# [Double-Column Extended Version] Alsatian: Optimizing Model Search for Deep Transfer Learning

Nils Strassenburg
nils.strassenburg@hpi.de
Hasso Plattner Institute
University of Potsdam
Germany

Boris Glavic
bglavic@uic.edu
University of Illinois, Chicago
USA

Tilmann Rabl
tilmann.rabl@hpi.de
Hasso Plattner Institute
University of Potsdam
Germany

## Abstract

Transfer learning is an effective technique for tuning a deep learning model when training data or computational resources are limited. Instead of training a new model from scratch, the parameters of an existing "base model" are adjusted for the new task. The accuracy of such a fine-tuned model depends on the suitability of the base model chosen. Model search automates the selection of such a base model by evaluating the suitability of candidate models for a specific task. This entails inference with each candidate model on task-specific data. With thousands of models available through model stores, the computational cost of model search is a major bottleneck for efficient transfer learning.

In this work, we present *Alsatian*, a novel model search system. Based on the observation that many candidate models overlap to a significant extent and following a careful bottleneck analysis, we propose optimization techniques that are applicable to many model search frameworks. These optimizations include: (i) splitting models into individual blocks that can be shared across models, (ii) caching of intermediate inference results and model blocks, and (iii) selecting a beneficial search order for models to maximize sharing of cached results. In our evaluation on state-of-the-art deep learning models from computer vision and natural language processing, we show that *Alsatian* outperforms baselines by up to ~14×.

**ACM Reference Format:**
Nils Strassenburg, Boris Glavic, and Tilmann Rabl. 2025. [Double-Column Extended Version] Alsatian: Optimizing Model Search for Deep Transfer Learning. In *Proceedings of ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

The increased accuracy of deep learning (DL) models compared to classical machine learning models has revolutionized many fields in research and industry. Nevertheless, developing a custom model from scratch is out of reach for many users due to limited availability of training data, limited computational resources, and a lack of experts with sufficient knowledge to develop new models. Deep transfer learning (DTL) [17], where a *base model* trained for one domain is fine-tuned for a different *target domain* using a training dataset $D_{target}$, has become a popular approach to address these challenges. DTL significantly reduces training time and requires less training data when compared to training from scratch [7, 17].

Finding the right base model is critical for DTL – it has been shown that fine-tuning a random or generic model typically results in poor performance while choosing a suitable model significantly improves model accuracy [41]. Therefore, a machine learning (ML) engineer tasked with developing a DL model for a labeled dataset $D_{target}$, ideally wants to automatically identify a suitable base model from a company-internal or a public model store. The selected model should follow a preferred architecture to ensure compatibility, ease integration, as well as leverage existing optimizations. Moreover, it should maximize the likelihood of successful transfer learning.

The amount of publicly available models has significantly increased in recent years [9]. On the one hand, this increases the chance that a suitable base model is available for DTL. On the other hand, keeping track of all available models is challenging even for ML experts. Automated *model search* systems have emerged as a potential solution for this problem. Model search has received significant attention from industry (e.g., Google/Google Brain [25, 30, 41], AWS [2, 11, 29, 36], and Facebook AI [36]) and the ML research community [4, 15, 16, 20, 42, 62, 63, 67]. These techniques automate the process of choosing a base model by evaluating the suitability of many models, such as those available in public model stores [21, 39, 56] for the target domain. Naive approaches that rely solely on model metadata are not effective for selecting a good base model [29, 41]. At the other end of the spectrum, fine-tuning all available models is prohibitively expensive. For example, Renggli et al. [42] report a runtime of more than 13 days for 100 models on a single GPU machine. The state-of-the-art are *feature-based model search* techniques [4, 29, 30, 41]. These methods first *extract features* for every model by applying the first blocks of the model (the so-called *feature extractor*) to the dataset $D_{target}$ and then *calculate a proxy score* for the suitability of the model for the new task by, for example, training a fully connected neural network layer [41] and measuring the accuracy of this proxy model on $D_{target}$.

Model selection based on proxy scores has been demonstrated to be effective for selecting suitable base models [41]. However, even though generating proxy scores is less costly than full fine-tuning, it remains a significant bottleneck in the model search process. This is also due to the large number of available models. For example, Hugging Face (HF) now hosts over one million public models [9]


*SIGMOD '25, June 22-27,2025, Berlin, Germany*

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

including hundreds to thousands of fine-tuned variants for popular architectures such as ResNet [18], ViT [13], and BERT [12].

Prior work proposes two techniques for reducing the runtime of model search: improving proxy scoring and speeding up feature extraction. Improved proxy scoring methods include using k-nearest neighbors classifiers [38, 42], heuristics [30], or recommendation models [29], to bypass the need to train a fully connected layer. However, feature extraction is the real bottleneck in model search, because it requires data preparation and inference with a large part of the model (see Section 3). To optimize feature extraction time, Renggli et al. [42] propose *successive halving* which progressively narrows the model search space using proxy scoring with subsets of $D_{target}$ of increasing size. Even with these optimizations, model search still takes multiple hours to days and it turns out that no matter whether successive halving is used or not, feature extraction is still the main bottleneck [20, 30, 62]. In early iterations (small subsets of $D_{target}$), model loading is the bottleneck while inference is the main bottleneck in later iterations (larger subsets of $D_{target}$).

Analyzing the characteristics of typical sets of candidate models, we observed substantial overlap between models, which allows for significant optimization of feature extraction. This potential has been ignored by prior model search systems. We analyzed 2,800 models from HF as well as best practices for transfer learning proposed in ML textbooks [7, 17, 66], tutorials [1, 6, 55], and recent research on transfer learning [22, 23, 27, 28, 31, 32, 35, 44]. A common thread is that many models are generated by fine-tuning popular architectures. While fine-tuning all layers of a model is a common approach, often only a subset of a model's parameters are adjusted leading to substantial overlap between the fine-tuned and base model.

In this paper, we address the challenge of efficiently managing and querying model data to improve the performance of the feature extraction step in model search. Drawing inspiration from closely related work in the database community [16, 35, 42] as well as result caching, multi-query optimization, and common subexpression elimination, we develop new techniques to exploit the overlap between models. These techniques can be used by any model search approach that utilizes feature extraction and also for other inference tasks with multiple, similar models. We implement these techniques in a model search system called *Alsatian* that applies three optimizations to speed up model search with successive halving: ① partial model access, ② caching of intermediate inference results, and ③ optimizing the search order.

Figure 1 shows an example model search using Alsatian. The candidate models are $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$, and $\mathcal{M}_4$, consisting of three blocks each. Note that these models share some blocks (e.g., $B_1$). To reduce storage requirements, Alsatian stores each block exactly once, no matter how many models the block belongs to. To calculate the proxy score for $\mathcal{M}_1$ we first read its blocks ($B_1$ to $B_3$) from Alsatian's model store and then apply the model to the current subset of $D_{target}$ (feature extraction). Alsatian caches the outputs (inference results) for blocks $B_1$ and $B_2$ as these results will be reused for $\mathcal{M}_3$. For $\mathcal{M}_3$, we only have to load $B_6$ and apply it to the cached result for $B_2$ to compute the feature extractor result for $\mathcal{M}_3$. That is, for $\mathcal{M}_3$ we ① avoid loading the whole model and ② inference with the full model. After calculating the proxy score for $\mathcal{M}_3$, we score models $\mathcal{M}_4$ and $\mathcal{M}_2$ in the same manner.

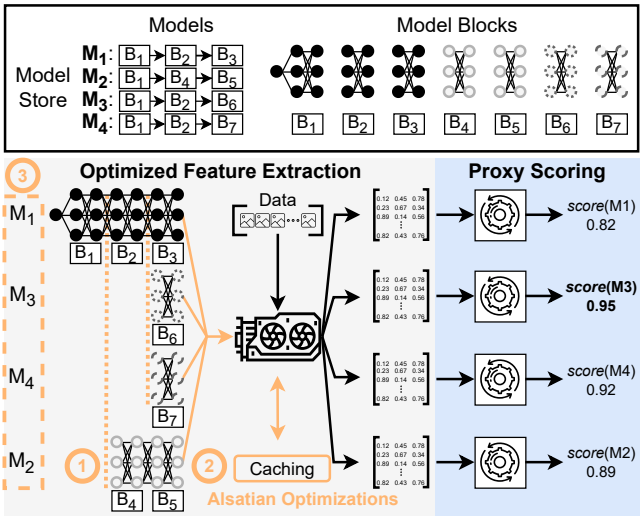



**Figure 1: Optimized feature based model search**

Note that if storage used for caching is limited, then a careful choice of search order (*execution plan*) can significantly impact performance. For example, assume that we can cache only one intermediate output at a time. Then using the search order $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$, $\mathcal{M}_4$, we have to compute $B_2$'s output twice. However, for the order $\mathcal{M}_1$, $\mathcal{M}_3$, $\mathcal{M}_4$, $\mathcal{M}_2$, $B_2$'s output is computed only once. Using metadata about the block structure of the candidate models, *Alsatian* determines a beneficial execution plan ③.

In summary, we make the following contributions:

(1) We analyze existing state-of-the-art model search techniques and identify I/O and compute as major bottlenecks.
(2) We propose a new approach that exploits the similarity between models by caching parts of models and caching (intermediate) inference results. These optimizations can be applied to any feature-based model search technique without changing its output.
(3) The problem of selecting a search order that minimizes execution time is computationally hard. We present an efficient heuristic algorithm that returns an optimal plan as long as the available space for caching is above a threshold that depends on the set of candidate models. Importantly, this condition often holds for real-world candidate models.
(4) We evaluate our approach implemented in Alsatian using synthetically generated as well as publicly available *computer vision* and *natural language processing* models and demonstrate that Alsatian outperforms the best baseline by up to 10.7×.

The remainder of this paper is structured as follows. In Section 2, we provide background on DTL and model search techniques, followed by an analysis of bottlenecks for model search in Section 3. In Section 4, we introduce our optimized model search approach Alsatian and then discuss its execution planner in Section 5. We evaluate Alsatian and compare it to existing approaches in Section 6, and then we give an overview of related work in Section 7. We conclude in Section 8. The source code for Alsatian and artifacts are publicly available.[1]

[1]https://github.com/hpides/alsatian

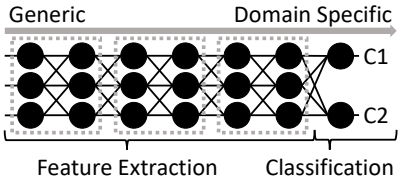



**Figure 2: Deep learning model divided into layers and blocks**

## 2 Background

In this section, we first describe how DL models are structured and give an overview of deep transfer learning (DTL). We then introduce feature-based model search and the successive halving approach we optimize in this work.

### 2.1 Deep Transfer Learning

***Structure of a DL Model.*** A model $\mathcal{M}$ is composed of blocks $B_1, \ldots B_m$ each consisting of one or more layers (e.g., transformer or bottleneck blocks [17, 18, 58]). For example, in PyTorch this hierarchical structure of models is typically achieved by nesting PyTorch modules to represent blocks and layers. Inference with a model on a dataset $D$ returns a prediction by evaluating $B_1$ on $D$, then feeding the result $B_1(D)$ into $B_2$ to compute $B_2(B_1(D))$ and so on. In domains such as computer vision and natural language processing, DL models can be logically divided into a feature extractor that is a prefix of the model's blocks and a problem-specific head [12, 13, 18, 43]. As we will discuss further in Section 6.3, the block structure of a model and what blocks belong to the feature extractor is typically explicitly declared (we validated this on ~ 2800 models from HF). For most models, the feature extractor makes up the largest part of the model. The head of the model typically contains one or more fully connected layers translating the output of the feature extractor into the actual prediction. Figure 2 shows an example of a DL model that consists of seven layers. The first six layers are grouped into three blocks and belong to the feature extractor while the last layer is the head which, in this example, predicts whether the extracted features represent an item in Class 1 or Class 2.

