Hasso Plattner Institute

Master Thesis

Model Management in Distributed Environments

Modellverwaltung in verteilten Umgebungen

by

Nils Straßenburg

First Supervisor

Prof. Dr. Tilmann Rabl

Chair for Data Engineering Systems

Second Supervisor

Prof. Dr. Bert Arnrich

Chair for Connected Healthcare

Advisor

Ilin Tolovski

Chair for Data Engineering Systems

Potsdam, July 20th, 2021

Acknowledgements

I would like to thank the following people for all of their kindness and support during my master thesis. Firstly, thanks to both my supervisors Ilin Tolovski and Tilmann Rabl, for their assistance, guidance, and reassurance throughout the process.

Thanks to Saoirse, Helen, and Patrick Moriarty for their huge help proofreading and grammatical insights! Thanks also to Bjarke and Claudia Lillinger for their additional proofreading support.

Zu guter Letzt möchte ich – wohl wissend, dass diese Arbeit primär auf Englisch verfasst ist – ein paar Worte an einen Teil meiner Unterstützer und Freunde in ihrer Muttersprache richten.

Meinen Eltern, Sabine und Frank Straßenburg, danke ich für ihre tatkräftige Unterstützung in jeglicher Lebenslage. Auch meiner restlichen Familie danke ich ganz herzlich. Dabei denke ich insbesondere an meine Großeltern.

Speziell erwähnen möchte ich zudem meinen Bruder im Geiste, Maximilian Kroschewski. Mit Abgabe dieser Arbeit gehen über sechs Jahre Studium zu Ende, eine lange Zeit, die wir zu großen Teilen in Vorlesungen, Übungen, Projekten, Seminaren und sogar im Auslandspraktikum Seite an Seite verbracht haben. Durch dich hat mir mein Studium noch mehr Spaß und Freude bereitet. Es war mir eine Ehre, ich werde es vermissen!

Abstract

Over the last few years, deep learning (DL) has revolutionized many domains by significantly outperforming previous approaches and became essential for many software products. To guarantee reliable and consistent performance, models that are used need not only to be adjusted, improved, and retrained but also documented, deployed, and monitored from a central location. An essential part of this set of processes, referred to as model management (MM), is to save and recover models; which ideally happens without loss of precision. Existing approaches in MM either compress the model with loss of precision or focus on metadata and use a naive approach to saving a model.

In this thesis we investigate if, and by how much, we can outperform a baseline approach capable of naively saving and recovering models without loss of precision, while focusing on metrics of storage consumption, time-to-save, and time-to-recover. We develop a set of approaches consisting of a baseline approach $\mathcal{B}$ that saves complete model snapshots, a parameter update approach $\mathcal{U}'_P$ that saves the updated model data, and a provenance approach $\mathcal{M}_{Prov}$ that saves the model's provenance instead of the model itself. In addition to these approaches, we also develop a probing tool to determine if we can reproduce the inference and training of a given model across different machines.

Evaluating all approaches in a distributed environment on different model architectures, model datasets, and model relations, we show that $\mathcal{M}_{Prov}$ outperforms the baseline by up to 70% and $\mathcal{U}'_P$ by up to 95.6% in terms of storage consumption. While both approaches have the potential to achieve a similar or slightly shorter time-to-save, they come with a longer time-to-recover. We find that if and by how much, a given approach outperforms a baseline strongly depends on factors such as the dataset used, the model architecture, and how many model parameters stay fixed across different model versions.

Zusammenfassung

Deep Learning erzielt deutlich bessere Ergebnisse als traditionelle Ansätze und macht es daher für viele Softwareprodukte unverzichtbar. Um eine zuverlässige und konsistente Güte der Ergebnisse zu gewährleisten, müssen alle verwendeten Modelle nicht nur kontinuierlich angepasst, verbessert und neu trainiert werden, sondern auch dokumentiert, bereitgestellt und von einem zentralen Ort aus überwacht werden. Ein wesentlicher Bestandteil dieser als Modellverwaltung bezeichneten Prozesse ist das Speichern und Wiederherstellen von Modellen. Idealerweise geschieht dies ohne Präzisionsverlust. Bei der Analyse existierender Ansätze zur Modellverwaltung ergab sich, dass die bestehenden Ansätze die Modelle entweder nicht verlustfrei komprimieren oder sich auf die Speicherung von Metadaten konzentrieren und für das Modell eine naive Speichermethode wählen.

In dieser Arbeit untersuchen wir, ob und um wie viel wir einen Basisansatz, der in der Lage ist, Modelle ohne Präzisionsverlust zu speichern und wiederherzustellen, übertreffen können. Hierbei konzentrieren wir uns auf den Speicherverbrauch sowie die Dauer zum Speichern und Wiederherstellen eines gegebenen Modells. Wir entwickeln eine Reihe von Ansätzen, bestehend aus einem Basisansatz $\mathcal{B}$, der komplette Modelle speichert, einem Parameter-Update-Ansatz $\mathcal{U}'_P$, der nur die aktualisierten Modelldaten speichert, und einem Modell-Herkunfts-Ansatz $\mathcal{M}_{Prov}$, der die Herkunft und den Entstehungsprozess des Modells anstelle des Modells selbst speichert. Neben den Ansätzen selbst implementieren wir eine Untersuchungssoftware, um zu testen, ob wir die Inferenz und das Training eines gegebenen Modells über verschiedene Maschinen hinweg reproduzieren können.

Wir evaluieren alle Ansätze in einer verteilten Umgebung und betrachten verschiedene Modellarchitekturen, Datensätze und Modellrelationen. Hierbei zeigen wir, dass $\mathcal{M}_{Prov}$ den Speicherverbrauch um bis zu 70% und $\mathcal{U}'_P$ den Speicherverbrauch um bis zu 95,6% im Vergleich zum Basisansatz reduziert. Während beide dieser Ansätze das Potenzial haben, eine ähnliche oder sogar etwas kürzere Zeit für das Speichern eines Modells zu benötigen, gehen sie mit einem längeren Wiederherstellungsprozess einher. Insgesamt stellen wir fest, dass es stark von Faktoren wie dem verwendeten Datensatz, der Modellarchitektur und der Anzahl an veränderten Parametern abhängt, ob und um wie viel ein bestimmter Ansatz den Basisansatz verbessert.

Contents

1. Introduction

Over the past few years, DL has revolutionized many domains by significantly outperforming previous non-DL approaches [13]. Nowadays, DL is essential for many software products such as face recognition, virtual assistants, or recommender systems. Even in safety-critical environments including autonomous driving, DL plays a key role [16].

When included as part of frequently used software, DL models change regularly. To guarantee reliable and consistent performance at all times, the models not only need to be adjusted, improved, and retrained but also documented, deployed, and monitored from a central location. The process surrounding this is called model management [44]. Model management is essential to develop and deploy models reliably and is not only helpful but necessary in safety-critical environments.

An essential reason to manage models is to document insights and reproduce results consistently. For this purpose, it is not sufficient to look at high-level metadata; it is necessary to reproduce or to load the actual model. Ideally, reproducing or loading a model is possible without loss of precision, meaning that we can access the same model we saved and not only an approximate version. This is especially beneficial for debugging purposes in safety-critical but also non-safety-critical environments.

Saving models without loss of precision can be easily achieved by saving a complete snapshot of every model, however, the problem with this approach is that a standard DL model can easily be larger than 100 MB which results in a non-negligible storage consumption for multiple frequently updated models. Although storage is cheap, it is nevertheless desirable to find easy and reliable ways to reduce the overall storage consumption. Furthermore, even for a single model it can make sense to save storage, for example, when we want to transfer it with limited available bandwidth.

Taking a look at related work in the field of model management and life cycle management we find that all approaches have one of the following three drawbacks: (1) they do not save the model but only high level metadata that is not sufficient to reproduce experiments exactly [10, 45, 53], (2) they do not focus on storage consumption and save models in a naive way [56] (3) if they optimize for storage consumption they focus on lossy compression of floating-point numbers which makes it impossible to recover the model parameters without loss of precision [28, 29].

The overall research question for this thesis is if and by how much we can outperform a baseline capable of saving and recovering models without loss of precision while focusing on the metrics of: storage consumption, time-to-save, and time-to-recover.

Contributions To answer the research question we make the following contributions:

1. We develop a baseline approach that is capable of saving and recovering models without loss of precision.

2. We develop a parameter update approach that saves only parameter updates instead of a complete model.

3. We discuss how to reproduce model training and implement a probing tool to in-

vestigate the reproducibility of models on a layer granularity across different environments and machines.

4. We use our knowledge on reproducible training for a provenance approach that saves a model without saving its parameters.

5. We extensively evaluate all approaches regarding storage consumption, time-to-save, and time-to-recover to discuss which approach to use for specific situations.

6. We integrate all our approaches in a Python framework to make them comfortably accessible and usable.

7. We developed a data schema that is generic enough to be easily extended for further approaches.

Thesis Structure In Section 2, we briefly discuss related work in the fields of model management, model saving and compression, storage-recreation tradeoffs, and reproducibility. Section 3 covers the foundations for developing all our approaches by giving a high-level overview of DL, discussing how to reproduce model training, and defining related terms.

Section 4 describes the use cases we want to cover with our approaches and set assumptions and focus. We present our baseline approach saving complete model snapshots in Section 5, a parameter update approach saving only the updated information of a model in Section 6, and a provenance approach that saves models based on their provenance data in Section 7.

In Section 8, we describe our experimental setup and evaluate all approaches extensively for storage consumption, time-to-save, and time-to-recover to discuss which approach is particularly suitable in specific scenarios.

In Section 9, we conclude our work and give an outlook on future work.

2. Related Work

In this section we briefly discuss related work that is relevant for developing our approaches. We focus on four fields: model management, model saving and compression, storage-recreation tradeoff, and the definition of reproducibility.

Model Management Model management and life cycle management tackle the problem of documenting and monitoring machine learning (ML) experiments and resulting models. While Schelter et al. [44] give an overview of conceptual, engineering, and data-processing related challenges in this field, *ModelDB* [55], *ModelDB-2.0* [56], *ModelHub* [28, 29], *ModelKB* [10], and *Runway* [53] 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* is considering saving, sharing, and reproducing of models as future work whereas, *Runway* only saves references to models and artifacts.

ModelDB, its successor *ModelDB-2.0*, and *ModelHub* are git like versioning tools for model management and life cycle management focusing on reproducing experiments and efficiently saving related data.

ModelDB-2.0 optionally saves model parameters but mainly focuses on saving the "*ingredients*" to reproduce experiments (e.g., the code, its configuration, the data used, and the environment). At the time of writing, to save the model parameters *ModelDB-2.0* uses a naive solution depending on the ML/DL framework used; the saved meta data is not extensive enough to reproduce model training at a level of detail desired. Research by Amazon [45] and Publio et al. [35] has defined schemata covering data related to DL experiments.

Saving and Compressing Models *ModelHub* sets its focus on efficiently saving model parameters in its so-called *Parameter Archival Storage System* and evaluates multiple compression techniques for floating-point numbers. *ISOBAR* [46] and *Pstore* [3] present further work on compressing floating-point numbers. Although these approaches are capable of reducing the storage consumption, they are not applicable in our scenario because they do not work losslessly.

Han et al. [18] and Joseph et al. [22] propose a technique to compress a model by pruning and sharing weights instead of compressing floating-point numbers. The drawback of these approaches for our setting is that they require extensive retraining and hyperparameter tuning. Moreover, although they almost reach the same results for accuracy, they alter the model to compress it.

Storage-Recreation Tradeoff Vartak et al. [54] define a system to save *model intermediates* (the output of distinct model layers) and formalize a storage-recreation tradeoff to address the problem of optimally using a limited amount of storage. Derakhshan et al. [8] define a similar tradeoff between saving and re-executing ML experiments in collaborative environments.

Definition of Reproducibility The National Academy of Science [32] and Barba et al. [2] 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 [9]. Hartley et al. [19] define repeatability, reproducibility, and replicability in the context of DL and present a basic approach implementing their guidelines for reproducible model training.

3. Background

In this section, we cover the foundations required to develop our approaches. We start with a high-level overview of DL and define related terms in Section 3.1. In Section 3.2 we define the terms recoverability, reproducibility, and provenance before we move on to a brief overview of the IEEE 754 standard on representing floating-point numbers in Section 3.3. Finally, we discuss in Section 3.4 how to reproduce the training of DL models.

3.1. Deep Learning

Deep learning (DL) is a sub-discipline of machine learning (ML), which is itself a sub-discipline of artificial intelligence (AI) [13]. In this thesis, we focus on supervised DL, meaning that models are trained using labeled input data [14].

In the following section, we do not explain the whole field of DL – for this, we refer to other authors [13] – but we explain the high-level steps of training a DL model to put essential terms used throughout this thesis into context. We adapt the example of developing a model to classify images according to whether they contain houses, cars, persons, or pets from LeCun et al. [26].

Before we can develop a model, we have to collect representative examples with corresponding labels; in our case, pictures of houses, cars, persons, and pets. We call this data collection a **dataset** that is usually split into a **training**, a **validation**, and a **test set**.

Next, we choose a **model architecture**. Most likely, we start with an existing architecture that has been shown to work in a similar context and adjust it. In our example, the starting point is probably a well-known convolutional neural network (CNN) [25].

The initial version of a **model** is unlikely to classify images containing houses, cars, persons, or pets very well. To change this, we have to **train** it by repeating the following steps several times: (1) We use a **data loader** to load a **batch** of samples and corresponding labels (in our example, the samples are images and the labels integers representing the image class) from the **training set**. (2) We let the model predict labels for the loaded images. (3) Using a **loss function** we measure how much the predicted labels differ from the actual labels. (4) We utilize an algorithm called **backpropagation** that uses the chain rule to efficiently compute the gradient for the computed loss with respect to every model's **parameters'**. The computed gradients give an indication of by how much and in which direction we have to change their associated parameter to reduce the overall loss – which is the ultimate goal. (5) Using an implementation of a gradient descent algorithm called an **optimizer** [14], we decide how exactly to update the model parameters. While simple optimizers base their decision only on the computed gradients, advanced optimizers often take information about past parameter updates into account too.

Having adjusted the model according to every sample in the training set once, we say that the model was trained for one **epoch**. After finishing an epoch, we typically validate the model's performance on the validation set (in our example, how well the model can predict whether a given image contains houses, cars, persons, or pets). Based on the result, we decide whether to train the model for another epoch or alternatively stop the training process and evaluate the model's performance on the test set.

Finally, we want to point out that the procedure above is a simplified description; in practice, all the steps can be considered research areas on their own.

3.1.1. Model and Dataset

The definitions for the terms *model* and *dataset* can be differently defined across domains and contexts. Because of these terms' importance for this thesis, we define them explicitly. For all other terms described above, we use the definitions given in Google's Machine Learning Glossary [14].

In the context of ML and DL Frameworks, the term *model* is overloaded. It could, for example, refer to the computational graph that defines "*how a prediction will be computed*" or to the weights and biases of the computational graph determined by training [14]. Throughout this thesis, we use the following definition:

Definition 1. *A **model** $M = (M_a, M_p)$ is the representation of what a machine learning system has learned from the training data [14], and consists of two pieces:*

- ***model architecture** (M_a): The computational structure for making a prediction*
- ***model parameters** (M_p) [1]: The specific weights and biases determined by training.*

If the model is a neural network (NN), the model architecture M_a consists of different layers that use corresponding parameters. We denote the layers of M_a as $M_a = (M_a^1, \ldots, M_a^n)$ and the parameters M_p as $M_p = (M_p^1, \ldots, M_p^n)$, where M_p^i denotes the parameters of layer M_a^i.

Definition 2. *We slightly adjust the definitions from Google's Machine Learning Glossary [14] to define dataset, training set, validation set, and test set. We explicitly point out that only the training set is used to fit the model parameters.*

*A **dataset** is a collection of examples. It is usually split into:*

- ***training set**: A subset of the dataset used to train a model. It is the only subset of the dataset that is used to fit the model parameters.*
- ***validation set**: A subset of the dataset, disjoint from the **training set** that is used for model validation.*
- ***test set**: The subset of the dataset, disjoint from **training set** and **validation set**, that is used to test a model after the model has gone through initial vetting by the **validation set**.*

[1] Not to be confused with hyperparameters

3.1.2. Model Relations

In DL it is common practice to use well-established models as a base to develop new models. We come up with precise definitions of how models can relate to each other in Definition 3 and illustrate them in Figure 1.

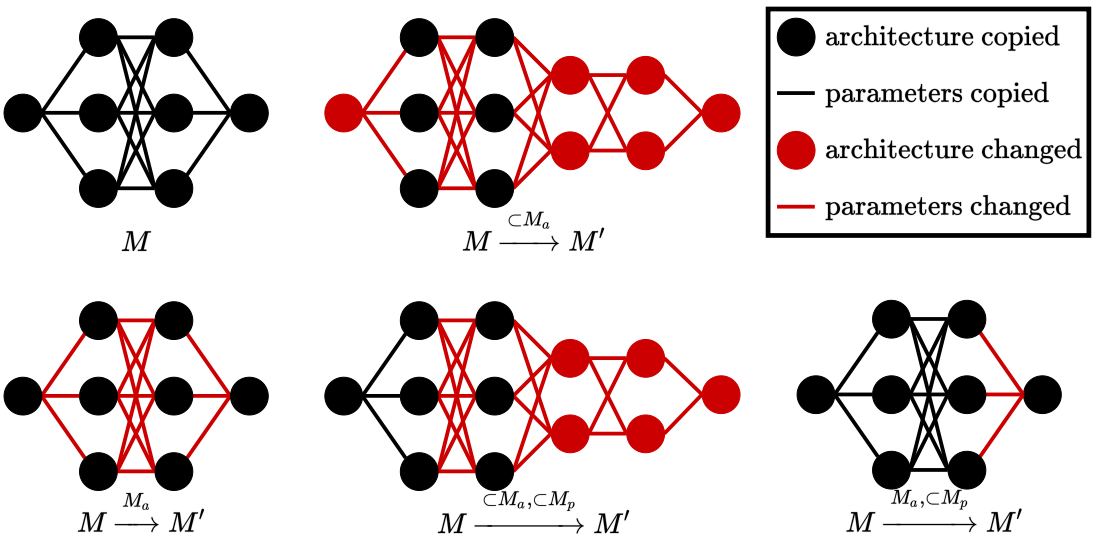


Figure 1: Model relations (left to right, top to bottom): *base model*, *derived architecture*, *version*, *extended*, and *fine-tuned version*.

Definition 3. *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*. We denote this as* $M \to M'$*.*
- *If* M *is the base model of* M'*, and* M' *copies parts of* M*'s architecture* $(M \to M' \wedge M_a \cap M'_a \neq \emptyset)$*, we say* M'*'s model* ***architecture is derived*** *from* M *and denote this as* $M \xrightarrow{\subset M_a} M'_a$ *or* $M \xrightarrow{\subset M_a} M'$
- *If* M'*'s architecture is derived from* M *in a way that* M' *has exactly the same model architecture as* M *but the parameters differ* $(M_a = M'_a \wedge M_p \neq M'_p)$*, we say* M' *is a* ***version*** *of* M *and denote it as* $M \xrightarrow{M_a} M'$*.*
- *If* M'*'s architecture is derived from* M *and also uses the same parameters for a subset of the derived part of the architecture, we say* M' ***extends*** M *and denote this as* $M \xrightarrow{\subset M_a, \subset M_p} M'$*.*
- *If* M' *is a version of* M*, and* M' *extends* M*, we say* M' *is a* ***fine-tuned version*** *of* M *and denote this as* $M \xrightarrow{M_a, \subset M_p} M'$*.*
- *If* $M_a = M'_a \wedge M_p = M'_p$ *we say* M *and* M' *are* ***equal****.*

3.2. Recoverability, Reproducibility, and Provenance

While the term *recoverability* is important throughout the whole thesis, the terms *reproducibility* and *provenance* are explicitly relevant for the approach presented in Section 7. In the following, we define these terms in the context of DL.

Definition 4. *A model is* ***recoverable*** *from given data, if the data provide sufficient information to reconstruct the model (meaning architecture and parameters) from them.*

Definition 5. *To define reproducibility for a model, we adapt the definition by Hartley et al. [19] that conforms with the terms' usage by the ACM [9]* [2] *saying: "Reproducibility is the ability to regenerate results using the original researchers' data, software, and parameters."*

Reproducible Model

- *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.*

With the definition of model *recoverability* 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.

Definition 6. *The* ***provenance*** *of a model is the history of the processes used to produce it, together with the input/training data. (adapted from [19, 31])*

3.3. IEEE 754 Standard for Floating-Point Arithmetic

In DL frameworks, it is common to represent model parameters as floating-point numbers. For example, in PyTorch [37], parameters are represented as tensors with the default tensor type *torch.FloatTensor*. A tensor of that type has the datatype *float32* and represents a floating-point number using 32 bits.

The most widely used standard for floating-point numbers is the IEEE 754 standard [21]. It represents 32-bit floating-point numbers by a *sign* (1 bit), an *exponent* (8 bit), and a *mantissa* (23 bit).

We can convert a number from the decimal to the binary IEEE 754 format by performing the following steps. As an example, we use the number 42.0. The conversion is shown in

[2] To prevent the reader from being confused when checking the term definitions in the sources, we need to mention here that when the work of Hartley et al. [19] was published, it did not conform with the ACM definitions. Due to inconsistencies in the term usage, especially for *recoverability* and *replicability* discussed by Barba et al. [2], the ACM adjusted their term definitions [9].

Figure 2: (1) We convert the number to its binary representation ($42.0_{10} \rightarrow 101010.0_2$). (2) We normalize the binary representation so that the number has the form $1.xxx * 2^e$ ($101010.0_2 * 2^0 \rightarrow 1.01010 * 2^5$). (3) If the number is positive, we set the sign to 0, otherwise to 1. (4) We calculate the IEEE 754 exponent by adding 127 to the exponent for our normalized representation and convert this number to binary ($e = 5_{10}, 5_{10} + 127_{10} = 132_{10} \rightarrow 10000100_2$). (5) We take the first 23 bits in our binary representation after the decimal point to form the mantissa; if needed, we round.

The fact that we use a finite number of bits to represent the floating-point numbers implies that we can, for example, represent 42_{10} precisely in binary, but 0.1_{10} only approximately. Figure 2 shows that the decimal value of the IEEE 754 representation for 0.1_{10} is not 0.1_{10} but a slightly higher number.

We discuss the implications of using this representation for floating-point numbers relevant for this thesis later in this section.

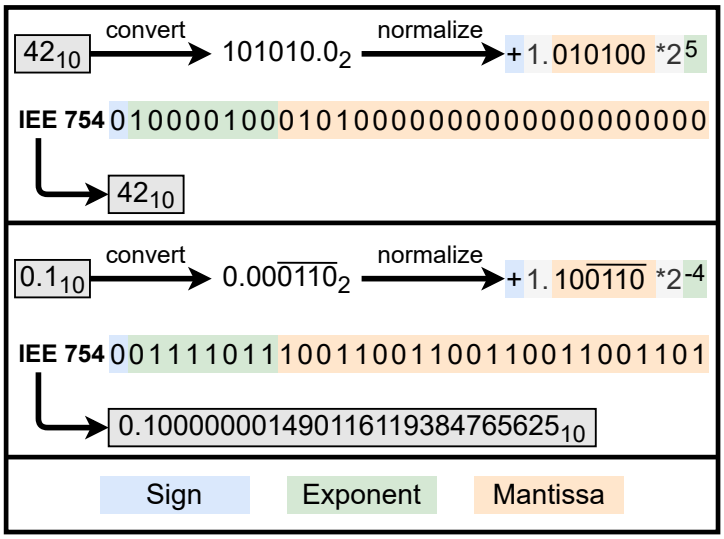


Figure 2: Example of converting the decimal numbers 42.0 and 0.1 to the IEEE 754 representation and back to the decimal representation.

3.4. Reproducibility in Deep Learning

For neural networks (NNs) it is neither possible to reproduce model inference nor to reproduce model training by default [41, 47]. This section will discuss factors that prevent the inference and training of NNs from being reproducible and ways to eliminate them.

3.4.1. Code, Parameters, and Data

If the experiment's code 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 to this problem is tracking the code used, the parameters, and the input data to use them consistently across executions. Applied to the deep learning (DL)

domain, we have to keep track of the model's code, the inference and training routine, the loss function(s) used, and the optimizer together with all parameters. To ensure the same input data, we have to keep track of the data itself and how it is provided by components such as the preprocessor or the dataloader.

3.4.2. Intentional Randomness

Randomness is often intentionally introduced as part of specific machine learning (ML) algorithms. Examples in DL are random data augmentation [48], random weight initialization [11], and regularization layers such as dropout [49].

Since Computers cannot generate real randomness, they use pseudorandom number generators (PRNGs) which is also the case for Python and related Frameworks. The output of PRNGs is entirely dependent on an initial value called *seed*. Therefore, setting the *seed* leads to reproducible pseudorandomness.

3.4.3. Floating-point Arithmetic

Taking the above discussions into account, we can see that given the same code and eliminating sources of randomness, the root cause of non-determinism are operators utilizing floating-point arithmetic (FPA) [41]. The reason for this is that, as seen in Figure 2, not all floating-point numbers can be represented exactly with a finite number of bytes; they must be approximated by rounding [12]. This implies that different implementations of the same operator or the order of calculation can lead to different results [33, 41].

Figure 3 demonstrates how different implementations of the same operator cause different results: we computed the dot-product using the serial and the parallel method. From a mathematical perspective, both computations should lead to the same result, but in practice, they differ. Figure 4 shows a similar scenario: here, we implemented the operator in the same way, but the input order differs, and so does the final result [3]. The dot-product is only one example, and many more operators, implementation details, and optimizations can cause non-reproducible results [33].

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.

[3]The code for both examples can be found in the thesis's GitHub repository. `https://github.com/slin96/master-thesis/blob/main/experiments/floating-point/dot_product_exp.ipynb`

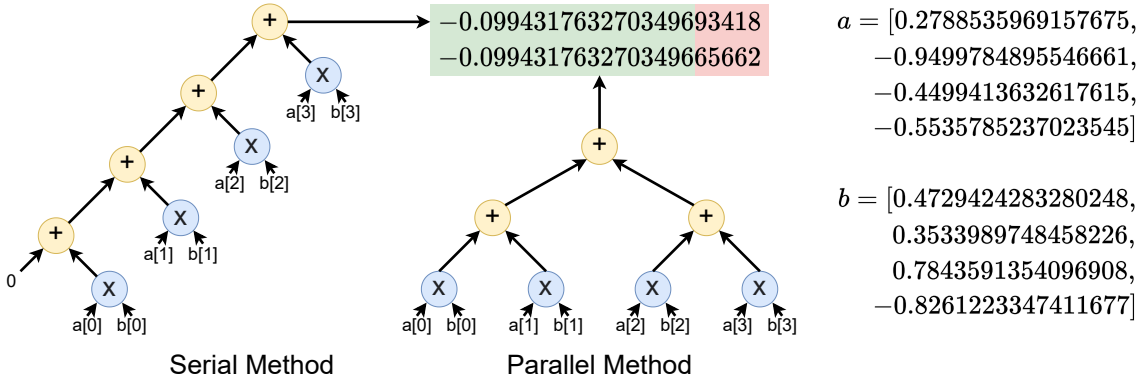


Figure 3: Example of different computation results for the dot-product comparing the serial and the parallel method (Inspired by Figure 3 and Figure 5 in [33]).

