiple GPUs. This explains the gaps visible in Figure 11b. Fusing these aspects into the cost model requires runtime-specifc variables (e.g., information about other queries executed in parallel), making it much more complex and potentially introducing extra latency when executing a query. It is therefore left as future work.



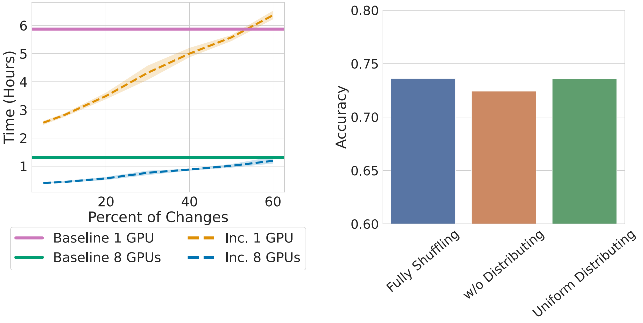
(a) 1 GPU

(b) 8 GPUs

Figure 11: Execution times for all settings. The variance in black illustrates the min and max over 4 independent runs, showing little fluctuation. The hatched bar indicates the selected plan based on the cost model.

6.5 Incremental Execution

In Figure 12a, we randomly change a fixed percentage of the samples (i.e., manipulating the features) and plot the time required to perform an incremental execution of Q2 on CIFAR. Unsurprisingly, after a significant fraction of changes (e.g., > 50%), users might want to enforce a re-execution from scratch (i.e., by building a new initial reader instead of using a change-reader). However, when a small fraction of the samples are changed and the query is run incrementally, SHiFT offers a significant speedup over the baseline. The computational performance of adding data follows the same trend as changing data. Note that the accuracy of any incremental query executions is heavily dataset- and distribution-dependent. The compute time for adding models to the query corresponds to the time needed to run a second independent query on these new models, due to the independence between the computation



(a) Execution time with respect to different number of feature changes for Q2 on CIFAR.

(b) Distribution of new training data (a full class) for Q2 on Retinopathy.

Figure 12: Incremental Execution.

on the new models and the old ones. Changing any number of labels in SHiFT is significantly cheaper compared to a re-execution. The reason lies in the large computational overhead of running inference compared to the cheap proxy computation. SHiFT only requires to re-run the latter for all models. Finally, to further show the importance of distributing new samples, we construct a data reader for the Retinopathy dataset consisting of only four out of five classes. We then append the fifth class via an add-reader and compare the post fine-tune accuracy using different shuffling variants in Figure 12b. This example illustrates a scenario in which the distributional difference between both readers, the initial one and the add-reader, leads to a good overall model being eliminated before seeing the new data if the samples are not distributed.

6.6 Using The Benchmark Module

We empirically validate the usefulness of the benchmark module of SHiFT be fine-tuning all VTAB-1K datasets [53] on a large set of more than 250 Huggingface transformer modules. Due to space limitations, we give the detailed time vs. post fine-tune accuracy comparison for all 19 datasets and a wide range of search queries, including randomly selecting one or two models, in Appendix F. The results confirm the usefulness of our guidelines, as well as the results given by previous work [14, 37].

7 OTHER RELATED WORK

ML Specific Data Management. The data-management has been working on improving the usability of ML in a flurry of work over the last decade, by focusing on different components of the ML development process. A few examples include data acquisition with *weak supervision* (e.g., Snorkel [35], ZeroER [48]), *debugging and validation* (e.g., TFX [6, 33], "Query 2.0" [49], Krypton [27]), *Model deployment* (e.g., MLFlow [51]), *knowledge integration* (e.g., DeepDive [54]), *data cleaning* (e.g., HoloClean [36], ActiveClean [16]), and *interaction* (e.g., NorthStar [15]). All these systems facilitate the ML development process without focusing on transfer learning.

Model Management. The data management community has also seen an intriguing line of work around *model management.* Systems like Cerebro [28] or ModelDB [44], and follow-up works [18, 21, 22, 40], are part of the main motivation to work on this new, transfer learning specific model management system. We hope that by lying the initial conceptual foundation of a model management system specifically for transfer learning, and by open sourcing SHiFT, we are able to trigger and facilitate future research in this area.

Dataset Search. Neural Data Server [7, 50] takes the approach of searching for *datasets* instead of pre-trained models to improve transfer learning. Other works such as Data2Vec [4] follow a similar goal by embedding a dataset and searching for similarity. Both approaches are orthogonal to our work, since we do not require access to the upstream datasets used to pre-train the models registered in SHiFT. Furthermore, it is unclear how these techniques can be used to distinguish models trained on the *same* upstream dataset.

Other Search Strategies. There are other search strategies omitted in Section 2 operating on semantical level (i.e., via a learned taxonomy) [52]. These methods are not well suited for searching in a pre-trained model hub, mainly given that they assume the input domain to remain fix, and datasets only to differ in their labels.