***Deep Transfer Learning.*** The layers of a DL model close to the input learn more generic features, while layers closer to the head learn more domain-specific features, as indicated in Figure 2 by the arrow above the model [61, 65]. This is why deep transfer learning (DTL) [17] reuses large parts of DL models to develop new models which speeds up training and requires significantly less training data. Two commonly used approaches for DTL are *feature-extraction* and *fine-tuning*. In both cases, we first select a *base model* to transfer parameters from and adjust its head for the new use case. This is done by freezing a certain number of blocks (i.e., the parameters of these blocks will not be adjusted during training) and training the model on the dataset $D_{target}$. For feature extraction, we freeze all layers of the feature extractor and only train a new head. For fine-tuning, we unfreeze parts of the feature extractor [7, 17, 40]. The optimal number of layers to tune is considered a hyperparameter for the training process and depends on many factors such as domain similarity, dataset size, and compute budget [35]. Freezing no or only a few layers allows us to adjust large parts of the model which might boost performance but increases the risk of overfitting and leads to slower training.

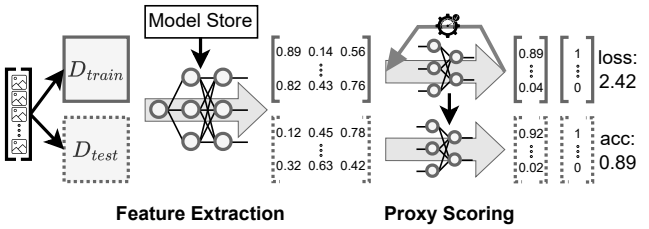



**Figure 3: Feature-based model scoring**

### 2.2 Model Search

***Model Search with Proxy Scoring.*** In model search, we are given a dataset $D_{target}$ for a target domain and a set of candidate models $\boldsymbol{M} = \{\mathcal{M}_i\}_{i=1}^{n}$. The goal is to select one of these models as a base model for transfer learning. The brute force approach for model search is to fine-tune every candidate model and select the fine-tuned model that performs best. This is prohibitively expensive. As we will explain in more detail in Section 7, model search techniques that rely on metadata alone are efficient, but often fail to identify a suitable base model. A technique that has been successfully employed in related work is to apply the *feature extractor* of each candidate model $\mathcal{M}_i$ to $D_{target}$ and then compute a *proxy score* to estimate the suitability of $\mathcal{M}_i$ as a base model for DTL. Different methods have been proposed for proxy scoring. For example, training a fully connected layer as a classifier on the feature extractor's output and using the classifier's accuracy as the model's proxy score [41].

Figure 3 shows an overview of the model search process with proxy scoring [42]. We split the target dataset $D_{target}$ into a training dataset $D_{train}$ and test dataset $D_{test}$. The feature extraction step consists of the following substeps:

- *Prepare data*: we load every data partition into memory and pre-process it to match the input dimensions of the model. For example, for images, this may require rescaling the image.
- *Prepare model*: We load every model's parameters from a model store into RAM and then move the model from RAM to GPU memory.
- *Inference*: we perform inference on the data using the feature extractor of the model to transform the training data $D_{train}$ and test data $D_{test}$ into feature matrices.

The *proxy scoring* step uses the feature matrices returned by the previous step to calculate a proxy score that estimates how well a fine-tuned version of the model performs on $D_{target}$.

- *Train a proxy model*: we train a fully connected layer as a proxy model using the features produced by the feature extraction step and the labels from $D_{train}$.
- *Evaluate proxy model*: we use the features produced for the test dataset $D_{test}$ by the feature extraction step to evaluate the proxy model's performance (e.g., [42] uses the model's accuracy as the proxy score).

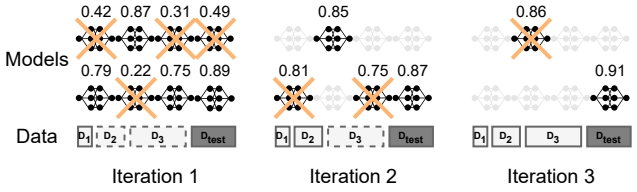



**Figure 4: Successive halving**

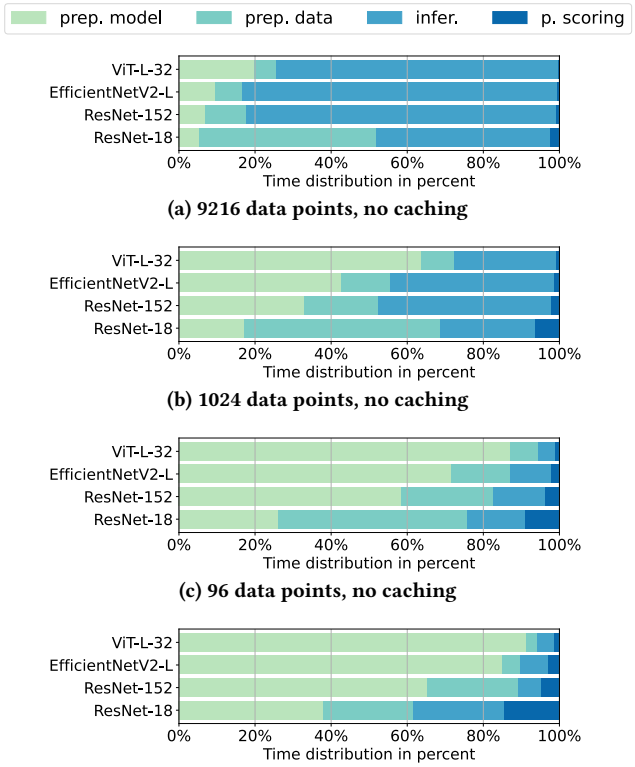
***Successive Halving and SHiFT.*** Most related work has focused on developing better or faster proxy scoring methods [20, 30, 62]. But, since feature extraction is the actual bottleneck, Renggli et al. [42] propose *successive halving* and implement this technique in their system called SHiFT. The system was evaluated on transfer learning tasks where all layers of the base model are fine-tuned, but the approach is also applicable to the other transfer learning variants discussed in Section 2.1. Successive halving prunes poorly performing models using proxy scoring with subsets of the data. We optimize successive halving in Alsatian.

Successive halving starts with the full set of candidate models $\boldsymbol{M}_0 = \{\mathcal{M}_i\}_{i=1}^{n}$ and prunes models until a desired number $k$ of models remains. The approach subsamples $D_{train}$ to create subsets $D_1, \ldots, D_l$ of exponentially increasing size (i.e., $|D_{i+1}| = 2|D_i|$). In each iteration $i$ of successive halving, the algorithm uses $D_{test}$ and $D_i$ to compute proxy scores for all models and filters out the $\frac{|\boldsymbol{M}_i|}{2}$ models from $\boldsymbol{M}_i$ with the lowest proxy scores. The result is the set of models $\boldsymbol{M}_{i+1}$ to be used in the next iteration. The initial dataset size $|D_1|$ is chosen such that in the last iteration the complete training dataset $D_{train}$ is used. This approach reduces the cost from $O(n \cdot m)$, where $n$ is the number of candidate models and $m$ is the size of $D_{train}$, to $O(\log n \cdot m)$.

Figure 4 shows an example with eight models. We start by splitting $D_{target}$ into four partitions: three partitions for training data $D_1$ to $D_3$ that double in size, and one partition $D_{test}$ for testing. In the first iteration, we extract features and compute proxy scores using the test data and the smallest training data partition $D_1$ to rank all models. Afterward, we prune the four models with the lowest proxy scores. In the second iteration, we compute proxy scores for the four remaining models using the smallest two training data partitions $D_1$ and $D_2$ and use the test data $D_{test}$ to compute scores and prune two more models. In the third iteration, we use all data ($D_1$ to $D_3$) for feature extraction and proxy scoring and prune one model leaving us with only one model. This model will be used for DTL.

## 3 Bottlenecks Analysis

To better understand the bottlenecks of model search with proxy scoring and the potential impact of caching (intermediate) inference results, we measure the runtime for computing proxy scores for a model in various settings. We use a server with a 64-core AMD Ryzen CPU with 2.48 GHz and an NVIDIA RTX A5000 GPU (for details see Section 6) and focus on the individual steps of: (1) preparing the model, (2) preparing the data including preprocessing such as resizing of images, (3) extracting features by performing inference on a target dataset, i.e., evaluating each block of the model on the output of the previous block or, in case of the first block on the target data $D_{target}$, and (4) calculating the proxy score by training and evaluating a fully connected layer over the extracted features for $D_{target}$ (which is split into $D_{train}$ and $D_{test}$ as described in Section 2.2). We vary the model architecture, the choice of which intermediate inference results to cache, and the dataset size by sampling from the ImageNet dataset [24].




**(a) 9216 data points, no caching**

**(b) 1024 data points, no caching**

**(c) 96 data points, no caching**

**(d) 1024 data points, reusing cached input for the 3rd last block**

**Figure 5: Runtime breakdown for scoring a single model**

***Effects of Dataset Size.*** We observe that the runtime for larger datasets (Figure 5(a)) is dominated by inference, especially for larger models such as ResNet-152, EfficientNetV2-L, and ViT-L-32. For smaller datasets, which are representative of early iterations of successive halving, the main bottleneck stems from preparing the model: for 96 data points as shown in Figure 5(c) and ResNet-152 (ViT-L-32) almost 60% (80%) of the time is spent on preparing the model. An exception are small models like ResNet-18 for which data preparation dominates, but even for ResNet-18 we observe that inference only has a significant impact for larger datasets.

***Caching.*** To simulate caching of intermediate inference results, we evaluated inference with the same models, but only for the last $n$ blocks of the feature extractor. Figure 5(d) shows the runtime distribution for preparing the data by loading cached intermediate inference results from SSD and reusing them so that we only evaluate the last $n = 3$ blocks of the feature extractor. Preparing the model is now the dominating factor: ~40% for ResNet-18 up to ~90% for ViT-L-32.

***Summary.*** The key insights of our bottleneck analysis are:

- Feature extraction (preparing the model, preparing the data, and performing inference) is the overall bottleneck.
- Without caching, inference is the bottleneck for large datasets and model preparation for smaller datasets. The exceptions are small models for which data preparation dominates.
- When caching intermediate inference results and, thus, only evaluating the last few blocks of a model's feature extractor, model preparation is the most significant bottleneck.

## 4 Alsatian: Efficient Model Search

In this section, we introduce the Alsatian framework. Based on the bottlenecks we identified for model search with proxy scoring, we optimize the performance of feature extraction through caching and the careful selection of a beneficial execution order without altering the model search result. We first present an overview of Alsatian in Section 4.1, discuss our model and data store in Section 4.2, and describe our execution engine in Section 4.3. Finally, we discuss Alsatian's planner in Section 5 that uses a simple, yet efficient, algorithm that produces optimal plans as long as sufficient memory is available for caching.

### 4.1 Alsatian Overview

Alsatian optimizes model search with successive halving as described in Section 2.2. We assume that we are given a storage budget $s_{cache}$ for caching and have a model store that allows access to models on a block-level granularity. A *model search query* consists of conditions for filtering models based on their metadata (e.g., consider computer vision models with less than 100M parameters), a target dataset $D_{target}$, and the result size, a parameter set by the user that dictates how many models should be returned as the final result of model search. The target dataset $D_{target}$ consists of labeled training examples for the task the returned models should solve, for example, a set of labeled images. Figure 6 shows a high-level overview of Alsatian. We start by passing the query and the dataset to the *model store* and the *data store* ①. The model store collects metadata for all models that fulfill the constraints and collects the initial set of models $\boldsymbol{M}_0$ ②. The data store splits $D_{target}$ into a test dataset $D_{test}$ and a training dataset $D_{train}$. It then determines the number of successive halving iterations based on the number of models in $\boldsymbol{M}_0$ such that the last iteration will return the requested number of result models. For that we create subsamples $D_1, \ldots, D_l$ of $D_{train}$ of exponentially increasing size; one for every iteration $i$ of successive halving ③.