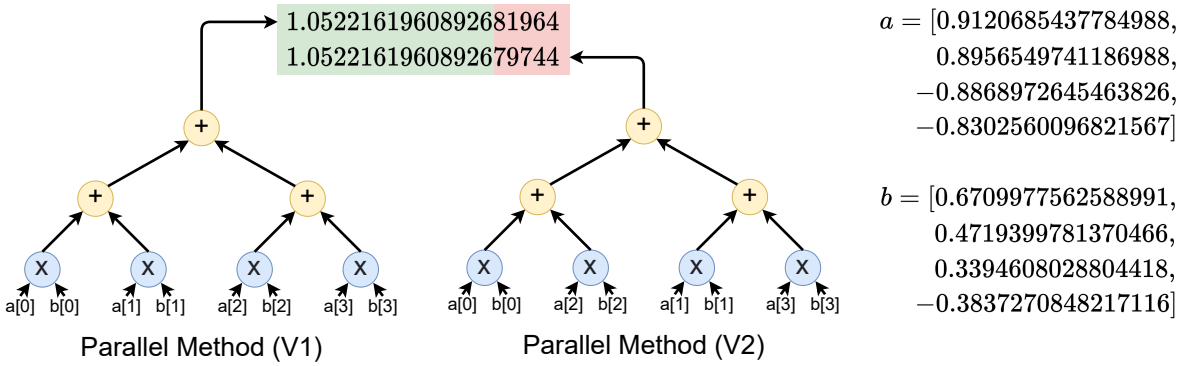


Figure 4: Example of different computation results for the dot-product comparing different computation orders for the parallel method (Inspired by Figure 5 in [33]).

Environment Throughout this thesis we use the DL framework *PyTorch* [37], mainly because it seems to be predominant in research [15, 17].

We define having the *same environment* as having the same: version of PyTorch and any other Framework used; version of the Python interpreter; operating system; and CPU. If we use hardware acceleration, we also have to have the same additionally installed software, driver, and chip (for example, GPU). This is illustrated in Figure 5 which demonstrates parts of the environment and their dependencies.

The highest layer is the framework used, in this case, PyTorch (only representative for all used frameworks). PyTorch is available in multiple different versions that differ in implementation and the target platform. To reproduce inference and training, we need the same PyTorch version.

PyTorch implements its high-level functions in Python and lower level DL related operations in C++. Since C++ is a compiled language, the platform we install PyTorch on already determines the exact machine code. This contrasts with Python. Python is an interpreted language, and the availability and behavior of Python language features depend on the installed Python interpreter. It is possible that we can reach reproducibility using different versions of Python, but it is not guaranteed. Therefore, we have to keep track of the Python interpreter used.

How certain floating-point operations are realized not only depends on the implementation in Python or C++ but also on the operations the CPU offers. Suppose we want to accelerate our code using a specific type of hardware; we have to keep track of the libraries, drivers used and of the chip itself. For NVIDIA GPUs, for example, the implementation and availability of operators depends on the installed CUDA [6] and cuDNN [7] versions which depend on both the installed driver and the GPU itself.

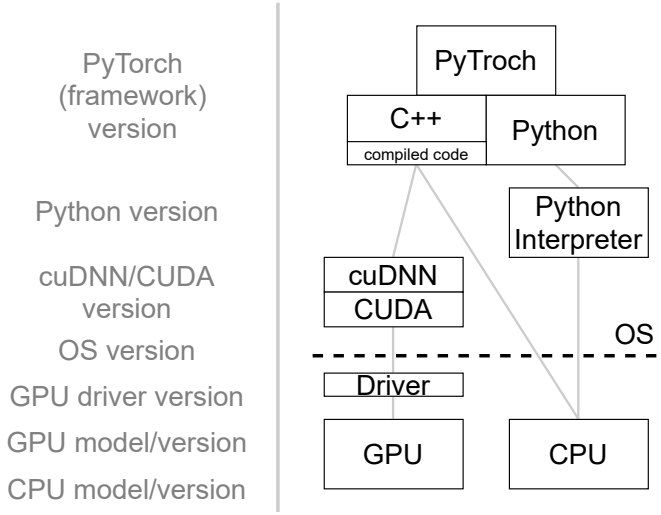


Figure 5: Experiment execution environment for processing on CPU and GPU.

Deterministic Operators By tracking the environment, we can execute our code in the same or a reproduced environment; but to reproduce inference and training, the operators must also be deterministic. By deterministic operators, we mean, for example, that floating-point calculations are always executed in the same order since otherwise, results might differ.

For PyTorch, there is a specific section about reproducibility in the official documentation [38]. The main method presented (*torch.set_deterministic()* [4]) is under active development, and its behavior might change in the future; for now, it guarantees that only deterministic operations are used and if there is no such implementation an error is thrown [5]. The documentation also states that a flag activating the benchmarking of cuDNN operations should be disabled and that it might be necessary to set the flag *CUBLAS_WORKSPACE_CONFIG* to a certain value.

According to the documentation, the transition from non-deterministic to deterministic results in a decrease in performance. In Section 8.3.4 we will analyze this for a set of models.

[4] In newer versions of PyTorch it was renamed to *torch.use_deterministic_algorithms()*

[5] There is, for example, no deterministic implementation for the operation *adaptive_avg_pool2d_backward_cuda* used in the Alexnet [25] (`https://github.com/pytorch/pytorch/issues/50198`, accessed 2021-03-05)

Reproducible Inference and Training Applying all the measures above – using identical code, parameters, and data; setting the seeds for all PRNGs; and making floating-point operations reproducible by always executing them in the same environment and only using their deterministic implementation – we can reproduce the inference and training of models on a given CPU and GPU.

To date, we have only tested the execution on a single GPU and not on multiple ones. Nevertheless, at least theoretically, it should be possible to reproduce inference and training in a multi GPU setting [41].

3.4.4. Probing Tool

To check the reproducibility of PyTorch models we developed a probing tool [6] . It executes the model twice using the same data and compares layer-wise the input and output tensors for the forward and backward pass. These intermediate results can also be saved and loaded so that we can be compare them across different machines.

As an example, we probed the PyTorch implementation of GoogLeNet [50]. In Figure 6(a) we can see that the dropout layer prevents reproducibility by generating different outputs in the forward pass. This, in turn, causes non-reproducible outputs for the rest of the forward and the entire backward pass.

Dropout layers are known sources of intentionally introduced randomness. This explains why, in this case, we can make the execution reproducible by setting the seeds as described in Section 3.4.2. Having done this, the probing tool no longer finds any differences and produces the output shown in Figure 6(b).

Other summary is: diff

l_name	fwd_idx	inp	out	grad_inp	grad_out
Conv2d	186	same	same	diff	diff
BNorm2d	187	same	same	diff	diff
BcConv2d	188	same	same	diff	diff
MaxPool2d	189	same	same	diff	diff
Conv2d	190	same	same	diff	diff
BNorm2d	191	same	same	diff	diff
BcConv2d	192	same	same	diff	diff
Inception	193	same	same	diff	diff
AAvgP2d	194	same	same	diff	diff
Dropout	195	same	diff	diff	diff
Linear	196	diff	diff	diff	diff

(a)

Other summary is: same

l_name	fwd_idx	inp	out	grad_inp	grad_out
Conv2d	186	same	same	same	same
BNorm2d	187	same	same	same	same
BcConv2d	188	same	same	same	same
MaxPool2d	189	same	same	same	same
Conv2d	190	same	same	same	same
BNorm2d	191	same	same	same	same
BcConv2d	192	same	same	same	same
Inception	193	same	same	same	same
AAvgP2d	194	same	same	same	same
Dropout	195	same	same	same	same
Linear	196	same	same	same	same

(b)

Figure 6: Probing a non-reproducible (a) and a reproducible (b) execution of the GoogLeNet (names are shortened and only the last eleven layers are displayed).

[6] https://github.com/slin96/mmlib/blob/master/examples/probe_example.py

4. MMlib outline

The overall research question for this thesis is if and by how much 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 will develop a baseline approach $\mathcal{B}$ in Section 5 and focus on two main ideas to improve it: (1) save only the model data that has changed compared to its base model. (2) Do not save the model, but only the model's provenance data.

Following idea (1), we develop the parameter update approach $\mathcal{U}_P$ in Section 6 and an improved version of it ($\mathcal{U}'_P$) in Section 6.7. For idea (2), we develop the provenance approach $\mathcal{M}_{Prov}$ presented in Section 7.

In this section, we introduce *MMlib*, a python library bundling all approaches we develop throughout this thesis (Section 4.1). Subsequently, we define the use cases that we cover with *MMlib* in Section 4.2 and set our assumptions and focus in Section 4.3.

4.1. MMlib Python Library

We develop all the approaches as part of a Python library that we call *MMlib* [7]. Next to providing the overall framework for our approaches, it aims to be minimalistic, intuitive to use, and generic enough to be extendable to implement future approaches beyond this thesis's scope.

As a DL framework, we choose the framework PyTorch [37]. The reasons are: (1) Looking at conference papers, PyTorch seems to be, for now, the predominant DL framework in research [15, 17]. (2) Our experience with PyTorch would reduce the development overhead. (3) In PyTorch, model parameters are easily accessible, simplifying the development of our approaches.

To make all approaches in *MMlib* interchangeable, they implement the interface *ModelSaveService* shown in Figure 7 (for a more extensive version see Figure 50 in the appendix). It defines two methods: *save_model* and *recover_model*. The method *save_model* saves a model by taking all information that we can not automatically extract (for example, the model itself) and returns a model identifier. The method *recover_model* takes a model identifier, recovers the model, and returns an object containing the recovered model and optionally metadata.

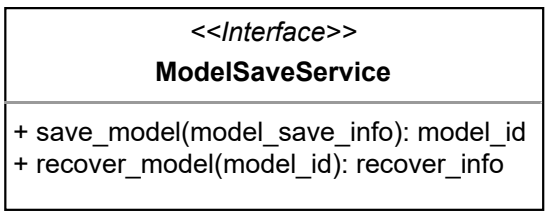


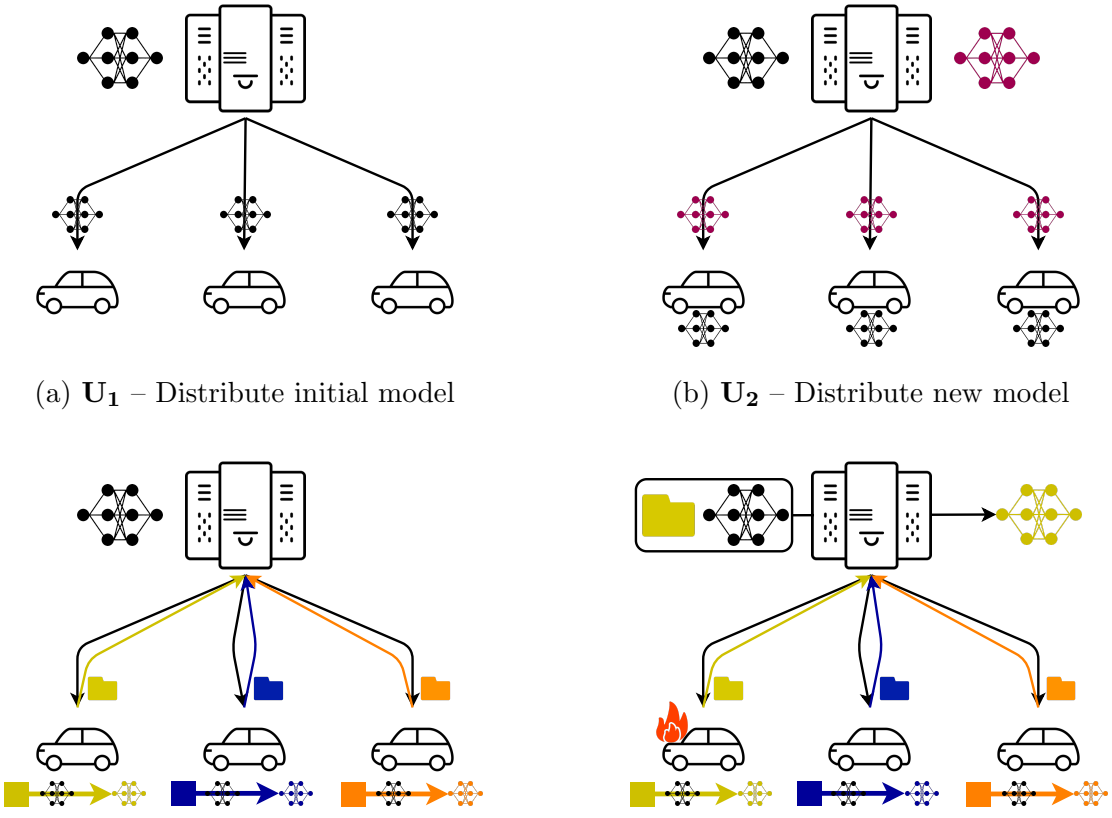
Figure 7: Interface *ModelSaveService* defining *save_model* and *recover_model*.

[7] https://github.com/slin96/mmlib

4.2. Use Cases

We develop *MMlib* to use it in a setting with a central *server* and multiple distributed devices that we refer to as *distributed nodes* or, if it is apparent in the context, just *nodes*. We initially develop models on the server and later distribute them to all nodes. Over time we need to adjust the models, which happens in two ways. (1) We update or retrain the models on the server and deploy this update to the nodes. (2) We update the model on the node using locally collected data. Regardless of whether we trained the models on the server or the node, the server should know of every model that exists and should also be able to recover it if necessary.

Having this setting in mind, we derive four use cases that *MMlib* should cover. We define them in the following, show them in Figure 8, and reference them throughout this thesis. [8]



(a) $\mathbf{U_1}$ – Distribute initial model

(b) $\mathbf{U_2}$ – Distribute new model

(c) $\mathbf{U_3}$ – Inform about new model

(d) $\mathbf{U_4}$ – Recover model

Figure 8: Use cases

[8] The fact that Figure 8 shows cars as nodes is just for illustrations reasons and does not imply that we focus on the domain of cars throughout this thesis.

U_1 – Distribute initial model (Figure 8(a)): We initially train a model on a central server using an extensive dataset. As soon as we have finished the training, we distribute the model to all nodes.

U_2 – Distribute derived model (Figure 8(b)): After we have initially deployed the model, we further improve it on the server. Possible directions are that we: enrich the training dataset, improve the training procedure, or slightly change what the model is used for. As soon as we have derived a new model, we send a model update to the nodes.

U_3 – Inform about locally derived model (Figure 8(c)): Alternatively, to adjust the server's model, we can also change the model on a single node. The aim is to improve the model, for example, for individual users' behavior or a changing environment; we most likely train the model on a small dataset that was locally collected. Before using the new model locally, we inform the central server about it by transferring whatever information is needed to recover the model at any point in time.

U_4 – Recover model (Figure 8(d)): If we want to recover a model, we have to do this from the already saved data. Reasons for which we might want to recover a model, could, for example, include: something went wrong on a node, and we want to investigate what model was used, or we want to use a specific model as a starting point for further development.

4.3. Assumptions and Focus

The presented use cases, as well as, the defined *ModelSaveService* are generic. To narrow down the scope of our thesis and the approaches we develop as part of *MMlib*, we set our assumptions and focus.

Use Case Frequency We expect to see that every node updates its model regularly while major model updates happen only occasionally. Moreover, we expect to see that recovering a model is rare compared to the other scenarios. These assumptions directly map to the expected frequency of the use cases. We expect to see U_3 frequently, U_1 and U_2 from time to time, and U_4 rarely.

Throughout this thesis, we will focus on the use cases U_3 and U_2. U_3 is the most frequent one and thus most relevant; U_2 is not as relevant but very similar to U_3. In addition, these use cases have the most potential for improvement. For U_2 and U_3, we can make use of the fact that the models are related to each other. For U_1, we always have to save the entire model, and for U_4, we have no other choice than to recover the model from the given data.

Saved Data In this thesis, we do not discuss what subset of all models we should save; we save all created and used models. Moreover, we set our focus on saving only the actual models. If we save any other data (for example, the dataset for the model provenance), we do this naively and declare more advanced approaches as future work.

Advanced Optimizers As mentioned in Section 3.1, simple optimizers base their decision on updating the model parameters only on the computed gradients. Advanced optimizers additionally take information about past parameter updates into account, which massively increases the size of their internal state and the storage consumed to save it.

If in a particular scenario, the use of such advanced optimizers is essential to achieve good training results, and we want to save models in a way so that they can be continued to be trained on any other machine, we would have to save the used optimizer and its state together with every model. This would most likely result in a storage consumption increased by a constant factor for every model.

Although we provide the functionality to save an optimizer regardless of its complexity, throughout this thesis and the evaluation, we focus, for simplicity, only on optimizers that do not save information about past parameter updates.

Model Relations For U_2 and especially U_3, it is most likely that the models are *versions* or *fine-tuned versions* of each other. Since U_2 and U_3 are the use cases we focus on, we optimize our approaches for model versions and fine-tuned model versions and discuss other model relations only when feasible.

Recoverability We save all models in a way that we can recover them without loss of precision. By this, we mean that a model we save and a model we recover are equal in the sense of Definition 3: they have the same architecture, the same parameters, and thus produce the same output when given the same input data.

5. Baseline Approach

In this section we develop a baseline approach ($\mathcal{B}$) that implements the *ModelSaveService* interface defined above. We see $\mathcal{B}$ as a starting point for more advanced approaches and as a reference point to measure and evaluate performance improvements. We start by presenting the overall idea in Section 5.1, move on to describe how to generate and extract the data to save a model in Section 5.2, and discuss how to guarantee reproducibility in Section 5.3. In Section 5.4, we present how we structure the data, in Section 5.5 how to persist it, and in Section 5.6 how to recover a model from it. Finally, we present our expectations for the approach's performance in Section 5.7.

5.1. Outline

As discussed earlier, any approach has to implement the primary operations to save a model and recover a model. The only information we want to save next to the model itself is a reference to the base model, if there is any.

With the baseline approach ($\mathcal{B}$), we aim to develop an approach that covers all requirements in a naive way. In detail, this means we want a minimal complexity for extracting, representing, saving, and recovering all information needed.

Applying this to saving a model means we do not include any context knowledge or optimization in the saving process. This explicitly implies that we do not consider that a derived model and its base model share information. Applying a minimal approach to saving metadata means we only save essential metadata like an identifier, a reference to the base model, and some checksums.

Without giving any implementation details, a potential baseline approach could save models as presented in Figure 9. To save a model, $\mathcal{B}$ extracts model metadata like the base model and checksums and serializes the corresponding (PyTorch model) object. Afterward, $\mathcal{B}$ generates an identifier and saves it together with the extracted metadata and a reference to the serialized model in a text-based file. To recover a model, $\mathcal{B}$ identifies the model metadata by its identifier, reads it, deserializes the model object, and finally performs checks to ensure it recovered the model correctly.

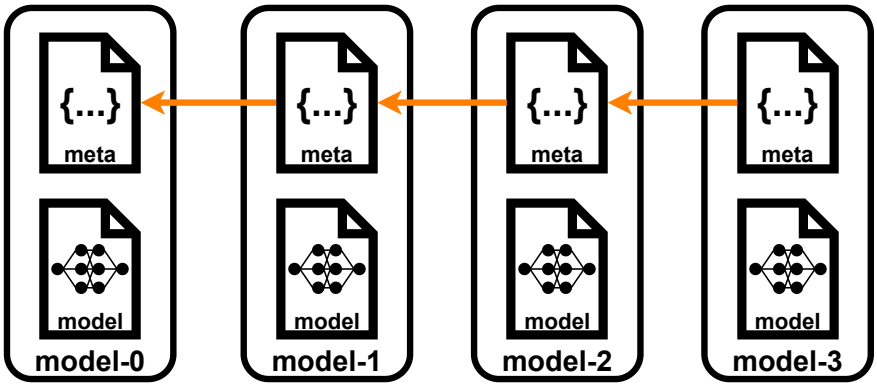


Figure 9: Overall idea for a baseline approach ($\mathcal{B}$).

5.2. Extract and Generate

In the previous section, we identified two groups of data $\mathcal{B}$ extracts and saves: the metadata and the serialized model object.

Collecting the metadata is almost effortless; the reference to the base model is given, and the identifier auto-generated. The only data we have to extract are checksums for the model parameters; therefore, we implemented a method to hash PyTorch tensor objects, which is the data structure PyTorch uses to represent model parameters.

The second step is to serialize the model given to us as a Python object. For PyTorch, we can serialize and deserialize a model using the operations *torch.save()* and *torch.load()* [40]; both operations use Python's pickle module [34] to (de)serialize the model object in a zipfile-based file format.

Although this intuitive approach works, it comes with a problem: when serializing an object, the pickle module does not include the object code; it saves a reference to the code file. Consequently, when deserializing the model, the code file must be present in the same directory path as when we serialized the model.

This might be acceptable for local development and check-pointing. But, especially when saving and recovering models across different machines, we cannot expect to have the same directory structure everywhere and reconstructing it every time adds overhead. This is why the recommended way to save a PyTorch model is to save the model's code (defining the model architecture) and the model's parameters separately [39].

To save the model's code, we save the file holding it. To save the parameters, we save the model's *state_dict*. The *state_dict* is a dictionary object every PyTorch model holds that maps each layer to its parameters. We can access the *state_dict* using *model.state_dict()* and save it using *torch.save()*.

This leads to the slightly adjusted version of $\mathcal{B}$, shown in Figure 10. Compared to our initial idea in Figure 9, we represent a model by splitting it into the code file (*model.py*) representing the model's architecture and a serialized version of the *state_dict* (*params.pt*).

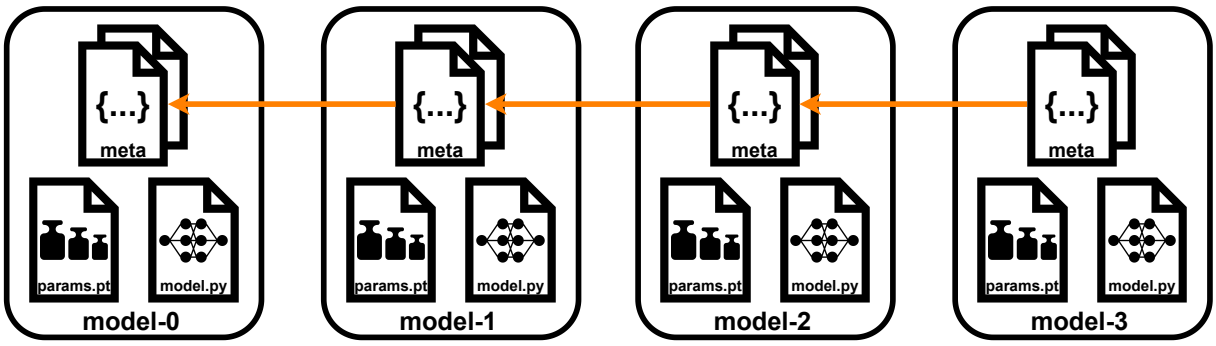


Figure 10: Updated idea for a baseline approach ($\mathcal{B}$).

5.3. Reproducibility

Model Architecture As discussed in Section 4.3, one of our requirements is reproducibility: we want to save and recover models without loss of precision.

To save a model, we could have chosen a more abstract or a more human-readable representation for the model architecture than saving the source code. One way could, for example, be to save the layer names and dimensions.

Most high-level or more abstract approaches would work to document the model's architecture roughly and give an intuitive indication as to how to reconstruct the model, but they do not work to save a model in a reproducible way. Moreover, such an approach would massively blow up our data schema and comes with a high probability of forgetting to represent details. The fact that there might be updates of the implementation, which would make an update of a data schema necessary, additionally increases the complexity.

This is why we decided to save the model's code which represents a precise definition of the architecture.

Environment Even if the model's code gives us a precise definition of the model architecture, in different versions of PyTorch the implementation of single layers might vary and lead to different behavior.

Thus, the model's architecture is only fully specified if we also save the PyTorch version that is dependent on the environment. We discussed what information about the environment is critical in Section 3.4. To extract it, we use Python's platform module [36] and PyTorch's *torch.utils.collect_env.get_env_info()* function [9].

Layer Hash as Checksum For every model, we optionally save a hash value for each layer's parameters. When recovering a model, we can compare the hashes of the revered layers with the saved ones to verify that there was no loss of precision.

While the hash information is, for $\mathcal{B}$, only an extra safety net, it is helpful to decrease the run time to save a model in later approaches (see Section 6.7).

5.4. Structure the Data

Before we present how $\mathcal{B}$ structures and saves the data to represent a model in a recoverable form, we sum up what data we need to save.

As metadata, we have to save a model identifier and a reference to the base model if existing. Optionally, we save the hash values for every model layer. To save the model architecture, we save its code, its class name (if the code defines multiple classes), and the environment (most importantly, the PyTorch version). To represent the parameters, we save the model's *state_dict* as a file.

To be open for future extensions, we do not save all the data in one object or file but split it into three entities (see Figure 11): a generic *ModelInfo*, the information to recover the model (*RecoverInfo-FullModel*), and the associated *Environment*.

[9] `https://github.com/pytorch/pytorch/blob/master/torch/utils/collect_env.py`, accessed 2021-05-11

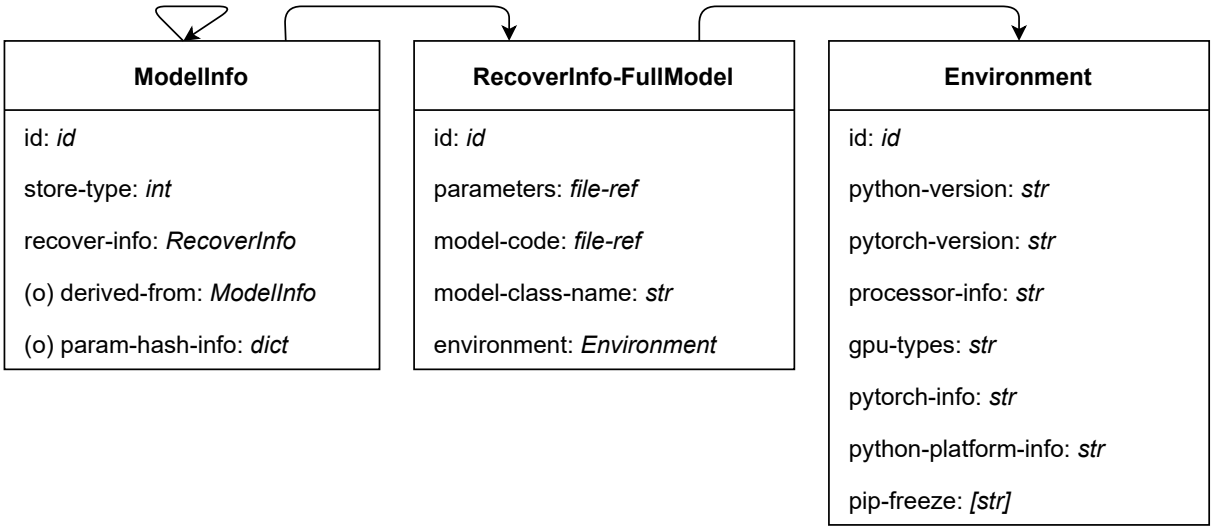


Figure 11: The storage schema for the baseline approach ($\mathcal{B}$).

