Efficiently Managing Deep Learning Models in a Distributed Environment

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

Ilin Tolovski
ilin.tolovski@hpi.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

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

ABSTRACT

Deep learning has revolutionized many domains relevant in research and industry, including computer vision and natural language processing by significantly outperforming previous state-of-the-art approaches. This is why deep learning models are part of many essential software applications. To guarantee their reliable and consistent performance even in changing environments, they need to be regularly adjusted, improved, and retrained but also documented, deployed, and monitored. An essential part of this set of processes, referred to as model management, is to save and recover models. To enable debugging, many applications require an exact model representation.

In this paper, we investigate if, and to what extend, we can outperform a baseline approach capable of saving and recovering models, while focusing on storage consumption, time-to-save, and time-to-recover. We present our Python library MMlib, offering three approaches: a baseline approach that saves complete model snapshots, a parameter update approach that saves the updated model data, and a model provenance approach that saves the model's provenance instead of the model itself. We evaluate all approaches in four distributed environments on different model architectures, model relations, and data sets. Our evaluation shows that both the model provenance and parameter update approach outperform the baseline by up to 15.8% and 51.7% in time-to-save and by up to 70.0% and 95.6% in storage consumption, respectively.

1 INTRODUCTION

Deep learning (DL) is essential for many scientific and industrial applications such as image recognition, virtual assistants, and recommender systems. Even in safety-critical environments such as autonomous driving, DL plays a key role [14, 37].

To guarantee reliable performance in dynamic environments at all times and to reproduce results consistently, DL models need to be regularly adjusted, improved, and retrained as well as documented, deployed, and monitored. The process surrounding this is defined as model management [31].

When managing models in safety-critical environments, it is crucial to save and recover exact representations of models that output the exact same results to enable debugging. Therefore, it is insufficient to look at high-level metadata; instead it is necessary to reproduce or to load the actual model.

One practical example are the models representing electric vehicles' battery health. These batteries consist of hundreds of individual cells, which need to be controlled by a battery management system (BMS). The BMS levels out slight differences in individual cells and controls the charging and discharging for the battery safety. Another function is the power prediction, e.g., for acceleration and braking or for calculation of the remaining driving range. This is typically done by a battery simulation model, which needs to be updated regularly per car as the battery changes with aging or other conditions. Because battery safety is very important in automotive setups, models need to be precise to prevent failure or accidents. Today's models are often simple and only deliver a rough estimation, therefore, large safety margins are necessary. The model parameters are initialized from laboratory measurements of other cells of the same type and updated from measurements during operation so that it can take some time until the model of a specific vehicle battery is adapted. In a laboratory setting, not all possible real-life scenarios can be tested, so the model needs to be adapted for new and unseen situations. In case of failure or to improve the precision across all batteries after the adaptation, the models need to be exactly reproducible in a central storage.

Saving the exact models can be achieved by saving a complete snapshot of every model. However, the issue with this approach is that a standard DL model often exceeds 100 MB. Taking into account that the complexity of state of the art models increases exponentially [26] this results in considerable storage consumption for frequently updated models. Therefore, it is desirable to find easy and reliable ways to reduce the overall storage consumption. Even for a single model, it is beneficial to save storage in cases when a transfer with limited available bandwidth is required.

Previous research has produced multiple approaches that track and save training metadata but do not or only naively save the model snapshots [8, 32, 39, 42]. There are other approaches [21] that save models in a more advanced way to reduce storage but focus on fast model recovery and use complex algorithms to optimally save a set of models that become unfeasible for the amount and frequency of models we aim to save.

In this paper, we explore different approaches to save and recover models and their effect on storage consumption, time-to-save, and time-to-recover. We present two advanced approaches that we evaluate against our baseline approach and bundle all approaches in an open source Python library *MMlib* [1].

We make the following contributions:

(1) We present three approaches for saving and recovering exact model representations: a baseline approach that saves each model independently, a parameter update approach that saves parameter updates instead of a complete models, and a model provenance approach that saves models without saving their parameters.
(2) We extensively evaluate all approaches and show that the model provenance and parameter update approach

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

outperform the baseline by up to 15.8% and 51.7% in time-to-save and by up to 70% and 95.6% in storage consumption, respectively.
(3) We analyze how the approaches perform in different distributed environments and find that all of them scale without a loss in performance.
(4) We publish the three approaches together with a model verification probing tool in *MMlib*, an open-source Python library.

The remainder of this paper is structured as follows. In Section 2 we provide background information and definitions on DL and how to reproduce experiments. In Section 3 we define the focus of our work and introduce three different approaches to save and recover models. We extensively evaluate all approaches in Section 4. In Section 5, we give an overview of the related work. Finally, in Section 6 we conclude our work and provide an outlook on future work.

2 BACKGROUND

In this section, we first cover DL foundations in Section 2.1 and give term definitions for recoverability, reproducibility, and provenance in Section 2.2. In Section 2.3 we discuss reproducibility in DL and present how to reproduce model inference and training in Section 2.4.

2.1 Deep Learning

Deep learning (DL) is a subset of machine learning (ML) [12]. In this paper, we focus on supervised DL, meaning that the models are trained using labeled input data [13]. In the following section, we give definitions for the most relevant terms in the scope of this paper and give a generic description of the DL model lifecycle.

DL Term Definitions. A *model* is "*the representation of what a machine learning system has learned from the training data*" [13]. A model $M = (M_a, M_p)$ consists of two pieces: the *model architecture* (M_a) which is the computational structure for making a prediction and the *model parameters* (M_p) representing the weights and biases.

In DL, it is a common practice to use well-established models as a base to develop new models [38]. Being aware that models can be related in various different ways, we define the model relations relevant for this paper and illustrate them in Figure 1.

Given two models $M = (M_a, M_p)$ and $M' = (M'_a, M'_p)$. If a model M' was created by modifying or using parts of M, we say M is the *base model* of M', or M' was *derived* from M and denote this as $M \rightarrow M'$.

If M''s architecture is identical to M's architecture but all parameters differ, we say M' is a *fully updated model version* of M. If only a subset of the parameters have changed we say M' is a *partially updated model version.*

If $M_a = M'_a \wedge M_p = M'_p$ we say M and M' are *equal.*

DL Model Lifecycle. DL models are used to make predictions on certain data. To create a model that makes good predictions, we have to train it on a representative dataset. This gives us an initial version of a model. Over time the distribution of the data will change, known as concept drift [43]. This effect decreases the quality of the model predictions and makes it necessary to train the model on an updated dataset. The training leads to an updated model which makes better predictions on the current data. Because concept drift is a continuous process, we repeat the training process regularly.

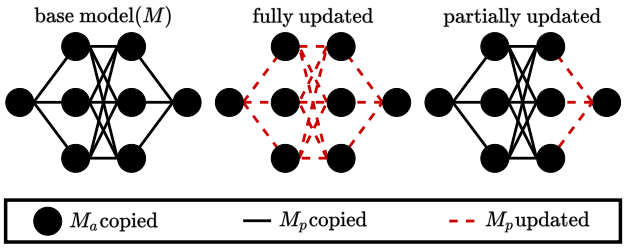


Figure 1: Model relations (left to right): *base model, fully updated model versions,* and *partially updated model version.*

2.2 Recoverability, Reproducibility, and Provenance

The National Academy of Science [23] and Barba et al. [3] define the terms repeatability, reproducibility, and replicability and discuss their inconsistent use in the broader scientific community. The Association for Computing Machinery (ACM) defines these terms within the field of computer science [19]. Hartley et al. [16] define repeatability, reproducibility, replicability, and provenance in the context of DL and present a basic approach implementing their guidelines for reproducible model training.

We base our understanding of reproducibility and provenance on the definitions of Hartley et al. [16]. The *inference of a model M is reproducible* if processing the same input data by M always produces exactly the same output. The *training of a model M is reproducible* if executing exactly the same training process for M (same input data, code, parameters, etc.) always results in exactly the same updated model. A *model M is reproducible* if inference and training of M are reproducible. The *provenance* of a model is the history of the processes used to produce it, together with the input/training data.

We define a model as *recoverable* from given data, if the data provide sufficient information to reconstruct the model (meaning architecture and parameters) from them. With this definition in mind, we want to point out that there are multiple different ways to *recover* a model M from given data D. The easiest way is where D contains M stored in a specific format. If M is *reproducible*, an alternative is that D provides sufficient model *provenance*, which we can use to recover M by reproducing its training. Framework independent formats like PMML [15], PFA [27], or ONNX [25] do not capture the model in a level of detail needed to reproduce model training.