***Successive Halving.*** In each iteration of successive halving, the *execution planner* requests metadata about the models $\boldsymbol{M}_i$ and the dataset $D_i$ to generate an execution plan for scoring these models using $D_i$. The execution planner generates a task tree (TT) for the current set of candidate models that encodes which block inference tasks have to be executed for feature extraction, the proxy scoring tasks, and all the dependencies between these tasks ④. Afterward, Alsatian generates an execution plan by traversing the TT ⑤. The execution engine evaluates the plan by iterating over the execution steps in the plan and performs operations such as accessing data chunks $D_i$ from the data store, reading model blocks from the model store, reading cached intermediate results for blocks or adjusting

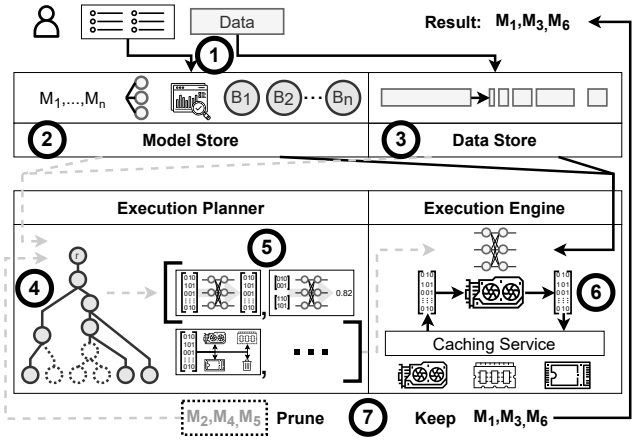



**Figure 6: Alsatian Overview**

the cache content using its caching service, extracting features, and calculating proxy scores ⑥. Based on the calculated proxy scores, the execution engine then prunes half of the models with the lowest proxy scores from the TT ⑦. The remaining models are the set of models $\boldsymbol{M}_{i+1}$ for the next iteration of successive halving. The process terminates once $|\boldsymbol{M}_i|$ is equal to the user-provided result size and Alsatian.

***Implementation.*** We implement Alsatian as a Python-based framework that uses PyTorch to execute inference and proxy scoring. We assume that models and blocks are represented as PyTorch *Modules*,[2] that the feature extractor is represented as a sequence of blocks using PyTorch *Sequential*[3] or *ModuleList*[4] objects, and that the model declares its feature extractor and classification head. For all models from PyTorch Hub and HF we analyzed, these conditions hold or can be achieved with minimal effort. In future work, we plan to extend our implementation to support arbitrary models by parsing their execution DAG.

As proposed in related work [42], we execute all core DL tasks (e.g., inference and training) on a GPU and all other tasks (e.g., execution planning and data preparation) on the CPU. We store $D_{target}$ on an SSD, and the models on an HDD. Analyzing inference time and intermediate result sizes on a model block granularity, we observe that there is no need to distinguish between GPU memory or CPU memory for caching. The reason is that accessing the cached intermediate from GPU memory or transferring it from CPU memory via the CPU-GPU interconnect is always significantly faster than recomputing the result. For example, the inference time of a single ViT-L-32 block is approximately 3 ms, but loading a batch of intermediate results from CPU to GPU memory takes only ~0.3 ms. We set the caching budget $s_{cache}$ for intermediate results to the sum of the system's GPU and CPU memory. Additionally, we cache model parameters on SSD to avoid having to repeatedly load them from HDD.

[2] https://pytorch.org/docs/stable/generated/torch.nn.Module.html
[3] https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html
[4] https://pytorch.org/docs/stable/generated/torch.nn.ModuleList.html

### 4.2 Model & Data Store

We store models as a sequence of blocks in the model store. For every block we record a series of characteristics: the block's architecture, the block's parameters, the block's output dimensions, the block's storage size, and hashes of the block's parameter and architecture forming a unique content-based identifier for a block. Overlap between models is determined based on the block hashes, and shared blocks are only stored once.

The data store provides access to chunks of the dataset $D_{target}$ and to complementary metadata such as the number of items in every chunk. To prepare the data, the data store requests the number of models that should be ranked and splits the data into multiple chunks: one test data chunk $D_{test}$ of a fixed size and multiple training data chunks $D_i$; one for every iteration of successive halving. The training data chunks are generated such that the size of chunk $D_i$ is twice the size of chunk $D_{i-1}$ and that the full training dataset is used in the last iteration of successive halving.

### 4.3 Execution Planner & Execution Engine

The execution planner (see Section 5) generates an execution plan $P$ consisting of a sequence of *execution steps* $P = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$. Each execution step falls into one of the three categories Block-Inference-Step (*BI*-Step), Score-Model-Step (*SM*-Step), or Modify-Cache-Step (*MC*-Step) and is executed by the execution engine.

A *BI*-Step performs inference for a sequence of model blocks $B_1$ to $B_k$. For such a step we record the input data partition of the first block in the sequence ($B_1$) and record where the output of $B_k$ should be cached. To execute a *BI*-Step, the execution engine prepares a model consisting of the model blocks $B_1$ to $B_k$, which are fetched from the model store. Afterward, the execution engine requests the input data from the data store or cached intermediate results from the caching service. Finally, it prepares the sequence of blocks, feeds the input data batch-wise through the blocks, and caches the output at the dedicated cache location. Note that the output of a *BI*-Step will be used at least once and, thus, has to be stored in the cache.

A *SM*-Step calculates the proxy score for a given set of extracted features (outputs of a block) and corresponding labels. We record what scoring method should be used (such as training and classification with a fully connected layer), a list of model IDs to score, and references to the relevant input features and labels. To execute a *SM*-Step, the execution engine requests all referenced training and test data features and labels from the caching service to train and evaluate the proxy model on the GPU. To avoid data preparation bottlenecks and fully utilize the GPU, the execution engine prepares input data and labels exactly once and caches them on the GPU or in CPU memory. This is possible even for datasets with multiple thousands of items because a single input feature uses less than 10KB and is thus relatively small compared to most intermediate inference results.

A *MC*-Step deletes intermediate inference results from the cache or moves such results to a different device (e.g., from CPU memory to SSD). To execute a *MC*-Step the execution engine forwards the request to the caching service that will move the data or mark it for deletion.

## 5 Execution Planner

The execution planner generates plans for a single iteration of successive halving. For each iteration, it generates a task tree (TT) encoding the block inference and proxy scoring tasks to be executed during this iteration and their dependencies. The planner then traverses the TT to create an execution plan $P = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ consisting of a sequence of execution steps $\mathcal{S}_i$ that is passed to the execution engine. We describe the generation of the TT in Section 5.1, our definition of an execution plan in Section 5.2, and present a heuristic algorithm to generate the execution plan in Section 5.3. In Section 5.4, we show that our algorithm is optimal when the caching budget $C$ is sufficiently large, and discuss how we handle the case when this condition does not hold.

### 5.1 Task Trees

***Task Tree Structure.*** Given a dataset, the TT $T = (V, E)$ is the data structure that represents all operations and their dependencies that we have to execute to extract features for a set of models and to calculate their proxy scores. Each vertex $v \in V$ of the TT represents an intermediate result and each edge $(v_1, v_2) \in E$ represents a computation (block inference or proxy scoring) that takes $v_1$ as input and returns $v_2$. We use $leaves(T)$ to denote all leaf nodes of $T$ which represent the proxy scores we have to compute. Additionally, we define a function $\textsc{mem} : V \to \mathbb{N}$ that associates each vertex (block inference result) with a storage cost; a function $\textsc{cost} : E \to \mathbb{N}$ that associates each edge (inference with a block or proxy scoring step) with a computational cost; and a function $\textsc{cmem} : E \to \mathbb{N}$ that associates each computation (edge) with a memory footprint. The execution planner will use the TT to determine an execution plan that minimizes the sum of the computation cost $\textsc{cost}$ for steps in the plan while ensuring that the total memory requirements of the cached intermediate inference results and storage costs for the computation (the blocks themselves) never exceed the cache size $s_{cache}$. Note that a block can be released from cache once its output has been produced, but outputs may be kept in cache for several execution steps. Furthermore, typically $\textsc{cmem}(B) \ll \textsc{mem}(R)$ where $R$ is the output of the block $B$. Thus, for simplicity, we ignore $\textsc{cmem}$ in the following.

***Building Task Trees.*** The two core operations the TT has to support are *adding* a new model (its metadata) and *pruning* a set of models. We add new models when collecting the models $\boldsymbol{M}_0$ relevant to our search query and we prune models after every iteration of successive halving.

Figure 7 shows an example of a TT constructed for three models $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$. On the top we show the models — each consisting of six blocks, some of which are shared. Below we show how these models are transformed into a TT. All TTs have a distinguished root node that represents the input data and have one edge for every block inference or proxy scoring task we have to process. Edges representing a block also store a set of model identifiers recording all models the block belongs to. In the middle of Figure 7, we show the situation where models $\mathcal{M}_1$ and $\mathcal{M}_2$ have been added to the tree already. Note how these two models share two nodes in the task tree for the two blocks ($B_1$ and $B_2$) they have in common. As model $\mathcal{M}_3$ shares a prefix of 4 blocks with $\mathcal{M}_2$, when adding this

model to the TT, new nodes are only created for the last two blocks of the model ($B_{11}$ and $B_{12}$).

To prune a model (e.g., $\mathcal{M}_2$ as shown in Figure 8) from the TT, we traverse the TT in breadth-first (BF) order and remove the model's identifier from all edges. Any edges whose set of models is empty after this step will be deleted as these block inference tasks will not have to be executed in any future iterations of successive halving.

## 5.2 Execution Plans

A TT for a dataset $D_k$ and set of models $\boldsymbol{M}_k$ records all tasks we have to execute in iteration $k$ of successive halving and their dependencies. As mentioned above, an execution plan $P$ consists of a sequence of steps $\mathcal{S}_1$ to $\mathcal{S}_n$. We use $C_i$ to denote the content of the cache after execution step $\mathcal{S}_i$. In each step $\mathcal{S}_i$ of an execution plan $P$ we either (1) compute the result $R_i$ of a block $B_i$ whose input $R_j$ is currently in the cache ($R_j \in C_i$) or which is applied to the input data $D_k$; (2) compute the proxy score based on a block's output $R_i$ that is currently in cache ($R_i \in C_i$); or (3) adjust the content of the cache by adding or dropping intermediate results. The cost of an execution plan $P$ with steps $\mathcal{S}_1, \ldots, \mathcal{S}_k$ is $\text{cost}(P) = \sum_{i=1}^{k} \text{cost}(\mathcal{S}_i)$. As proxy scores have to be computed exactly once for each model $\mathcal{M} \in \boldsymbol{M}_k$ in every execution plan for a TT $T$, we can ignore their cost when determining an optimal plan as it is common to every execution plan. Furthermore, as the cost of cache operations is neglectable compared to the cost of block inference steps, we ignore them. Thus, the cost of a plan is simply the sum of the costs of its block inference steps, say for blocks $B_1$ to $B_l$: $\sum_{i=1}^{l} \text{cost}(B_i)$. Note that we may have to execute a block inference step of a TT $T$ more than once if the result of the block is needed more than once and we do not have sufficient cache to keep the result of the block in cache until it is needed again.