The *ModelInfo* is the abstract representation of a model; its *id* is also the model identifier. It holds a reference to the *RecoverInfo* and a field called *store-type* to specify in what way we saved the model (becomes relevant for future approaches). Optionally, the *ModelInfo* refers the parameter hashes for every layer and to another *ModelInfo* object to define the base model.

The *RecoverInfo-FullModel* holds all information that is necessary to recover a model saved by $\mathcal{B}$. This includes the model *parameters* and *code* saved as files, the model *class name*, and a reference to the *Environment*.

The *Environment* holds the Python version, PyTorch version, information about the processor, information about GPUs (if available), and a list of all Python libraries and their version numbers. *Pytorch-info* and *python-platform-info* are both long strings holding all information we can access about Python respectively PyTorch. Not all of it necessary, but it helps in case we have to debug.

5.5. Persist the Data

Going back to the schema in Figure 11, we have to save two types of data: metadata and files. We decided not to define a fixed relational database schema to save the metadata because we developed a prototype that changes frequently. Instead, we represent every entity in our schema as a JSON [23] document that is persisted in a MongoDB [30].

We do not save files in the MongoDB. By chunking them, it would be possible, but MongoDB is not designed to save large files, unlike most other databases. Instead, we save the files to a local or a distributed file system and reference them in the entities' JSON documents.

To be open to other or more advanced ways to save documents and files, $\mathcal{B}$ does not access a MongoDB or implementation to save files directly. Instead, it accesses implementations of generic interfaces (for a class diagram see Figure 51 in the appendix).

5.6. Recover a Model

Holding a model's identifier, we can recover every model saved using $\mathcal{B}$ independently from all other models: we load the *ModelInfo* and recursively the *RecoverInfo* and the *Environment*.

We check if the present PyTorch and Python versions are the same using the environment information. If this is the case, we recover the model architecture by creating a model instance using the class name and the code file. Finally, we load the parameters into the model and optionally check if they match the hashes in the *param-hash-info*.

5.7. Expectations

Having discussed $\mathcal{B}$, we expect to see that the **storage consumption**, the **time-to-save (TTS)** a model, and the **time-to-recover (TTR)** a model are primarily dependent on the **model architecture and the number of parameters**.

In Section 8 we will see that $\mathcal{B}$ meets these expectations.

6. Parameter Update Approach

In this section we present the parameter update approach $\mathcal{U}_P$. It improves the baseline approach $\mathcal{B}$ by saving only the updated information of a derived model instead of a complete snapshot. We start with presenting the overall idea in Section 6.1 and describe how we generate and extract the data $\mathcal{U}_P$ saves in Section 6.2. After we briefly discuss how we achieve reproducibility in Section 6.3, we present how to structure and persist the extracted data in Section 6.4 and how to recover a given model from this data in Section 6.5. We finalize this section by giving our expectations for $\mathcal{U}_P$'s performance in Section 6.6.

6.1. Outline

We focus on model versions and fine-tuned versions as part of U_2 and U_3. If we save a model M, it was most likely derived from a base model B and it is probable that M and B share a significant amount of information. If a model M is a version of B, they share at least the model architecture ($B \xrightarrow{B_a} M$). If M is a fine-tuned version, they additionally share a subset of the parameters ($B \xrightarrow{B_a, \subset B_p} M$).

The baseline approach $\mathcal{B}$ (Section 5) does not consider that some models share information and, therefore, permanently saves a complete snapshot of every model. This changes with $\mathcal{U}_P$. The overall idea is to identify redundant information and save it only once. Especially for fine-tuned model versions, we expect to see an improvement because it is the parameters that have the most significant impact on a model's storage footprint.

Figure 12 shows an example: We use $\mathcal{B}$ to save a initial complete snapshot of *model-0*. Afterward we use $\mathcal{U}_P$ to save only the updated metadata and parameters for every derived model version or fine-tuned model M (*model-1*, *model-2*, or *model-3*). For all information that has not changed like the model architecture and a subset of the parameters, we refer to the base model.

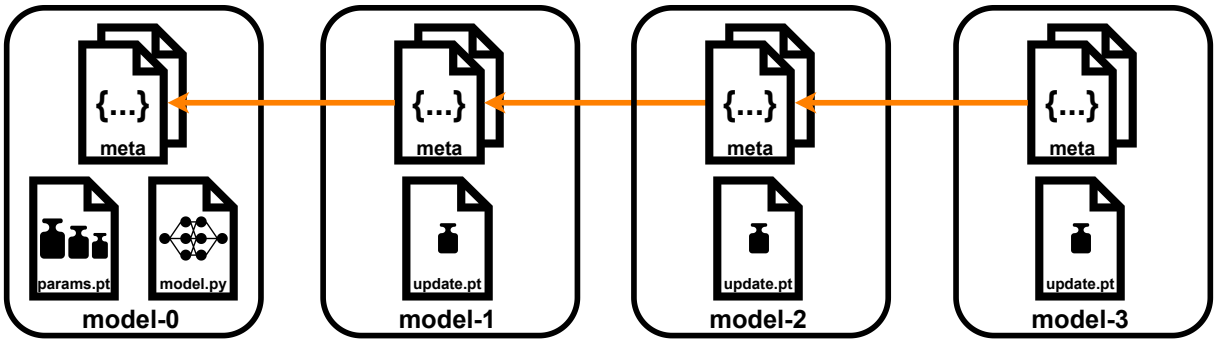


Figure 12: We save the initial model as a complete snapshot using $\mathcal{B}$, for all derived models we use $\mathcal{U}_P$ to only save the updated information.

6.2. Extract and Generate

For every initial model, we use the baseline approach $\mathcal{B}$; we already discussed how to generate and extract the necessary data in this case in Section 5.2. For derived models, the main task is to check which of the parameters have changed – we know that the architecture did not change because we focus on model versions and fine-tuned versions; the metadata changes for every model.

Let us assume we derived a model M from a base model B ($B \rightarrow M$). To extract the updated parameters, we take M's *state_dict*, compare it to B's *state_dict*, and delete all layers' parameters that have not changed. We call this pruned version of M's *state_dict* the *parameter update* and serialize it to a file with the same method we use for a normal *state_dict – torch.save()*.

Every time we retrain only a part of a base model B to derive a new model M, we freeze B's parameters layer-wise. For the non-frozen layers, it is unlikely that only a subset of the parameters changes. This is why we calculate the *parameter update* layer-wise and not on a single parameter granularity.

6.3. Reproducibility

In Section 8 we will see that $\mathcal{B}$ can save and recover models without loss of precision. The critical part is to save and extract a representation of the model architecture (Section 5.3).

Since $\mathcal{U}_P$ utilizes $\mathcal{B}$'s logic to save an initial model and does not save the model architecture, we can assume that $\mathcal{U}_P$ can also save and recover models without loss of precision.

6.4. Structure the Data

As it was for $\mathcal{B}$, we save initial models (for example, model-0 in Figure 12) by using the entities: *ModelInfo*, *RecoverInfo-FullModel*, and *Environment*.

To save a model using $\mathcal{U}_P$, we have to save: metadata including a reference to the base model and a pruned *state_dict*, the *parameter update*. We extend the schema presented in Section 5.4 by adding the new entity *RecoverInfo-ParameterUpdate*, as shown in Figure 13. It holds two fields: the *parameter update* referenced as a file and the *update-type* indicating how we computed the parameter update.

The *store-type* in the *ModelInfo* indicates which of the two recover info entities it references and thus if we can recover a model independently of its base model.

To persist the data, we use the same approach for dictionaries and files as presented in Section 5.5.

6.5. Recover a Model

Using $\mathcal{U}_P$, we save every model version ($B \xrightarrow{B_a} M$) or fine-tuned version ($B \xrightarrow{B_a, \subset B_p} M$) by saving a parameter update. To recover a model M, we load its base model B and apply the parameter to it.

Saving multiple models using $\mathcal{U}_P$ leads to a chain of models referring to each other. To recover the last model in the chain, we have to recover all its base models. Taking

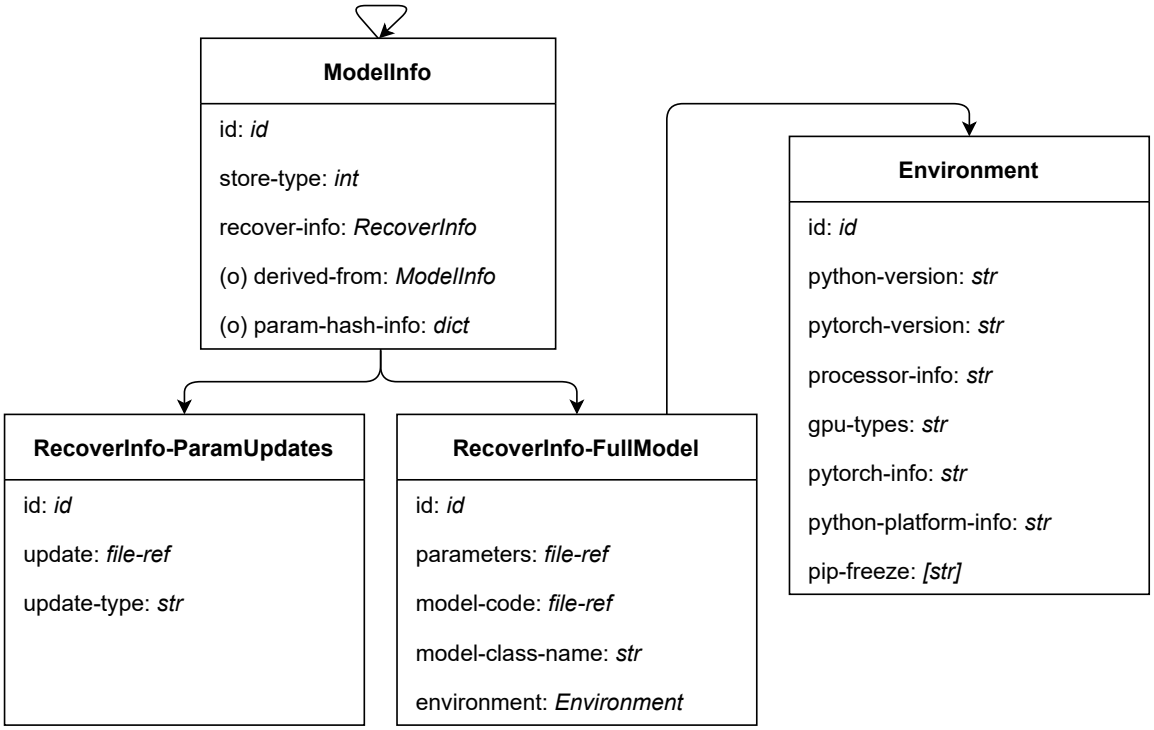


Figure 13: Extended storage schema used by $\mathcal{U}_P$.

the example from Figure 12: To recover model-3, we have to recover model-2; to recover model-2, we have to recover model-1; and to recover model-1, we have to recover model-0. This makes recovering a model saved using $\mathcal{U}_P$ a recursive process. The pseudo-code in Algorithm 1 describes the abstract logic for it.

The function *is_full_model* checks if we saved a model as a complete snapshot using $\mathcal{B}$. If this is the case, we can recover it independently of all other models. If not, we first have to recover the base model to apply the parameter update on it.

These chains of recovering models can be infinitely long in practice, resulting in a very computationally intense recovery process. However, for the scope of this thesis we assume that recovering a model happens rarely and focus on reducing the storage footprint.

Algorithm 1 Recursively recover a model saved using $\mathcal{U}_P$

```
if is_full_model(model_id) then
    model = get_full_model(model_id)
    return model
else
    base_model_id = get_base_model_id(model_id)
    base_model = recover_model(base_model_id)
    parameter_update = get_parameter_update(model_id)
    model = base_model.apply(parameter_update)
    return model
end if
```

6.6. Expectations

Having discussed $\mathcal{U}_P$, we expect it to behave as follows:

- **Storage consumption**
  - For model versions the storage consumption for the same setting decreases only marginally compared to $\mathcal{B}$ because all parameters change.
  - For fine-tuned model versions the storage consumption for the same setting decreases compared to $\mathcal{B}$. By how much it decreases depends on how many parameters the models share.
  - Although we ideally save only a subset of the parameters, they are still the biggest factor for the overall storage consumption.
- The **time-to-recover (TTR)** a model increases because of the recursive recovery process compared to $\mathcal{B}$ with every additional model we reference as a base model.
- Saving a model requires recovering its base model to generate the parameter update. Therefore, the **time-to-save (TTS)** a model increases compared to $\mathcal{B}$ according to the increasing TTR.

In Section 8 we will see that $\mathcal{U}_P$ meets all these expectations.

6.7. Improved Parameter Update Approach

In this section we will present an improved version of $\mathcal{U}_P$ we name $\mathcal{U}'_P$. As the naming suggests the over all idea is the same for both approaches. The main difference is $\mathcal{U}'_P$'s sophisticated way of computing the parameter update. It works without recovering the base model and by this noticeably reduces the time it takes to save a model.

Slow Saves To save a model M ($B \rightarrow M$) we have to access M's and B's *state_dict* to compute the parameter update. If B is not in memory, we have to recover B which is a recursive process whose time requirement increases with the number of base models. This noticeably slows down the process of saving models and is critical because it is an essential part of the most frequent use case U_3.

Improved Parameter Update Approach We develop $\mathcal{U}'_P$ to address this problem. It differs from $\mathcal{U}_P$ in only one aspect: it does not compute the parameter update based on the base model's *state_dict* but based on the *param-hash-info* saved in the base model's *ModelInfo*.

This massively reduces the overhead of generating a parameter update. The *param-hash-info* is small, and to access it, we only need to recover the direct base model's *ModelInfo*. We do not have to recursively recover every base model until we eventually reach a complete model saved using $\mathcal{B}$.

Parameter Hash Info We structure the *param-hash-info* as a Merkle tree [27]. It enables us to determine if two models are the same in constant time and identify the layers that differ, in most cases, in logarithmic time.

Figure 14 shows an example of a Merkle tree representing the *param-hash-info* for a model with four layers. We walk through the example to explain how we construct it.

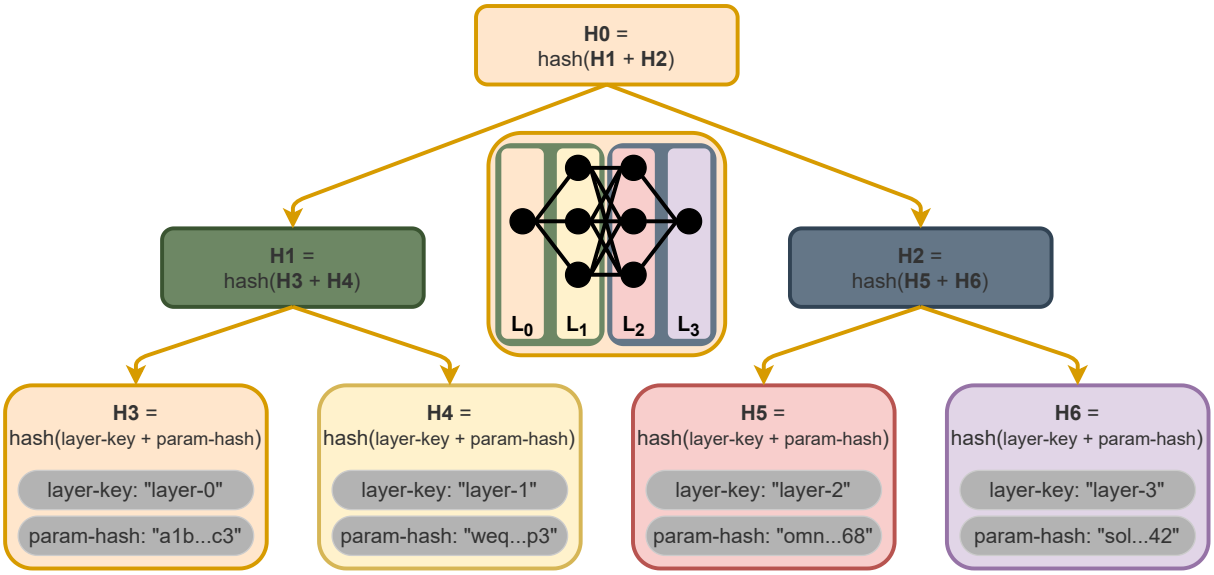


Figure 14: An example for a Merkle tree created from a four layered model.

Let us assume we have a model $M = (M_a, M_p)$, with a four layer architecture $M_a = (M_a^0, \ldots, M_a^3)$ and corresponding parameters $M_p = (M_p^0, \ldots, M_p^3)$. The Merkle tree consists of two types of nodes: *leaf nodes* and *non-leaf nodes*.

Every *leaf node* represents one layer $L_i = (M_a^i, M_p^i)$ of the model and holds the following information: (1) the *layer-key* identifying the layer in the model's *state_dict*, (2) the hash of the layer parameters ($param\text{-}hash = hash(M_p^i)$), and (3) a hash combining the layer key and the parameter hash: $hash(layer\text{-}key + param\text{-}hash)$. Two *leaf nodes* have the same hash if and only if they represent equal layers. Thus, comparing two *leaf nodes'* hash values is sufficient to check if their represented layers are equal.

Every *non-leaf node* in the Merkle tree represents a subset of the model layers. It holds a reference to its left child, a reference to its right child, and a hash value generated by hashing the concatenation of the left and the right child's hash. Two *non-leaf nodes* hold the same hash if and only if they represent an equal subset of layers meaning that the layers in the subsets are pairwise equal to each other. Thus, comparing two *non-leaf nodes'* hash values is sufficient to check if the represented subsets of layers are equal.

Calculate Parameter Update We can calculate a parameter update by referring only to its base model's parameter Merkle tree.

Assume the following scenario: We have a base model B and a fine-tuned version M of B ($B \xrightarrow{B_a \subset B_p} M$). Both models have four layers: L_0, L_1, L_2, and L_3, and only the layers L_2 and L_3 differ between B and M – for example, because they have been retrained. At

the time we want to save M, we hold a reference to the model object representing M, but for B, we only hold the *model_id*.

The first step to save M is to recover B's *ModelInfo*. This does not include recovering any *RecoverInfo* and also no recursive recovery of other models that might have to be recovered to recover B.

Having recovered B's *ModelInfo*, we can access B's *param-hash-info* and compare it to M's *param-hash-info* that we can easily create. Both *param-hash-infos* will look similar to the Merkle tree in Figure 14. They hold the same values except for the leaf nodes: H5 and H6 and the non-leaf nodes H2 and H0.

Starting to compare both Merkle trees from the root nodes, we note that they differ and continue with comparing their root nodes' left (H1) and right (H2) children. We notice that: (1) the hashes of H1 are the same, so all child nodes of H1 have to be the same, and (2) the hashes of H2 differ.

We compare H5 and H6 in both trees and find that they both differ too. Since they are leaf nodes we are done. We know that the parameter update has to contain only *layer-2* and *layer-3*; all other layers are equal.

For this small example, the process does not seem much faster than comparing all layers separately. Nevertheless, using a Merkle tree pays off, especially when only the last layers of a model have changed.

To give a better intuition why, we show two examples of a Merkle tree that represents a 64-layered model in Appendix A. In Figure 48 only the last two layers differ, in Figure 49 the last two and four layers in the middle differ.

To find out whether two models are equal – we do this every time we recover a model – using an iterative approach, we have to, in the worst case, perform 64 checks (one for every layer) to be sure no layer has changed. Using the Merkle tree we only have to compare the root node.

To identify which layers have changed using an iterative approach we always have to perform 64 checks. Using a Merkle tree it depends on how many nodes have changed and how they are positioned. In Figure 48 we have to do 13 checks, and 29 in Figure 49.

Improved Performance With the updated approach $\mathcal{U}'_P$ we can compare with little effort what layers of a derived model M have changed compared to its base model B. We do not have to have B in memory, nor do we have to recover B's parameters or any of B's base models. Using $\mathcal{U}'_P$, we expect to massively speed up the time-to-save for derived model versions and fine-tuned versions.

7. Model Provenance Approach

In this section we present the model provenance approach $\mathcal{M}_{Prov}$ that saves models based on their provenance data. We start with presenting the overall idea in Section 7.1. In Section 7.2 we discuss how to track the model provenance data and how to structure them in Section 7.3. We explain how to recover a model from its provenance data in Section 7.4 and finally, give our expectations for $\mathcal{M}_{Prov}$'s performance in Section 7.5.

7.1. Outline

All approaches we presented so far ($\mathcal{B}$, $\mathcal{U}_P$, and $\mathcal{U}'_P$) are based on saving model parameters. Their performance is, therefore, highly dependent on the architecture of the model they save or recover. With $\mathcal{M}_{Prov}$, we develop an approach that does not save model parameters but how they were created by training; we save the model provenance, which is, according to Definition 6, *the history of the processes used to produce the model, together with the input/training data.*

When we recover a model, we have to return a model equal to the saved model, which explicitly includes the model parameters. Not saving the parameters but only the model provenance implies that we must reproduce the training process to recover the model parameters in the overall model recovery process. In Section 3.4, we showed that this is possible – even across different machines – if we track the following provenance information: the model architecture or a reference to the base model, detailed information about the training process, the dataset we used for training, and the environment we trained the model in. This brings us to our overall idea for $\mathcal{M}_{Prov}$ that we show in Figure 15. As for all previous approaches, we save the first model (model-0) using $\mathcal{B}$; we save the metadata, the model architecture, and the model parameters.

For every derived model M ($B \rightarrow M$), we save: a reference to the base model B, the dataset the model was trained on, detailed information about the training process, and the training environment, including information about available hardware and installed software [10].

[10] We also save the environment when using B, but mainly the PyTorch version is required and not necessarily the whole environment information. For $\mathcal{M}_{Prov}$, the environment information is essential to enable us to reproduce model training. This is why in Figure 15, we explicitly include the environment as part of the info that we have to version to save a model.

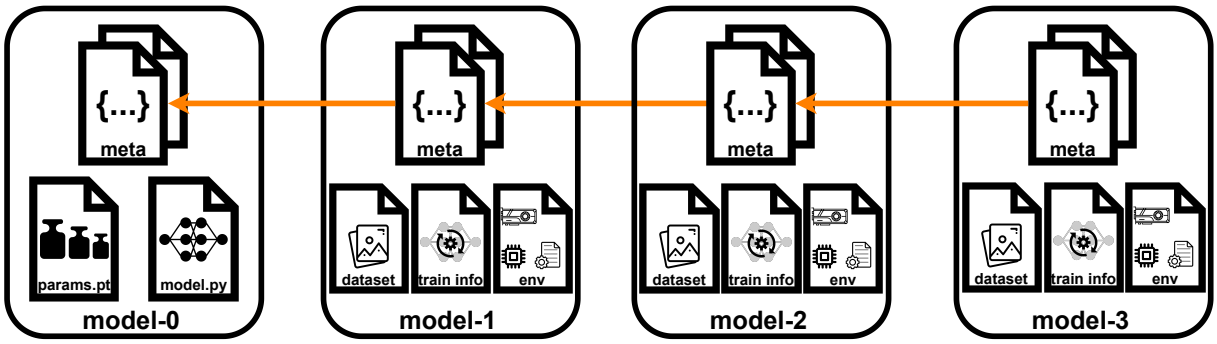


Figure 15: Saving models using $\mathcal{M}_{Prov}$. We save the initial model as a complete snapshot using $\mathcal{B}$, for all derived models, we use $\mathcal{M}_{Prov}$ to save the model provenance data comprising the training dataset and detailed information on the training process and environment.

7.2. Track Model Provenance

To save a model, we save its provenance; to save the model provenance, we have to generate and extract all provenance data. This includes the model architecture, the training environment, the training data, and the training process.

Throughout this thesis, we focus on model versions and fine-tuned versions. For these relations, all derived models have the same architecture, and only the parameters change. We always save the first model using $\mathcal{B}$ and thus, do not have to discuss extracting the model architecture for $\mathcal{M}_{Prov}$. For the environment, we already described how to extract and represent it in Section 5.2; this does not change for $\mathcal{M}_{Prov}$. To save the dataset in a compact form, we access the raw training data and compress it to a single file.

While we can almost effortlessly generate and extract the data for the model architecture, the environment, and the dataset, it is challenging to generate and extract the data to represent the training process. Revising the general training process described in Section 3.1 and how we would usually implement it in PyTorch, we see that we can represent the training process by three different groups of data: (1) parametrized objects without an internal state [11] 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.

In the following, we discuss how to save all data of a training process by walking through the example of tracking the training of a model on the ImageNet dataset.

[11] 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.

Parametrized Objects The overall idea to save and recover a parametrized object is to wrap it in a *wrapper object*. The wrapper object holds the reference to the wrapped object and is responsible for saving and recovering it. In our implementation, we call this simple version of wrapper RestorableObjectWrapper (ROW). We show an example of how to represent a data loader object using a ROW in Figure 16.

In code, such a wrapper 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 PyTorch); the initialization arguments; arguments that are read from a configuration file; and arguments that are references to other objects [12] (for example, in PyTorch a data loader needs a reference to a dataset object). The wrapper persists all its fields except the instance to save a representation of the wrapped object. The wrapper loads all its fields from the file system or a database and initializes an object with the given class name and parameters to recover the instance.

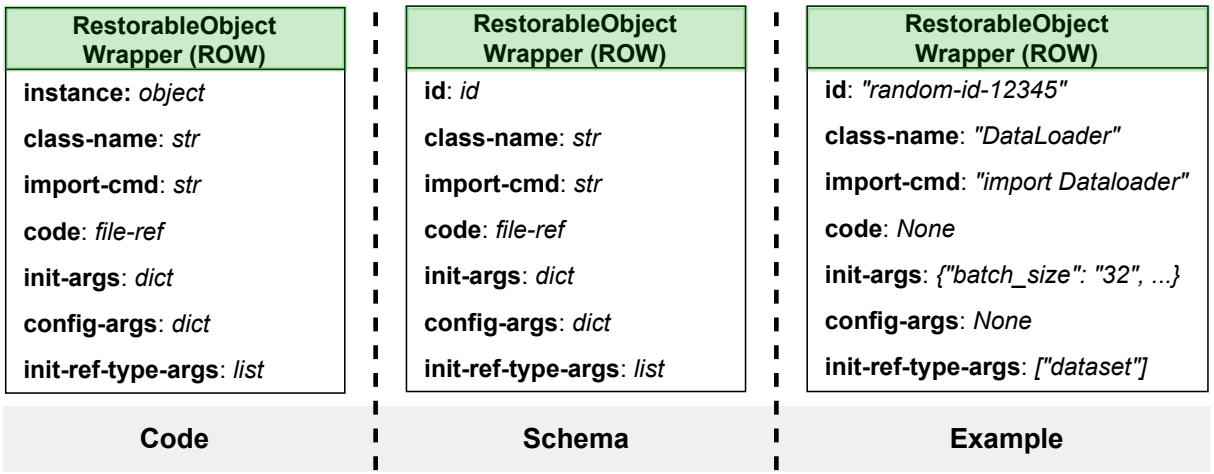


Figure 16: RestorableObjectWrapper implemented in code (left), represented in schema (middle), and example for *DataLoader* (right).