2.3 Reproducibility in Deep Learning

For almost all DL models it is neither supported to reproduce model inference nor to reproduce model training by default [28, 33]. In this section we discuss the factors that prevent the inference and training of DL models from being reproducible and ways to eliminate them.

Code, Parameters, and Data. If the source code defining the model or the parameters differ across executions, we cannot expect to get the same results. The same holds for the input data; if the input data varies between runs, we cannot expect to reproduce results. The solution is to track the source code, the parameters, and the input data to use them consistently across executions. Applied to the deep learning (DL) domain, this includes tracking the model's code, the inference and training routine, the

loss function(s) used, and the optimizer together with all parameters. Additionally, we have to ensure the same training input. This requires tracking the raw dataset and how it is provided by components such as the preprocessor or the dataloader.

Intentional Randomness. Randomness is often intentionally introduced as part of specific machine learning (ML) algorithms. Examples in DL are random data augmentation [34], random weight initialization [10], and regularization layers such as dropout [35]. Since computers cannot generate real randomness, they use pseudorandom number generators (PRNGs). The output of PRNGs is entirely dependent on an initial value called *seed*. Therefore, setting the *seed* leads to reproducible pseudorandomness.

Floating-point Arithmetic. A further source of non-determinism that prevents reproducibility in DL is floating-point arithmetic [28]. Not all floating-point numbers can be represented exactly with a finite number of bytes; they must be approximated by rounding [11]. This implies that different implementations of the same operator or the order of calculation can lead to different results [24, 28]. An example is shown in Figure 2 where the calculation of the dot-product with the serial and the parallel method leads to similar but different results.

To reproduce calculations using floating-point arithmetic, we have to ensure that: (1) the operator is executed in the same environment, meaning on top of the same software and hardware, and (2) the implementation of the operation itself is deterministic and reproducible [28].

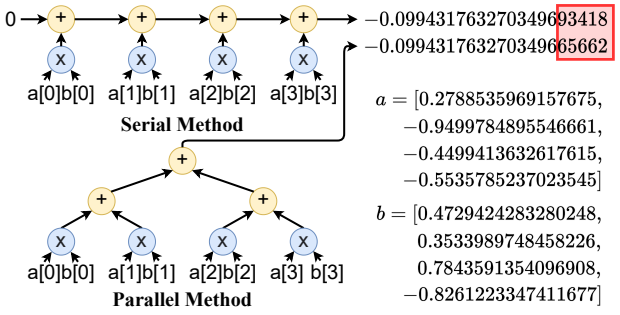
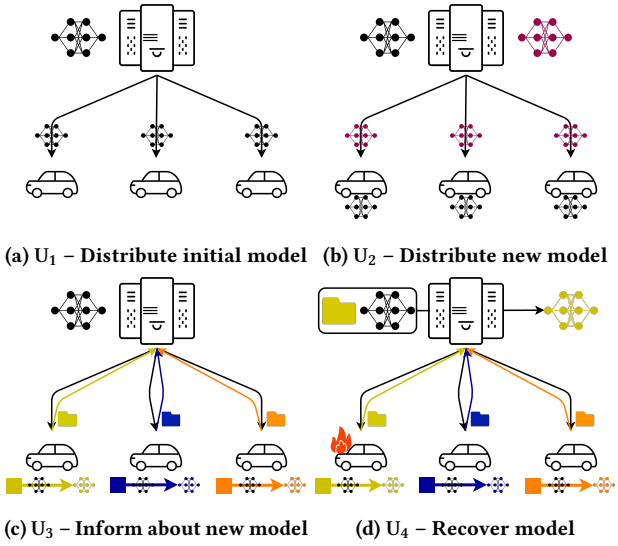


Figure 2: Example of different computation results for the dot-product comparing the serial and the parallel method (Inspired by Figure 3 and Figure 5 in [24]).

2.4 Reproducible Inference and Training

To reproduce model inference and training, we have to use identical code, parameters, and data; set the seeds for all PRNGs; and making floating-point operations reproducible by always executing them in an equivalent environment and only using their deterministic implementation. To check if a given model is reproducible in a given setup, we developed a probing tool for PyTorch [7] models and integrated it in *MMlib*. Inspired by the tool Riach [28] presented for TensorFlow [1] models, our probing tool executes a given PyTorch model twice using the same data to compare layer-wise the input and output tensors for the forward and backward pass. These intermediate results can be saved and loaded which enables us to also verify the model reproducibility across different machines.

We used the probing tool to check if popular computer vision models are reproducible. For a majority of models, we find that we can reproduce inference and training. For non reproducible



(a) U_1 – Distribute initial model **(b) U_2 – Distribute new model**

(c) U_3 – Inform about new model **(d) U_4 – Recover model**

Figure 3: Use cases

models, we find that the reason is their use of deprecated layers where PyTorch does not provide a deterministic implementation. We conclude that reproducing model inference and training is possible by applying the measures presented above if there is a deterministic implementation for every layer.

3 MMLIB: MODEL MANAGEMENT LIBRARY

The research question we address in this paper is if and to what extent we can outperform a baseline capable of saving and recovering deep learning (DL) models without any loss in terms of storage consumption, time-to-save (TTS), and time-to-recover (TTR). To investigate this question, we develop three approaches that all cover the same operations and provide them in a Python library that we call *MMlib*.

We develop *MMlib* with a distributed setting in mind that consists of a central *server* and multiple distributed devices that we refer to as *nodes*. *MMlib* covers the four use cases shown in Figure 3. We initially develop models on the server and distribute them to all nodes ($\mathbf{U_1}$). Over time we update the models in two ways. (1) We update the models on the server and deploy this update to the nodes ($\mathbf{U_2}$). (2) We update the model on the node using locally collected data and inform the server about this model before we use it ($\mathbf{U_3}$). Regardless of whether we trained the models on the server or the node, the server has to monitor every model that exists and has to be able to losslessly recover it when requested ($\mathbf{U_4}$).

Within this setting, we assume that every node updates its model regularly while major model updates happen only occasionally. We also assume that recovering a model occurs rarely compared to the other scenarios. Mapped to the use cases this means U_3 happens frequently, U_1 and U_2 from time to time, and U_4 rarely. For U_3 the most likely model relations are partially and fully updated model version (see Section 2.1) which is why we focus on these model relations throughout this paper.

In MMlib the focus is on result reproducibility and model recoverability. Therefore, we save an exact representation of all models. This means that a model we save and a model we recover are equal with regards to the definition in Section 2.1. For now,

we save all models created and do not optimize for other data like training or log data.

Having set our focus, we develop three approaches. First, we present a baseline approach (BA) (Section 3.1) and implement two strategies to improve it. (1) The first is to save derived models by only saving the model data that has changed compared to its base model, which leads to the parameter update approach (PUA) (Section 3.2). (2) The second strategy is to not save derived models, but only their provenance data, which results in the model provenance approach (MPA) (Section 3.3).

3.1 Baseline Approach

The BA covers all requirements with minimal complexity. Therefore, we take the BA as a reference point to evaluate the two other approaches. This implies that the BA ignores the similarity of a base model and its derived model and does not apply any advanced measures to optimize storage consumption.

Model Representation. Using the BA, we represent a model using three types of information: metadata about the model, the model architecture, and the model parameters.

The model metadata includes an identifier, a reference to the base model, and optionally checksums to verify that a model was correctly recovered. The reference to the base model is given, and the identifier auto-generated. To generate checksums we hash the tensor objects, which is the data structure used to represent model parameters.

We represent the model architecture by its implementation in code and detailed environment information. This includes the framework version, all third-party libraries, the language interpreter, operating system kernel, as well as the driver versions, and the hardware specification. This is necessary to guarantee reproducibility because, as discussed in Section 2, the implementation and the behavior of certain layers might differ across framework versions and underlying hardware. To represent the model parameters, we serialize the model's internal data structure that maps each layer to its parameters.

Model Storage. To save a model, we have to save two types of data: metadata and files. The metadata consists of the environment information, a reference to the base model and optionally hash values. The files are the model code and the serialized model parameters.

We save the extracted model metadata in different JSON [6] documents that we organize in a hierarchical structure. We identify each document by a generated identifier and persist them in a document database like MongoDB [22]. To save files, we use a shared file system and insert an automatically generated file identifier as a reference in the appropriate JSON document.

Model Recovery. To recover a model holding its identifier, we recursively load all associated JSON documents and files. We explicitly exclude loading documents holding base model information since the BA saves every model independently. The recovered documents and files provide sufficient information to recover the model losslessly and, if provided, we can use the saved checksums to verify this.