8 CONCLUSION

We presented SHiFT, the first downstream task-aware search engine for transfer learning. Using our custom query language SHiFT-QL, users can generically define different model search strategies. Based on a cost-model, we automatically optimize prominent search queries and show significant speedups. Furthermore, by caching intermediate results, we allow our users to efficiently execute similar queries incrementally. In the future, we hope that SHiFT, together with our benchmark module, will enable researchers to easily implement and evaluate newer search strategies.

ACKNOWLEDGMENTS

CZ and the DS3Lab gratefully acknowledge the support from the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number MB22.00036 (for European Research Council (ERC) Starting Grant TRIDENT 101042665), the Swiss National Science Foundation (Project Number 200021_184628, and 197485), Innosuisse/SNF BRIDGE Discovery (Project Number 40B2-0_187132), European Union Horizon 2020 Research and Innovation Programme (DAPHNE, 957407), Botnar Research Centre for Child Health, Swiss Data Science Center, Alibaba, Cisco, eBay, Google Focused Research Awards, Kuaishou Inc., Oracle Labs, Zurich Insurance, and the Department of Computer Science at ETH Zurich.

REFERENCES

[1] Alessandro Achille, Michael Lam, Rahul Tewari, Avinash Ravichandran, Subhransu Maji, Charless C Fowlkes, Stefano Soatto, and Pietro Perona. 2019. Task2vec: Task embedding for meta-learning. *IEEE International Conference on Computer Vision* (2019), 6430–6439.

[2] Leonel Aguilar Melgar, David Dao, Shaoduo Gan, Nezihe M Gürel, Nora Hollenstein, Jiawei Jiang, Bojan Karlaš, Thomas Lemmin, Tian Li, Yang Li, et al. 2021. Ease. ML: A Lifecycle Management System for Machine Learning. In *11th Annual Conference on Innovative Data Systems Research.* CIDR.

[3] Hossein Azizpour, Ali Sharif Razavian, Josephine Sullivan, Atsuto Maki, and Stefan Carlsson. 2015. Factors of transferability for a generic convnet representation. *IEEE transactions on pattern analysis and machine intelligence* 38, 9 (2015), 1790–1802.

[4] Alexei Baevski, Wei-Ning Hsu, Qiantong Xu, Arun Babu, Jiatao Gu, and Michael Auli. 2022. Data2vec: A general framework for self-supervised learning in speech, vision and language. *arXiv preprint arXiv:2202.03555* (2022).

[5] Yajie Bao, Yang Li, Shao-Lun Huang, Lin Zhang, Lizhong Zheng, Amir Zamir, and Leonidas Guibas. 2019. An Information-Theoretic Approach to Transferability in Task Transfer Learning. *IEEE International Conference on Image Processing* (2019), 2309–2313.

[6] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1387–1395.

[7] Tianshi Cao, Sasha (Alexandre) Doubov, David Acuna, and Sanja Fidler. 2021. Scalable Neural Data Server: A Data Recommender for Transfer Learning. *NeurIPS* (2021).

[8] Zhun Deng, Linjun Zhang, Kailas Vodrahalli, Kenji Kawaguchi, and James Zou. 2021. Adversarial Training Helps Transfer Learning via Better Representations. *arXiv preprint arXiv:2106.10189* (2021).

[9] Aditya Deshpande, Alessandro Achille, Avinash Ravichandran, Hao Li, Luca Zancato, Charless Fowlkes, Rahul Bhotika, Stefano Soatto, and Pietro Perona. 2021. A linearized framework and a new benchmark for model selection for fine-tuning. *arXiv preprint arXiv:2102.00084* (2021).

[10] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. 2021. Mlinspect: A data distribution debugger for machine learning pipelines. In *Proceedings of the 2021 International Conference on Management of Data*. 2736–2739.

[11] 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*. PMLR, 2790–2799.

[12] Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial intelligence and statistics*. PMLR, 240–248.

[13] Bojan Karlaš, Ji Liu, Wentao Wu, and Ce Zhang. 2018. Ease. ml in action: Towards multi-tenant declarative learning services. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2054–2057.

[14] Simon Kornblith, Jonathon Shlens, and Quoc V Le. 2019. Do better Imagenet models transfer better? *IEEE Conference on Computer Vision and Pattern Recognition* (2019).

[15] Tim Kraska. 2018. Northstar: an interactive data science system. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2150–2164.

[16] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. 2016. Activeclean: Interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment* 9, 12 (2016), 948–959.

[17] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009). http://www.cs.toronto.edu/~kriz/cifar.html

[18] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1717–1722.

[19] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel. 2016. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record* 44, 4 (2016), 17–22.

[20] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* 18, 1 (2017), 6765–6816.

[21] Liangde Li, Supun Nakandala, and Arun Kumar. 2021. Intermittent human-in-the-loop model selection using cerebro: a demonstration. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2687–2690.

[22] Side Li and Arun Kumar. 2021. Towards an optimized GROUP by abstraction for large-scale machine learning. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2327–2340.