Objects with State Having discussed how to wrap parametrized objects without an internal state, we now move on to parametrized objects with an internal state. We call the wrapper for these types of objects StateFileRestorableObjectWrapper (SFROW). We show an example for wrapping an optimizer object in Figure 17.

A SFROW holds all the fields of a ROW and additionally a reference to a state file which can be any file that represents the state of the wrapped object. In PyTorch, for example, some classes implement the functionality to export a *state_dict*.

To save the wrapped object, the SFROW generates the object's state file and persists all fields except the instance. To recover the instance, it loads all fields from the file system or database, recovers the object, and recovers the objects' state from the state file.

[12] 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.

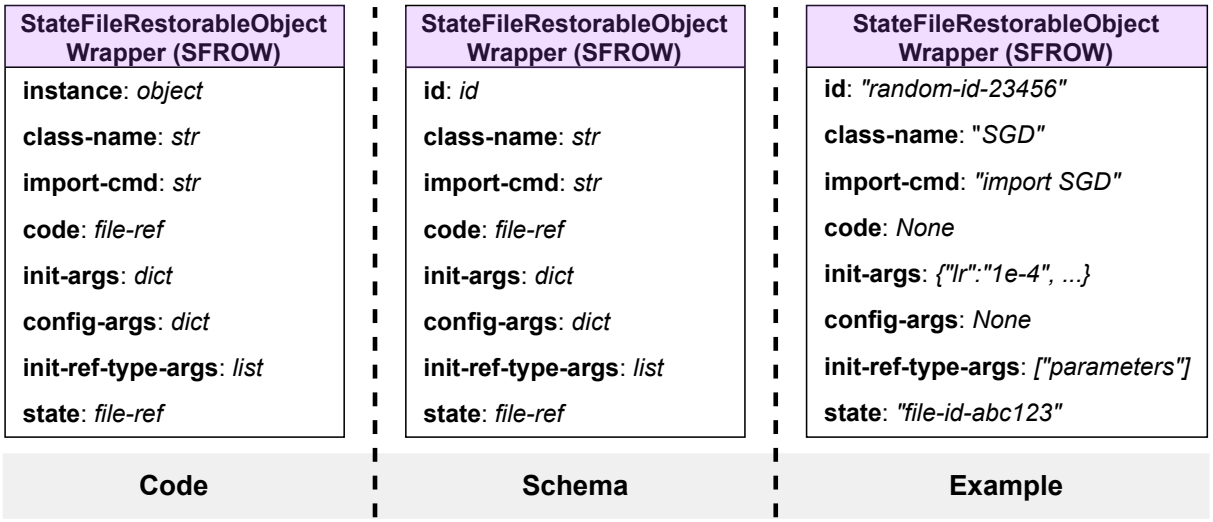


Figure 17: StateFileRestorableObjectWrapper implemented in code (left), represented in schema (middle), and example for *Optimizer* (right).

Train Logic To represent the training logic, we define a class that implements the interface *TrainService*. Every *TrainService* defines the logic to train a given model in its *train* method and references all objects that are relevant for it in a wrapped form (for example optimizer and data loader) in its *state dict*. All objects in the state dict are either ROWs or SFROWs and together define the state of the training process. We name the corresponding wrapper for *TrainServices* StateDictRestorableObjectWrapper (SDROW) and show an example *TrainService* in Figure 18.

To save an instance of a *TrainService*, the SDROW first persists all wrapped objects in the *TrainService*'s state dict and saves the references to the persisted wrappers together with the *TrainService*'s class name and code or import command.

To recover a *TrainService*, the SDROW first creates an instance of the *TrainService* and afterward recovers its state dict by recovering all wrapped objects.

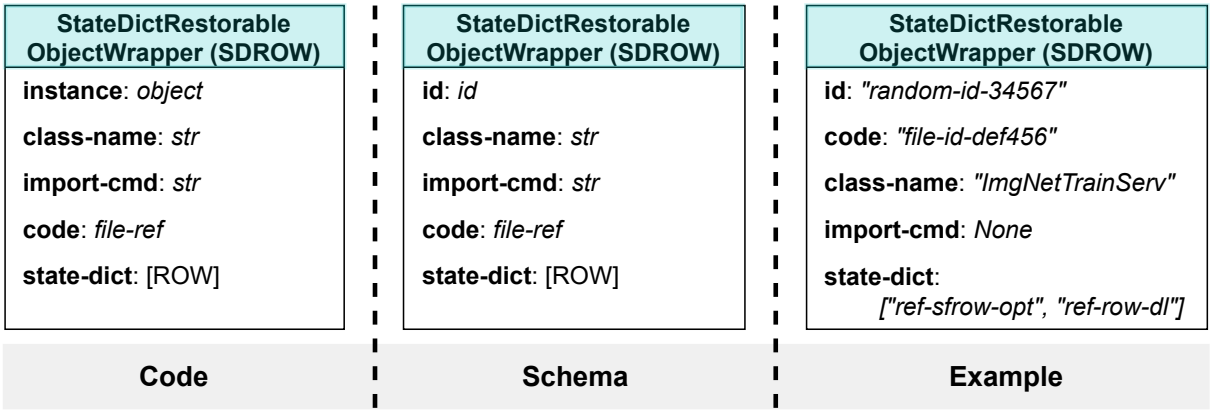


Figure 18: StateDictRestorableObjectWrapper implemented in code (left), represented in schema (middle), and example for *ImageNetTrainService* (right).

Train Info Knowing how we can save the state of the *TrainService*, we have one issue when recovering it: we can not recover the *state_dict* without knowing how the wrapped objects reference each other.

This is why for every *TrainService* we can not just use the standard implementation of the SDROW, we have to implement a custom subclass (for this example we name it *ImagenetTrainWrapper*). The custom subclass defines in what order and with what references we have to initialize the objects in the *TrainService*'s *state_dict* by implementing its method *restore_instance*.

The *ImagenetTrainWrapper* is also an object, and to save it, we wrap it. We do not define a special wrapper but represent the information by the high-level schema entity *TrainInfo* that also holds the key-value arguments used for training. We show an example in Figure 19.

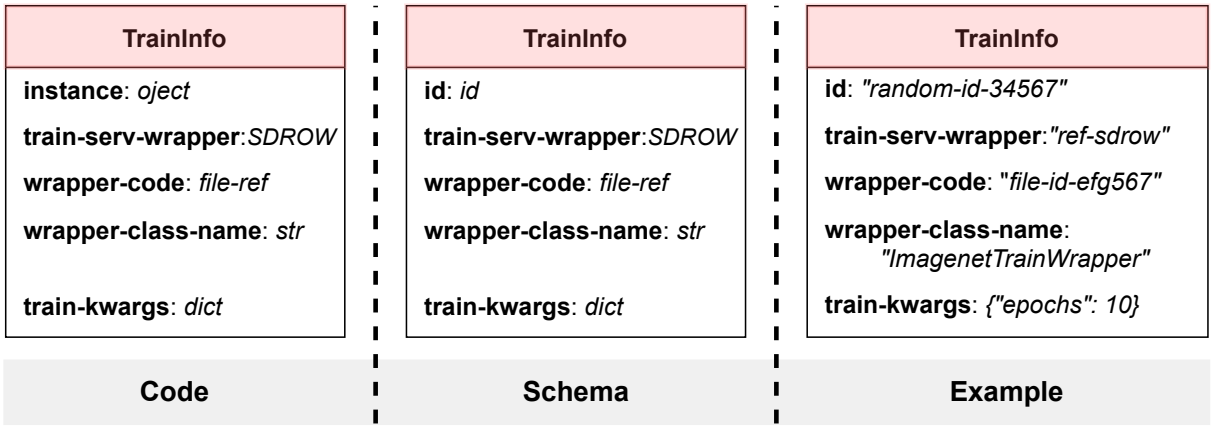


Figure 19: *TrainInfo* implemented in code (left), represented in schema (middle), and example for *ImageNetTrainService* (right).

Track Provenance Now, knowing how to represent all provenance data separately, we show in Figure 20 (for the extended version see Figure 52 in the appendix) how everything works together. The colors for the tracked objects are matching the colors of their corresponding wrappers from the previous figures. In the example, we assume that the training service only needs the data loader and optimizer.

The data loader is an object without an internal state and wrapped in ROW, the optimizer has an internal state and is wrapped in a SFROW. The *ImagenetTrainService* defines the training logic for training a model on the ImageNet data in its *train* method and holds a reference to the data loader and optimizer in its state dict. The *ImagenetTrainService* is wrapped in the custom *ImagenetTrainWrapper* which is a subclass of SFROW.

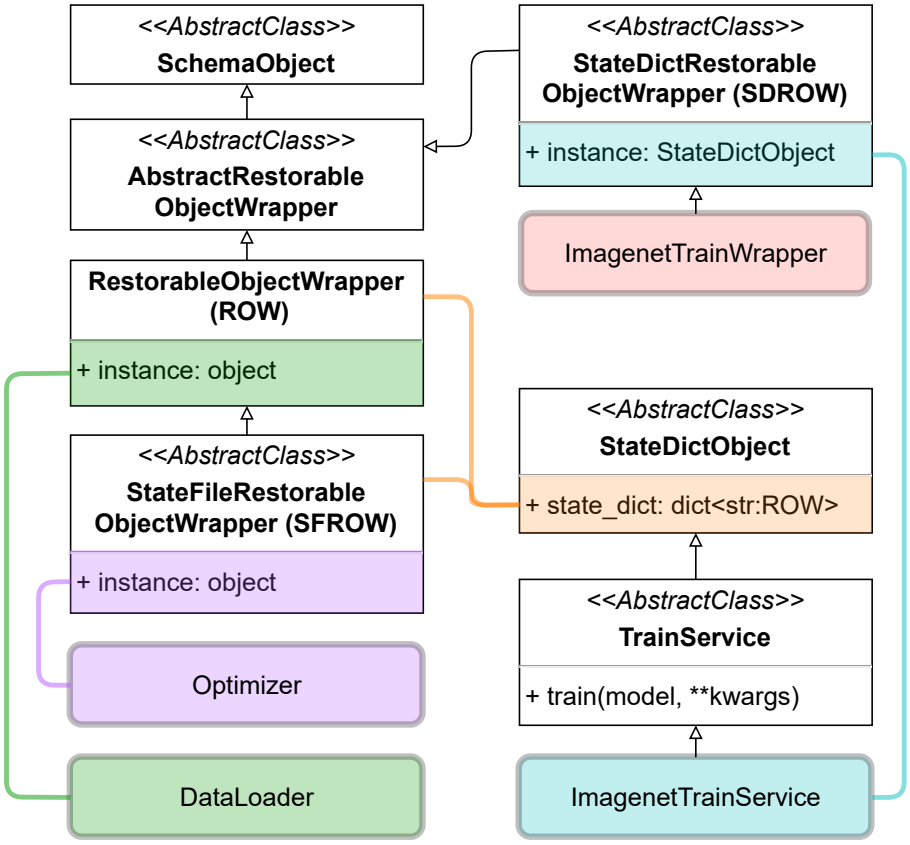


Figure 20: Simplified class diagram for tracking the model provenance data.

7.3. Structure the Data

Having discussed how to represent the model provenance data and explicitly the training logic, we look at how to integrate all data in our schema. Figure 21 shows a reduced version of the storage schema (for the entire schema, we refer to the Figure 53 in the appendix).

Missing entities that we discussed in the previous section are: ROW, SFROW, and SDROW. Entities we already know from previous approaches are *ModelInfo*, *RecoverInfo-FullModel*, and *Environment*. Newly added entities are *RecoverInfo-Provenance*, *Dataset*, and *TrainInfo*.

RecoverInfo-Provenance is the high-level representation of the provenance data and the third alternative after *RecoverInfo-FullModel* and *RecoverInfo-ParamaterUpdate* (see Figure 13) to represent the information to recover a model. It references the *Environment* the model was trained in, the *TrainInfo* that we extensively discussed above, and the *Dataset* holding a reference to the compressed training data.

We use the same method as for all previously presented approaches to persisting the entities. We save files in a local or shared file system and metadata as documents in a MongoDB.

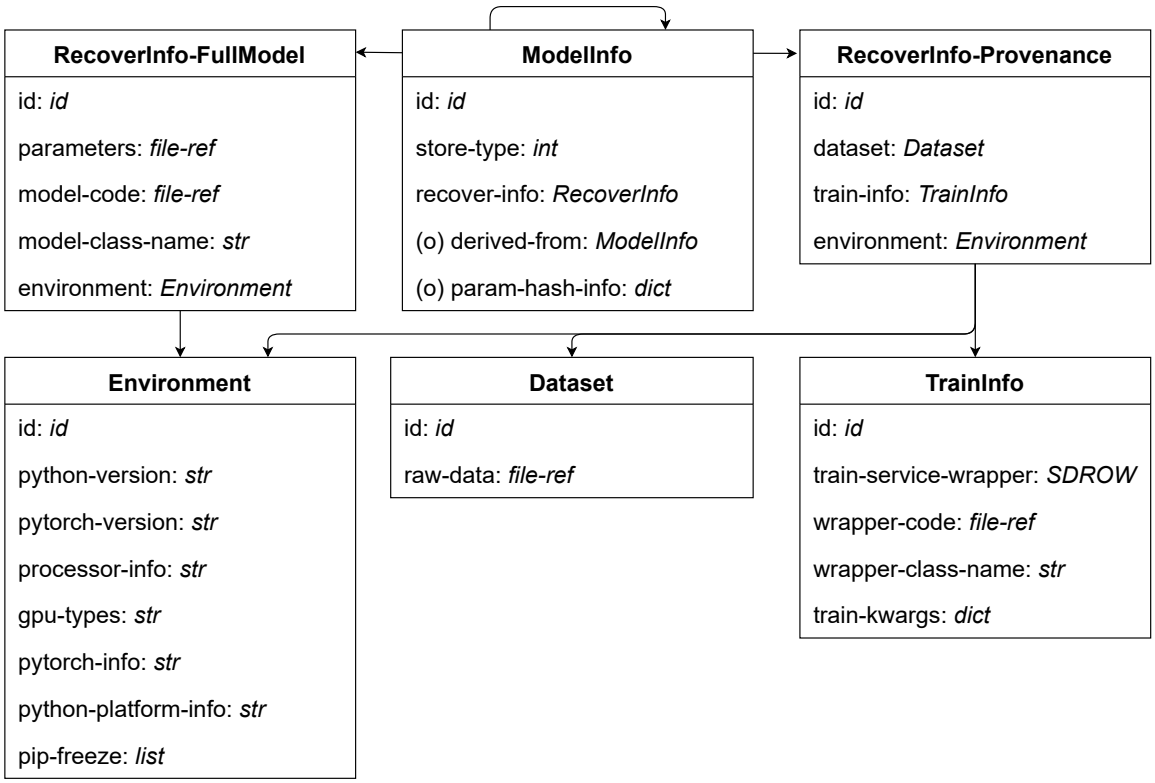


Figure 21: Reduced version of the storage schema for the provenance approach ($\mathcal{M}_{Prov}$).

7.4. Recover a Model

With $\mathcal{M}_{Prov}$, we save a model M ($B \rightarrow M$) always by referencing its base model B. Therefore, to recover M, we have to recover B and, if B is also derived, all B's base models. This makes recovering M a recursive process that is almost identical to the one for $\mathcal{U}_P$ presented in Section 6.5. The only difference is that $\mathcal{M}_{Prov}$ reproduces the model training instead of applying a parameter update.

In Algorithm 2 we describe this process in pseudo-code. First, we check in line two if the model we want to recover was saved as a complete model using $\mathcal{B}$; if this is the case, we can recover it without recovering any base models. If the model is dependent on a base model, we recover the base model and use it as a starting point to reproduce the training process, which transforms the base model to the model we want to recover. We reproduce the training by accessing the saved provenance data and by training deterministically using the techniques presented in Section 3.4.

Algorithm 2 Recursively recover a model saved using $\mathcal{M}_{Prov}$

```
1: if is_full_model(model_id) then
2:     model = get_full_model(model_id)
3:     return model
4: else
5:     base_model_id = get_base_model_id(model_id)
6:     base_model = recover_model(base_model_id)
7:     train_service = recover_train_service(model_id)
8:     model = train_service.train(base_model)
9:     return model
10: end if
```

7.5. Expectations

Having discussed $\mathcal{M}_{Prov}$, we expect it to behave as follows:

- The training dataset is the largest factor for the overall **storage consumption**.
- The **storage consumption** is mostly independent of the model architecture.
- The **time-to-save (TTS)** a model depends mainly on the size of the training dataset.
- The **time-to-recover (TTR)** a model increases with the number of base models because of the recursive recovery process; it mainly depends on the training time.

In Section 8 we will see that $\mathcal{M}_{Prov}$ meets all these expectations.

8. Evaluation

In this section we first describe the experimental setup we used to evaluate all approaches. Afterwards we present and extensively analyze how every approach performs in terms of storage consumption, time-to-save (TTS), and time-to-recover (TTR). Finally, we will compare the approaches to provide suggestions as to which approach is particularly suitable in certain scenarios.

8.1. Experimental Setup

In this section, we define the sequence of use cases, the set of model architectures, and the datasets we used to evaluate all our approaches.

8.1.1. Evaluation Flow

To evaluate our approaches, we defined an *evaluation flow* shown in Figure 22(a). This is a sequence of operations corresponding to the use cases presented in Section 4.2 and consists of the three entities (*server*, *node*, and *database*) communicating with each other.

(1) The *evaluation flow* starts with U_1: the server initially trains a model, saves it in the database, and informs the node about it. (2) The node recovers the model, adjusts it on a locally collected dataset, saves it on the database, and informs the server about it (U_3). We repeat this four times and declare each of the repetitions of U_3 as $U\_3\_1\_n$ with $n \in \{1, 2, 3, 4\}$. (3) The server takes the model that it created in U_1, trains it on a new dataset, saves it in the database, and informs the node about it (U_2). (4) The node recovers the model and again runs four iterations of U_3 named $U\_3\_2\_n$ with $n \in \{1, 2, 3, 4\}$. (5) The server recovers each model that was created and executes checks to make sure we are able to recover all models without loss of precision.

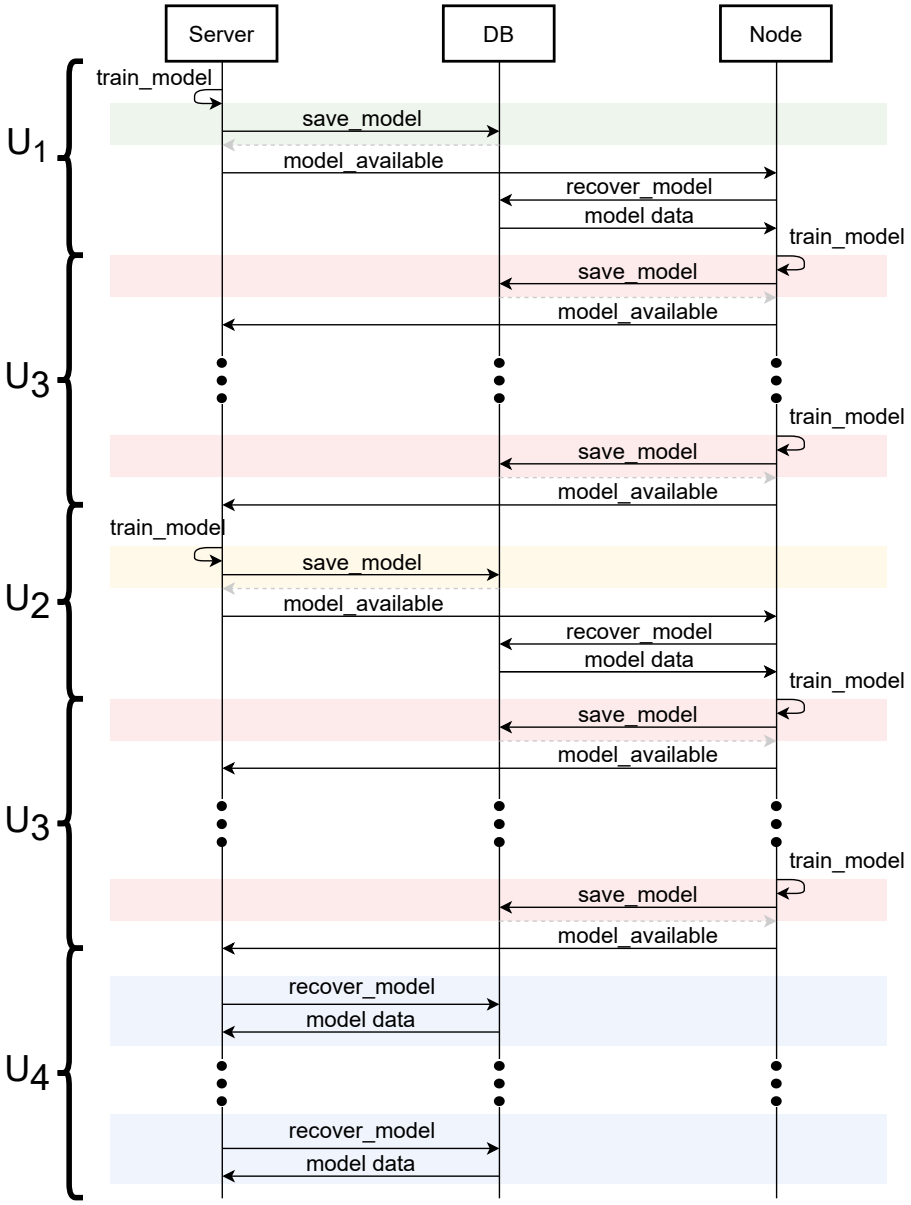
Each execution of the evaluation flow leads to a set of ten models, their relation is shown in Figure 22(b). The root model is created in U_1, and the directly derived red models are created during the iterations of $U\_3\_1\_n$. The yellow model is derived from the root model in U_2 and the red models derived from it are created in the iterations of $U\_3\_2\_n$. It is important to note that the green model (created in U_1) is transitively a base model of all other models.

The *evaluation flow* gives information about how and when different parties save and inform about models. However it does not define what models we train and exchange, and it also does not define on what dataset we train the models on.

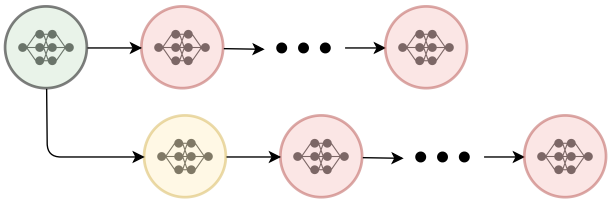
8.1.2. Models

Architectures To evaluate the approaches we decided on the set of model architectures $\mathcal{M}$ listed in Table 1. They all belong to the computer vision (CV) domain, an essential field under active research, and cover a wide range of parameters used, layers, and storage sizes

At the time of publishing GoogLeNet and all the ResNets were amongst the best performing approaches in the ImageNet challenge [42]. While GoogLeNet and ResNet were



(a) The evaluation flow consisting of a sequence of use cases.



(b) Models that are created during the execution of the evaluation flow.

Figure 22: Evaluation flow and derived models.

not developed with any hardware or other resource constraints in mind, the MobileNetV2 is a network aiming to be applicable in mobile and resource-constrained environments and was also evaluated on the ImageNet dataset.

Due to ImageNet's great popularity, many publications and sources analyzed the ideas and performance for all models we have selected [1, 4, 24] which make them very well understood, tested, and often used in research. The ResNet architectures in particular are often used as a starting point or baseline for new approaches.

With the models being used regularly, the *PyTorch* framework provides implementations together with pretrained parameters for all model architectures in $\mathcal{M}$ [51].

Versions and Fine-tuned Versions In the evaluation, we focused on model versions and fine-tuned versions. While for model versions we trained all parameters of the model, for fine-tuned versions we had to decide which subset of the model parameters to train.

All selected model architectures follow the same concept: almost all layers are convolutional layers and extract features from the input images. Only the last layer is fully connected and maps the extracted features to the final output which is usually an image class represented as an integer.

If models are to be be slightly adjusted or we want to apply transfer learning [57] but only have a small dataset, one approach is to retrain only the linear classifier part (meaning the fully connected layers) of the model [52]. We adopted this approach to select the parameters and layers we retrain in the case of fine-tuned versions. The column *fine-tuned* in Table 1 indicates how many of the parameters we retrained for fine-tuned versions.

Name	Layers	Params	Fine-tuned	Size	Publication
MobileNetV2	53	3,504,872	1,281,000	14.3 MB	Sandler et al. [43]
GoogLeNet	22	6,624,904	1,025,000	26.7 MB	Szegedy et al. [50]
ResNet-18	18	11,689,512	513,000	46.8 MB	He et al. [20]
ResNet-50	50	25,557,032	2,049,000	102.5 MB	He et al. [20]
ResNet-152	152	60,192,808	2,049,000	241.7 MB	He et al. [20]

Table 1: Set of selected model architectures ($\mathcal{M}$) for the evaluation with the number of layers, trainable parameters, trainable parameters in case of a fine-tuned model version, the size of the weights saved as a pickle dump, and the publication.

8.1.3. Datasets

Having selected the set of models we used for the evaluation, we needed to define datasets that matched the capabilities of the model architectures and the concept of the individual use cases. Table 2 shows the datasets we selected or created, together with the number of images they contain, their sizes, and the corresponding use case.

U_1 is about distributing an initial model to the nodes. In real-world scenarios, the initial model is extensively trained and optimized to achieve the best possible performance.

To save resources, we use the pre-trained parameters provided by *PyTorch* that were generated by training on the ImageNet training dataset from 2012 [42].

In U_2 we distribute a 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 simulate this, we trained the initial models on the ImageNet validation dataset from 2012.

U_3 is about training a model on a locally collected dataset. These locally observed data are likely to be from a different distribution than the initial training set and slightly biased. To simulate this we created two different datasets: *Coco-food-512* and *Coco-outdoor-512*. Both are subsets of the Coco dataset [5] and contain 512 images each matching one of the categories in the ImageNet dataset [13].

The *Coco-food-512* dataset contains images of the categories: pizza, broccoli, banana, orange, and hot dog. The *Coco-outdoor-512* dataset contains images of the categories: traffic light, kite, umbrella, parking meter, and backpack.

In U_4 we recover the model from the data communicated in U3. Thus, there is no dataset required to simulate this use case.

We evaluated all approaches in various different configurations. Our provenance approach $\mathcal{M}_{Prov}$ includes compressing and saving the datasets on which we trained the model. The ImageNet validation dataset is about 6.3 GB, and too large to evaluate a high number of experiments and configurations in a reasonable amount of time. Therefore, we created and used a subset of the ImageNet validation dataset that we call *mini ImageNet validation* dataset.