3.2 Parameter Update Approach

In contrast to the BA, the parameter update approach (PUA) utilizes the fact that derived models are related to each other and saves only the information that differ compared to the base model.

Model Representation. The PUA saves all model information for an initial model exactly as the BA does it. For derived models, the PUA represents every model by a reference to its base model and the changed information. Parts of a model that could change are, for example, the model code and environment (both affecting the model architecture) or the model parameters. Given that we focus on partially and fully updated model versions, the model code does not change and the most important and frequently updated information are the model parameters.

We assume that whenever a new model M is derived from a base model B ($B \rightarrow M$), a subset of the model parameters are declared as not-trainable on a layer granularity. We further assume that all trainable parameters will change at least marginally. To extract the updated parameters of M, we take M's parameters, compare them layer-wise to B's parameters, and delete all layers' parameters that have not changed. We call the pruned version of M's parameters the *parameter update* and serialize it. Except that we save the parameter update instead of a full model snapshot, the PUA's saving process is identical to the BA's.

Model Recovery. The PUA saves a model M ($B \rightarrow M$) by only saving the updated information compared to its base model B. To recover M, we load its base model B and merge its parameter information layer-wise with the policy of prioritizing M's parameter information in case of merge conflicts.

Using the PUA, recovering a model is a recursive process. To recover a model M ($B \rightarrow M$), we have to recover B. If B is also referencing a base model, we also have to recover it and so on. This recursive process can be computationally intense and long. This is not a problem for recovering a specific model since we assume it happens rarely. Nevertheless, without further measures, it passively slows down saving a model.

To save a model M ($B \rightarrow M$) using the PUA, we have to compare it to B. If B is not in memory, we have to recover B and all its base models which can take a significant amount of time. To eliminate this problem, we always save a hash value for every layer of a saved model when using the PUA. Having this information, we can identify the changed layers by only recovering and comparing the direct base model's hash values instead of recursively recovering it fully.

We organize the layer hash values in a Merkle tree [20] where every model layer is represented by a leaf node. A leaf node holds the parameter hash for one layer. A non-leaf node combines the two hashes of its child nodes to a new hash value and represents multiple layers. The root node represents the whole model. In Figure 4 we show an example of a model with eight layers.

Having a Merkle tree for two models enables us to determine if the models have equal weights by only comparing the trees' root nodes. This is beneficial to check if a model was correctly recovered. For bigger models, the Merkle tree makes it also more efficient to find out what parts of the model have changed.

In the example in Figure 4, the last two layers of the model have changed which results in changed hash values for the nodes marked in red. Here we need only seven instead of eight comparisons to find what layers have changed. For more realistic architectures with more layers the benefits become even clearer. In the same scenario but for a model with 64 layers the number reduces from 64 to 13, and for a model with 128 layers from 128 to 15.

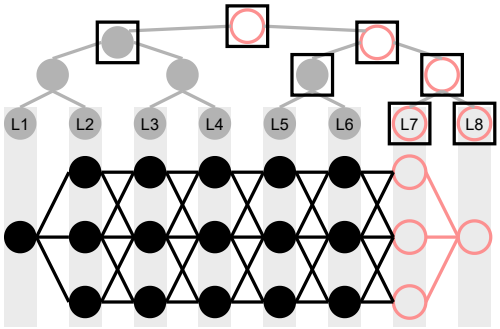


Figure 4: Merkle tree representing layer hash values. Changed layers and nodes are represented in hollow red circles. Nodes that need to be compared to find the changed layers are marked with black boxes.

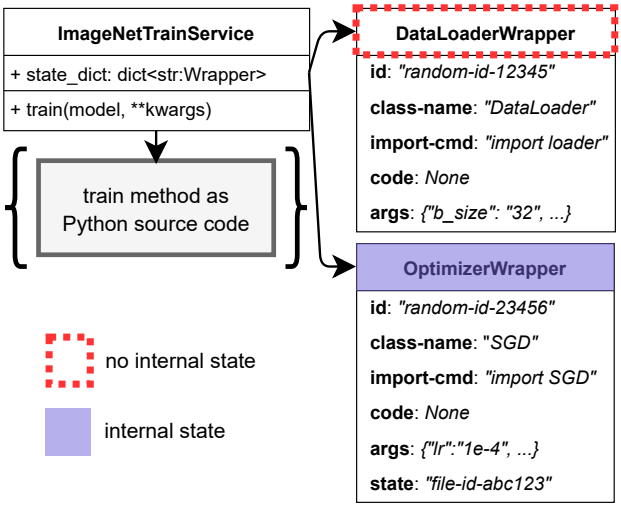


Figure 5: Example of a Train Service

3.3 Model Provenance Approach

Both, the BA and the PUA are based on saving model parameters. Therefore, their performance is highly dependent on the architecture of the model they save or recover. In this section, we present the model provenance approach (MPA). Instead of saving model parameters, the MPA saves the model provenance, which describes the processes and data used to create the model.

We first specify the single parts of the provenance data before we describe the high-level steps of collecting and saving all information together with run time estimations. Afterward, we describe implementation details about representing the training process, tracking the training environment, and saving the dataset.

Model Representation. The MPA saves the first model with the same logic the BA uses. For all following models, the MPA represents a model by its provenance data. The provenance data consists of (1) information about the training process, (2) a detailed specification of the training environment, (3) the training data, and (4) a reference to the base model.

Given our assumption of partially and fully updated model versions, the model architecture does not change across derived models. Therefore, the model architecture is defined by the reference to the base model and the environment information. This is why we do not include the model architecture in the model provenance data.

Model Provenance Tracking. We represent the training process by the training source code. For every object referenced as part of the training process, we save its state before the training starts. The run time of this process depends linearly on the complexity of the training process and the complexity of the referenced objects. We collect detailed training environment information by calling various library functions. This takes constant time for a given environment.

To save the used dataset, we compress it to a single file, save it, and reference the file in our metadata. The run time of this step depends on the size of the dataset and the used compression algorithm. The reference to the base model is given; we do not need to extract or generate it. This leads to a constant run time close to zero.

Training Process. We represent the training process by three different groups of data: (1) parametrized objects without an internal state [2] to save objects like the dataloader, (2) parametrized objects with an internal state to save objects like the optimizer, and (3) the overall training logic to define how the objects relevant for training (for example, optimizer, dataloader, and dataset) interact with each other and what hyper-parameters we use.

To save and recover a parametrized object we wrap it in a *wrapper object* that is responsible for saving and recovering it. In our implementation, a *wrapper object* holds the following information for its wrapped object: a reference to it; its class name; the code or, if the object is defined in a library, the import command (for example, for objects defined in a library); the initialization arguments; arguments that are read from a configuration file; and arguments that are references to other objects. [3] To represent its wrapped object a wrapper serializes all the information listed above except the wrapped object instance. To wrap objects holding an internal state the wrapper additionally holds a reference to a state file which can be any file that represents the state of the wrapped object.

To represent the training logic, we provide an interface called *TrainService*. Every *TrainService* defines the logic to train a given model in its *train* method and references all objects that are relevant for it wrapped in wrapper objects. To represent an instance of a *TrainService*, we serialize all referenced wrappers and additionally the *TrainService*'s class name and code or import command. Figure 5 shows the example of an implementation of a *TrainService* that we call *ImageNetTrainService*. For the dataloader and the optimizer it refers to their wrapper objects. Compared to the stateless dataloader, the optimizer has an internal state that is saved in a state file. For the train logic the *TrainSevrvie* refers to a train method implemented in Python.

For the provenance data, we represent every wrapper object and the training service as a separate JSON document. We save all the documents and files exactly as we do it for the BA and the PUA.

[2] With internal state we mean, in this case, a state that we can not recover by just initializing the object with the same constructor arguments.
[3] Here we just save that reference objects are part of the constructor arguments. How they are handed over is managed by saving the training process.

Environment Tracking. We extract information about the training environment including the DL framework version, all third-party libraries, the language interpreter, operating system kernel, the driver versions, and the hardware specification.

In our current implementation of MMlib, we call various library functions provided by the DL framework and Python to collect all environment information. To the best of our knowledge, there is no approach available that tracks and collects the entire environment information needed for the MPA. Existing approaches track subsets of the environment. ReproZip [4], for example, is a tool that collects detailed information about installed software, but it does not extract information about the available hardware. A full integration or adaptation of ReproZip to track the software dependencies for the MPA is part of our future work.