An execution plan $P$ is *valid* if it produces proxy scores for all models in $\boldsymbol{M}_k$ and there does not exist a step $\mathcal{S}_i \in P$ where $C_i$ exceeds the cache budget $s_{cache}$. We call an execution plan *optimal* if it is valid and has the minimal cost among all valid plans. The minimum cache requirement $C_{min}$ of an execution plan $P$ is equal to the maximal size of the cache $C_i$ across all execution steps of the plan. Thus, $P$ is valid if $C_{min} \leq s_{cache}$. While generating an optimal execution plan that minimizes $C_{min}$ is NP-hard in general [3, 60], we present a simple heuristic algorithm and analyze its minimum caching requirements. We demonstrate that this algorithm is optimal when $s_{cache}$ is larger than the generated plan's minimum caching requirement $C_{min}$, independent of the computational costs of individual blocks. Fortunately, as we demonstrate experimentally, this condition often holds. Furthermore, we discuss how to reduce the caching requirement of a plan generated by our algorithm by splitting the input data $D_k$ into multiple chunks that are processed independently and argue that this is potentially more effective than applying a more expensive planning algorithm.

## 5.3 The DFS+ Optimization Algorithm

Our optimization algorithm which we call *DFS+* generates an execution plan using a depth-first search (DFS) traversal of the TT. *DFS+* starts at the root node and traverses one edge at a time (executes one block inference step) in DFS order. For any intermediate

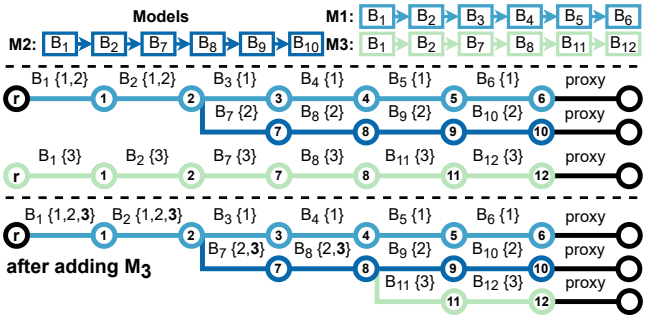



**Figure 7: Example of adding models to a task tree**

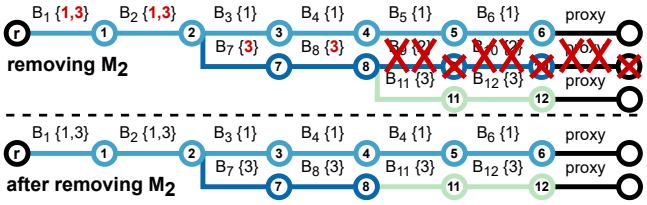



**Figure 8: Example of removing models from a task tree**

result $R$ (a node $v$) created by inference with a block $B$ (corresponding to the edge ending in $v$), we cache $R$ until all children of $v$ have been traversed (all blocks or proxy scoring steps that take $R$ as input have been executed). Once all children of a node $v$ have been traversed, we can release $R$ from the cache as it will not be needed again. *DFS+* traverses the children of a node in increasing order of their minimum cache requirement, i.e., the smallest possible $s_{cache}$ required to traverse the subtree rooted at the child with *DFS+*. We explain how to determine the minimum cache requirement for a node in the following. The rationale for using DFS order is that DFS first processes all nodes in a subtree before traversing to a sibling of a node and, thus, allows us to release the node from the cache as soon as possible. The motivation for processing children in increasing order of their minimum cache requirements (s) is that once the last child (the one with the highest minimum cache requirement) has been processed we can delete its parent from the cache to free up space to process the subtree with the highest minimum cache requirement.

***Determining Minimum Cache Requirements for Subtrees.*** We determine the minimum cache requirement for any subtree of a TT $T$ through a top-down traversal followed by a bottom-up traversal and another top-down traversal of $T$. During the top-down traversal, we accumulate the total memory required to produce the intermediate inference result corresponding to a node $v$ which requires caching all ancestors of the node $v$ and $v$ itself. For each node $v$ we memorize this cost. Intuitively, this cost will be the size of the cache right after reaching the node if we ignore that we can delete nodes from the cache after their last child has been processed. In the bottom-up traversal, we set the minimum cache requirement for each node to the maximum requirement in the subtree rooted at the node. At this point each node is associated with a cache requirement that is equal to the largest cache that will be created while traversing the subtree rooted at the node, still ignoring that nodes can be released once their last child in *DFS+* traversal order

has been processed. In the second top-down traversal, we adjust the cache requirements to take this into account. For that, the cost of the parent $v'$ of a node $v$ is removed from all nodes below $v$ if $v$ has the largest cache requirement among all children of its parent $v'$ as we can then release the parent right after processing $v$.

Figure 9 shows an example of these three steps. The TT contains intermediate block results 0 to 7 (the orange values shown on the left indicate their size). Figure 9(a) step (1): we traverse the TT using DFS and accumulate every path's cache requiremnt from the root to the leaves indicated as the numbers on the edges. For example, to reach node 7, we need to compute and cache intermediates 0, 1, 3, and 7 with a total cost of 9 + 4 + 2 = 15. Figure 9(b) shows step (2). We recursively propagate the maximum caching requirement of every sub-tree up to its root. For example, the minimum cache requirement propagated to intermediate 2 is 19 = $max(19, 17)$ and the minimum cache requirement propagated to intermediate 1 is also 19 = $max(19, 15)$. Figure 9(c) shows step (3). We adjust the cost of the descendents of every node $v$ that has the highest requirement among all its siblings. For instance, the requirement for node 4 is adjusted to 10 as we can release 1 with cost 9 from the cache immediately after materializing intermediate 2. The minimum cache requirement $C_{min}$ for an execution plan $P$ generated by *DFS+* is equal to the maximum minimum cache requirement among nodes in the task tree.

***Generating the Execution Plan.*** Continuing with the example shown in Figure 9(c), once the cache requirements have been determined and we have confirmed that $C_{min}$ for the plan $P$ will be less than the caching budget $s_{cache}$, we generate and execute the plan $P$. As mentioned above when traversing the tree to generate the plan, we always traverse the children of a node in increasing order of the minimum cache requirements, for example, we process intermediate 3 before intermediate 2. This leads to the overall traversal sequence of 0, 1, 3, 7, 6, 2, 5, 4 with a maximum caching requirement of 15. The reason why the budget does not exceed 15 when computing intermediates 4 and 5 is that once we computed intermediate 2, we can remove intermediate 1 from the cache because all paths depending on it already have been computed. This allows us to, for example, traverse from node 2 to node 4 with a budget of 10 instead of 19. We map the generated traversal order to an execution plan $P$ by adding one Block-Inference-Step or Score-Model-Step for every edge we traverse and a Modify-Cache-Step whenever we can release a cached intermediate.

## 5.4 Optimality of DFS+

Note that *DFS+* traverses each edge (block) exactly once. As any execution plan has to execute each block at least once and as the cost of a plan is the sum of its block execution costs, any plan produced by *DFS+* is optimal (has minimal compute cost) as long as it is valid (does not exceed the cache budget $s_{cache}$). Note that this argument does not rely on the computational costs for individual blocks meaning optimality is guaranteed independent of what the compute costs are. While we demonstrate in Section 6 that available memory for caching is typically sufficiently large, we nonetheless provide a solution that handles the case when the minimum cache requirement of the plan produced by *DFS+* exceed $s_{cache}$.

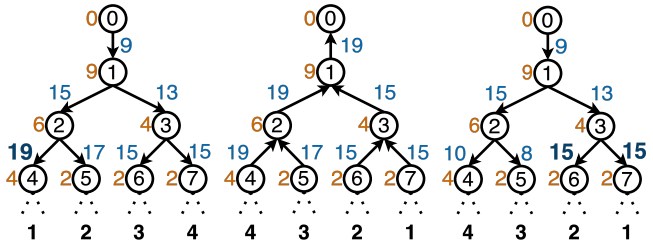



**Figure 9: Example for computing minimum cache requirements to determine a child traversal order for *DFS+*: (a) accumulate size of the cache used to traverse a path from the root to a node using a top-down traversal; (b) determine the minimum cache requirement for each subtree as the maximum cache size in the subtree; (c) adjust the minimum cache requirements of edges ending in a node $v$ whose parent has the highest cache requirement among all its siblings by removing the storage size of the parent from all edges below $v$. Intermediate storage costs are shown in orange, the calculated costs in blue, the maximum is marked in bold, and the traversal order is shown in black below the paths.**

Instead of resorting to a more expensive planning algorithm, we deal with plans exceeding the cache size by splitting the input data for the TT into smaller chunks as the size of intermediates is proportional to the input data size. If the plan for a data chunk $D_k$ exceeds the cache budget, we execute the execution plan $i > 1$ times, each time over a partition of $D_k$ of size $d_i = |D_k| \cdot (1/i)$. This reduces the cache requirement by a factor $d_i$, but is associated with the overhead of preparing each involved model blocks $i$ times. However, because the consecutive runs are executed without any interruption, most model blocks are likely cached by the OS and as a backup, we also cache them on SSD. That is, the additional overhead introduced by partitioning $D_k$ is small as long as each partition is sufficiently large. However, this is often the case exceeding the caching budget $s_{cache}$ is more likely to happen in late iterations of successive halving as the dataset size and, thus, also the cache requirements are proportional to the size of $D_k$ which increases exponentially in the number of iterations. Even though the number of models decreases exponentially over the iterations this also decreases the potential for sharing which in turn often causes the growth in data size to outweigh the decrease in the number of models.

# 6 Evaluation

In this section, we evaluate Alsatian and compare its performance to several baselines. We first describe our evaluation setup. Afterward, we analyze Alsatian's end-to-end search time in Section 6.1 to Section 6.3 across different computer vision model architectures, model candidate sets, and dataset sizes. In Section 6.4 we investigate how Alsatian performs with limited memory and in Section 6.5 we evaluate the model search time for models in the natural language processing domain. We conclude the section with a discussion of our results in Section 6.6.

***Hardware and Software.*** We evaluate our system on a server with an AMD Ryzen 3995WX 64-Core CPU with 2.48 GHz, two NVIDIA RTX A5000 GPUs, 64GB RAM, a SAMSUNG MZVL21-T0HCLR-00BL7 SSD, and a RAID 5 setup with three WDC WD-10EZEX-08W HDDs. Experiments are run inside a docker container based on the nvidia/cuda:11.3.0-devel-ubuntu20.04 image with docker version 24.0.2, Python 3.8, and PyTorch 2.2.0. We limit the container's resources to one GPU, half of the server's CPU cores, and the read speed of the model store to 200MB/s. We start with cold cache and execute 3-5 runs. The variance of our measurements is so low that we report median values, but no confidence intervals.

***Baselines.*** We compare Alsatian with two baselines presented by Renggli et al. [42]. The first one iterates sequentially through all models, performs feature extraction, and uses a fully connected layer for proxy scoring. We refer to this approach as *Base*. The second approach follows the same pattern as *Base* but applies successive halving. We refer to this approach as *SHiFT* even though the SHiFT approach, as presented by Renggli et al. [42] adaptively decides between *Base* and successive halving based on a cost model. We implement Alsatian as a standalone system built on top of PyTorch. In contrast, *SHiFT* is implemented as a server-client system that distributes computations (inference and proxy scoring) and executes them in docker containers. To guarantee a fair comparison, we re-implement *Base* and *SHiFT* as a standalone PyTorch based system.