Name	Shortname	Images	Size	Use case
ImageNet validation	$INet_{val}$	50,000	6.3 GB	U_2
mini ImageNet validation	$mINet_{val}$	1,400	200 MB	U_2
Coco-food-512	CF-512	512	94.3 MB	U_3
Coco-outdoor-512	CF-512	512	71.6 MB	U_3

Table 2: All datasets used throughout the evaluation with their: name, shortname, number of images, size, and the corresponding use case.

8.1.4. Execute Experiments

We defined one experiment as a full run of the evaluation flow for a given approach, model architecture, model relation, and dataset. We define a single experiment as a tuple (a, m_a, m_r, d_{U_3}), with

- a defining the **approach** ($a \in \{\mathcal{B}, \mathcal{U}_P, \mathcal{U}'_P, \mathcal{M}_{Prov}\}$)
- m_a defining the model **architecture** ($m_a \in \mathcal{M}$)
- m_r defining the **model relation** ($m_r \in \{version, \textit{fine-tuned version}\}$)

[13] For both datasets we give a detailed description of how we created them in our GitHub repository `https://github.com/slin96/master-thesis`

- d_{U_3} defining the used **dataset** for U_3 $d \in \{\text{CF-512,CO-512}\}$.

For all results we present, we executed every possible experiment five times and took the median computation time and storage consumption, respectively.

Setup We simulated every instance in the evaluation flow (*node*, *server*, and *database*) by one separate machine to test the case of saving a model on one machine and recovering it on another. All machines had identical hardware and software: 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 were connected via 100G InfiniBand and had access to the same external storage.

Model Training To simulate the changing model parameters, we trained for ten epochs on $INet_{val}$ and for five epochs per iteration of U_3 on CF-512 or CO-512.

To evaluate all approaches for all model architectures, model relations, and datasets, we had to execute 400 experiments – 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 an extensive evaluation feasible, we decided to train the models before the actual experiments and load them from snapshots instead of repeating the training procedure each time.

We trained the models deterministically on a DGX A100 machine with 1 TB RAM using one of eight NVIDIA A100-SXM4-40GB GPUs. The training times, relevant for the evaluation of $\mathcal{M}_{Prov}$ are shown in Table 3 in Section 8.3.3.

It is out of this thesis's scope to optimize the training. For example, we set the number of workers for the dataloader to 0. By choosing a higher number of workers and other additional optimization steps, we would expect to see a significant decrease in training time.

8.2. Storage Consumption

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

8.2.1. Baseline

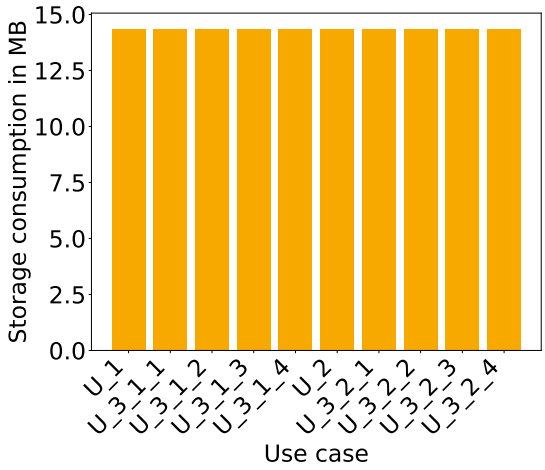
In Figure 23(a), we show the storage consumption for MobileNetV2 model versions trained on CF-512 across different use cases. The numbers indicate that the use case has no effect on the storage consumption. Looking at other scenarios we noticed throughout our evaluation that the storage consumption is also independent of the dataset and the model relation; we have even seen this effect for all model architectures in $\mathcal{M}$. Taking the numbers from Figure 23(c) into account, the only exception is a 43 Byte lower storage consumption for U_1.

The baseline approach ($\mathcal{B}$) 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, which explains the constant storage consumption across use cases, datasets, and model relations. We can explain the 43 Byte difference between models saved in U_1 and all other use cases by the fact that U_1 is the only use case where the metadata does not hold a reference to a base model.

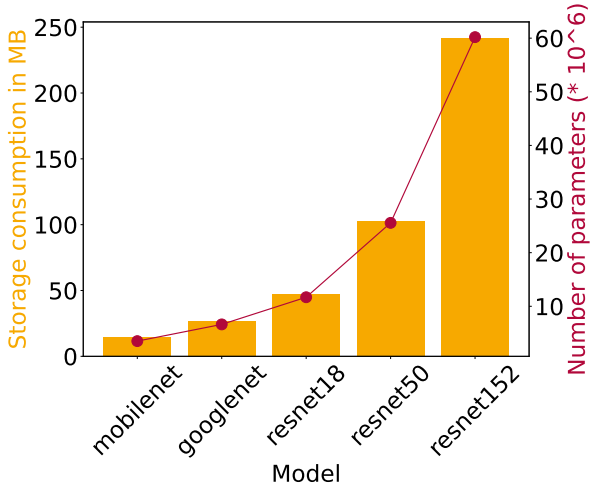
The data shown in Figure 23(c) imply that the storage consumption is dependent on the model architecture. For example, saving a MobileNetV2 takes approximately 14 MB whereas saving a ResNet-18 takes about 47 MB. The data in Figure 23(b) suggest that the critical factor is the number of parameters; with an increased number of parameters the storage consumption increases as well. Figure 23(d) confirms this by showing that the parameters take the majority of the total storage needed to save a model using $\mathcal{B}$.

While the metadata and the model architecture take up little space – the metadata is saved as multiple JSON files, the architecture is defined by a code file – each model parameter is represented by a 32-bit floating-point number. This explains the correlation between the storage consumption and the number of parameters.

In summary, we see 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.**



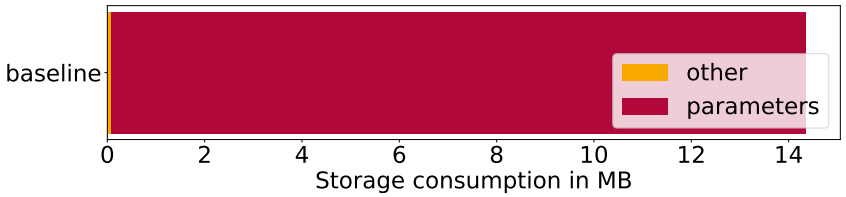
(a) Median storage consumption per use case for saving MobileNetV2 model versions. U_3 models were trained on the Custom-Coco-Food-512 dataset.



(b) Storage consumption and number of parameters per model.

Model	use cases 1	use cases 2	use cases 3
MobileNetV2	14,345,728	14,345,771	14,345,771
GoogLeNet	26,769,975	26,770,018	26,770,018
ResNet-18	46,873,903	46,873,946	46,873,946
ResNet-50	102,627,679	102,627,722	102,627,722
ResNet-152	241,902,565	241,902,608	241,902,608

(c) Median storage consumption (in Bytes) per model and use case.



(d) Distribution of MobileNetV2 storage consumption divided into *parameters* and *other*.

Figure 23: Storage consumption using $\mathcal{B}$.

8.2.2. Parameter Update Approach

$\mathcal{U}_P$ and $\mathcal{U}'_P$ only differ in how they compute the parameter update but not in what data they save. This is why, in this section, all numbers and results for the storage consumption are representative of both approaches. In the following, when we refer to $\mathcal{U}_P$ we implicitly also refer to $\mathcal{U}'_P$.

Looking at the numbers in Figure 24(a), we see that using $\mathcal{U}_P$ decreases the storage consumption (compared to $\mathcal{B}$) for saving a version of the MobileNetV2 by approximately 9.7 KB, and for the ResNet-152 by approximately 10.2 KB for all use cases except for U_1. For both model architectures, this is a decrease of less than one percent. Across all models $\mathcal{M}$ we see: the more complex the model architecture the more the storage consumption decreases in comparison to a complete model snapshot; nevertheless, the decrease is only marginal.

For *model versions*, all model parameters are trainable and it is unlikely that a subset of them remains unmodified. $\mathcal{U}_P$ only saves the updated parameters. However, if all parameters are changed, a parameter update is equivalent to a full snapshot of all parameters, and $\mathcal{U}_P$ does not improve the storage consumption compared to $\mathcal{B}$.

$\mathcal{U}_P$ does not save the model code and metadata for every model and instead refers to its base model. Larger models have more layers, which results in a larger code file and in more metadata. This is why $\mathcal{U}_P$ saves a marginal amount of space, which increases slightly with the complexity of a model for all use cases except U_1.

Comparing Figure 24(b) with Figure 24(c) and Figure 24(d) with Figure 24(e), the numbers show that for fine-tuned model versions the storage consumption significantly improves compared to $\mathcal{B}$. Considering the number of trainable parameters for each model architecture (shown in Table 1), we find that a lower amount of trainable parameters results in reduced storage consumption. According to Figure 24(a) the storage consumption decreases, with the exception of U_1, by 63.7% for the MobileNetV2, and by 95.6% for the ResNet-152 architecture for all use cases.

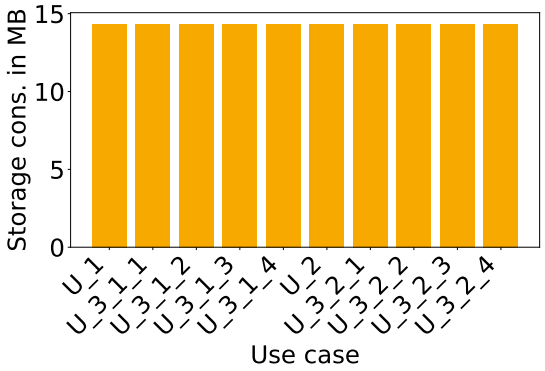
For *fine-tuned model versions*, only some of the model parameters change during training. The parameter update does not have to contain all model parameters but only the subset that has been changed. This results in a decreased storage consumption.

The distributions of the storage consumption in Figure 24(f) suggest, that also for $\mathcal{U}_P$ the model parameters are the critical factor for the overall storage consumption regardless of the model relation. $\mathcal{U}_P$ is, just as $\mathcal{B}$, based on saving the model parameters which is why the storage consumption is dependent on the model architecture and not on other factors like the training dataset.

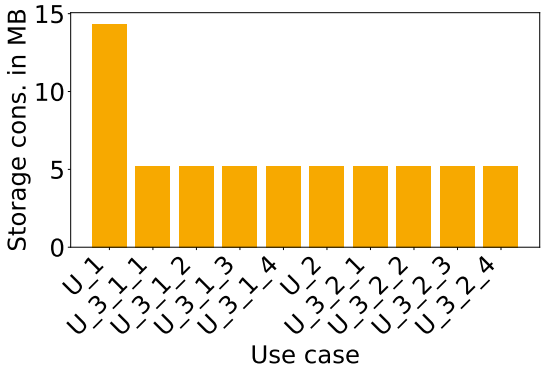
Overall we see that **for model versions $\mathcal{U}_P$ marginally improves the storage consumption, whereas for fine-tuned model versions the improvement is noticeably higher.**

Model	Relation	Approach	use case 1	use case 2	use case 3
MobileNetV2	–	$\mathcal{B}$	14,345,728	14,345,771	14,345,771
MobileNetV2	version	$\mathcal{U}_P$ and $\mathcal{U}'_P$	14,345,728	14,336,054	14,336,054
MobileNetV2	fine-tuned	$\mathcal{U}_P$ and $\mathcal{U}'_P$	14,345,728	5,202,622	5,202,622
ResNet-152	–	$\mathcal{B}$	241,902,565	241,902,608	241,902,608
ResNet-152	version	$\mathcal{U}_P$ and $\mathcal{U}'_P$	241,902,565	241,892,452	241,892,452
ResNet-152	fine-tuned	$\mathcal{U}_P$ and $\mathcal{U}'_P$	241,902,565	8,399,218	8,399,218

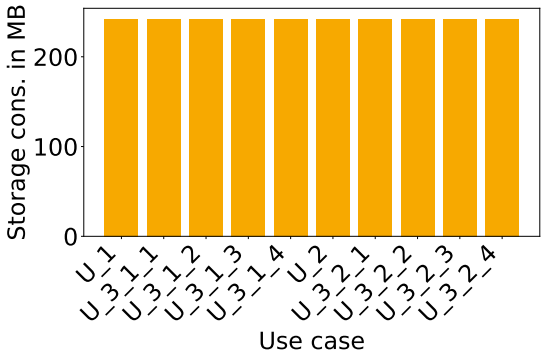
(a) Median storage consumption for MobileNetV2 and ResNet-152 (in Bytes) per model, model relation, approach, and use case.



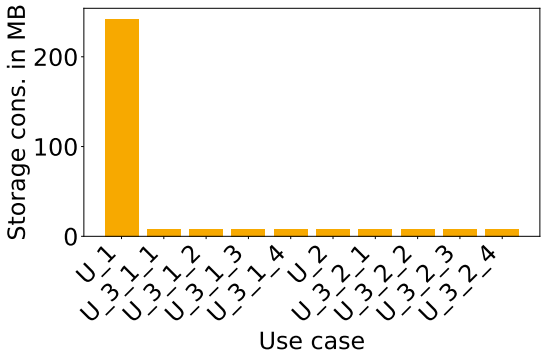
(b) Median storage consumption per use case for saving MobileNetV2 model versions. U_3 models trained on the Custom-Coco-Food-512 dataset.



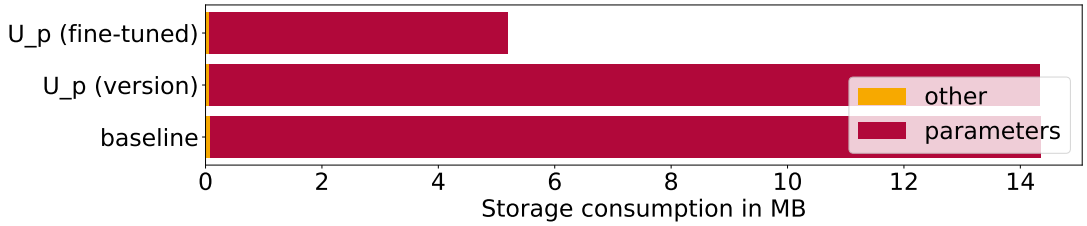
(c) Median storage consumption per use case for saving MobileNetV2 fine-tuned model versions. U_3 models trained on the Custom-Coco-Food-512 dataset.



(d) Median storage consumption per use case for saving ResNet-152 model versions. U_3 models trained on the Custom-Coco-Food-512 dataset.



(e) Median storage consumption per use case for saving ResNet-152 fine-tuned model versions. U_3 models trained on the Custom-Coco-Food-512 dataset.



(f) Distribution of MobileNetV2 storage consumption divided into *parameters* and *other*.

Figure 24: Storage consumption using $\mathcal{U}_P$.

8.2.3. Provenance Approach

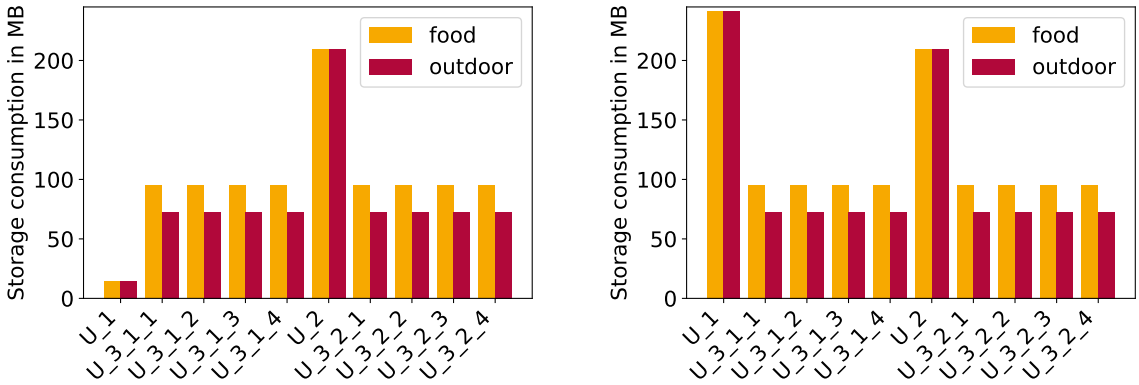
Comparing Figure 25(a) and Figure 25(b), we see that the storage consumption per use case is similar, although the numbers in Figure 25(a) apply to the MobileNetV2 architecture, and the numbers in Figure 25(b) for the ResNet-152 architecture. Both suggest that the model architecture and number of parameters do not meaningfully influence the storage consumption.

This is due to the fact that $\mathcal{M}_{Prov}$ saves the model provenance data. Parts of the provenance data depend on the model architecture, but this data is only metadata (for example the code file and hash values for all layers) and thus very small in size. The reason for the large difference between Figure 25(a) and Figure 25(b) for U_1's storage consumption is that for U_1 all approaches save the model using the baseline approach.

The numbers in Figure 25(a) and Figure 25(b) suggest that the storage consumption mainly depends on the dataset that we used to train the model. For a MobileNetV2 the dataset is responsible for more than 99.9% of the storage consumption and whenever we used a larger dataset also the storage consumption increases. CF-512, for example, is approximately 23 MB larger than CO-512, which is roughly the difference between the storage consumption for all U_3s. For U_2, we always used the smaller version of the ImageNet data ($mINet_{val}$) and see no difference in the storage consumption.

As mentioned earlier, $\mathcal{M}_{Prov}$ saves the model provenance data. In addition to metadata the provenance data also contain the training dataset which, relative to the other provenance data, is significantly larger. This explains why the storage consumption mainly depends on the size of the training dataset.

In summary, we see that **the storage consumption mainly depends on the training dataset and is almost independent of the model architecture.**



(a) MobileNetV2

(b) ResNet-152

Figure 25: Median storage consumption using $\mathcal{M}_{Prov}$ per use case for saving model versions. U_3 models trained on the Custom-Coco-Food-512 and Custom-Coco-Outdoor-512 dataset, U_2 models trained on a reduced version ($\sim$ 200 MB) of the ImageNet validation dataset.

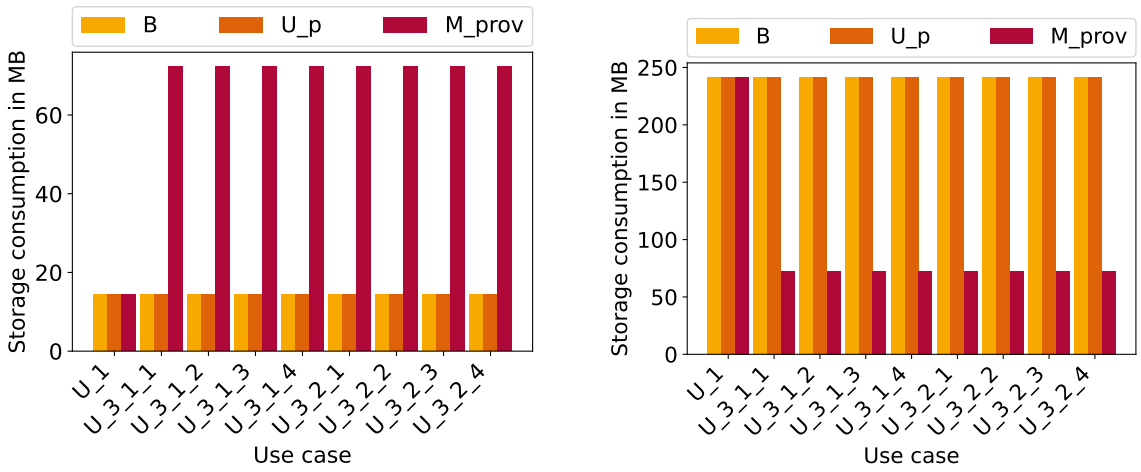
8.2.4. Comparing Approaches

In the following section, we compare the approaches' storage consumption in different scenarios. To avoid peaks in the figures, which make it hard to analyze the rest of the data, we do not include U_2 and refer to the detailed analysis in earlier sections. We also do not differentiate between $\mathcal{U}_P$ and $\mathcal{U}'_P$ since their storage consumption per model is identical.

Model Versions As Figure 26(a) and Figure 26(b) indicate, when saving model versions $\mathcal{U}_P$ shows only negligible improvement in storage consumption compared to $\mathcal{B}$.

Whether $\mathcal{M}_{Prov}$ would improve the storage consumption, depends on how large the provenance data (mainly dependent on the dataset) is in comparison to the number of parameters in the model architecture.

In Figure 26(a), we see that if saving a model using $\mathcal{B}$ consumes less storage than saving the provenance data, $\mathcal{M}_{Prov}$ does not improve the storage consumption. If it is the other way around, as shown in In Figure 26(b), $\mathcal{M}_{Prov}$ does save storage compared to $\mathcal{B}$ and also to $\mathcal{U}_P$ (in this case approximately 70%).



(a) MobileNetV2

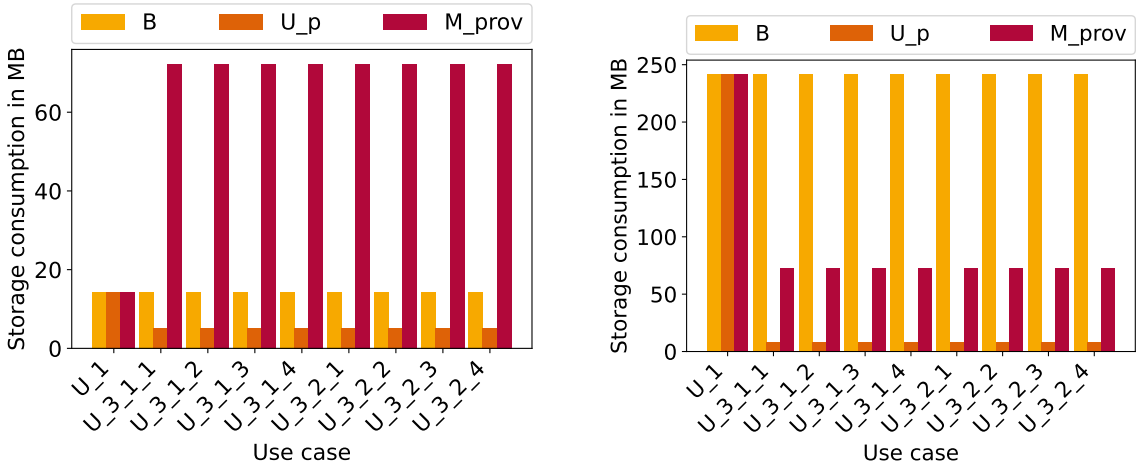
(b) ResNet-152

Figure 26: Median storage consumption per use case for saving model versions across approaches. All models in U_3 were trained on the Custom-Coco-Outdoor-512 dataset.

Fine-tuned Model Versions The storage consumptions in Figure 27(a) and Figure 27(b) show that $\mathcal{U}_P$ has the potential to drastically reduce the storage consumption compared to $\mathcal{B}$ for fine-tuned model versions. The important factor is how many of the model parameters we retrain; the more parameters change the closer we get to the case of model versions and the storage consumption increases.

For $\mathcal{M}_{Prov}$ the storage consumption mainly depends on the size of the provenance data and not on the model parameters. This is why it does not make any difference for this

approach if we save fine-tuned model versions or model versions. The only aspect that changes is: since the storage consumption for $\mathcal{U}_P$ improves for fine-tuned model versions $\mathcal{M}_{Prov}$'s performance decreases compared to $\mathcal{U}_P$.



(a) MobileNetV2

(b) ResNet-152

Figure 27: Median storage consumption per use case for saving fine-tuned model versions across approaches. All models in U_3 were trained on the Custom-Coco-Outdoor-512 dataset.

Other Model Relations Although we only evaluated model versions and fine-tuned model versions in our experiments, we want to discuss what we expect to see for the other model relations defined in Definition 3.

For derived model architectures, all parameters and the model architecture change. This lets us assume that when using $\mathcal{U}_P$ there is even less potential to reduce the storage consumption than for model versions.

For extended models, we expect to see that the number of parameters that remain unchanged will be the crucial factor in reducing the storage consumption using $\mathcal{U}_P$.

Using $\mathcal{M}_{Prov}$, the storage consumption does not depend on the model relation. Thus, for models with derived architectures and extended models we expect similar results to those presented above.

8.3. Time to Recover

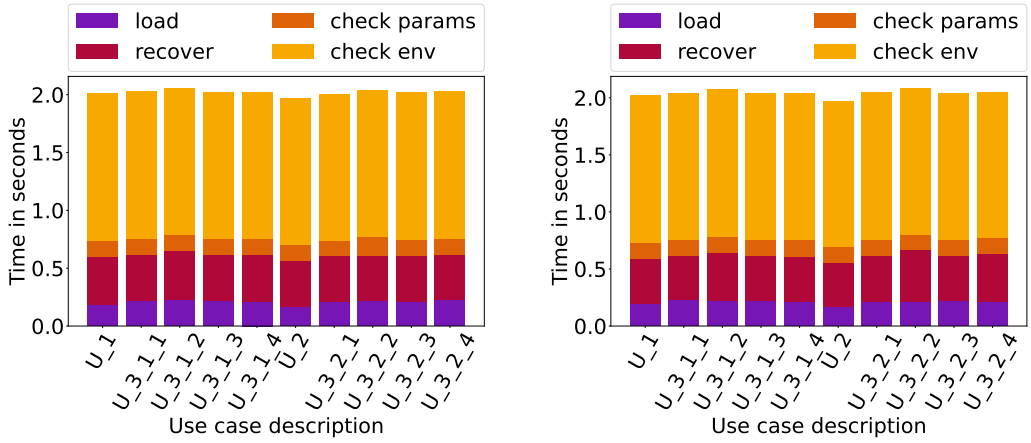
In this section, we analyze the amount of time it takes to recover a given model for all approaches. With *time-to-recover (TTR)* we refer to the time that is needed to recover a specific model in U_4, which also includes the time it takes to load the data. We identify the models throughout the evaluation 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 in U_4 to recover the model that we saved in U_2.

As we will see, for $\mathcal{U}_P$ the time-to-save a model depends on the TTR of its base model. Therefore, we discuss the TTR before the time-to-save.

8.3.1. Baseline

In the following figures, we present representative numbers for the TTR using $\mathcal{B}$ divided into four categories: *load* (time to load all data necessary to recover the model), *recover* (time to recover the model from the loaded data), *check params* (time to check if all parameters of the recovered model are identical to the ones of the saved model), and *check env* (time to check if the environment the model has been saved in is equal to the environment it is recovered in)

Figure 28(a) shows the TTR for a ResNet-50 with the U_3 models being trained on *CF*-512. All TTRs are, independently of the use case, approximately two seconds long. Checking the environment takes the longest (over one second), followed by recovering the model, loading the data, and checking the model parameters. Except for the load time, all times are close to constant. Comparing Figure 28(a) to Figure 28(b), the data suggest that the TTR for $\mathcal{B}$ is not only independent of the use case but also independent of the dataset and model relation.