Managing Data sets. MMlib compresses datasets to a file, saves the file, and references it in the provenance data. If a dedicated external system manages these datasets, as presented by Agrawal et al. [2], we do not have to compress the dataset but only save the reference to the managed dataset as part of the provenance data.

Model Recovery. With the MPA, we always save a model M by referencing its base model B. This makes recovering M a recursive process that is almost identical to that of the PUA. The only difference is that the MPA reproduces the model training step-by-step instead of applying a parameter update which can take a significant amount of time.

4 EVALUATION

In this section, we evaluate the three approaches presented in Section 3 on several datasets and different model architectures in a distributed environment. We first describe our setup in Section 4.1, before we evaluate every approaches' storage consumption in Section 4.2, time-to-save (TTS) in Section 4.3, and the time-to-recover (TTR) in Section 4.4. We evaluate the impact of a deterministic training on the training time in Section 4.5. In Section 4.6 we analyze the approaches' performance in large distributed environments and finally, discuss the best approach to use under different assumptions in Section 4.7.

4.1 Setup

Evaluation Flow. We evaluate all our approaches on a sequence of use cases (Section 3) that we call the *evaluation flow*. The evaluation flow starts with U_1 and continues with four iterations of U_3 ($U_{3\text{-}1\text{-}n}$ with $n \in \{1, 2, 3, 4\}$) followed by U_2 and then again four iterations of U_3 ($U_{3\text{-}2\text{-}n}$ with $n \in \{1, 2, 3, 4\}$). That is, each execution of the evaluation contains of ten executed use cases that each lead to a new model. Except for the first model created in U_1, all models are derived from the model generated in the previous use case which are transitively derived from the model created in U_1. In Figure 6 we show these model relations.

Datasets. As mentioned, we derive a new model by training in every use case. We list the datasets that we use to model the use cases for the evaluation in Table 1 and give information about how many images they include, what size they have and the associated use case.

In U_1 we distribute an extensively trained initial model to the nodes. To save resources, we use the pre-trained parameters provided by *PyTorch* that were generated by training on the ImageNet training dataset from 2012 [29]. In U_2 we distribute a (partially or fully updated) version of the initial model. The idea is that the server has collected more data and improved the model's performance through further training. To represent this, we train the initial models on the ImageNet validation dataset ($INet_{val}$) from 2012 [29]. U_3 is about training a model on a locally collected dataset. It is likely that the locally collected data is slightly biased and its' distribution differs from that of the training data. To this end, we created two different datasets: *Coco-food-512* (*CF*-512) and *Coco-outdoor-512* (*CO*-512). Both are subsets of the Coco dataset [18] and contain 512 images each matching one of the categories in the ImageNet dataset. [4] We train the model for U_2 for ten epochs on $INet_{val}$ and then retrain for five epochs per iteration of U_3 on *CF*-512 or *CO*-512.

Models. We list the set of model architectures we use for the evaluation in Table 2. They are extensively analyzed architectures in the computer vision domain, an active field of research. We select them to cover a wide range of layers, storage sizes and model complexities. For the implementation of the models we slightly adapt the implementations of PyTorch. For fully updated model versions we retrain all model parameters and for partially updated model versions only the last fully connected layers.

[4] For both datasets we give a detailed description of how we created them in our GitHub repository.

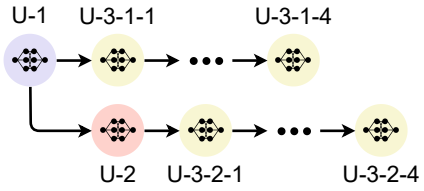


Figure 6: Models created during the execution of the evaluation flow.

Table 1: All datasets used throughout the evaluation with their short name, number of images, size, and the corresponding use case.

Short name	Images	Size	Use case
$INet_{val}$	50,000	6.3 GB	U_2
$mINet_{val}$	1,400	200 MB	U_2
CF-512	512	94.3 MB	U_3
CO-512	512	71.6 MB	U_3

Table 2: Set of selected model architectures [17, 30, 36] for the evaluation with the number of trainable parameters, trainable parameters in case of a partially updated model version, and the size of the parameters.

Name	#Params	part. updated	Size
MobileNetV2	3,504,872	1,281,000	14.3 MB
GoogLeNet	6,624,904	1,025,000	26.7 MB
ResNet-18	11,689,512	513,000	46.8 MB
ResNet-50	25,557,032	2,049,000	102.5 MB
ResNet-152	60,192,808	2,049,000	241.7 MB

Executing Experiments. To evaluate all approaches we perform multiple experiments. One experiment is a full run of the evaluation flow for a given approach, model architecture, model relation, and dataset. We execute every experiment five times for all presented results and take the median computation time to save and recover a model, respectively. The storage consumption is constant across multiple runs of the same experiment.

To model a distributed setup, we simulate the evaluation flow using three different machines. One to model a *node*, one to model a *server*, and a third one to run an instance of MongoDB. This allows us to test for all approaches the storage of a model on one machine and have it identically recovered on another. All machines have the same hardware and software setup: 96 GB RAM, two Intel Xeon Gold 5220S processors with 18 cores, Python 3.5.8, PyTorch 1.7.1, and torchvision 0.8.2. They are connected via 100G InfiniBand and have access to the same external storage.

Pretrain Models. To evaluate all approaches for all model architectures, model relations, and datasets, we execute 80 different experiments, each executed five times to provide median values. The model training (especially for U_2) can take multiple hours per experiment and always leads to the same model. Thus, to make the extensive evaluation feasible, we train the models before the actual experiments and load them from snapshots instead of repeating the training procedure each time.

The MPA's performance is highly dependent on the size of the dataset. For U_2, we use a significantly larger dataset than for U_3 (see Table 1). This leads to a peak in storage consumption, TTS, and TTR for the MPA in U_2 while the other approaches show similar results to U_3. Including the data for U_2 in comparison plots, such as Figure 7, Figure 10, and Figure 11, does not add any information but massively decreases the plots' readability. For this reason, we exclude the results for U_2 from them.

4.2 Storage Consumption

In this section, we analyze the amount of storage that every approach consumes to save a given model. The *storage consumption* of a model does not include the amount of storage that is used to save its base model.

Figure 7(a) and 7(b) show the storage consumption across the use cases and approaches for fully and partially updated MobileNetV2 versions trained on *CF*-512. Figure 7(c) and 7(d) show the same setting for partially and fully updated ResNet-152 versions.

Baseline Approach. The numbers in Figure 7 indicate that neither the use case nor the model relation has an impact on the storage consumption. During our evaluation we found the same not only for the MobileNetV2 and the ResNet-152 but for all model architectures. We also found that the storage consumption is independent of the dataset.

These observations are expected and can be explained by the fact that the BA saves a complete snapshot of each model including its metadata, architecture, and parameters. The size of these data does not change over the use cases and is independent of the training dataset used as well as of the model relation.

Comparing Figure 7(b) and 7(d), we see that the storage consumption is dependent on the model architecture. The MobileNetV2 architecture consumes approximately 14MB while the ResNet-152 consumes over 240MB. Figure 8 shows the storage consumption using the baseline approach and number of parameters for all our model architectures. It confirms the observations

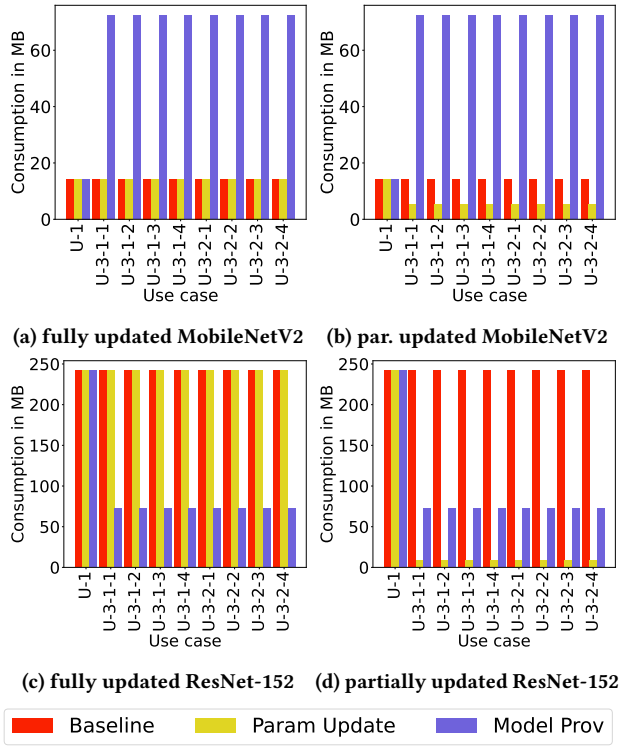


Figure 7: Comparison of the storage consumption across approaches

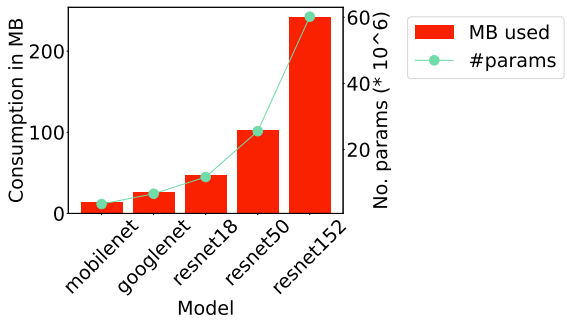


Figure 8: Storage consumption and number of parameters per model using the baseline approach

and indicates that the storage consumption increases proportionally with an increased number of parameters.