***Models.*** We use three sets of candidate models for our evaluation. Models in $\mathcal{M}_{syn}$ use architectures from real world models with synthetically generated parameters; $\mathcal{M}_{train}$ are models using real architectures trained on real datasets; and $\mathcal{M}_{HF}$ are publicly available models from Hugging Face (HF).

For $\mathcal{M}_{syn}$, we focus on the architectures listed in Table 1. The first three models, ResNet-18, ResNet-152, and EfficientNetV2-L, are convolutional image classification models commonly used in related work on model search [25, 29, 30, 36, 41, 42, 62]. The last two models, ViT-L-32 and BERT, are popular transformer-based architectures for image and text classification, respectively. We generate $\mathcal{M}_{syn}$ by adding models for every architecture as follows: We first add a model pre-trained on ImageNet to the model set. We then generate a new model by randomly picking a base model from the already existing models, copying its parameter into a new model, sampling a random number $n$ of blocks to fine-tune from a distribution, randomly initializing blocks marked for fine-tuning, and add the new model to our model set. We consider three different truncated normal distributions for $n$ representing the scenarios of ~50% of retrained layers (*Top 50%*), ~25% of retrained layers (*Top 25%*), and fine-tuning the last few layers (*Top Layers*) that were mentioned frequently in related work [19, 22, 23, 27]. These *synthetic models* allow us to gain detailed insights since they differ from *real models* only in their parameter values but neither in their model architecture nor in their number of parameters which guarantees equivalent computational demands for every substep of feature extraction and proxy scoring.

To collect models for $\mathcal{M}_{train}$ we follow the same procedure as for $\mathcal{M}_{syn}$ but fine-tune the models for 20 epochs using an Adam optimizer with a learning rate of 0.001. As data we randomly pick one of the ImageWoof [14], Stanford Dogs [24], Stanford Cars [26], CUB-200 Birds [59], or the Food 101 [5] datasets for every fine-tuning process.

**Table 1: Set of selected model architectures with the number of parameters, and model size in MB.**

| Name | #Parameters | Size | Reference |
|---|---|---|---|
| ResNet-18 | 11,689,512 | 46.8 MB | [18] |
| ResNet-152 | 60,192,808 | 241.7 MB | [18] |
| EfficientNetV2-L | 118,515,272 | 476.7 MB | [45] |
| ViT-L-32 | 306,535,400 | 1,226.3 MB | [13] |
| BERT | 109,482,240 | 438.0 MB | [12] |

For $\mathcal{M}_{HF}$, we collect over 2800 models from HF across 28 architectures for the task categories of image classification, object detection, image feature extraction, and image-to-text. For every task category, we consider all models where its base model appears on the first two result pages (the most popular models based on downloads) and has more than 5 registered fine-tuned variants[5].

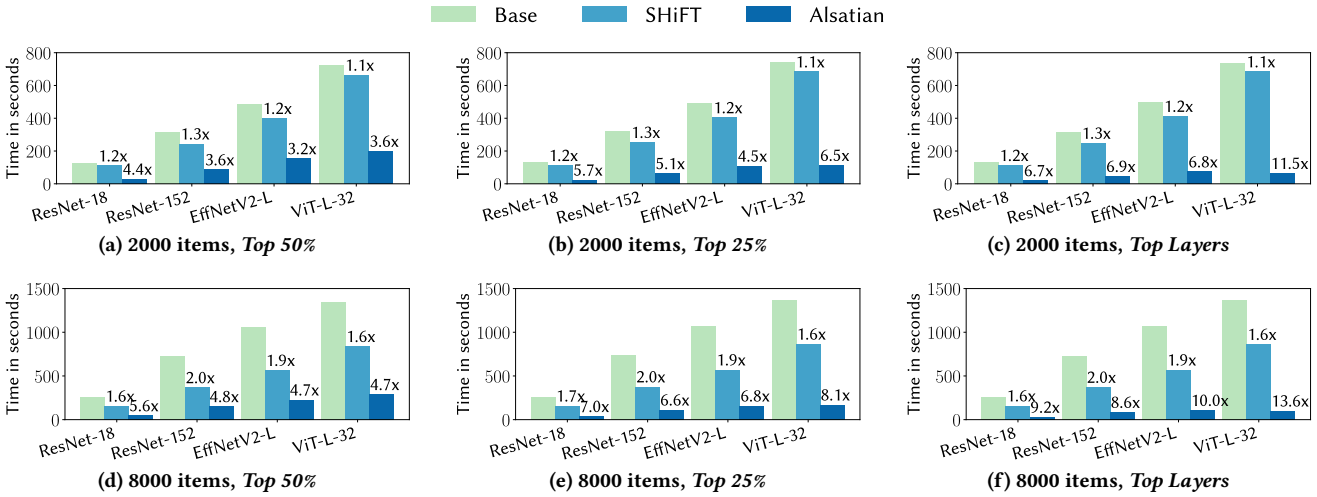
## 6.1 End-to-end Search Time

In this section, we use $\mathcal{M}_{syn}$ and analyze the end-to-end time of Alsatian compared to *Base* and *SHiFT* for searching through 35 models. We vary the model architecture, the model similarity distribution, and the number of items in the target dataset by sampling subsets of the ImageNet dataset [24]. The left column of Figure 10 shows the runtime of model search over 2000 data items for several model similarity distributions. The right column of Figure 10 shows the same setting but with 8000 data items. Figure 11 shows a time breakdown for a subset of the scenarios for the steps of *prepare data*, *prepare model*, *inference*, and *proxy scoring*. **Overall we observe that Alsatian outperforms the baselines for all settings with improvements of up to 13.6× compared to *Base* and 10.7× compared to *SHiFT*.**

***Model Similarity Distribution.*** Figure 10 shows that for a given architecture and a constant number of data items, Alsatian's search time drastically decreases with more overlap between the models while *Base*' and *SHiFT*'s performance are not affected. Comparing the numbers for 2000 data items in Figure 10(a), Figure 10(b), and Figure 10(c) observe that *SHiFT* is only up to 0.3× faster than the baseline and its performance does not improve when models are more similar. In contrast, Alsatian already outperforms *Base* and *SHiFT* by 3-4× for *Top 50%* and is more than one order of magnitude faster for *Top Layers*. The trends for 8000 data items shown in Figure 10(d), Figure 10(e), and Figure 10(f) are similar, with the difference that *SHiFT* outperforms *Base* by up to 2×.

The reason for *Base*'s and *SHiFT*'s constant performance across different model similarity distributions is that both approaches process every model individually and do not exploit model overlap. Comparing the results from Figure 11(a) with Figure 11(b), we observe that the cost of the substeps is not affected by model overlap. In contrast, Alsatian plans the execution to maximize caching and reuse of intermediate computations, reducing the time to prepare the data, to prepare the model, and to perform inference. As shown

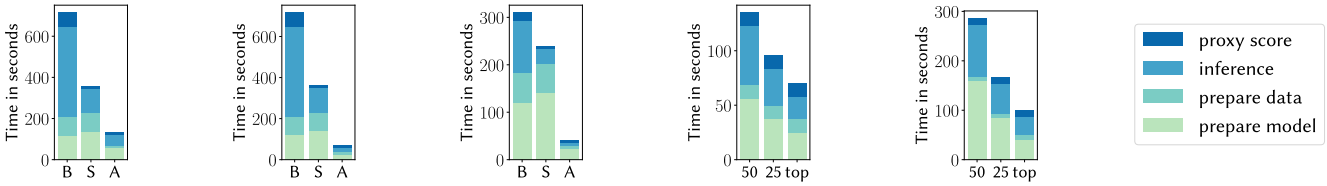
[5]For a detailed list of model architectures see our repository.




(a) 2000 items, *Top 50%* (b) 2000 items, *Top 25%* (c) 2000 items, *Top Layers*

(d) 8000 items, *Top 50%* (e) 8000 items, *Top 25%* (f) 8000 items, *Top Layers*

**Figure 10: End-to-end times for searching through 35 models across different architectures and number of items in the dataset.**




(a) ResNet-152, 8000 items, *Top 50%* (b) ResNet-152, 8000 items, *Top Layers* (c) ResNet-152, 2000 items, *Top Layers* (d) Alsatian, ResNet -152, 8000 items (e) Alsatian, ViT-L-32, 8000 items

**Figure 11: Time breakdowns for different configurations across approaches number of items, and model similarity distributions. B - *Base*, S - *SHiFT*, A - Alsatian; 50 - *Top 50%*, 25 - *Top 25%*, top - *Top Layers***

in Figure 11(d) and Figure 11(e), the performance of these three substeps improves when model similarity is increased. The only step with constant cost is proxy scoring because the number and size of the extracted features are not affected by our optimizations and the results of proxy scoring cannot be reused.

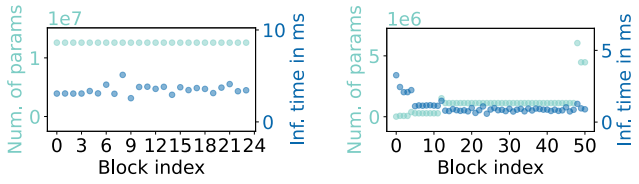
***Dataset Size.*** Next, we analyze the impact of the size of $D_{train}$ on performance. Both *SHiFT* and Alsatian scale better to large datasets than *Base*. When we increase the dataset size from 2000 to 8000 for the scenario of *Top Layers* as shown in Figure 10(c) and Figure 10(f), the absolute model search time increases for all methods. For 2000 items *SHiFT* is slightly faster than *Base* and for 8000 items *SHiFT*'s speedup is 1.5-2×. For Alsatian, the improvements over *Base* are up to 11.5× for 2000 items, and more than 13.6× for 8000 items.

The reasons for the increased cost of *Base* is evident from the breakdowns in Figure 11(b) and Figure 11(c). The dataset size increases the input size for data preparation, inference, and proxy scoring. This scales these subsetps proportionally to the dataset size while the time to prepare the model stays constant.

*SHiFT* and Alsatian scale better to larger datasets than *Base* because they use successive halving. This reduces the inference time but requires repeatedly preparing models or parts of models and loading them to DRAM and GPU memory. As shown in Figure 11(c), for small datasets *SHiFT*'s improvement of inference time is small which can only partially compensate for the overhead in model preparation. In contrast, as shown in Figure 11(b), for larger datasets the inference time gets proportionally larger which compensates for model preparation overheads. These effects were also observed by Renggli et al. [42]. Alsatian has lower model preparation overhead because it only loads and prepares parts of models. This reduces the model preparation time and increases the likelihood of model parts being available in the OS cache.

***Model Architecture.*** Evaluating how the model architecture affects search time, we observe that: (1) Searching through larger models takes longer. (2) The relative speedups for the *Top 50%* distribution are largest for the convolutional architectures, but for the *Top 25%* and the *Top Layers* distribution most significant for the transformer-based architecture ViT-L-32.

The reason for (1) is that larger models have more parameters, perform more complex computations, and have larger features. This increases the model preparation, inference, and proxy scoring time. To explain (2), we analyze the number of parameters and inference time for our models at the granularity of individual blocks. As shown in Figure 12(a), the number of parameters and the inference time is close to constant across all blocks for the ViT-L-32 architecture. However, for convolutional models such as the ResNet-152 the first few layers are most expensive in terms of inferences. The number of parameters is low for early layers, constant for most of the middle blocks and very high for the last few blocks (Figure 12(b)). As shown in Figure 11(e) and Figure 11(d), for both architecture




(a) ViT-L-32 (b) ResNet-152

**Figure 12: Number of parameters and inference time per architecture block.**

types the inference time decreases by a similar amount with more similar models. For the ViT-L-32 architecture the "prepare model" time decreases proportionally with model overlap, for ResNet-152 the "prepare model time" decreases less because the last few blocks are not shared and are the largest blocks in terms of parameters.