(a) model versions, U_3 trained on the Coco-food-512

(b) fine-tuned model versions, U_3 trained on the Coco-outdoor-512

Figure 28: Time-to-recover (TTR) for ResNet-50 using $\mathcal{B}$ divided into four categories: *load* is the time to load all data necessary to *recover* a given model. *check params* and *check env* specify the time it takes to check if the parameters and the environment of the recovered model are equal to the model parameters and environment when the model was saved.

While these results mostly meet our expectations, it is surprising to see that checking the environment takes the longest. It consists of two operations we would expect to be quick: (1) collecting the environment information on the current machine, and (2) comparing it to the environment information saved with the model. While analyzing this anomaly, we found that the operation *torch.utils.collect_env.get_env_info()*, which we use to get detailed information about PyTorch, takes more than one second. Why this is the case, is out of the scope of this thesis but for future implementations it would be beneficial to find the cause in order to speed up recovering models. To avoid this long TTR, we could deactivate the corresponding check, for example, if we are certain that all machines have an equivalent environment.

The root cause for the varying total TTR is the time to load all data. We see the reason for its variation in the fact that it strongly depends on multiple external factors like the performance of the shared file system, the performance of the database (in our case MongoDB), and the network connection.

The machines we use for the experiments are well equipped, well connected, and the shared filesystem is rapidly accessible. In real-world environments, the potential for external factors to influence the load time is higher and we would expect to see it fluctuating more.

Recovering and checking the weights are local operations without external dependencies resulting in constant computation times.

Figure 29 illustrates how the TTR changes based on the model architecture (we left out the *check env* time for better comparability). The displayed data demonstrate that the total, but also the more specific, times increase with the number of model parameters. 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.

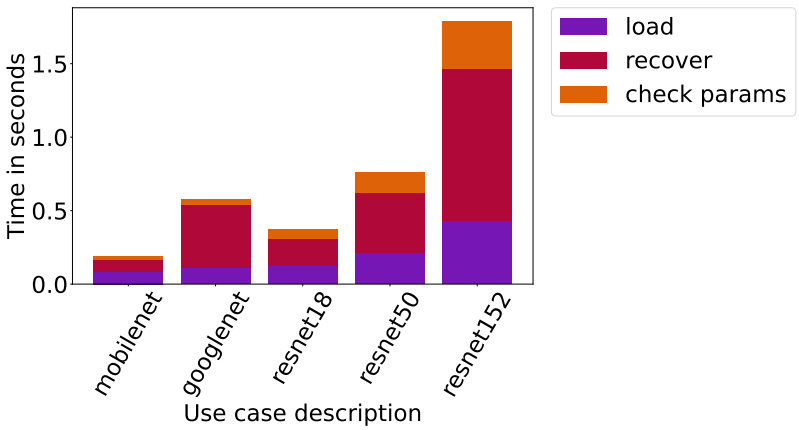


Figure 29: Time-to-recover (TTR) for different model architectures, *check env* time excluded.

The reason for this peak – in comparison to all other models in $\mathcal{M}$ – is a disproportional high computation time for GoogLeNet's initialization routine. We found that it takes approximately seven times longer than initializing a ResNet-18. The developers of

PyTorch confirm this issue by printing a warning about the long initialization time caused by the use of the *scipy* framework for initializing GoogLeNet [14].

With the exception of GoogLeNet, we can summarize our findings as follows: **the time-to-recover (TTR) depends solely on the model architecture and the number of parameters.**

8.3.2. Parameter Update Approach

$\mathcal{U}_P$ and $\mathcal{U}'_P$ only differ in how they compute the parameter update but not in how they recover models. As in previous sections, we only refer here to $\mathcal{U}_P$ and by this implicitly also refer to $\mathcal{U}'_P$.

Moreover, note that for all U_1s, $\mathcal{U}_P$ uses $\mathcal{B}$ to save and recover the model; for the first model we save, there is no base model on which to calculate an update.

In the following figures, we present representative numbers for the TTR using $\mathcal{U}_P$ divided into five categories: *recover base*, *load*, *recover*, *check parameters*, and *check env.* *Recover base* describes the time it takes to recover the model's base model; all other categories are identical to the ones from Section 8.3.1.

The data in Figure 30(a) indicate that, when recovering a model M (with $B \rightarrow M$), the time to recover B is the longest and increases over time, which forms two staircase patterns starting at U_1 and U_2. The TTR of a given model M is approximately the TTR of its base model B plus the time to load, recover, and perform both checks.

The setting in Figure 30(b) differs from the one in Figure 30(a) only in the model relation. The comparison suggests that for fine-tuned model versions the TTR is shorter and increases by a smaller factor. It does not change how the categories develop in relation to each other across the use cases.

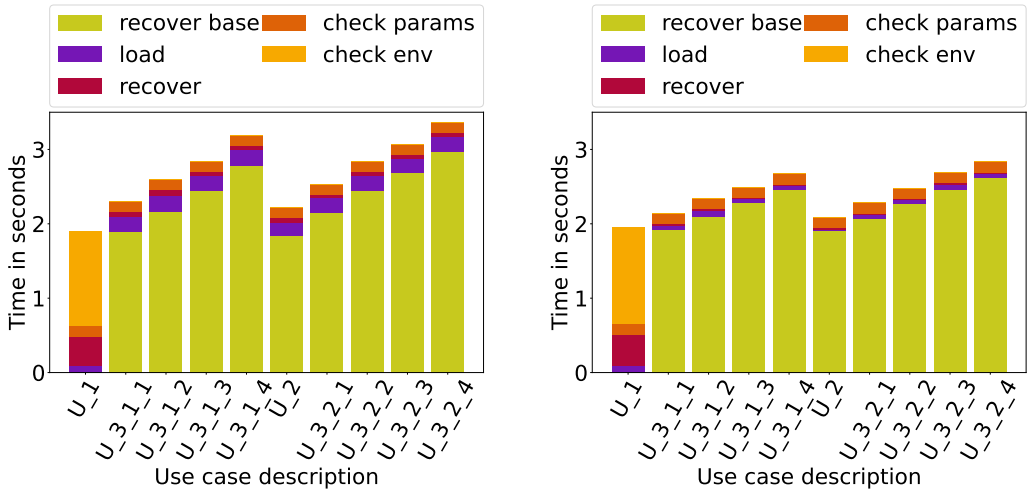
As described in Section 6.5, recovering a model M (with $B \rightarrow M$) that we saved using $\mathcal{U}_P$ is a recursive process and includes recovering B. Taking into account how the models that we create during the evaluation flow relate to each other, we can explain that the staircase patterns start at U_1 and U_2. To recover a model $U_{3\_1\_n}$, $\mathcal{U}_P$ has to recover all its base models: $U_{3\_1\_(n\text{-}1)}$ to $U_{3\_1\_1}$, and U_1. To recover a model $U_{3\_2\_n}$, $\mathcal{U}_P$ has to recover all its base models: $U_{3\_2\_(n\text{-}1)}$ to $U_{3\_2\_1}$, U_2, and U_1. The higher n, the more base models $\mathcal{U}_P$ has to recover and the higher the TTR.

For fine-tuned model versions $\mathcal{U}_P$ has to load the same number of base models, but a smaller amount of data. This results in a shorter TTR for model versions.

The data in Figure 31 indicate an increased TTR for more complex model architectures; also the detailed times (*recover base*, *load*, *recover*, and *check params*) follow this pattern. The only exception is the GoogLeNet architecture. Here, the TTR for the base model is disproportionately high, which is unexpected. The reason is GoogLeNet's long initialization time (already discussed in Section 8.3.1).

Overall, **the time-to-recover (TTR) depends on the model architecture, the number of parameters, the model relation, and the use case.**

[14] `https://github.com/pytorch/vision/blob/ef711591a5db69d36f904ab5c39dec13627a58ad/torchvision/models/googlenet.py#L78,accessed2021-06-10`



(a) model versions (b) fine-tuned versions

Figure 30: Time-to-recover (TTR) for ResNet-50 using $\mathcal{U}_P$, categories same as in previous figures, *recover base* describes the time it takes to recover a model's base model.

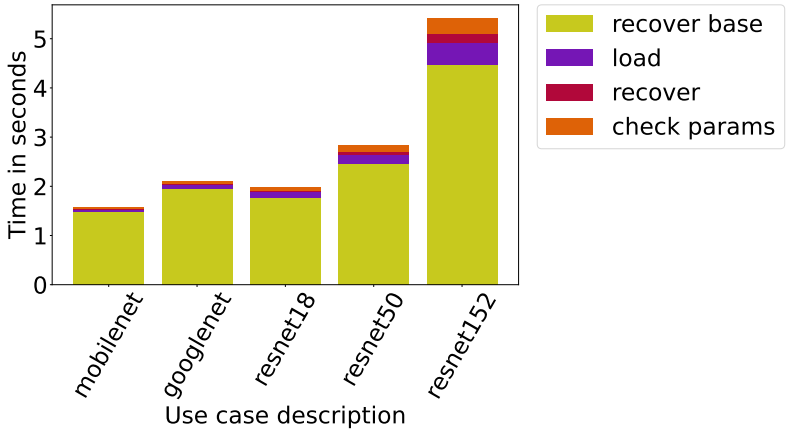


Figure 31: Time-to-recover (TTR) for different model architectures.

8.3.3. Provenance Approach

To evaluate the TTR for $\mathcal{M}_{Prov}$, we ran the model training that is part of the recovery process only for two epochs with two batches; this drastically reduces the training time. To give an insight into how long a recovery process (without any optimization, for example, for data loading) would actually take, we list times for training a full epoch per dataset and model in Table 3.

The following figures give insights into the TTR using $\mathcal{M}_{Prov}$. The times are divided into four categories: *recover base* (time to recover the base model), *load* (time to load the model provenance information), *training* (time to reproduce the model training), *recover full model* (time to recover a full model, only relevant for U_1).

In Figure 32(a), we show TTR for ResNet-50 model versions using the CF-512 dataset. It is apparent that *recover base*, followed by *training*, influences the TTR most. The *load*

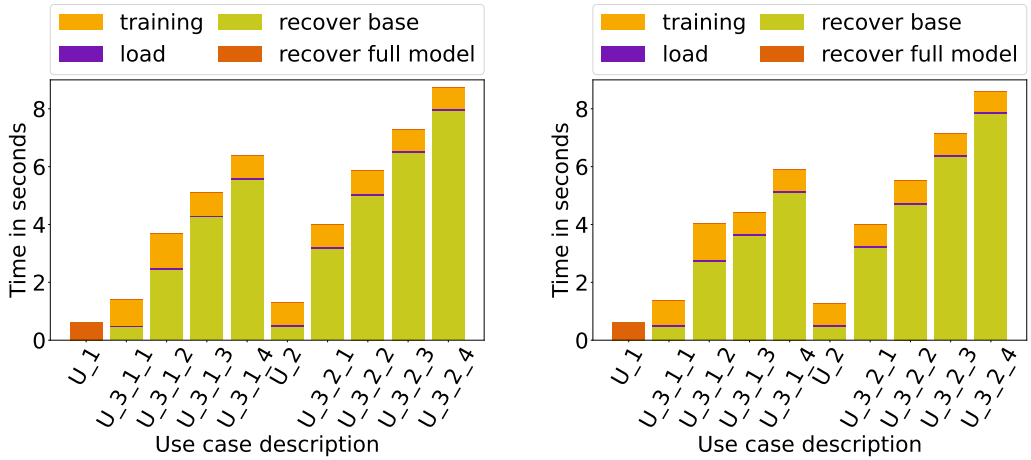
Model	ImageNet-val	food-512	outdoor-512
MobileNetV2	555.51	4.65	5.44
GoogLeNet	689.23	7.22	6.44
ResNet-18	640.06	6.72	5.89
ResNet-50	617.75	6.41	5.93
ResNet-152	1013.78	9.57	9.37

Table 3: Training times for one epoch in seconds per model and dataset. The training was performed deterministically and without any optimization for training time on a NVIDIA A100-SXM4-40GB GPU.

time is short and taking into account that the real training and TTR are much longer than displayed, makes it negligible.

Starting at U_1 and U_2 we see a steadily increasing *recover base* time forming a staircase pattern similar to the one seen for $\mathcal{U}_P$. For Figure 32(b), where we used the smaller dataset *CO*-512, we see the same patterns as in Figure 32(a), but with slightly shorter times.

Similar to $\mathcal{U}_P$, recovering a model using $\mathcal{M}_{Prov}$ is a recursive process. The more base models a given model M transitively refers to, the longer it takes to recover its direct base model. The reasoning is the same as described in Section 8.3.2 and explains the staircase patterns for the *recover base* time.



(a) model versions

(b) fine-tuned versions

Figure 32: Time-to-recover (TTR) for ResNet-50 using $\mathcal{M}_{Prov}$. The TTR divided into four categories: *recover base* specifies the time to recover the base model, *load* describes the time to load the model provenance information, *training* is the time to reproduce the model training, and *recover full model* defines the time to recover a full model (only relevant for U_1). Models in U_3 trained on the Coco-outdoor-512 dataset.

Figure 33 provides an overview of the TTR per model architecture. It is striking that for all models, except the GoogLeNet, the TTR and all categories tend to increase with more complex architectures. Taking into account that reproducing training takes the most time – clearly seen by looking at the number for $U_{3\_1\_1}$ where recovering the base model does not include model training – we can explain why the model architecture strongly influences the TTR.

Figure 33: Time-to-recover (TTR) for different model architectures, categories same as in Figure 32.

Overall we see that **the time-to-recover (TTR) mainly depends on the model architecture and the corresponding time to train the model, which is also part of the *recover base* time.**

8.3.4. Training Times

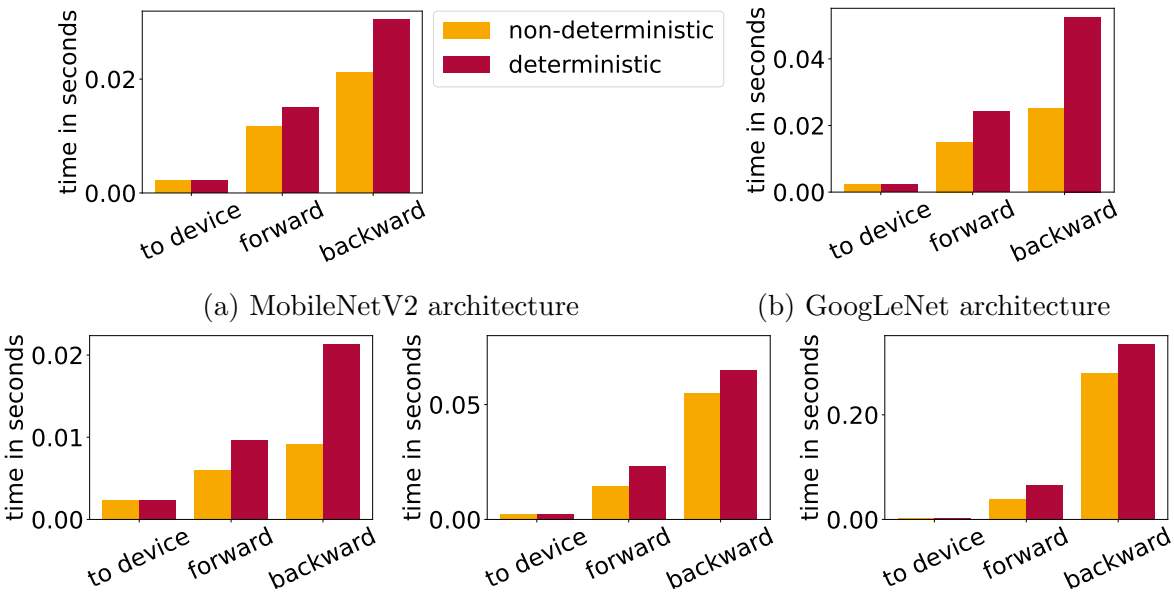
Deterministic Training Having trained all models deterministically, we did not know how and if a deterministic execution influences the training times compared to a non-deterministic execution. We also did not know if the effects are the same across different model architectures. The only information we find in the PyTorch documentation [38] is: "*deterministic operations are often slower than nondeterministic operations [...]*".

To get more detailed insights into how deterministic training performs compared to a non-deterministic execution, we set up a separate experiment and executed two training runs per model; a deterministic one and a non-deterministic one.

Figure 34 compares the median training time for one batch in a deterministic and non-deterministic mode for all models in $\mathcal{M}$. The data indicate that, in general, training a model deterministically is slower in the forward and backward pass.

The size of the relative difference, varies across the model architecture. We see the most drastic relative differences for the backward pass of the GoogLeNet and the ResNet-18. Finding out why this is the case is out of our scope. Most likely it would require a detailed analysis of the model architectures and the effect of deterministic training on the single layers.

Summarizing, we found that **using only deterministic operations slows down the training; by how much depends on the model architecture.**



(a) MobileNetV2 architecture (b) GoogLeNet architecture

(c) ResNet-18 architecture (d) ResNet-50 architecture (e) ResNet-152 architecture

Figure 34: Median models training time for one batch of 64 images, in *deterministic* and *non-deterministic* mode. The training was performed on a NVIDIA A100-SXM4-40GB GPU.

Inconsistent Training Times Comparing the training times in Figure 34 with the ones in Table 1, it is somewhat counterintuitive that the number in Table 1 suggests that training the ResNet-18 is slower than training the ResNet-50, but in Figure 34 the opposite is the case.

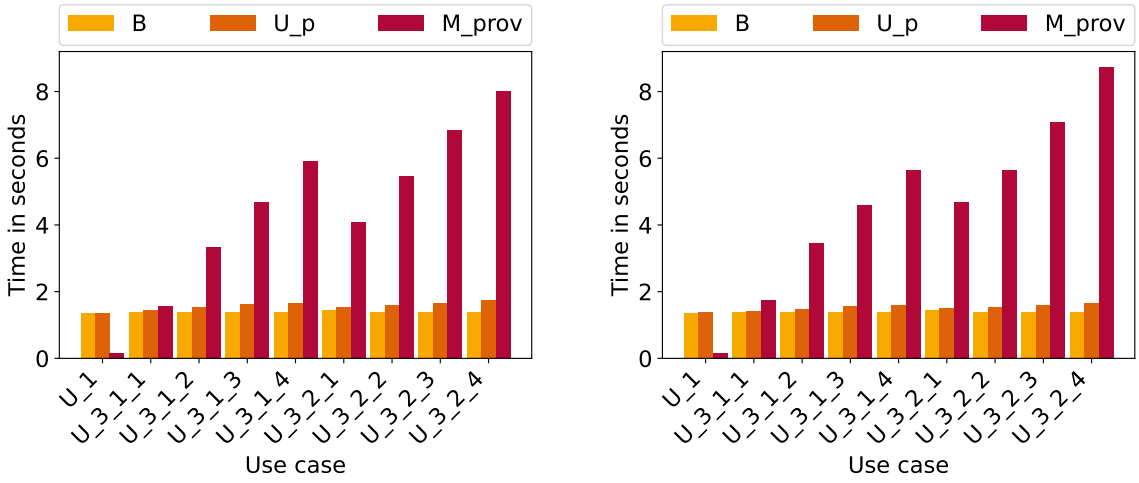
Although not knowing the exact reason, we have arguments letting us assume that the findings in Figure 34 are more reliable. Firstly, the numbers for the *training* in Figure 33 confirm the findings in Figure 34. Secondly, the experiment that led to the numbers in Figure 34 was more extensive and had exclusive access to the machine and the GPU. For the experiment that led to the numbers in Table 1 we only had exclusive access to the GPU but not the whole machine. This might have caused some variation in the data loading, which is a nonnegligible amount of time, given that we did not optimize the data loading process.

8.3.5. Comparing Approaches

In Figure 35 and Figure 36, we show the TTR for all approaches across different scenarios. As we have already discussed in Section 8.3.3, for $\mathcal{M}_{Prov}$ we only simulated the model training. When analyzing the data we have to consider that the TTR for $\mathcal{M}_{Prov}$ are actually higher.

Using $\mathcal{B}$, the TTR is constant across use cases, datasets, and model relations. For $\mathcal{U}'_P$ and $\mathcal{M}_{Prov}$ the TTR forms a staircase pattern which, is most evident for $\mathcal{M}_{Prov}$ – especially having in mind that the actual training times are longer.

In none of the cases do $\mathcal{U}'_P$ or $\mathcal{M}_{Prov}$ outperform $\mathcal{B}$. However, it is interesting that the TTR for $\mathcal{U}'_P$ are 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 35(a) we would have to save a MobileNetV2 that transitively refers to 18 base models; for Figure 35(b) this number even rises to 25.



(a) model versions

(b) fine-tuned versions

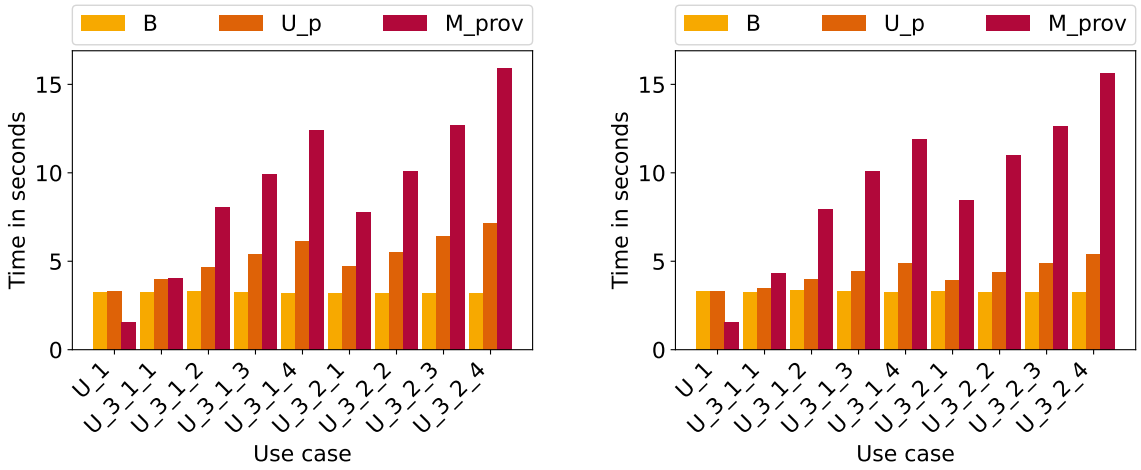
Figure 35: Comparison of the median time-to-recover (TTR) for the MobileNetV2 across approaches. All models in U_3 were trained on the Custom-Coco-Outdoor-512 dataset.

For $\mathcal{M}_{Prov}$, the TTR is long, caused by the model training. Although we could probably reduce it by optimizing, for example, the data loading, it is unlikely that $\mathcal{M}_{Prov}$ outperforms $\mathcal{B}$ or $\mathcal{U}'_P$ in any realistic scenario. How much longer is the TTR for $\mathcal{M}_{Prov}$, depends on many factors including the model architecture, the size of the dataset, and the available hardware.

For U_1, all approaches use $\mathcal{B}$'s logic and should lead to a similar TTR. It is surprising to see that $\mathcal{M}_{Prov}$'s TTR in U_1 is noticeably lower than for the other approaches.

To extensively evaluate $\mathcal{M}_{Prov}$ in different scenarios, as already mentioned above, we only simulated the training process. With simulated training, we cannot expect the recovered model to be equal to the one we saved. To avoid errors, we deactivated some automatic checks for $\mathcal{M}_{Prov}$ that, for example, include a check of the current environment.

From Section 8.3.1, we know that checking the environment can take more than a second and, therefore, the turned-off checks are the reason for $\mathcal{M}_{Prov}$'s lower TTR in U_1.



(a) model versions

(b) fine-tuned versions

Figure 36: 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.

8.4. Time to Save

In this section, we analyze the amount of time it takes to save a given model for all approaches. With TTS we refer to the time that is needed to save a model in use case U_1, U_2, or U_3.

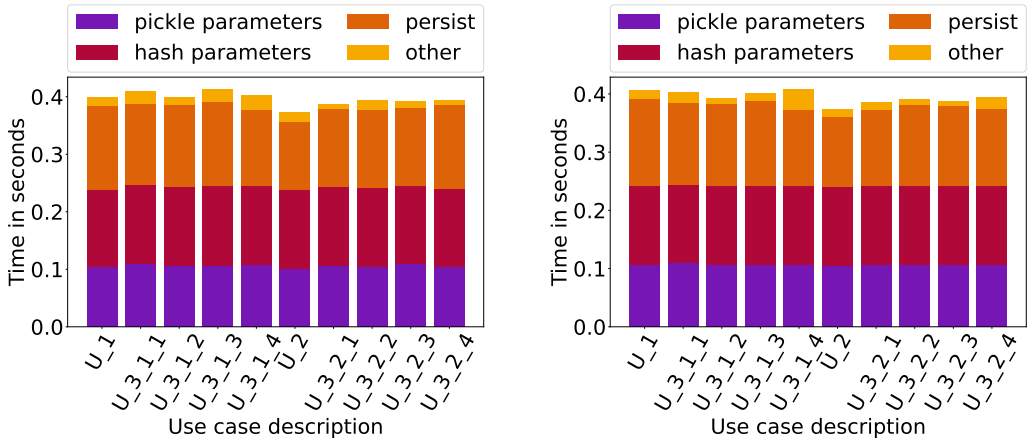
We identify the models throughout the evaluation by the use case they were saved in. For example, if we refer to the TTS of (the model) U_2, we refer to the time that we needed in U_2 to save the model.

8.4.1. Baseline

In the following figures, we present the TTS for models in different scenarios using $\mathcal{B}$ divided into four categories. *Pickle parameters*, *hash parameters*, and *persist* are coherent operations doing what their names suggest. The category *other* comprises all operations that are part of the saving process but are not of importance or do not belong to a specific category.

In Figure 37(a), we show the TTS for ResNet-50 version with all U_3 models being trained on CF-512. The times to hash the model parameters are close to constant, the times to pickle the parameters vary slightly, while the times to persist the model vary the most of the three categories of interest. Comparing Figure 37(a) with Figure 37(b) we find the same trends for the same model architecture but different datasets and model relations. This also includes the tendency of slightly higher TTS for U_1 and the first iteration of U_3.

For a given model architecture $\mathcal{B}$ extracts, processes, and saves an equivalent amount of data for every model. In an ideal world we would expect the TTS to be the same across



(a) model versions, U_3 models trained on the Coco-food-512 dataset

(b) fine-tuned versions, U_3 models trained on the Coco-outdoor-512 dataset

Figure 37: Time-to-save (TTS) for ResNet-50 using $\mathcal{B}$ divided into four categories: *pickle parameters*, *hash parameters*, *persist*, and *other* (comprising all operations that are part of the saving process but are not of interest).

all use cases, datasets, and model relations. The reason for the varying TTS is that it is dependent on external factors, which is also the case for the persist time. (We already discussed the effect of external factors extensively in previous sections.)

Figure 38 indicates a strong dependency of the TTS on the model architecture and thus the number of parameters. The total TTS but also the times for the different categories steadily increases with larger model architectures.