In summary, for the BA we observe that **the storage consumption depends on the number of parameters and is, for a given model, independent of the use case, the dataset, and the model relation.**

Parameter Update Approach. Analyzing the storage consumption for the parameter update approach (PUA) in Figure 7(a) and 7(c), we observe that using the PUA does not significantly decrease the storage consumption compared to BA.

The BA saves the architecture and a complete snapshot of the parameters for every model. The PUA only saves the updated parameters and refers to its base model for the unchanged parameters and the model architecture. For fully updated model versions all model parameters are trainable and change between related models. Therefore, the parameter update is equivalent to

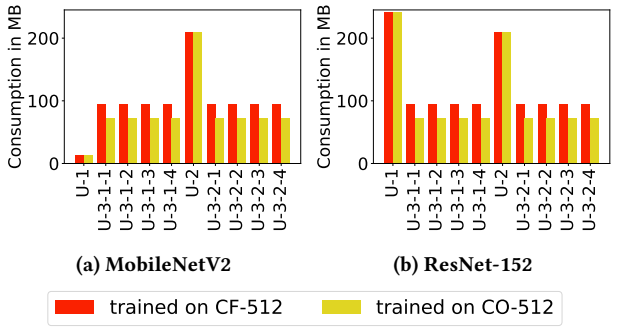


Figure 9: Comparing the storage consumption using the MPA across datasets.

a complete snapshot of all parameters, and the PUA only reduces the storage consumption by the amount of storage used to save the model architecture. With the model parameters being very dominant for the storage consumption of both the BA and the PUA, the differences in Figure 7(a) and 7(c) are insignificant.

On the other hand, Figure 7(b) and 7(d) show that for partially updated model versions the storage consumption significantly improves compared to the BA. With the exception of U_1, the PUA lowers the storage consumption by 63.7% for the partially updated MobileNetV2, and by 95.6% for the partially updated ResNet-152 architecture. As presented in Table 2, for partially updated model versions, a significantly lower number of parameters is trainable. The parameter update does not have to contain all model parameters but only the subset that has been changed which results in a decreased storage consumption.

Overall, we observe that **for fully updated model versions the parameter update approach only marginally improves the storage consumption, whereas for partially updated model versions the improvement is noticeably higher.**

Model Provenance Approach. Comparing the MPA's storage consumption for the MobileNetV2 in Figure 7(a) and 7(b), we see that the MPA consumes significantly more storage than the BA and the PUA. The opposite is the case for the ResNet-152. In Figure 7(c) and 7(d), we see that the MPA outperforms the BA and the PUA by up to 70% for fully updated model versions.

Overall, the critical factor is the ratio of provenance data size and model parameter size. If the provenance data is small compared to the size of the model parameters, the MPA will outperform the BA. If it is the other way around, the MPA will consume more storage than the other two approaches.

Figure 9 shows the storage consumption using the MPA for the MobileNetV2 and the ResNet-152 when trained on *CF*-512 and *CO*-512. We observe that the storage consumption per use case is similar, although the numbers in Figure 9(a) are for the MobileNetV2 architecture, and the numbers in Figure 9(b) are for the ResNet-152 architecture.

We observe that the model architecture and number of parameters do not significantly influence the storage consumption. The storage consumption mainly depends on the size of the dataset that we use to train the model. For a MobileNetV2, the dataset is responsible for more than 99.9% of the storage consumption, and whenever we use a larger dataset also the storage consumption increases. *CF*-512, for example, is approximately 23 MB larger than *CO*-512, which is approximately the difference between the storage consumption for all U_3s. For U_2, we always use the

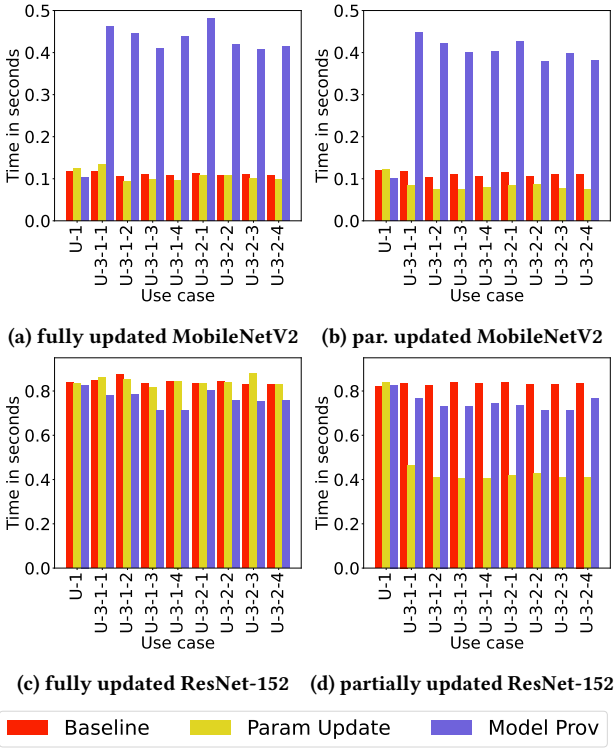


Figure 10: Comparison of the median time-to-save (TTS) across approaches. All models in U_3 were trained on the Custom-Coco-Outdoor-512 dataset.

smaller version of the ImageNet data ($mINet_{val}$) and see no difference in the storage consumption. For U_1 every approach uses the BA's logic which leads to different storage consumptions across model architectures.

This observation is expected because the MPA saves only the model provenance data. In addition to small metadata, the provenance data also contain the training dataset which is, relative to the other provenance data, significantly larger.

For the model provenance approach, we observe that **the storage consumption mainly depends on the training dataset and is almost independent of the model architecture. Whether the MPA outperforms the BA depends on the ratio between the training set size and model size.**

4.3 Time to Save

In this section, we analyze the amount of time it takes to save a given model for all approaches. The time-to-save (TTS) includes the time to extract the model data and the time to persist them. Throughout this section, we identify the models by the use case they were saved in. For example, if we refer to the TTS of (model) U_2, we refer to the time that we needed in U_2 to save the model.

Baseline Approach. In Figure 10, we compare the TTS using the different approaches in the same settings. For the BA we find that the TTS strongly depends on the number of parameters. Saving a full MobileNetV2 takes around 0.1 s, saving a full ResNet-152 takes around 0.8 s. This can be explained by the fact that the main steps when saving a model are to hash and serialize the parameters and to persist all model data which mostly consists of serialized parameters.

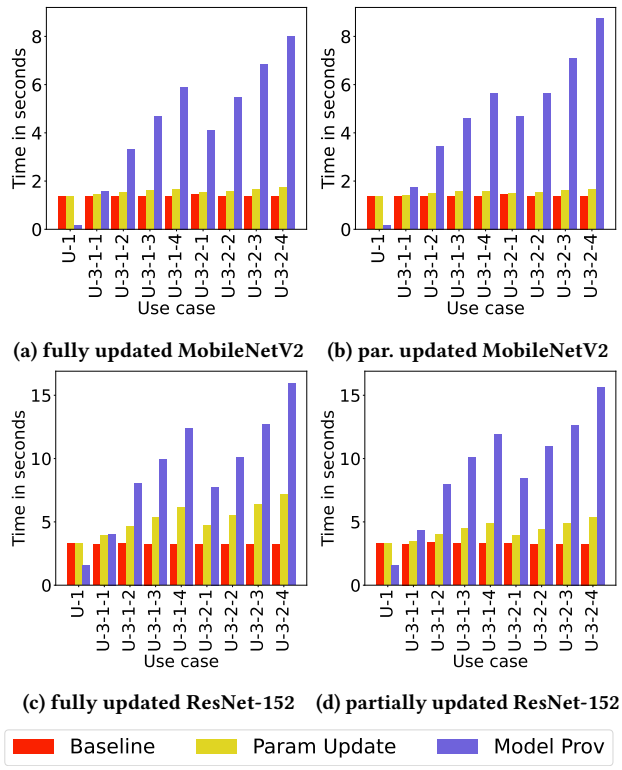


Figure 11: Comparison of the median time-to-recover (TTR) for the ResNet-152 across approaches. All models in U_3 were trained on the Custom-Coco-Outdoor-512 dataset.