## 6.2 Real Models

We now evaluate how the findings for $\mathcal{M}_{syn}$ translate to real models by picking 2000 data items, a *Top 25%* model similarity distribution, and the candidate model set $\mathcal{M}_{train}$.

The results for synthetic and real models in Figure 10(b) and Figure 13 are similar. The reason is that the synthetic and real models do not differ in computational complexity and number of parameters. The experiments with synthetic and real models only differ in pruning decisions. For *Base* and *SHiFT* we see an equivalent search time for both settings because both approaches process every model individually and are thus not affected by pruning decisions. For Alsatian, we sometimes see a faster and sometimes a slower search time for real models. The reason is that different pruning orders lead to different amounts of overlapping parameters which has a direct impact on the model preparation and inference time and thus on the search time.

## 6.3 Hugging Face Models

We use $\mathcal{M}_{HF}$ to evaluate our approach on ~2800 real models from HF and to validate that (i) publicly available models are structured in blocks and this structure can be automatically detected without human intervention, (ii) there is significant overlap between model, and (iii) Alsatian significantly improves the end-to-end search performance for these models.

***Explicit Declaration of Model Architecture.*** All models in $\mathcal{M}_{HF}$ explicitly declare their feature extractor as a *backbone* or *encoder* component. These feature extractors are structured in blocks defined as *PyTorch Modules* and are combined into a larger model using either a *PyTorch ModuleList* or a *PyTorch Sequential* object (as for the models on PyTorch's website) making it easy identify the model's structure and access individual model blocks.

***Model Overlap.*** We observe that freezing parts of a model during fine-tuning, as suggested in textbooks and related work [17, 22, 23, 27, 28], is common in $\mathcal{M}_{HF}$. For object detection models, the most common approach is freezing the initial blocks of the image feature extractor, followed by training only the final layers or using custom freezing scheme. For example, for Facebook's DETR-50

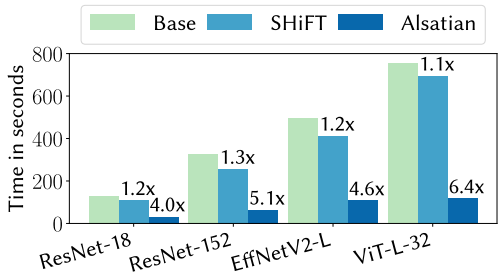
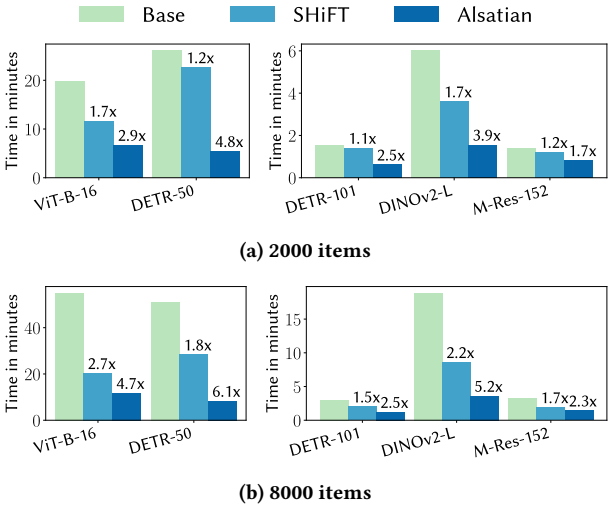



**Figure 13: Search over 35 models per architecture and 2000 items, *Top 25%* model similarity distribution, and use five different datasets for fine-tuning.**

architecture, 87% of the 420 models have significant overlap with the base model [47]. For 84% the first blocks of the image feature extractor are frozen, for 3% only the last few layers of the feature extractor are adjusted. For Microsoft's conditional object detection model and a variation of Facebook's DETR-50, 90% and 94% of the available models have significant overlap [46, 48]. For image classification, feature extraction, and image-to-text models, we see less strict fine-tuning patterns and block overlap. Most training methods fine-tune the entire model or a large portion, followed by approaches that train only the final layers or use a custom number of frozen blocks. As a result, for models like *TrOCR* or *YOLOS-small*, we find few instances with overlap [51, 54]. However, we also find cases like *ResNet-18*, *ResNet-152*, and *DINOv2-L*, where for up to 63% of the models, only the last layers were fine-tuned [49, 50, 53].

***Model Search.*** We select a subset of $\mathcal{M}_{HF}$ with ~500 model across twelve model architectures, focusing on architectures similar to the ones from $\mathcal{M}_{syn}$ (such as vision transformers and ResNet-based architectures [49, 50, 52]) and some new architectures (such as DINOv2, and detection transformer (DETR) models from Facebook and Microsoft [46, 47, 53]). For ten of the architectures, we have less than 100 models and include all of them. For the large ViT architecture we use ~100 models, and for the smaller object detection architecture, approximately ~200 models. We evaluate two scenarios: a per-architecture search similar to the previous experiments and searching through all 500 HF models. In both cases, *Alsatian* outperforms *Base* and *SHiFT*.

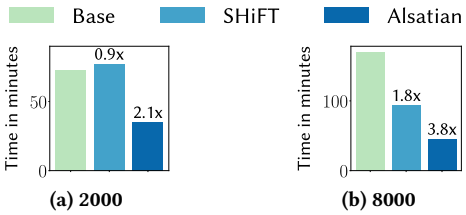
Figure 14 shows a subset of the single architecture search results. For a dataset size of 2000 (Figure 14(a)), *Alsatian* improves *Base* from 1.7× for the HF ResNet-152 up to 4.8× for DETR-50. Similarly, *Alsatian* improves *SHiFT* between 1.6× and 4×. For dataset size 8000 (Figure 14(b)) *Alsatian*'s improvements over *Base* is up to 6.1×, and slightly decrease compared to *SHiFT* with an improvement of up to 3.4×. The variation in improvements is caused by the variation in model overlap across the architectures. For the ResNet-152 architecture, there are fewer identical blocks and, thus, less opportunity for sharing results. For DINOv2 and DETR-50 there are many shared blocks leading to better performance for *Alsatian*. For the larger target dataset (8000 data points) inference is the bottleneck, reducing the impact of repeatedly loading models during successive halving. This explains the increased performance of *Alsatian* and *SHiFT* compared to *Base*.




**(a) 2000 items**

**(b) 8000 items**

**Figure 14: Search over Hugging Face models per architecture.**




**(a) 2000**

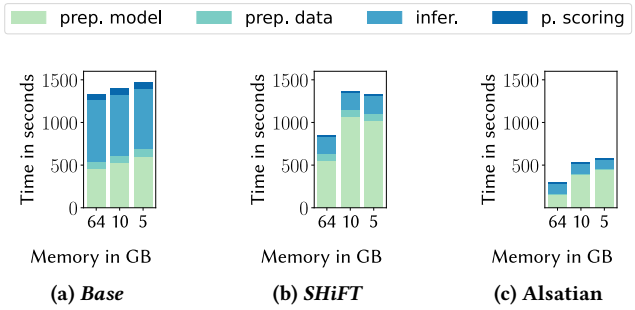
**(b) 8000**

**Figure 15: Search over approximately 500 Hugging Face models of different architectures.**

Figure 15 shows that *Alsatian* outperforms *Base* and *SHiFT* when searching over all HF models: approximately 2.1× for 2000 items and up to 3.8× for 8000 items in Figure 15(b). The reason for *SHiFT*'s poor performance and *Alsatian*'s smaller advantage compared to single architecture search experiments is that the combined model parameter size of all models exceeds the caching budget for model caching. Consequently, both *Alsatian* and *SHiFT* load a subset of the model parameters multiple times from storage. Since the relative time spent on model loading is much larger for scenarios with little data, this effect is stronger for 2000 items than for 8000 items.

## 6.4 Memory Budget Experiments

To evaluate how the methods perform with limited memory, we choose $\mathcal{M}_{syn}$'s most memory-intense workload – a search over 35 ViT-L-32 models following the *Top 50%* distribution – and execute it on the same setup as before while limiting the memory to 64GB, 10GB, and 5GB and prevent swapping. For the execution, we limit the number of dataloader workers (which have a significant memory footprint) to three. This is to have the same setup across configurations and to guarantee that *Base* and *SHiFT* do not run out of memory for the 5GB configuration. We further assume $S_{\text{cache}}$ is sufficient to store all final model outputs. The reason is that neural networks usually transform high-dimensional data into lower-dimensional features and while intermediate results for




**(a) *Base*** **(b) *SHiFT*** **(c) Alsatian**

**Figure 16: Model search for 35 ViT-L-32 models following *Top 50%* distribution using limited available memory for caching.**

blocks early in the sequence can be large, the final model output is typically small (e.g., 256KB with batch size 32 for ResNet-152).

Figure 16 shows a breakdown of the search time for the same set of 35 ViT-L-32 models varying the memory limit. Regardless of the memory budget, Alsatian outperforms the other approaches significantly even though the models we search through overlap by less than 50%. Figure 16(a) shows that, as expected, *Base* is least affected by reducing the memory budget. *Base* does not actively cache any intermediates while the amount of input data and the complexity of the computation stays constant. Thus, the time to prepare the data, to perform inference, and to calculate the proxy score are almost constant regardless of the memory budget. The slight increase in end-to-end time stems from an increase in model preparation time. *Base* accesses every model twice (once for training and once for testing) and first loads the model into CPU memory and from there to the GPU. For the 64GB configuration there is enough capacity for data preparation and to cache the entire model. But with less memory, once the model is on the GPU, most of the CPU memory is used for data preparation. This prevents the OS from caching the entire model and slows down preparing the model for the second time.

*SHiFT* and Alsatian are both more affected by a limited memory budget than *Base*. The time needed to prepare the data, to perform inference, and to calculate the proxy score are almost constant, but the time to prepare the model increases. The reason for the increase is – same as for *Base* – that with limited memory the OS can cache only a limited amount of models. With *SHiFT* and Alsatian both iterating over many models multiple times these approaches are much more affected by a limited memory budget than *Base*. Even though Alsatian and *SHiFT* access the same models. Alsatian is less affected by limited memory because *SHiFT* always accesses entire models while Alsatian loads models partially and also implements caching of blocks on SSD.

The fact that *SHiFT* is faster with 5GB of memory compared to 10GB of memory can be explained by a detailed analysis of the individual iterations of successive halving. The 5GB configuration is only faster for the first two iterations of successive halving. For the remaining iterations, the 10GB variant is faster than the 5GB variant. Looking at the time distribution per execution step, with 5GB of memory, PyTorch's data loaders cache the input data less aggressively than with 10GB which slightly slows down the data

preparation step. This has the effect that the 10GB variant has less I/O budget for loading the model from persistent storage during the first successive halving iterations, which increases the time to prepare the models more than it saves by caching more input data. Later, once many models are pruned from the search space, larger parts of the models fit in the cache, and preparing the model is not the bottleneck anymore because of the increased amount of data per model.

## 6.5 Language Models

To see how the methods perform in other domains we extend our evaluation to natural language processing. We chose the BERT [12] model from $\mathcal{M}_{syn}$ which uses a transformer architecture and is, thus, conceptually similar to modern large language models.

Analog to the previous analysis of vision models, we consider three model similarity distributions (*Top 50%*, *Top 25%*, and *Top Layers*) and run a search over 2000 and 8000 items of the Large Movie Review sentiment classification dataset [33]. Figure 17 shows the search time for 2000 items.

We observe similar trends as in the previous experiments on vision models. In particular, the behavior is similar to that of ViT-L-32 which has the most similar architecture to BERT. *Base*'s and *SHiFT*'s performance is not influenced by a change in model similarity distribution because these approaches compute models individually. *SHiFT* is slightly faster than *Base* because it can prune poorly performing models early while Alsatian outperforms *Base* by more than 13.8×.