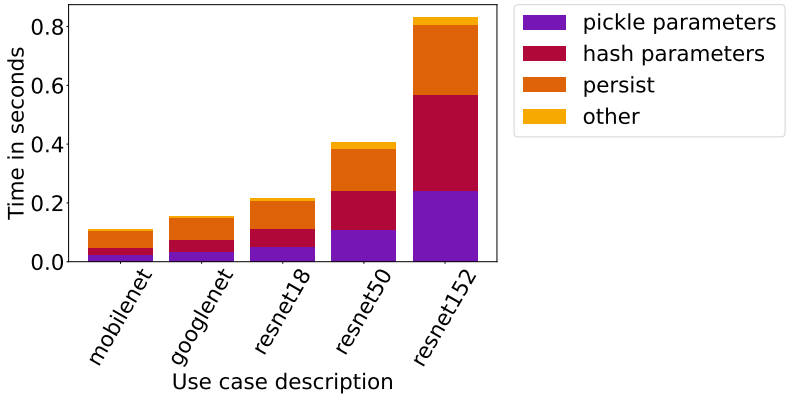


Figure 38: Time-to-save (TTS) per model architecture using $\mathcal{B}$.

In Section 8.2.1 we have seen that the main factor for saving a model using $\mathcal{B}$ is the number of model parameters. $\mathcal{B}$ does not save the dataset, and the number of parameters is independent of the model relation. Therefore, it meets our expectation to see that the TTS is independent of the dataset and the model relation but strongly correlates with the model architecture and the number of parameters. It is also expected that the single

categories depend on the model architecture since an increased number of parameters results in more parameters to pickle, hash, and persist.

Taken together, the results suggest that **the time-to-save (TTS) increases with the number of parameters, varies slightly across use cases, but shows similar trends across datasets and model relations.**

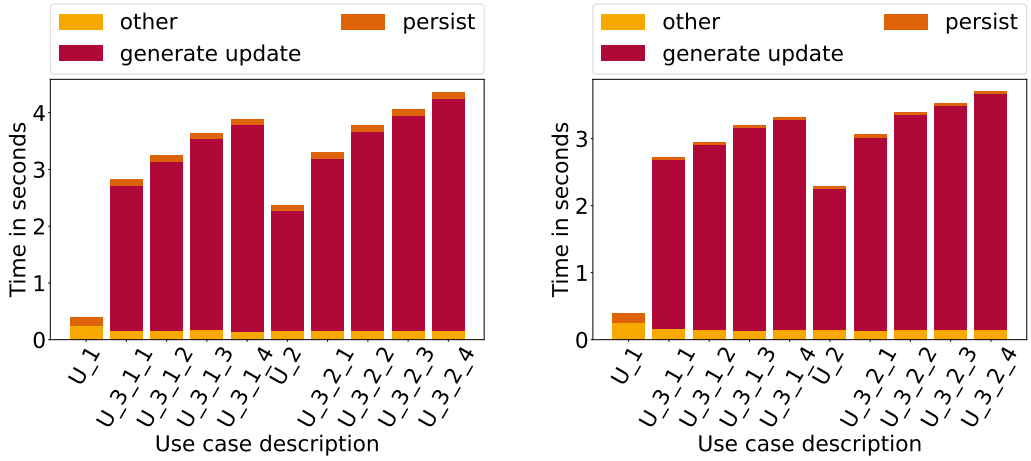
8.4.2. Parameter Update Approach

$\mathcal{U}_P$ and $\mathcal{U}'_P$ differ in how they compute the parameter update which influences the TTS. Unlike our approach in previous sections, we do not implicitly refer to $\mathcal{U}'_P$ when referring to $\mathcal{U}_P$.

In the following figures, we illustrate the TTS for $\mathcal{U}_P$ across three categories: *persist*, *generate update*, and *other*. *Generate update* describes the time to generate the parameter update based on the base model; *persist* and *other* have the same meaning as in Section 8.4.1.

The data in Figure 39(a) suggest that for the TTS of U_2 and U_3 generating the parameter update is the most time-consuming operation; it forms a staircase pattern similar to the one we have seen for the time-to-recover in Section 8.3.2.

Comparing Figure 39(a) with Figure 39(b), we notice slightly lower TTSs for fine-tuned model versions, whilst again forming a similar staircase pattern to the TTS for model versions.



(a) model versions

(b) fine-tuned versions

Figure 39: Time-to-save (TTS) for the ResNet-50 using $\mathcal{U}_P$ divided into three categories: *persist*, *generate update*, and *other* (comprising all operations that are part of the saving process but are not of interest)

The data in Figure 40 indicate an increased TTS for model architectures with a higher number of parameters. The only exception is – again – the GoogLeNet; its *generate update* time is disproportionally high.

Given the observations relating to time-to-recover already discussed in Section 8.3.2, all our findings are in line with expectations. To save a model M (with $B \rightarrow M$), $\mathcal{U}_P$

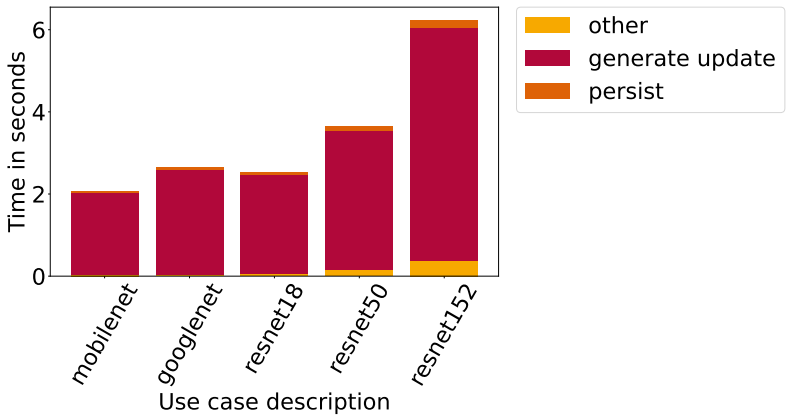


Figure 40: Time-to-save (TTS) per model architecture using $\mathcal{U}_P$.

has to compute the parameter update between B and M. Therefore, $\mathcal{U}_P$ recovers B's parameters, which implies that saving a model M takes at least as long as it takes to recover its base model B. This dependency explains the staircase pattern, the slightly lower times for fine-tuned model versions, and the higher time-to-save for the GoogLeNet.

Together the results suggest that **the time-to-save (TTS) depends on the model architecture, the model relation, and on the time-to-recover (TTR) of the base model which depends on the use case.**

8.4.3. Improved Parameter Update Approach

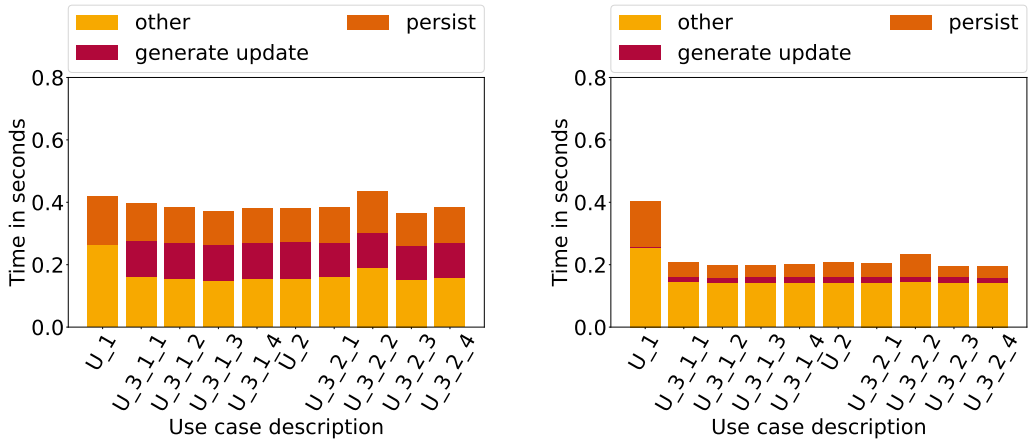
$\mathcal{U}_P$ generates the parameter update based on a recovered base model; $\mathcal{U}'_P$ uses the parameter hash info saved in the *ModelInfo* entity. The following figures show the TTS for $\mathcal{U}'_P$ with the categories being the same as in Section 8.4.2.

The numbers in Figure 41(a) and Figure 41(b) indicate that for $\mathcal{U}'_P$ the *generate update* and consequently the total TTS no longer depends on the use case. This is due to the fact that accessing the parameter hash information does not require a recursive recovery process.

Comparing Figure 41(a) with Figure 41(b), the results show that for $\mathcal{U}'_P$ the TTS equally depends on the number of parameters in the parameter update and thus on the model relation and, as we see in Figure 42, also on the model architecture.

As described above, the improved logic to generate a parameter update no longer includes recovering the base model. Therefore, no model has to be initialized during the saving process, which eliminates the anomaly for the GoogLeNet architecture that we have observed, for example, in Figure 40.

In summary, **the time-to-save (TTS) for $\mathcal{U}'_P$ still depends on the model architecture and model relation, but does not depend on the time-to-recover of the base model anymore.**



(a) model versions

(b) fine-tuned model versions

Figure 41: Time-to-save (TTS) for the ResNet-50 using $\mathcal{U}'_P$ divided into three categories: *persist*, *generate update*, and *other* (comprising all operations that are part of the saving process but are not of interest).

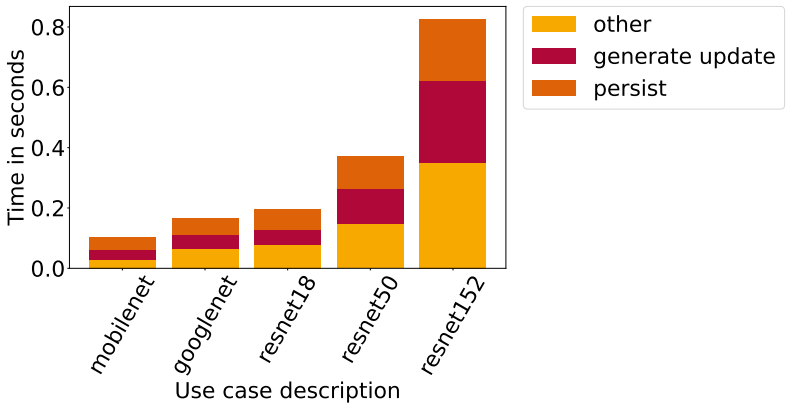
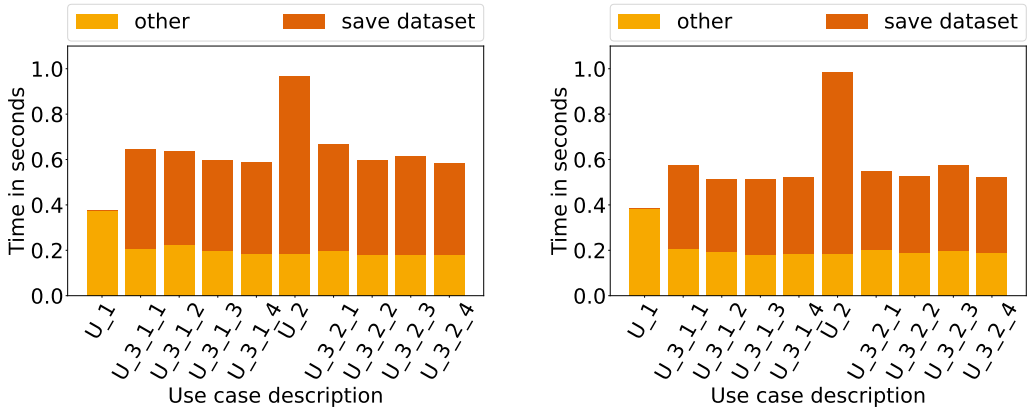


Figure 42: Time-to-save (TTS) per model architecture using $\mathcal{U}'_P$.

8.4.4. Provenance Approach

In the following, we present the TTS for $\mathcal{M}_{Prov}$ divided into two categories. *Save dataset* describes the time to save the dataset which is part of the provenance data; *other* comprises all other operations.

The numbers in Figure 43(a) and Figure 43(b) suggest varying TTSs that are dominated by the time to save the dataset across all use cases. In both figures, we used the smaller dataset $mINet_{val}$ for U_2; for U_3 we used CF-512 in Figure 43(a) and the slightly smaller CO-512 in Figure 43(b).



(a) Coco-food-512 dataset

(b) Coco-outdoor-512 dataset.

Figure 43: Time-to-save (TTS) for ResNet-50 versions using $\mathcal{M}_{Prov}$ divided between the time to save the dataset (*save dataset*) and all *other* operations that are part of the saving process.

Because the dataset is the most extensive subset of the provenance data, we would expect to see that TTS decreases whenever we use smaller datasets.

Given the same training dataset, in an ideal world both the *save dataset* and *other* time would be the same for all iterations of U_3; in practice, both times are dependent on external factors such as the connection to the database or file-system – we have already seen and discussed a similar effect in Section 8.4.1.

The data in Figure 44 suggest that the model architecture influences the *other* and by this the total TTS. This effect is also expected, since *other* includes the time to generate the hash values for the model parameters.

In summary, we see that **the time-to-save (TTS) varies across the use cases and depends on the model architecture, but mainly on the training dataset size.**

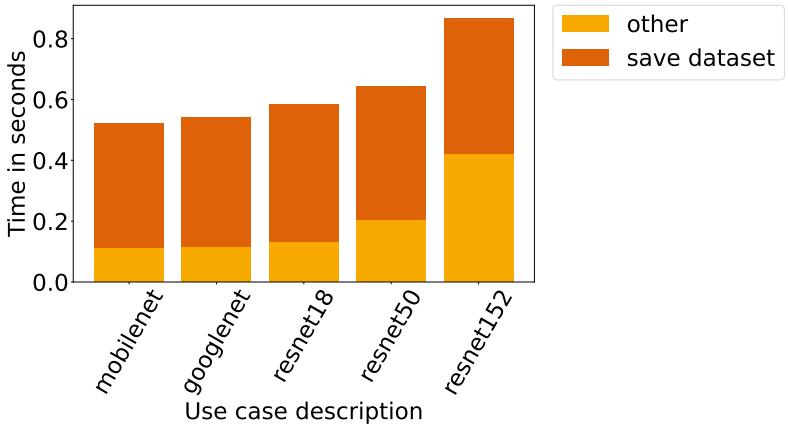


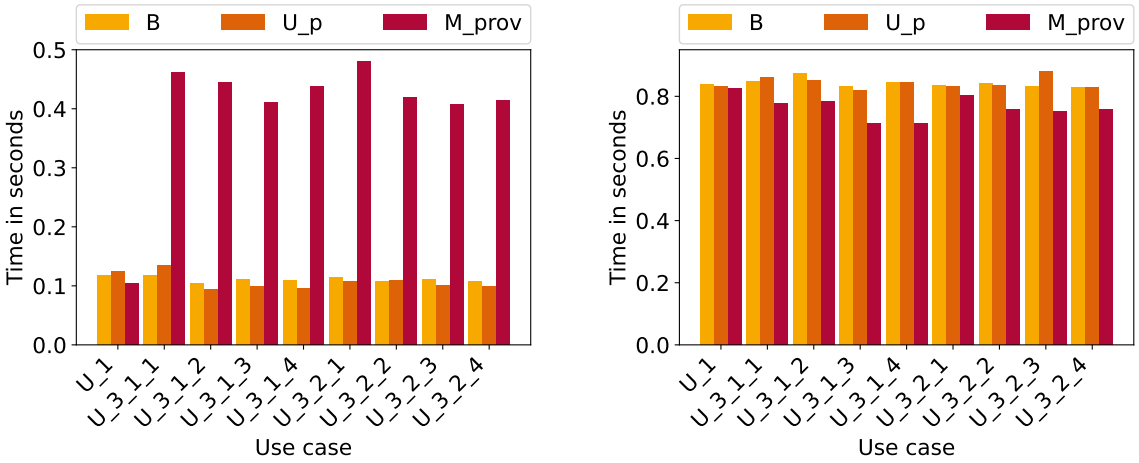
Figure 44: Time-to-save (TTS) for different model architectures.

8.4.5. Comparing Approaches

In Figure 45 and Figure 46, we compare the TTS using the different approaches in the same settings. $\mathcal{U}'_P$ outperforms $\mathcal{U}_P$ in all cases, therefore, we exclude $\mathcal{U}_P$ from the comparison to improve the readability.

Model Versions For $\mathcal{B}$ and $\mathcal{U}'_P$, the TTS depends on the model architecture. Comparing Figure 45(a) with Figure 45(b) shows that for model versions $\mathcal{U}'_P$ does not improve $\mathcal{B}$'s TTS because the parameter update and a complete parameter snapshot are equal in size.

For $\mathcal{M}_{Prov}$ we know that the TTS depends on the model architecture, but mainly on the training dataset used . The times in Figure 45(b) suggest that $\mathcal{M}_{Prov}$ has the potential to outperform $\mathcal{B}$ and $\mathcal{U}'_P$; especially when $\mathcal{M}_{Prov}$ uses less storage than $\mathcal{B}$ and $\mathcal{U}'_P$. Figure 45(a) shows that there are also scenarios where $\mathcal{M}_{Prov}$ is drastically outperformed by $\mathcal{B}$ and $\mathcal{U}'_P$; this is mainly the case when $\mathcal{M}_{Prov}$'s storage consumption is relatively high.



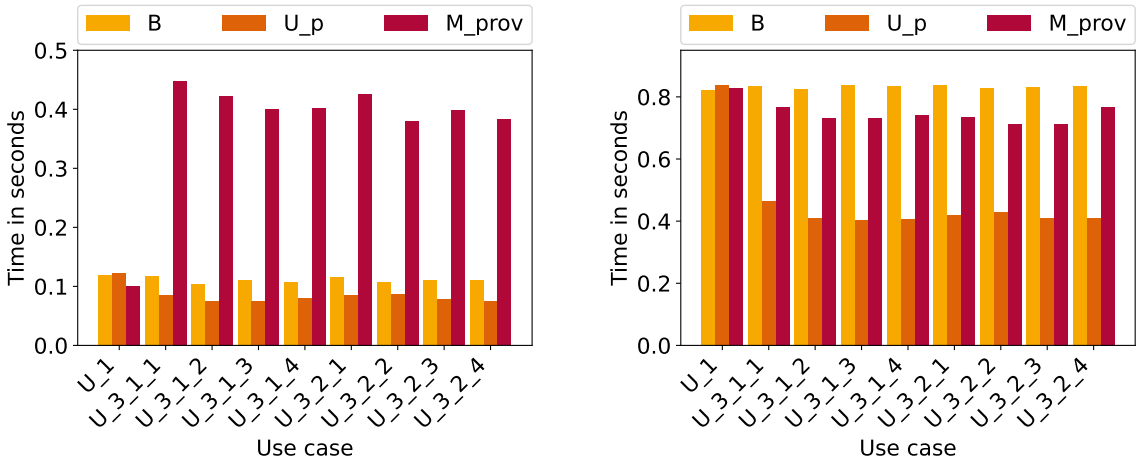
(a) MobileNetV2

(b) ResNet-152

Figure 45: Comparison of the median TTS for model version across approaches. All models in U_3 were trained on the Custom-Coco-Outdoor-512 dataset.

Fine-tuned Versions For $\mathcal{U}'_P$ we know that the TTS reduces for fine-tuned versions depending on how many parameters we retrain. Thus, $\mathcal{U}'_P$ outperforms $\mathcal{B}$ for fine-tuned model versions depending on the model architecture and the amount of retrained parameters. We can see this effect for the MobileNetV2 in Figure 46(a) and for the ResNet-152 in Figure 46(b).

$\mathcal{B}$'s, as well as $\mathcal{M}_{Prov}$'s TTS does not depend on the model relation. Since $\mathcal{M}_{Prov}$'s performance does not change but $\mathcal{U}'_P$'s improves, it becomes more unlikely that $\mathcal{M}_{Prov}$ will have a shorter TTS than $\mathcal{U}'_P$.



(a) MobileNetV2

(b) ResNet-152

Figure 46: Comparison of the median time-to-save (TTS) for fine-tuned versions across approaches. All models in U_3 were trained on the Custom-Coco-Outdoor-512 dataset.

8.5. Discussion

In this section, we utilize the knowledge we gained throughout the evaluation to discuss the most suitable approach for a given scenario.

8.5.1. Approach Applicability

Before we can decide what approach to use for a given scenario, we have to make sure to only consider applicable approaches. $\mathcal{B}$ and $\mathcal{U}'_P$ are approaches with low complexity and work in almost every scenario. $\mathcal{M}_{Prov}$ only works as intended in scenarios that enable us to reproduce model training. Whether this is the case, depends, as discussed in Section 3.4, mainly on the environment and the operations used implementing the model layers. If it is not possible to reproduce the training, $\mathcal{M}_{Prov}$ does not entirely stop working, but it is not guaranteed that we can recover models without loss of precision. Moreover, when using $\mathcal{M}_{Prov}$, we should always be aware that it is most likely that the time-to-recover (TTR) is significantly higher than for $\mathcal{B}$ and $\mathcal{U}'_P$.

8.5.2. Choice of Approach

Optimized Time-to-recover In each of the scenarios presented, we can find one approach ($\mathcal{U}'_P$ or $\mathcal{M}_{Prov}$) that outperforms our baseline $\mathcal{B}$ in terms of storage consumption and time-to-save (TTS). The drawback of these approaches is an increased TTR that is marginally higher for $\mathcal{U}'_P$ and drastically for $\mathcal{M}_{Prov}$. Therefore, if, for a given setting, the TTR has the highest priority, $\mathcal{B}$ is the preferred choice.

Optimized Storage Consumption and Time-to-save For the settings we focus on in Section 4.3, we assume to recover models rarely, but save derived models frequently. In this situation, we can neglect the TTR to a certain extent and concentrate on the TTS and storage consumption.

Which of both approaches ($\mathcal{U}'_P$ or $\mathcal{M}_{Prov}$) is the best choice, highly depends on the scenario we operate in. If the models we save are used in a domain where datasets are large, $\mathcal{U}'_P$ is most likely the preferred choice. If we have a domain with complex models but small datasets or frequent model training on small datasets, $\mathcal{M}_{Prov}$ is the best approach for storage consumption and TTS. Especially for $\mathcal{M}_{Prov}$ it is crucial to be aware of the long TTR due to model training that is dependent on the domain and model architecture.

When deciding on an approach, we should also consider the overall environment. To give one example: maybe we save the collected data for model retraining in a central database anyway. This would make $\mathcal{M}_{Prov}$ very attractive in terms of storage consumption because the training data (by far the largest subset of the provenance) is saved regardless of the approach we choose.

Storage-Recreations Tradeoff When choosing an approach for a specific scenario, we always have a storage-recreation tradeoff problem. Either, we chose $\mathcal{B}$ that does not optimize the storage consumption, but the TTR. Or, we choose $\mathcal{U}'_P$ or $\mathcal{M}_{Prov}$ to reduce the storage consumption and accept their effect on the TTR that is mainly influenced by the time required to recreate the models. For a given scenario, we always have to answer how much TTR (and resources) we want to invest to save storage and (if we have to transmit the model) bandwidth.

Let us assume the following scenario: we have a model M and saving it, requires 300 MB using $\mathcal{B}$. For every fine-tuned version, $\mathcal{U}'_P$ needs 100 MB to save the update. The training/provenance data is 5 MB. Every training to update the model takes about 10 hours. Let us also assume we create 100 models where every new model is a fine-tuned version of its previous model.

Of all approaches, $\mathcal{B}$ would have the highest storage consumption, at 30 GB, but we could recover every model independently, resulting in a low TTR. $\mathcal{M}_{Prov}$ has, of all the approaches, the minimal storage consumption of 0.8 GB, but its TTR would be, on average, more than 20 days. In between $\mathcal{B}$ and $\mathcal{M}_{Prov}$, we have $\mathcal{U}'_P$. It Consumes 10.2 GB and has a longer TTR than $\mathcal{B}$, but still significantly shorter than $\mathcal{M}_{Prov}$.

When choosing the right approach for this scenario, we have to ask: How badly do we want to reduce storage consumption (or consumed bandwidth), and how much computational resources are we willing to invest in recovering models.

Derakhshan et al. [8] and Vartak et al. [54] formalize similar tradeoffs in the context of ML experiments. In future work, we could adapt their formalization to our scenario.

$\mathcal{M}_{Prov}$'s performance is particularly dependent on the domain we operate in because it usually roughly determines the 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 $\mathcal{M}_{Prov}$ would be short training times, small datasets, and large models.

Adaptive Approach So far, we have only discussed the most suitable approach for a given situation, but have not addressed that we could also combine approaches to form an adaptive approach. A possible direction could be, for example, to use a heuristic that decides individually for the best approach ($\mathcal{B}$, $\mathcal{U}'_P$, or $\mathcal{M}_{Prov}$) for every model. A simple heuristic could be based on the fact that $\mathcal{B}$ and $\mathcal{U}'_P$ mainly depend on the model parameters, whereas $\mathcal{M}_{Prov}$ primarily depends on the dataset. A more complex one could be based on a formalized tradeoff combined with some given parameters like maximum storage consumption or TTR.

To give a concrete example, we show a combined approach in Figure 47 that minimizes the storage consumption under the constraint of a maximum TTR. Such an approach could use $\mathcal{U}'_P$ or $\mathcal{M}_{Prov}$ as the default approach, and whenever the TTR passes a certain threshold (as it is the case for M_3), it saves the next model using $\mathcal{B}$ to reduce the TTR for the following models.

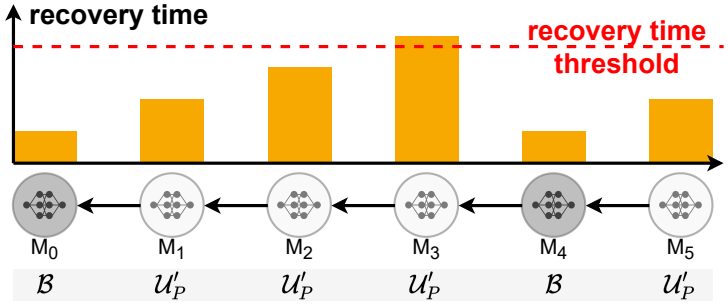


Figure 47: Combining $\mathcal{B}$ and $\mathcal{U}'_P$ to save a chain of derived models. In grey the used approach, in orange the resulting TTR.

9. Conclusion

In this section we conclude our work. We first summarize our contributions and results in Section 9.1; and then, give an outlook on future work in Section 9.2.

9.1. Summary

The overall research question for this thesis was if, and by how much, we can outperform a baseline approach 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).

Throughout the thesis, we have shown that $\mathcal{M}_{Prov}$ outperforms the baseline $\mathcal{B}$ by up to 70% and $\mathcal{U}'_P$ by up to 95.6% in terms of storage consumption while both having a similar or slightly shorter TTS but an increased TTR. We have also seen that the degree of performance improvement highly dependent on factors such as model architecture, model relation, and size of training dataset. To come to this conclusion, we developed and extensively evaluated three approaches: the baseline approach $\mathcal{B}$, a parameter update approach $\mathcal{U}'_P$, and a provenance approach $\mathcal{M}_{Prov}$.