[23] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. 2018. Ease. ml: Towards multi-tenant resource sharing for machine learning workloads. *Proceedings of the VLDB Endowment* 11, 5 (2018), 607–620.

[24] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3431–3440.

[25] Amiel Meiseles and Lior Rokach. 2020. Source Model Selection for Deep Learning in the Time Series Domain. *IEEE Access* (2020).

[26] Thomas Mensink, Jasper Uijlings, Alina Kuznetsova, Michael Gygli, and Vittorio Ferrari. 2021. Factors of influence for transfer learning across diverse appearance domains and task types. *arXiv preprint arXiv:2103.13318* (2021).

[27] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. 2019. Incremental and approximate inference for faster occlusion-based deep cnn explanations. In *Proceedings of the 2019 International Conference on Management of Data*. 1589–1606.

[28] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2159–2173.

[29] Cuong V Nguyen, Tal Hassner, Cedric Archambeau, and Matthias Seeger. 2020. LEEP: A New Measure to Evaluate Transferability of Learned Representations. *International Conference on Machine Learning* (2020).

[30] Maria-Elena Nilsback and Andrew Zisserman. 2008. Automated flower classification over a large number of classes. In *2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*. IEEE, 722–729.

[31] Laurel Orr, Atindriyo Sanyal, Xiao Ling, Karan Goel, and Megan Leszczynski. 2021. Managing ML pipelines: feature stores and the coming wave of embedding ecosystems. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3178–3181.

[32] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering* (2009).

[33] Neoklis Polyzotis, Martin Zinkevich, Sudip Roy, Eric Breck, and Steven Whang. 2019. Data validation for machine learning. *Proceedings of Machine Learning and Systems* 1 (2019), 334–347.

[34] Joan Puigcerver, Carlos Riquelme, Basil Mustafa, Cedric Renggli, André Susano Pinto, Sylvain Gelly, Daniel Keysers, and Neil Houlsby. 2021. Scalable Transfer Learning with Expert Models. *International Conference on Learning Representations* (2021).

[35] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment* 11, 3 (2017), 269–282.

[36] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1190–1201.

[37] Cedric Renggli, André Susano Pinto, Luka Rimanic, Joan Puigcerver, Carlos Riquelme, Ce Zhang, and Mario Lucic. 2022. Which model to transfer? finding the needle in the growing haystack. *IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2022).

[38] Luka Rimanic, Cedric Renggli, Bo Li, and Ce Zhang. 2020. On convergence of nearest neighbor classifiers over feature transformations. *Advances in Neural Information Processing Systems* 33 (2020), 12521–12532.

[39] 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*. 15–18.

[40] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. 2018. On Challenges in Machine Learning Model Management. *Data Engineering* (2018), 5.

[41] Sebastian Schelter, Stefan Grafberger, Shubha Guha, Olivier Sprangers, Bojan Karlaš, and Ce Zhang. 2022. Screening Native ML Pipelines with "ArgusEyes". (2022).

[42] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. 2018. A survey on deep transfer learning. *International Conference on Artificial Neural Networks* (2018).

[43] Anh T Tran, Cuong V Nguyen, and Tal Hassner. 2019. Transferability and hardness of supervised classification tasks. *IEEE International Conference on Computer Vision* (2019), 1395–1405.

[44] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 1–3.

[45] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[46] Zirui Wang. 2018. Theoretical Guarantees of Transfer Learning. *arXiv preprint arXiv:1810.05986* (2018).

[47] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *Journal of Big data* (2016).

[48] Renzhi Wu, Sanya Chaba, Saurabh Sawlani, Xu Chu, and Saravanan Thirumuruganathan. 2020. Zeroer: Entity resolution using zero labeled examples. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1149–1164.

[49] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. 2020. Complaint-driven training data debugging for query 2.0. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1317–1334.

[50] Xi Yan, David Acuna, and Sanja Fidler. 2020. Neural Data Server: A Large-Scale Search Engine for Transfer Learning Data. *CVPR* (2020).

[51] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.

[52] Amir R Zamir, Alexander Sax, William Shen, Leonidas J Guibas, Jitendra Malik, and Silvio Savarese. 2018. Taskonomy: Disentangling task transfer learning. *IEEE Conference on Computer Vision and Pattern Recognition* (2018).

[53] Xiaohua Zhai, Joan Puigcerver, Alexander Kolesnikov, Pierre Ruyssen, Carlos Riquelme, Mario Lucic, Josip Djolonga, Andre Susano Pinto, Maxim Neumann, Alexey Dosovitskiy, et al. 2019. The Visual Task Adaptation Benchmark. *arXiv preprint arXiv:1910.04867* (2019).

[54] Ce Zhang, Christopher Ré, Michael Cafarella, Christopher De Sa, Alex Ratner, Jaeho Shin, Feiran Wang, and Sen Wu. 2017. DeepDive: Declarative knowledge base construction. *Commun. ACM* 60, 5 (2017), 93–102.