Parameter Update Approach. For the PUA, the TTS depends on the number of parameters in the parameter update. For fully updated model versions the PUA's TTS is similar to the BA's for both model architectures because the amount of parameters to save is equivalent to the number of the total parameters. When the number of parameters in the parameter update decreases, as it is the case for partially updated model versions, also the TTS decreases. In this case, the PUA outperforms the BA by up to 28.5% and 51.7% for MobileNetV2 and ResNet-152, respectively.

Model Provenance Approach. The times in Figure 10(c) suggest that the MPA has the potential to outperform the BA and the PUA by up to 15.8%; especially when the MPA uses less storage than the BA and the PUA. Figure 10(a) shows that there are also scenarios where the MPA is drastically outperformed by the BA and the PUA, which is mainly the case when the MPA's storage consumption is relatively high.

4.4 Time to Recover

In this section, we analyze the *time-to-recover (TTR)* which describes the amount of time it takes to load the model data and to recover the model from it. We identify the models by the use case they were saved in. For example, if we refer to the *TTR* of (the model) U_2, we refer to the time that we needed to recover the model that we saved in U_2.

Baseline Approach. In Figure 11, we show the TTR for all approaches across different scenarios for the MobileNetV2 and the ResNet-152. We observe that for the BA the TTR is not dependent on the use case and the model relation. This is due to the fact that the BA saves every model independently of other models.

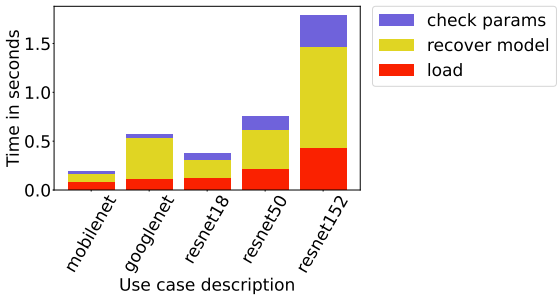


Figure 12: Baseline time-to-recover (TTR) for different model architectures in $U_{3\text{-}1\text{-}3}$, *check env* time excluded.

In Figure 12 we show the TTR with respect to the model architecture. The process to recover a model consists of four steps: loading the model data, recovering the model from the data, verifying if the environment matches the given information, and verifying if the model parameters have been correctly recovered. We found that verifying the environment takes over one second and adds a constant time to the recover process regardless of the model architecture. For clarity we exclude this time from Figure 12 and only show the remaining categories.

We can see that the different steps but also the overall TTR strongly depend on the model architecture. The more parameters, the more data to load, and the more data to process, to recover, and verify. The only exception is the TTR for GoogLeNet, although it has fewer parameters than the ResNet-18 its *recover* time is noticeably higher and consequently also its total TTR. The reason for this peak – in comparison to all other models – is a disproportional high computation time for GoogLeNet's initialization routine. We found that initializing an instance of a GoogLeNet takes approximately seven times longer than initializing a ResNet-18, thus, it significantly slows down the model recovery phase.

Parameter Update Approach. In Figure 11 we show that the PUA does not outperform the BA for any use case, model architecture or model relation. However, it is interesting that the TTR for the PUA is not drastically higher even though they increase with every iteration of U_3. For example, to approximately double the TTR of U_1 in Figure 11(a) we would have to save a MobileNetV2 that transitively refers to 18 base models; for Figure 11(b) this number rises to 25.

The reason for the increasing TTR per iteration of U_3 is described in Section 3.2; recovering a derived model M (with $B \rightarrow M$) that is saved using the PUA is a recursive process and includes recovering its base model B. Taking into account how the models that we create during the evaluation flow relate to each other (Figure 1), we can explain the two staircase patterns. To recover a model $U_{3\text{-}1\text{-}n}$, the PUA has to recover all its base models: $U_{3\text{-}1\text{-}(n\text{-}1)}$ to $U_{3\text{-}1\text{-}1}$, and U_1. To recover a model $U_{3\text{-}2\text{-}n}$, the PUA has to recover all its base models: $U_{3\text{-}2\text{-}(n\text{-}1)}$ to $U_{3\text{-}2\text{-}1}$, U_2, and U_1. The higher n, the more base models the PUA has to recover and the higher the TTR. For partially updated model versions the PUA has to load the same number of base models, but a smaller amount of data which, results in a shorter TTR compared to fully updated model versions.

When using the parameter update approach, **the time-to-recover (TTR) depends on the model architecture, the number of parameters, model relation, and use case.**

Model Provenance Approach. To recover a model saved using the MPA, we have to reproduce the model training which can take a significant amount of time. For the MPA, we ran the model training only for two epochs with two batches. The time consumption for this training procedure might be not as high as for realistic use cases, but it is sufficient to ensure that our prototype is capable of recovering all data necessary to repeat the model training. Moreover, it results in a training time that is proportional to the complexity of the model architecture and the model relation.

We verified that our models are reproducible across machines by using our probing tool introduced in Section 2.4. To test that the MPA is capable of reproducing model training that lead to equal models we performed a separate experiment that loads the same models twice using the MPA and compares them.

Figure 11 shows that the MPA is not capable of outperforming the BA or the PUA. We additionally have to keep in mind that we only simulated and not fully executed model training to make an extensive evaluation feasible.

The MPA's TTR shows a similar staircase pattern as the PUA. The reason is the model dependencies we already described for the PUA. The only difference is that the MPA does not apply a parameter update but reproduces the model training.

In a detailed analysis we found that the main factor for the TTR is the model training. Since the MPA reproduces the complete training process, this leads to consistently slower TTR. How much longer the TTR for the MPA is, depends on many factors including the model architecture, the size of the dataset, the number of epochs, and the available hardware. For U_1, all approaches use the BA's logic which results in a similar TTR. We observe that the MPA's TTR in U_1 is noticeably lower than for the other approaches. Since we use a simulated model training for the MPA, we deactivated a set of automatic checks that also include a check of the current environment. The environment check takes more than a second and not performing it is the reason for the MPA's lower TTR in U_1.

For the model provenance approach we see that **the time-to-recover (TTR) mainly depends on the model architecture and the corresponding time to train the model.**

4.5 Deterministic Training

For the MPA, it is critical that the training is executed deterministically to be able to reproduce the model. To analyze the impact of a deterministic execution on the training time, we perform multiple training runs in a deterministic and non-deterministic way. As models, we use the ResNet-18, the ResNet-50, and the ResNet-152. We train them for five epochs on the *CO*-512 dataset using an NVIDIA A100-SXM4-40GB GPU with batch size 64. This is equivalent to one iteration of U_3.

Figure 13 shows the median training times of five runs for the time consumed to load the data on the GPU, the forward pass, and the backward pass. The data indicate that training a model deterministically is slower in the forward and the backward pass while the time to load the data to the GPU is not influenced. For the ResNet-50 and ResNet-152 we see that a deterministic training is not significantly slower. For ResNet-18 this is not the case, here especially the backward pass takes more than twice the time. The reason for this difference is that in our implementations the ResNet-50 and the ResNet-152 architecture make use of the same layers, while the ResNet-18 uses a similar but not identical set of layers.

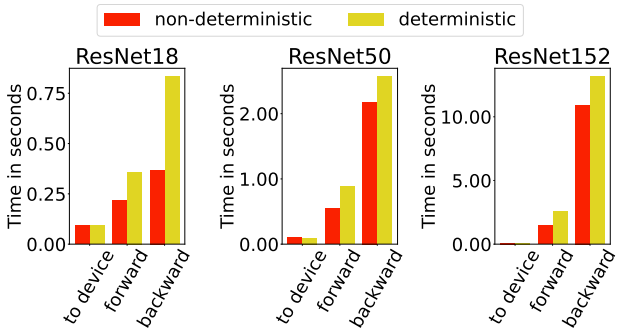


Figure 13: Median times for loading the data to the GPU, processing the forward pass, and processing the backward pass for five epochs on *CO*-512 in *deterministic* and *non-deterministic* mode.

Performing a more extensive experiment over ten times the number of epochs, we find that the times per batch are close to constant for all models. Therefore, we can expect to see the same relative slowdown due to deterministic training regardless of the number of epochs or number of images in the dataset.