The baseline approach $\mathcal{B}$ saves a complete snapshot of a model with its storage consumption, TTS, and TTR mainly depending on the number of model parameters. The first advanced approach we developed is $\mathcal{U}'_P$. This only saves the part of a model that has changed compared to the base model; in our cases, this is only the model parameters. By doing this, $\mathcal{U}'_P$ outperforms $\mathcal{B}$ in all our experiments (for model versions and fine-tuned model versions) in both storage consumption and TTS. How much it outperforms $\mathcal{B}$ mainly depends on how many of the model parameters stay constant compared to the base model. Both $\mathcal{B}$ and $\mathcal{U}'_P$ save model parameters. The more parameters change compared to the base model, the closer $\mathcal{U}'_P$'s performance is to $\mathcal{B}$'s for both storage consumption and TTS.

With $\mathcal{M}_{Prov}$, we developed an approach that does not save model parameters but instead the provenance of the model, which has the training data as its largest subset. Being independent of the model parameters, $\mathcal{M}_{Prov}$ can outperform $\mathcal{B}$ and $\mathcal{U}'_P$ in terms of storage consumption and TTS in those cases where the provenance data is small compared to the model parameters. While having the potential to outperform $\mathcal{B}$ in storage consumption and TTS, both $\mathcal{U}'_P$ and $\mathcal{M}_{Prov}$ come with a recursively increasing TTR. For $\mathcal{U}'_P$, the TTS increases moderately; for $\mathcal{M}_{Prov}$, it increases drastically depending on the time to reproduce the model training.

In addition to answering the research question by extensively evaluating the different approaches, we made additional contributions while developing both the baseline and advanced approaches. We discussed how, and under which conditions, we can reproduce model training and developed a probing tool to test the reproducibility of model architectures across different machines. We integrated all our approaches into a Python framework to make them easily accessible and usable. We developed a data schema that is generic enough to be easily extended for further approaches.

9.2. Future Work

Extended Evaluation We extensively evaluated and analyzed how all approaches perform in different scenarios and configurations. However, we limited ourselves to the domain of computer vision, a set of five model architectures, two model relations, and two datasets. Deep learning is a technique that we can apply to almost every domain and data. Thus, it would be interesting to see how our approaches perform in other fields implying other data and model architectures.

Hardware and Environment In our evaluation, we limited ourselves to a specific hardware setup. Other choices of hardware and network connectivity would probably influence TTS and TTR. It would be interesting to see how our approaches react to limited computational power or a constrained network connection.

Reduce Time to Train Especially for reproducible model training, the hardware and the model implementations determine the execution time. In this context, it would be beneficial to know if there are specific layers that drastically slow down the training time. If we find slow layers, finding ways to improve or replace them without decreasing the models' predictive performance would be desirable.

Compatible Environments To guarantee reproducible training, we require the same environment, but it would be interesting to see if there are compatible setups.

Especially for $\mathcal{M}_{Prov}$ it would be beneficial to find a pair of setups that is compatible in terms of reproducibility but differs in terms of computational power. Having found such a pair, we could use the setup with low computational power on distributed devices and the more powerful setup on a central server to recover models in a fraction of the time it would take on one of the distributed devices.

Further Approaches In our thesis, we developed three approaches and tested them naively, assuming that only one would be used. In practice, we would assume that real-world performance improvement would be maximized by using them adaptively in a hybrid implementation as described in Section 8.5.2.

Moreover, future work could explore what implications it would have to relax requirements such as the exact reproducibility of models.

A. Merkle Tree Examples

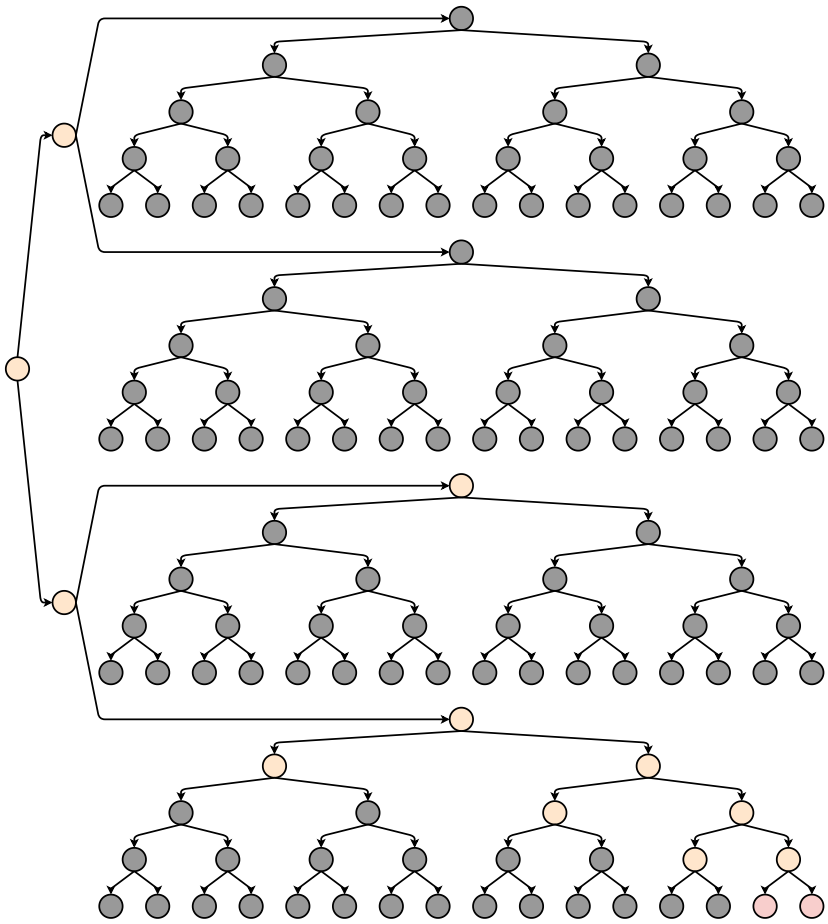


Figure 48: Abstract representation of a Merkle tree containing the hashes for a 64-layer model. The last two layers (red) have changed; to find this out we have to compare 13 hashes in total.

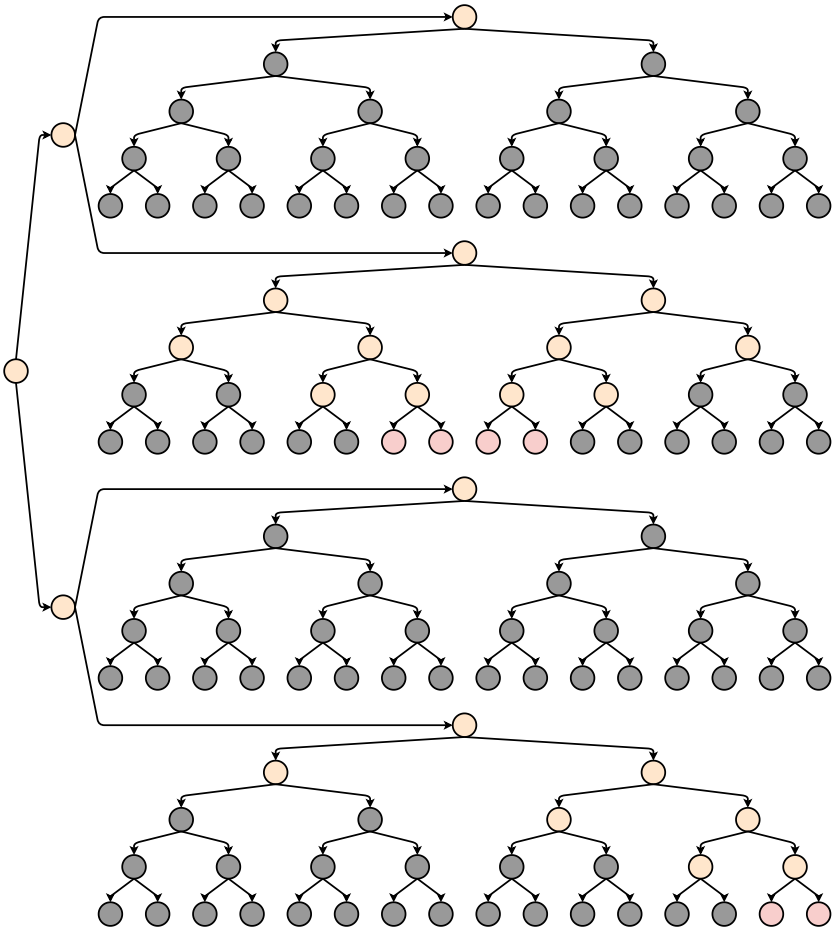


Figure 49: Abstract representation of a Merkle tree containing the hashes for a 64-layer model. The last two, and four middle layers (red) have changed; to find this out we have to compare 29 hashes in total.

B. Class Diagrams and Data Schema

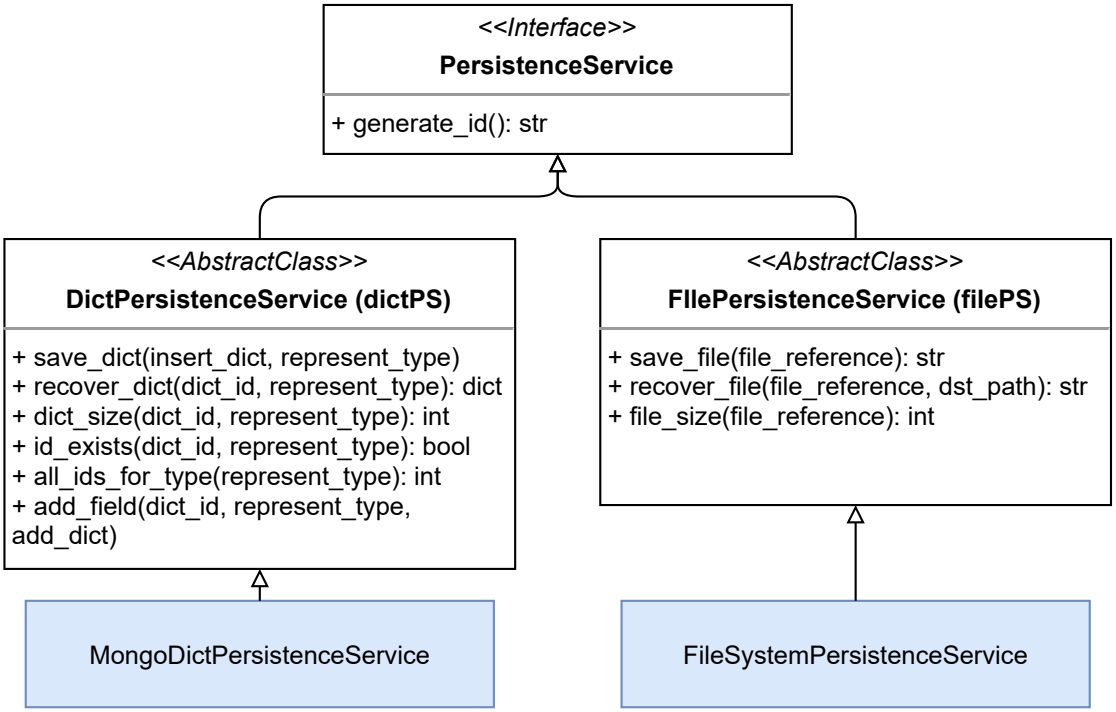


Figure 50: *SaveInfo* classes and the *AbstractSaveService* defining the interface for all developed approaches.

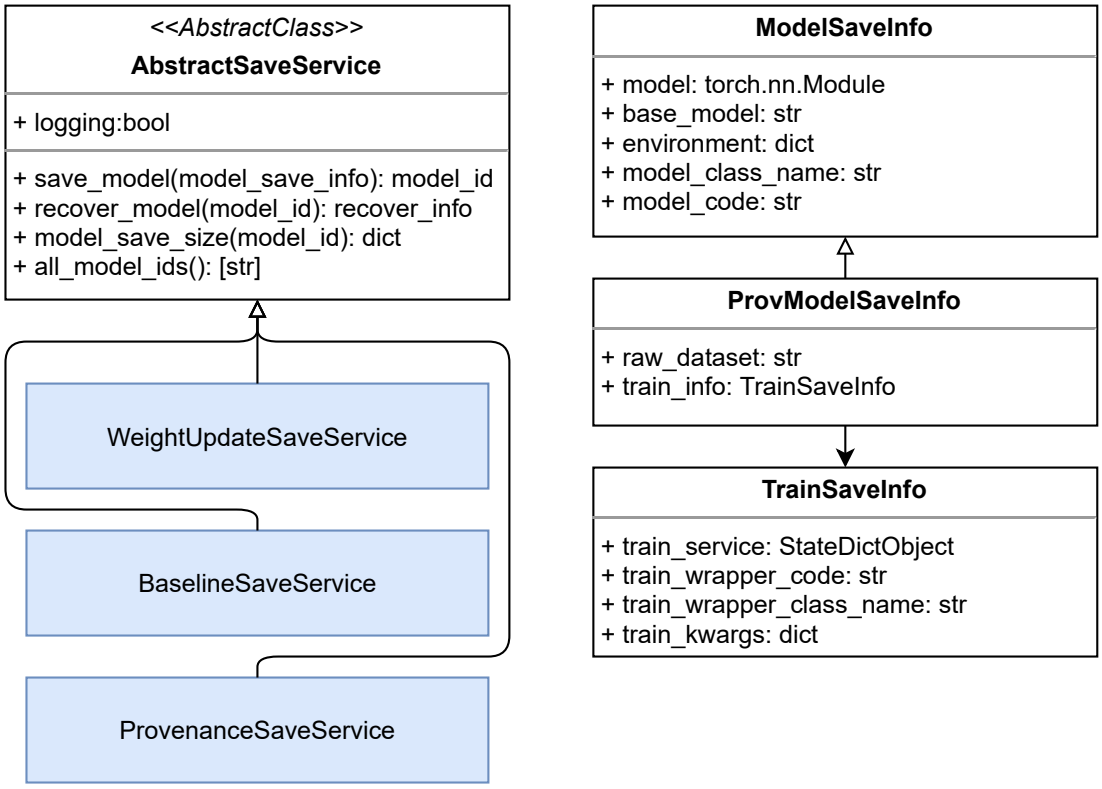


Figure 51: Class diagram for persistence related classes.

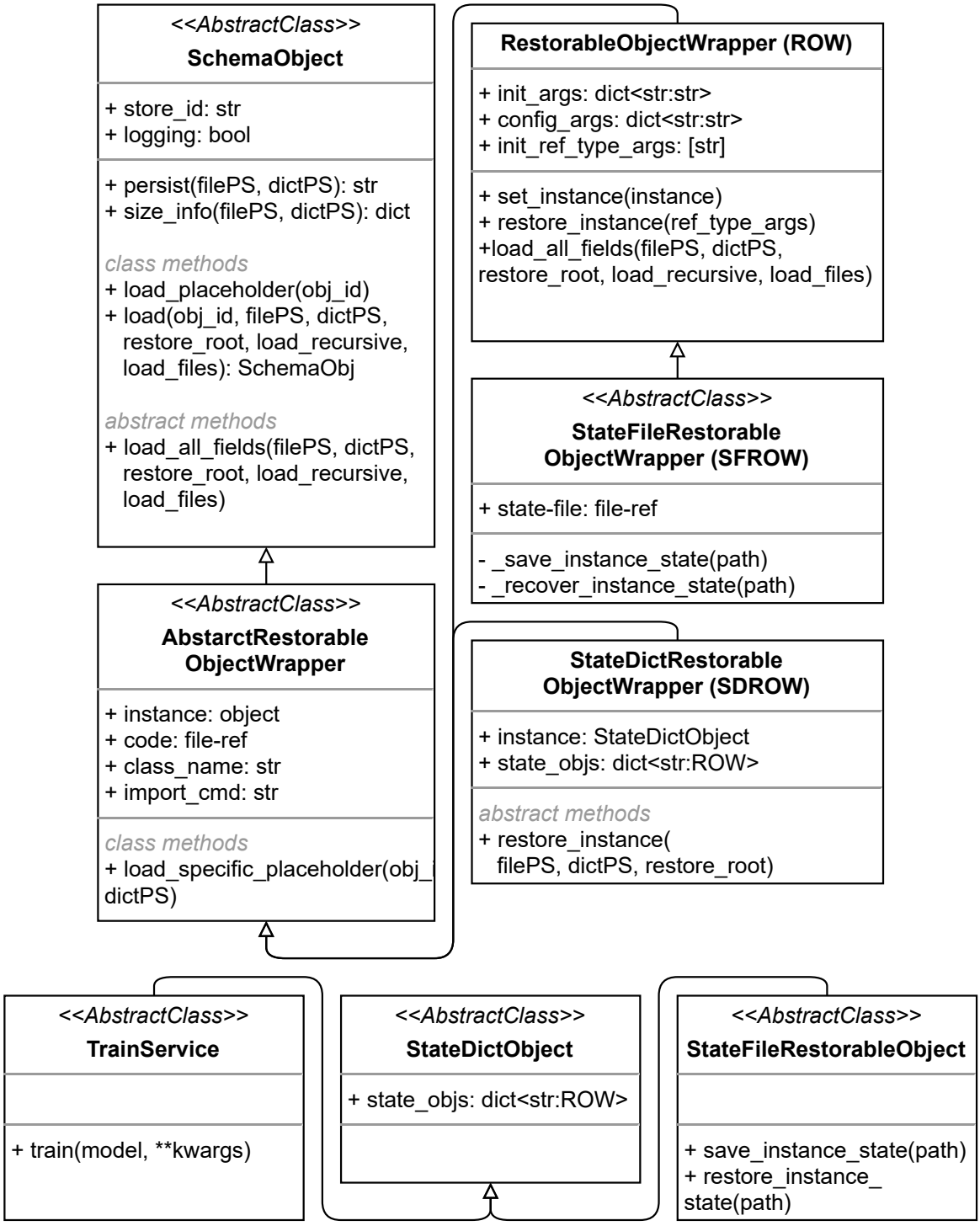


Figure 52: Class diagram of all *SchemaObjects* to track the provenance data.

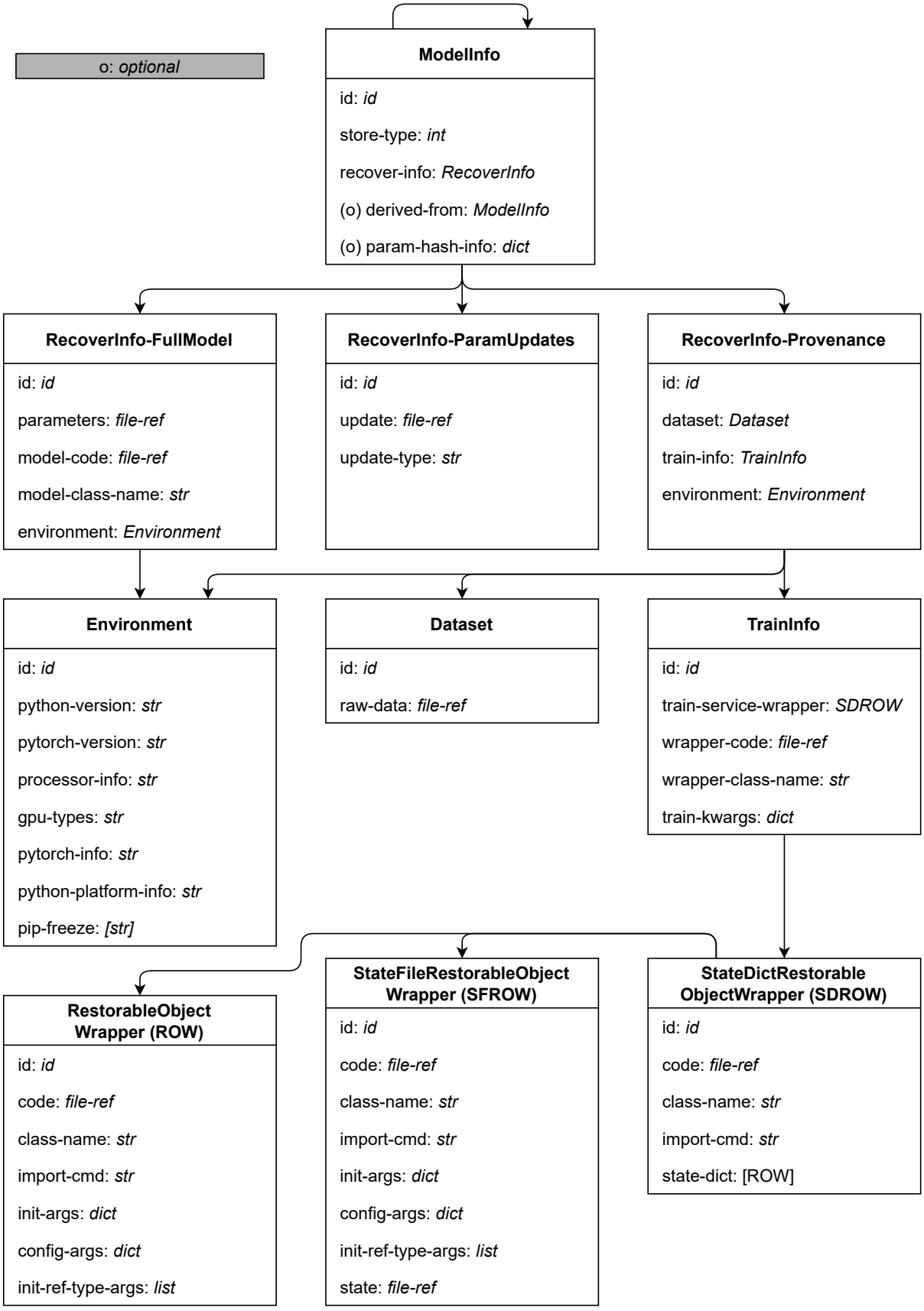


Figure 53: The full database schema.

References

[1] Md Zahangir Alom, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S. Awwal, and Vijayan K. Asari. The history began from alexnet: A comprehensive survey on deep learning approaches, 2018.

[2] Lorena A Barba. Terminologies for reproducible research. *arXiv preprint arXiv:1802.03311*, 2018.

[3] Souvik Bhattacherjee, Amol Deshpande, and Alan Sussman. Pstore: an efficient storage framework for managing scientific data. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2014.

[4] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications, 2017.

[5] Coco common objects in context. `https://cocodataset.org/`. accessed: 2021-01-12.

[6] Nvidia cuda toolkit. `https://developer.nvidia.com/cuda-toolkit`. accessed 2021-07-14.

[7] Nvidia cudnn. `https://developer.nvidia.com/cudnn`. accessed 2021-03-05.

[8] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. Optimizing machine learning workloads in collaborative environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1701–1716, 2020.

[9] Association for Computing Machinery. Artifact review and badging. `https://www.acm.org/publications/policies/artifact-review-and-badging-current`. Accessed: 2020-10-31.

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

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

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

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

[14] Google. Machine learning glossary. `https://developers.google.com/machine-learning/glossary`. accessed: 2021-04-30.

[15] The Gradient. The state of machine learning frameworks in 2019. `https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/`. accessed: 2021-05-06.

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

[17] Jeff Hale. Is pytorch catching tensorflow? `https://towardsdatascience.com/is-pytorch-catching-tensorflow-ca88f9128304`. accessed: 2021-05-06.

[18] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[19] Matthew Hartley and Tjelvar SG Olsson. dtoolai: Reproducibility for deep learning. *Pat-*

terns, 1(5):100073, 2020.

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

[21] IEEE. Ieee-754, standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–58, 01 2008.

[22] Vinu Joseph, Ganesh L. Gopalakrishnan, Saurav Muralidharan, Michael Garland, and Animesh Garg. A programmable approach to neural network compression. *IEEE Micro*, 40(5):17–25, Sep 2020.

[23] Introducing json. `https://www.json.org/json-en.html`. accessed: 2021-03-10.

[24] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516, 2020.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

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

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

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

[30] mongodb. `https://www.mongodb.com/`. accessed: 2021-03-10.

[31] Luc Moreau, Paul Groth, Simon Miles, Javier Vazquez-Salceda, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, et al. The provenance of electronic data. *Communications of the ACM*, 51(4):52–58, 2008.

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

[33] NVIDIA. Cuda toolkit documentation – floating point and ieee 754 compliance for nvidia gpus. `https://docs.nvidia.com/cuda/floating-point/index.html`. accessed: 2021-01-04.

[34] Pickle — python object serialization. `https://docs.python.org/3/library/pickle.html`. accessed 2021-04-06.

[35] Gustavo Correa Publio, Diego Esteves, Agnieszka Ławrynowicz, Panče Panov, Larisa Soldatova, Tommaso Soru, Joaquin Vanschoren, and Hamid Zafar. Ml-schema: Exposing the semantics of machine learning with schemas and ontologies. *arXiv preprint arXiv:1807.05351*, 2018.

[36] Platform — access to underlying platform's identifying data. `https://docs.python.org/3/library/platform.html`. accessed 2021-05-11.

[37] Pytorch. `https://pytorch.org/`. accessed: 2021-04-30.

[38] Pytorch reproducibility. `https://pytorch.org/docs/1.7.1/notes/randomness.html`. accessed 2021-03-05.

[39] Pytorch saving and loading models. `https://pytorch.org/tutorials/beginner/savin`

g_loading_models.html. accessed 2021-05-11.
[40] Pytorch docs serialization. https://pytorch.org/docs/stable/torch.html?highlight=save#torch.save. accessed 2021-05-11.
[41] Duncan Riach. Determinism in deep learning. NVIDIA's GPU Technology Conference, 2019.
[42] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.
[43] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.
[44] Sebastian Schelter, Felix Bießmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. On challenges in machine learning model management. *IEEE Data Eng. Bull.*, 41(4):5–15, 2018.
[45] Sebastian Schelter, Joos-Hendrik Boese, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems workshop at NIPS*, 2017.
[46] Eric R Schendel, Ye Jin, Neil Shah, Jackie Chen, Choong-Seock Chang, Seung-Hoe Ku, Stephane Ethier, Scott Klasky, Robert Latham, Robert Ross, et al. Isobar preconditioner for effective and high-throughput lossless data compression. In *2012 IEEE 28th international conference on data engineering*, pages 138–149. IEEE, 2012.
[47] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28:2503–2511, 2015.
[48] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48, 2019.
[49] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.
[50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
[51] Pytorch torchvision.models. https://pytorch.org/vision/0.8/models.html. accessed: 2021-07-15.
[52] Cs231n convolutional neural networks for visual recognition - transfer learning. https://cs231n.github.io/transfer-learning/. accessed: 2021-07-15.
[53] Jason Tsay, Todd Mummert, Norman Bobroff, Alan Braz, Peter Westerink, and Martin Hirzel. Runway: machine learning model experiment management tool, 2018.
[54] Manasi Vartak, Joana M F. da Trindade, Samuel Madden, and Matei Zaharia. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1285–1300, 2018.
[55] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Modeldb: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages

1–3, 2016.

[56] VertaAI. Modeldb: An open-source system for machine learning model versioning, metadata, and experiment management. `https://github.com/VertaAI/modeldb`. Accessed: 2020-10-31.

[57] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):1–40, 2016.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die digital eingereichte Version ist mit der vorliegenden Arbeit identisch.

Ort, Datum	Unterschrift (Nils Straßenburg)