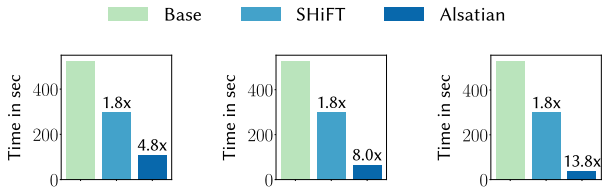
## 6.6 Discussion

We have shown that Alsatian effectively eliminates bottlenecks in feature-based model search. Alsatian outperforms related work by more than an order of magnitude and shows significant improvements even with highly restricted memory resources. Importantly, our approach does not alter the results produced by feature extraction and can be adopted by any existing feature-based model search system.

Despite the growing popularity and availability of foundation models, model search remains a critical task for two key reasons. First, foundation models are often fine-tuned for custom tasks, creating many similar models. Thus, efficient model search is essential to select the most promising candidates for DTL. Second, for simple and frequently repeated tasks, smaller and more specialized "traditional" models often match or even surpass foundation models in terms of accuracy, while requiring significantly fewer resources [8]. This creates a strong incentive to focus on creating such specialized models.

One direction to extend Alsatian is to support distributed execution of feature extraction. A simple way is to add execution steps to a queue and let multiple accelerators or servers work on processing the queue which is also what Renggli et al. propose [42]. A more sophisticated approach is to partition the search workload based on the task tree. By distributing an entire subtree of the task tree to a single accelerator or sever, we minimize duplicated caching of intermediates and model parameters.

Another direction for future work is auto-tuning hyper-parameters for inference. Currently, Alsatian uses a fixed configuration




**(a)** ***Top 50%*** **(b)** ***Top 25%*** **(c)** ***Top Layers***

**Figure 17: Search over 35 BERT models using 2000 items**

for parameters like batch size, number of workers, and successive halving that result in good performance for our hardware setup. In the constrained memory experiment, the number of workers and the batch size impact data preparation and inference time either directly through a higher degree of parallelism or indirectly by allocating or freeing resources for caching. Tuning these parameters on the fly for new target systems or using historic execution traces has the potential to further speed up feature extraction and model search.

# 7 Related Work

***Model Search.*** The ML community has proposed several methods for estimating the transferability of models and their feature extractors to new tasks. Most baselines are metadata or feature-based. Metadata-based methods include heuristics that use the model's performance on a fixed set of datasets as a predictor for the model's performance on a new dataset [41] or the number of parameters in combination with the number of items in the dataset to determine whether a model is suitable for a dataset [4]. Feature-based method follow the two-step process of feature extraction and proxy scoring as described in Section 2. Several proxy scores have been proposed including using a linear classifier of fully connected neural network layer [30, 42], logistic classifier [4], KNN classifier [4, 42], applying recommendation models [29], and metrics using the Pearson product-momentum correlation in combination with the Spearman correlation [4], Gaussian mixture models [30], or the logarithm of maximum evidence [62]. Our optimizations can be applied by any model search framework that uses proxy scoring.

***Model Search Systems.*** To the best of our knowledge, the only two model search systems are SHiFT [42] and Sommelier [16]. We already described SHiFT in Section 2.2 and used it as a baseline in our experimental evaluation. Sommelier indexes models based on their resource consumption and functional equivalence in terms of prediction performance to answer queries like: "given a ResNet model find a model that consumes 20% less resources and whose accuracy is no more than 5% less than the reference model". Answering this type of query is an orthogonal to the problem we solve in this work.

***Caching Intermediates for ML Pipelines.*** Caching and reuse of intermediate results for ML pipelines has been studied in [10, 35, 37, 57, 60]. Closest to our work is Nautilus [35]. Nautilus is a system that optimizes the parallel training of deep learning models for transfer learning by identifying and eliminating redundant computations. The authors first fuse different models into a multi-model graph. This inspired us to build our task tree which acts as an

index structure on top of the models in the model store. However, Nautilus uses an integer linear program (ILP) to decide what intermediates of the fused models to materialize and greedily combines models to maximize GPU utilization which does not fit searching through deep learning models with successive halving.

***Model Stores.*** Model stores like PyTorchHub [39] or HuggingFace [21] as well as ML lifecycle management tools like MLFlow [64] or ModelDB [57, 64] provide access to models but do not support feature-based model search. They use the deep learning framework's default model serialization formats which are not designed for fast fine granular access to model layers as needed for Alsatian. MMlib [44] or ModelHub [34] include more sophisticated ways of saving and accessing models but optimize in directions that are orthogonal to our work. MMlib focuses on fast model archival and a reduced storage footprint while sacrificing fast model loading. ModelHub optimizes for reducing the storage footprint for a set of models by using compression and finding optimal spanning trees leading to increased access times for full precision models.

## 8 Conclusion

In this paper, we present Alsatian, a system that optimizes the execution of feature-based model search techniques for deep learning models. Alsatian analyses the model search space, plans the model search using a task tree, and allows fine-granular model access to optimize model search by caching model parameters and intermediates. In our evaluation of state-of-the-art computer vision and language models, we show that Alsatian speeds up model search by more than one order of magnitude.

## Acknowledgments

This work was partially funded by the German Research Foundation (ref. 414984028 and ref. 556566056), the European Union's Horizon 2020 research and innovation programme (ref. 957407), and is in part supported by NSF awards IIS-2420577 and IIS-2420691.

## References

[1] 2024. CS231n Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/transfer-learning/

[2] 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. In *Proceedings of the IEEE/CVF international conference on computer vision*. 6430–6439.

[3] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of dataset versioning: Exploring the recreation/storage tradeoff. In *Proceedings of the VLDB endowment. International conference on very large data bases*, Vol. 8. NIH Public Access, 1346. Issue: 12.

[4] Daniel Bolya, Rohit Mittapalli, and Judy Hoffman. 2021. Scalable diverse model selection for accessible transfer learning. *Advances in Neural Information Processing Systems* 34 (2021), 19301–19312.

[5] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. 2014. Food-101–mining discriminative components with random forests. In *Computer vision–ECCV 2014: 13th European conference, zurich, Switzerland, September 6-12, 2014, proceedings, part VI 13*. Springer, 446–461.

[6] Sasank Chilamkurthy. 2024. Transfer Learning for Computer Vision Tutorial — PyTorch Tutorials 2.2.2+cu121 documentation. https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

[7] Francois Chollet. 2021. *Deep learning with Python*. Simon and Schuster.

[8] Timothy Dai, Austin Peters, Jonah B Gelbach, David Freeman Engstrom, and Daniel Kang. 2024. tailwiz: Empowering Domain Experts with Easy-to-Use, Task-Specific Natural Language Processing Models. In *DEEM*. 12–22.

[9] Clément Delangue. 2023. Hugging Face just crossed 1,000,000 free public models. https://x.com/clementdelangue/status/1839375655688884305?s=43

[10] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1701–1716.

[11] 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).

[12] 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)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*. https://openreview.net/forum?id=YicbFdNTTy

[14] Fastai. 2024. Imagenette: A smaller subset of 10 classes from Imagenet. https://github.com/fastai/imagenette

[15] Lukas Garbaciauskas, Max Ploner, and Alan Akbik. 2024. Choose Your Transformer: Improved Transferability Estimation of Transformer Models on Classification Tasks. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). 12752–12768.

[16] Peizhen Guo, Bo Hu, and Wenjun Hu. 2022. Sommelier: Curating DNN models for the masses. In *Proceedings of the 2022 International Conference on Management of Data*. 1876–1890.

[17] Aurélien Géron. 2022. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.".

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[19] 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.

[20] Long-Kai Huang, Junzhou Huang, Yu Rong, Qiang Yang, and Ying Wei. 2022. Frustratingly Easy Transferability Estimation. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 9201–9225. https://proceedings.mlr.press/v162/huang22d.html

[21] Hugging Face. 2024. Hugging Face: Machine Learning Platform. https://huggingface.co/

[22] Digvijay Ingle, Rishabh Tripathi, Ayush Kumar, Kevin Patel, and Jithendra Vepa. 2022. Investigating the Characteristics of a Transformer in a Few-Shot Setup: Does Freezing Layers in RoBERTa Help?. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, Jasmijn Bastings, Yonatan Belinkov, Yanai Elazar, Dieuwke Hupkes, Naomi Saphra, and Sarah Wiegreffe (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates (Hybrid), 238–248. https://doi.org/10.18653/v1/2022.blackboxnlp-1.19

[23] Ibrahem Kandel and Mauro Castelli. 2020. How deeply to fine-tune a convolutional neural network: a case study using a histopathology dataset. *Applied Sciences* 10, 10 (2020), 3359. https://doi.org/10.3390/app10103359 Publisher: MDPI.

[24] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Li Fei-Fei. 2011. ImageNet Dogs Dataset. http://vision.stanford.edu/aditya86/ImageNetDogs/

[25] Simon Kornblith, Jonathon Shlens, and Quoc V Le. 2019. Do better imagenet models transfer better?. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2661–2671.

[26] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 2013. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*. 554–561.

[27] Jaejun Lee, Raphael Tang, and Jimmy Lin. 2019. What would elsa do? freezing layers during transformer fine-tuning. *arXiv preprint arXiv:1911.03090* (2019).

[28] Yoonho Lee, Annie S Chen, Fahim Tajwar, Ananya Kumar, Huaxiu Yao, Percy Liang, and Chelsea Finn. 2022. Surgical fine-tuning improves adaptation to distribution shifts. *arXiv preprint arXiv:2210.11466* (2022).

[29] Hao Li, Charless Fowlkes, Hao Yang, Onkar Dabeer, Zhuowen Tu, and Stefano Soatto. 2023. Guided recommendation for model fine-tuning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3633–3642.

[30] Yandong Li, Xuhui Jia, Ruoxin Sang, Yukun Zhu, Bradley Green, Liqiang Wang, and Boqing Gong. 2021. Ranking neural checkpoints. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2663–2673.

[31] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Meng Shi, Gang Chen, and Feifei Li. 2023. SmartLite: A DBMS-Based Serving System for DNN Inference in Resource-Constrained Environments. *PVLDB* 17, 3 (2023), 278–291.

[32] Abhilasha Lodha, Gayatri Belapurkar, Saloni Chalkapurkar, Yuanming Tao, Reshmi Ghosh, Samyadeep Basu, Dmitrii Petrov, and Soundararajan Srinivasan. 2023. On surgical fine-tuning for language encoders. *arXiv preprint arXiv:2310.17041* (2023).

[33] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Portland, Oregon, USA, 142–150. http://www.aclweb.org/anthology/P11-1015

[34] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. 2017. Towards unified data and lifecycle management for deep learning. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 571–582.

[35] Supun Nakandala and Arun Kumar. 2022. Nautilus: An Optimized System for Deep Transfer Learning over Evolving Training Datasets. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia PA USA, 506–520. https://doi.org/10.1145/3514221.3517846

[36] Cuong Nguyen, Tal Hassner, Matthias Seeger, and Cedric Archambeau. 2020. LEEP: A New Measure to Evaluate Transferability of Learned Representations. In *Proceedings of the 37th International Conference on Machine Learning*. PMLR, 7294–7305. https://proceedings.mlr.press/v119/nguyen20b.html

[37] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event China, 1426–1439. https://doi.org/10.1145/3448016.3452788

[38] Joan Puigcerver, Carlos Riquelme Ruiz, Basil Mustafa, Cedric Renggli, André Susano Pinto, Sylvain Gelly, Daniel Keysers, and Neil Houlsby. 2021. Scalable Transfer Learning with Expert Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=23ZjUGpjcc

[39] PyTorch Team. 2024. Models and Pre-trained Weights. https://pytorch.org/vision/stable/models.html

[40] Sebastian Raschka. 2024. *Build a Large Language Model (From Scratch)*. Manning.