In summary, we find that **using only deterministic operations slows down the training. The impact of the deterministic training time depends on the model architecture.**

4.6 Distributed Evaluation Flows

In this section, we investigate how the different approaches perform in a scenario with more nodes and many more models than used in previous experiments.

We refer to the evaluation flow used so far as the *standard* evaluation flow. As described in Section 4.1, the standard evaluation flow consists of one iteration of U_1 and U_2, four iterations of $U_{3\text{-}1}$ and $U_{3\text{-}2}$, and considers one node. This adds up to ten saved and recovered models per execution and allows us to extensively evaluate all approaches for many different model architectures, datasets, and model relations.

We define three larger distributed evaluation flows listed in Table 3. For DIST-5, we consider five nodes, for DIST-10 ten nodes, and for DIST-20 20 nodes. Next to the increased number of nodes, we execute ten (instead of four) iterations of $U_{3\text{-}1}$ and $U_{3\text{-}2}$ for all new evaluation flows. This leads to a number of 102 (DIST-5), 202 (DIST-10), or 402 (DIST-20) saved and recovered models per run.

Table 3: Overview of different evaluation flows.

NAME	#NODES	#MODELS
STANDARD	1	10
DIST-5	5	102
DIST-10	10	202
DIST-20	20	402

For all results presented in the following, we execute each evaluation flow using the MobileNetV2 architecture and fully updated model versions as the model relation. We run every experiment three times to provide median values and use the same setup as described in Section 4.1.

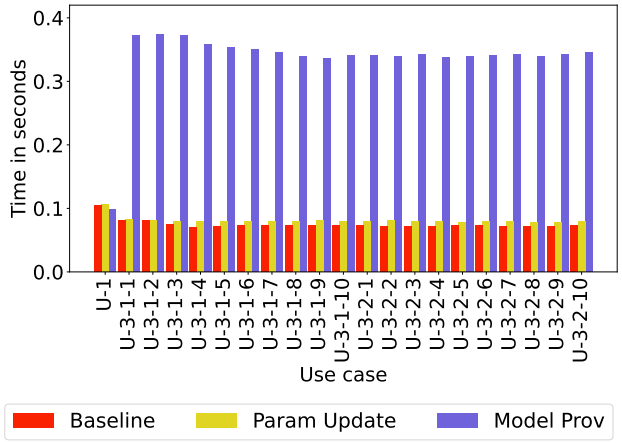


Figure 14: Comparison of the median time-to-save (TTS) for fully updated MobileNetV2 versions across approaches on the DIST-20 evaluation flow. All models in U_3 were trained on the Custom-Coco-Outdoor-512 dataset.

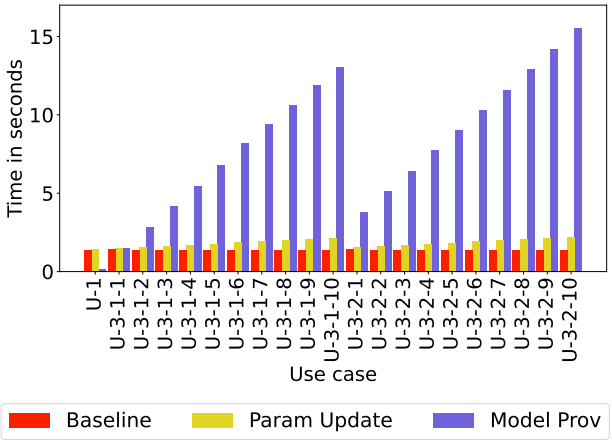


Figure 15: Comparison of the median time-to-recover (TTR) for fully updated MobileNetV2 versions across approaches on the DIST-20 evaluation flow. All models in U_3 were trained on the Custom-Coco-Outdoor-512 dataset.

Storage Consumption. Analyzing the raw numbers for DIST-5, DIST-10, and DIST-20, we see that the storage consumption for the BA strongly depends on the number of parameters. For the PUA, the critical factor is the size of the parameter update, and for the MPA, the storage consumption is dominated by the training dataset. In detail, we find that the storage consumption is, for a given approach and use case, constant across all evaluation flows in Table 3 and does not vary across different executions of the same experiment.

We conclude that the detailed discussion of the storage consumption for the standard evaluation flow in Section 4.2 also holds for all other evaluation flows we have evaluated. This shows once more the potential of the PUA and the MPA to outperform the BA in terms of storage consumption for certain scenarios.

Time-to-save. All distributed evaluation flows (DIST-5, DIST-10, and DIST-20) show very similar numbers and trends. The only difference between the distributed evaluation flows is the amount of measured times per use case which dependents on the number of used nodes.

Figure 14 shows the TTS for the DIST-20 evaluation flow. Executing the evaluation flow with 20 nodes, results in 20 measurements per iteration of U_3. The numbers for the individual nodes are very similar for a given use case. To increase the readability, we aggregate the values per iteration of a use case by taking the median time of all nodes.

We see that the TTS depends on the approach and varies only slightly across the different iterations of the use cases. Saving a fully updated MobileNetV2 version using the BA or the PUA takes just under 0.1 seconds. Saving the same models using the MPA takes under 0.4 seconds.

The reason that the TTS for the PUA is not lower than the TTS for the BA is that we save fully updated and not partially updated model versions. Thus, both the BA and the PUA save an almost equal amount of data which takes approximately the same amount of time.

The TTS for the MPA is significantly higher than the TTS for both other approaches because the MPA has to save significantly more data. The BA and the PUA have to save around 14 MB, whereas the MPA has to save around 72 MB.

Comparing these numbers to the ones from Section 4.3, we see that we can find approximately equal TTSs and the same trends for the larger evaluation flows and the standard evaluation flow.

Time-to-recover. For the TTR, we also show and analyze the results for DIST-20 and take the median times of all nodes per use case. The evaluation of DIST-5, DIST-10, and DIST-20 show equal numbers and trends.

In Figure 15, we show that the TTR for the BA stays on a constant level across all use cases. The reason is that the BA saves all models independently of each other.

The TTR for the PUA and the MPA follows a staircase pattern starting at U_1 and $U_{3\text{-}2\text{-}1}$. This is due to the PUA's and the MPA's recursive recovery process and how the saved models are related to each other. The TTR for the MPA is significantly higher because we have to repeat the model training, whereas, for the PUA, we apply parameter updates.

Overall, the patterns and absolute numbers we see in Figure 15 for DIST-20 match the numbers and trends we already have seen for the standard evaluation flow in Figure 11(a). The only difference between the standard and the DIST-20 evaluation flow is that for DIST-20 we have ten instead of four iterations of U_3. This leads to higher maximum TTRs.

Summarizing the results of evaluating all approaches on more extensive evaluation flows, we find that the **storage consummation is constant regardless of the evaluation flow. The TTS and TTR show similar numbers and the same trends across the standard and all of the more extensive evaluation flows. This shows that all approaches scale to evaluation flows similar to our motivational example.**

4.7 Discussion

In this section, we discuss which approach is the most suitable for a given scenario, as well as the storage-retraining trade-off and the impact of combining the three approaches.

Optimized Time-to-recover. In each of the scenarios presented, we find one approach (PUA or MPA) that outperforms the BA in terms of storage consumption and time-to-save (TTS). The drawback of these approaches is an increased TTR that is marginally higher for the PUA and drastically higher for the MPA. Therefore,

if, for a given setting, the TTR has the highest priority, the BA is the preferred choice.

Optimized Storage Consumption and Time-to-save. If we assume that models are rarely recovered, but derived models are saved frequently, we can focus on the TTS and the storage consumption.

If we have no information about the hardware environment and if the dataset is larger than the model, the PUA is the preferred choice. If we work in a domain with large models, but small datasets (for example, natural language processing) or frequent model training on small datasets, the MPA is the best approach for storage consumption and TTS.

Especially for the MPA, it is crucial to consider the overall environment and setup. A scenario in which the MPA could be the preferred choice is when the training data is saved and transferred to a central server regardless of the approach we choose (e.g., for analysis purposes). In this case, the MPA would not have to save the dataset and the storage consumption reduces to the training information.

The MPA's performance is particularly dependent on the domain we operate in because this usually determines the approximate size of the dataset, the complexity of the model, and the time to train the model. For example, for natural language processing, we would expect complex models and long training times but small datasets. For video processing, we would expect to see large datasets and moderately complex models and training times compared to natural language processing. The perfect domain for the MPA would be short training times, small datasets, and large models.

Storage-Retraining Tradeoff. When choosing an approach for a specific scenario, we always have a storage-retraining tradeoff problem. Either, we chose the BA that does not optimize the storage consumption, but the TTR. Or, we choose the PUA or the MPA to reduce the storage consumption and accept their effect on the TTR.

For a given scenario, we should always consider how much TTR (and resources) we want to invest to save storage and bandwidth in case we have to transmit the model.

Adaptive Approach. A possible direction for future work is to use a heuristic that decides which is the most suitable approach (BA, PUA, or the MPA) for every model. One heuristic could be based on the fact that the BA and the PUA mainly depend on the model parameters, whereas the MPA primarily depends on the dataset. A more complex heuristic could be based on a formalized tradeoff as presented by Derakhshan et al. [5] and Vartak et al. [40] combined with some given parameters, such as, maximum storage consumption or TTR.

5 RELATED WORK

Model and life cycle management tackle the problem of documenting and monitoring ML experiments and resulting models. While Schelter et al. [31] give an overview of the conceptual, engineering, and data-processing related challenges in this field, *ModelDB* [41], *ModelDB-2.0* [42] *ModelHub* [21], *ModelKB* [9], and *Runway* [39] describe software solutions for model management and life cycle management.

Both *ModelKB* and *Runway* focus on automatically managing ML experiments by saving corresponding metadata but not the model itself. *ModelKB* considers saving, sharing, and reproducing of models as future work, whereas *Runway* only saves references to models and artifacts. Both approaches do not provide metadata detailed enough to reproduce the model training.

ModelDB and its successor *ModelDB-2.0* are git-like versioning tools for model management. *ModelDB-2.0* focuses on saving the *ingredients of a model*, such as code artifacts, training configuration, and the environment. In contrast with MMlib, the level of detail for the saved information is not sufficient to reproduce model training in a way that would lead to the exact same models. In addition to metadata and code, *ModelDB-2.0* optionally saves model parameters. MMlib's baseline serializes a model by using the functionality offered by PyTorch. At the time of writing, *ModelDB-2.0* makes use of the same method. Therefore, MMlib's baseline and *ModelDB-2.0* can be seen as equivalent approaches when it comes to saving a model.

ModelHub's parameter archival storage (PAS) "minimizes [the] storage footprint and accelerates query workloads with minimal loss of accuracy" [21]. It is the approach closest to MMlib, but its design is orthogonal in two aspects. (1) We designed MMlib to recover the exact same models. In contrast to that, the ModelHub authors assume that it is in most cases sufficient to recover an approximate version of a model. This is why ModelHub saves model parameters in a segmented format. The first step to recover a model is to load the most significant bits of the model parameters. Only when requested, ModelHub will load the remaining segments of the parameters to recover the complete model from external storage. (2) MMlib was designed under the assumption of frequent model updates (U_3) but rarely occurring model recoveries (U_4) that, thus, can be time-consuming. ModelHub is optimized for recovering (approximate) models (U_4) and does not focus on frequently saving new models (U_3). This can be seen by the multi-stage model recovery process described above or by their complex algorithms (worse than quadratic run time) for saving a given set of models.

6 CONCLUSION

In this paper, we investigate the problem of storing and recovering machine learning models exactly in distributed environments. We present three approaches to efficiently manage DL models with regards to storage consumption, time-to-save (TTS), and time-to-recover (TTR). We develop a baseline approach (BA) and two advanced approaches: a parameter update approach (PUA) and a model provenance approach (MPA). Our evaluation shows that the provenance approach outperforms the baseline by up to 15.8% in TTS and 70.0% in storage consumption. The parameter update approach also outperforms the baseline by up to 51.7% in TTS and 95.6% in storage consumption. In distributed environments of different sizes, all approaches show consistent results with respect to storage consumption, TTS, and TTR across all experiments. Based on our findings on how the model architecture, model relation, and training dataset size influence the approaches' performance, we discuss the best approach for given scenarios and present ideas to combine them to optimize performance further. All approaches and the probing tool to verify the reproducibility of model architectures are publicly available in our Python library *MMlib*.

ACKNOWLEDGMENTS

This work was partially funded by the German Ministry for Education and Research (ref. 01IS18025A and ref. 01IS18037A), the German Research Foundation (ref. 414984028), and the European Union's Horizon 2020 research and innovation programme (ref. 957407).

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal, Sethu Raman, and others. 2019. Data platform for machine learning. In *Proceedings of the 2019 International Conference on Management of Data*. 1803–1816.

[3] Lorena A Barba. 2018. Terminologies for reproducible research. *arXiv preprint arXiv:1802.03311* (2018).

[4] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 2085–2088. https://doi.org/10.1145/2882903.2899401 event-place: San Francisco, California, USA.

[5] 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.

[6] ECMA. 2017. ECMA-404: The JSON Data Interchange Syntax. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

[7] Facebook. 2021. PyTorch. https://www.pytorch.org

[8] Gharib Gharibi, Vijay Walunj, Rakan Alanazi, Sirisha Rella, and Yugyung Lee. 2019. Automated management of deep learning experiments. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*. 1–4.

[9] Gharib Gharibi, Vijay Walunj, Sirisha Rella, and Yugyung Lee. 2019. ModelKB: Towards Automated Management of the Modeling Lifecycle in Deep Learning. In *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. 28–34.

[10] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 249–256.

[11] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)* 23, 1 (1991), 5–48. Publisher: ACM New York, NY, USA.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

[13] Google. 2021. Machine Learning Glossary. https://developers.google.com/machine-learning/glossary

[14] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics* 37, 3 (2020), 362–386. Publisher: Wiley Online Library.

[15] Alex Guazzelli, Michael Zeller, Wen-Ching Lin, Graham Williams, and others. 2009. PMML: An open standard for sharing models. *R J.* 1, 1 (2009), 60.

[16] Matthew Hartley and Tjelvar SG Olsson. 2020. dtoolAI: Reproducibility for Deep Learning. *Patterns* 1, 5 (2020), 100073. Publisher: Elsevier.

[17] 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.

[18] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2015. Microsoft COCO: Common Objects in Context. _eprint: 1405.0312.

[19] Association for Computing Machinery. 2021. Artifact Review and Badging. https://www.acm.org/publications/policies/artifact-review-and-badging-current

[20] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.

[21] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. 2017. Modelhub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1393–1394.

[22] MongoDB. 2021. MongoDB. https://www.mongodb.com

[23] National Academies of Sciences, Engineering, and Medicine and others. 2019. *Reproducibility and replicability in science*. National Academies Press.

[24] NVIDIA. 2021. CUDA Toolkit Documentation – Floating Point and IEEE 754 Compliance for NVIDIA GPUs. https://docs.nvidia.com/cuda/floating-point/index.html

[25] ONNX. 2021. ONNX. https://onnx.ai/

[26] OpenAI. 2018. AI and Compute. https://openai.com/blog/ai-and-compute/

[27] Jim Pivarski, Collin Bennett, and Robert L Grossman. 2016. Deploying analytics with the portable format for analytics (PFA). In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 579–588.

[28] Duncan Riach. 2019. Determinism in Deep Learning. Published: NVIDIA's GPU Technology Conference.

[29] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, and others. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115, 3 (2015), 211–252. Publisher: Springer.

[30] 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.

[31] Sebastian Schelter, Felix Bießmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. 2018. On Challenges in Machine Learning Model Management. *IEEE Data Eng. Bull.* 41, 4 (2018), 5–15.

[32] Sebastian Schelter, Joos-Hendrik Boese, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. 2017. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems Workshop at NIPS*. 27–29.

[33] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015), 2503–2511.

[34] Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data* 6, 1 (2019), 1–48. Publisher: Springer.

[35] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958. Publisher: JMLR. org.

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[37] Tesla. 2021. Tesla AI Day. https://www.youtube.com/watch?v=j0z4FweCy4M&t=8607s

[38] Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 242–264.

[39] Jason Tsay, Todd Mummert, Norman Bobroff, Alan Braz, Peter Westerink, and Martin Hirzel. 2018. Runway: machine learning model experiment management tool.

[40] Manasi Vartak, Joana M F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data*. 1285–1300.

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

[42] VertaAI. 2021. ModelDB: An open-source system for Machine Learning model versioning, metadata, and experiment management. https://github.com/VertaAI/modeldb

[43] Geoffrey I Webb, Roy Hyde, Hong Cao, Hai Long Nguyen, and Francois Petitjean. 2016. Characterizing concept drift. *Data Mining and Knowledge Discovery* 30, 4 (2016), 964–994. Publisher: Springer.