[41] Cedric Renggli, André Susano Pinto, Luka Rimanic, Joan Puigcerver, Carlos Riquelme, Ce Zhang, and Mario Lučić. 2022. Which model to transfer? finding the needle in the growing haystack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9205–9214.

[42] Cedric Renggli, Xiaozhe Yao, Luka Kolar, Luka Rimanic, Ana Klimovic, and Ce Zhang. 2022. SHiFT: an efficient, flexible search engine for transfer learning. *Proceedings of the VLDB Endowment* 16, 2 (2022), 304–316. Publisher: VLDB Endowment.

[43] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[44] Nils Strassenburg, Ilin Tolovski, and Tilmann Rabl. 2022. Efficiently Managing Deep Learning Models in a Distributed Environment. In *Proceedings 25th International Conference on Extending Database Technology ( EDBT 2022 )*. OpenProceedings.org. https://doi.org/10.48786/EDBT.2022.12

[45] Mingxing Tan and Quoc Le. 2021. Efficientnetv2: Smaller models and faster training. In *International conference on machine learning*. PMLR, 10096–10106.

[46] Hugging Face team. 2025. Conditional DETR model with ResNet-50 backbone. https://huggingface.co/microsoft/conditional-detr-resnet-50

[47] Hugging Face team. 2025. DETR (End-to-End Object Detection) model with ResNet-50 backbone. https://huggingface.co/facebook/detr-resnet-50

[48] Hugging Face team. 2025. DETR (End-to-End Object Detection) model with ResNet-50 backbone (dilated C5 stage). https://huggingface.co/facebook/detr-resnet-50-dc5

[49] Hugging Face team. 2025. ResNet. https://huggingface.co/microsoft/resnet-18

[50] Hugging Face team. 2025. ResNet-152 v1.5. https://huggingface.co/microsoft/resnet-152

[51] Hugging Face team. 2025. TrOCR (base-sized model, fine-tuned on SROIE). https://huggingface.co/microsoft/trocr-base-printed

[52] Hugging Face team. 2025. Vision Transformer (base-sized model). https://huggingface.co/google/vit-base-patch16-224-in21k

[53] Hugging Face team. 2025. Vision Transformer (large-sized model) trained using DINOv2. https://huggingface.co/facebook/dinov2-large

[54] Hugging Face team. 2025. YOLOS (small-sized) model. https://huggingface.co/hustvl/yolos-small

[55] TensorFlow. 2024. Transfer Learning and Fine-Tuning. https://www.tensorflow.org/tutorials/images/transfer_learning

[56] TensorFlow Team. 2024. TensorFlow Hub: A Repository of Trained Machine Learning Models. https://www.tensorflow.org/hub

[57] 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 (HILDA '16)*. Association for Computing Machinery, New York, NY, USA, 1–3. https://doi.org/10.1145/2939502.2939516

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[59] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. 2022. CUB-200-2011. https://doi.org/10.22002/D1.20098

[60] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. Helix: Holistic optimization for accelerating iterative machine learning. *arXiv preprint arXiv:1812.05762* (2018).

[61] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks? *Advances in neural information processing systems* 27 (2014).

[62] Kaichao You, Yong Liu, Jianmin Wang, and Mingsheng Long. 2021. Logme: Practical assessment of pre-trained models for transfer learning. In *International Conference on Machine Learning*. PMLR, 12133–12143.

[63] Kaichao You, Yong Liu, Ziyang Zhang, Jianmin Wang, Michael I. Jordan, and Mingsheng Long. 2022. Ranking and Tuning Pre-Trained Models: A New Paradigm for Exploiting Model Hubs. *J. Mach. Learn. Res.* 23 (2022), 209:1–209:47.

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

[65] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*. Springer, 818–833.

[66] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. 2023. *Dive into Deep Learning*. Cambridge University Press.

[67] Zhi Zhang, Qizhe Zhang, Zijun Gao, Renrui Zhang, Ekaterina Shutova, Shiji Zhou, and Shanghang Zhang. 2024. Gradient-based Parameter Selection for Efficient Fine-Tuning. In *CVPR*. 28566–28577.

# 9 Appendix

## 9.1 Optimized $C_min$ estimation

For most task trees that we see in practice, DFS+ estimates the optimal/minimum $C_{min}$. In this Section, we discuss some edge cases that we found while working on Alsatian. We describe how they lead to non-optimal estimates of $C_{min}$ and how relevant these cases are in practice. We do not claim that the set of presented edge cases is complete. Extending DFS+ without significantly increasing its runtime is declared future work.

*Scenario 1.* Consider the simplified task tree in Figure 18. It is identical to the one in Section 5.3 except that the storage costs of node 2 and node 3 is 1. Applying DFS+ in this scenario we will traverse the nodes in the order 0, 1, 3, 7, 6, 2, 4, 5, and $C_{min} = 12$.

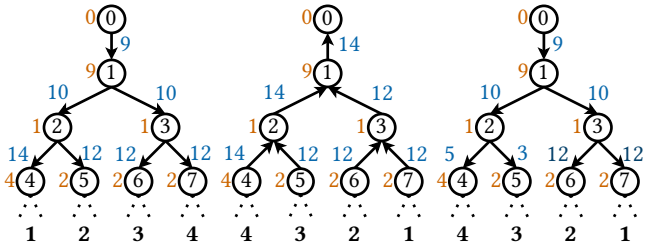


**Figure 18: DFS+ traversal**

However, in Figure 19, we see that with the traversal order of 0, 1, 2, 3, 7, 6, 5, 4 we can reduce $C_{min}$ to 11: We traverse the edge from node 0 to node 1 with a cost of 9, then go to node 2 and node 3 (deviating from a DFS traversal) requiring a caching budget of 11. However, we now can release node 1 and traverse nodes 6 and 7 with budget 4 (2 for node 2 and 3, and 2 for node 6/7), and nodes 5 and 4 with a budget of 3 and 5, respectively.

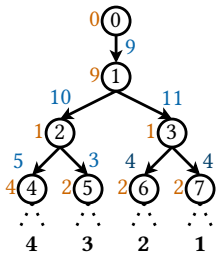


**Figure 19: DFS+ traversal**

*Scenario 2.* Looking at another example in Figure 20 which differs to Section 5.3 by the property that the combined cost of nodes 4 and 5 is lower than their parent's (node 2) cost, and the same for nodes 6 and 7.

Applying DFS+ leads to $C_{min} = 16$. However, with an alternative solution, we can reduce $C_{min}$ (explained for a branch of nodes 1, 2, 4, 5): We skip the caching of nodes 2 and instead combine nodes 2, 4, and 5 into one model (with two output heads) that we execute in a batched fashion. While this increases the memory footprint for the execution (which can usually be neglected if models don't get too large) it reduces the caching budget to $11 + b = 9$ (for node 1) + 1 (node 4) + 1 (node 5) + $b$ (the size of one batch of node 2).

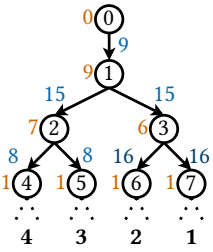


**Figure 20: DFS+ traversal**

*Likelihood of Scenarios 1 and 2.* The critical property for scenarios 1 and 2 to appear is that given a node ($n$) and its children ($c_1, \cdots, c_m$), the sum of its children's storage costs is below $n$'s storage cost ($\text{MEM}(c_1) + \cdots + \text{MEM}(c_m) < \text{MEM}(n)$).

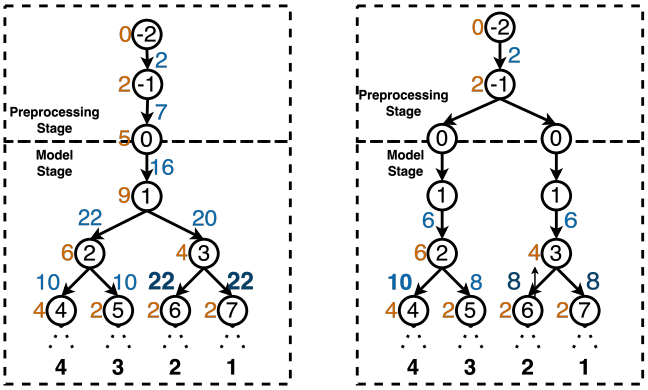
Analyzing a larger set of commonly used models including all the models used for our evaluation we find this case to be very unlikely. For transformer-based architectures, all block intermediates have the same size, meaning: $\text{MEM}(c_1) + \cdots + \text{MEM}(c_m) = m \times \text{MEM}(n)$. For convolutional architectures, such as ResNet, the output of a block is either equivalent in size to the previous output (block is in the same stage) or $2 - 2.5\times$ smaller. This means the only way for $\text{MEM}(c_1) + \cdots + \text{MEM}(c_m) < \text{MEM}(n)$ to be true is if $m \leq 2$ which is an edge case. If $m = 1$, we do not even cache the node, if $m = 2$, the condition is only true if both children are more than $2\times$ smaller.

*Scenario 3.* In our paper, we only consider model intermediates, but not intermediates created as part of the model input data pipeline because we assume models to overlap to a certain degree.

However, if we also consider fully fine-tuned models or models with very little overhead, we might see a situation as shown in Figure 21(a). Compared to previous settings we also consider the input data pipeline by adding the nodes -1 and -2 and assigning new intermediate sizes. Note that this scenario is no different from previous examples if the intermediate sizes always get smaller from root to bottom. We thus show the case where the input data intermediates -1 and 0 are smaller than the first model intermediate.

Traversing the tree in Figure 21(a) using DFS+, we do not consider caching anything before node 1 which might lead to a non-optimal estimation of $C_{min}$ but only leads to a non-valid plan if $C_{min} > s_{cache}$. If this is the case, in our current approach we fall back to reducing the dataset size and traverse the tree in $n$ runs to reduce $C_{min}$ by a factor of $n$. An alternative solution could be to allow recomputation of intermediate and by this trade compute for caching budget. For example, in Figure 21(b) we compute the intermediates associated with nodes 0 and 1, to avoid caching intermediates with size 9 of node 1. Such an optimization only makes sense if (1) even after extending our cache (to for, example, SSD) recomputation is still cheaper than caching, (2) the updated plan is faster than our current solution of traversing the tree in multiple iterations, and (3) the time savings justify the implementation and runtime overhead of a more complex planning algorithm than

DFS+. We plan to investigate the presence of these properties in depth as part of future work.




(a) a (b) b

**Figure 21: Caption**

*Concrete example ImageNet dataset.*

- *Size of compressed images:* The images are stored on disk and in a compressed format (e.g.: jpg or png). The average file size of one image is around $150KB$, assuming a batch size of 32, a batch of image files is roughly $5MB$ ($32 \times 150KB = 4800KB$).
- *Size of raw cropped images:* The images are usually converted into a row format with three 32-bit floats of size $224 \times 224$ pixels. A batch of 32 image is thus roughly $19MB$ ($32(\times 224 \times 224 \times 3) \times 4Byte = 19,267,584Byte$).
- *Vit_L_32*, *Vit_B_32*: intermediate size $\approx 6MB$
- *ResNet18*, *ResNet34*: most intermediates under $15MB$
- *MobileNet*, *EfficientNet small and large*: after 2 blocks below $10MB$
- *ViT_B_16*: intermediate size $\approx 19MB$
- *Vit_L_16*: intermediate size $26MB$ ($7MB$ larger)
- *ResNet152*, *ResNet101*, *ResNet50*: after 50% of layers intermediate size $\approx 25MB$

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009