Short Paper

Efficient Multi-Model Management

Nils Strassenburg[1] Dominic Kupfer[2] Julia Kowal[2] Tilmann Rabl[1]
[1]Hasso Plattner Institute, University of Potsdam [2]TU Berlin
{nils.strassenburg, tilmann.rabl}@hpi.de {dominic.kupfer, julia.kowal}@tu-berlin.de

ABSTRACT

Deep learning models are deployed in an increasing number of industrial domains, such as retail and automotive applications. An instance of a model typically performs one specific task, which is why larger software systems use multiple models in parallel. Given that all models in production software have to be managed, this leads to the problem of managing sets of related models, i.e., multi-model management. Existing approaches perform poorly on this task because they are optimized for saving single large models but not for simultaneously saving a set of related models.

In this paper, we explore the space of multi-model management by presenting three optimized approaches: (1) A baseline approach that saves full model representations and minimizes the amount of saved metadata. (2) An update approach that reduces the storage consumption compared to the baseline by saving parameter updates instead of full models. (3) A provenance approach that saves model provenance data instead of model parameters. We evaluate the approaches for the multi-model management use cases of managing car battery cell models and image classification models. Our results show that the baseline outperforms existing approaches for save and recover times by more than an order of magnitude and that more sophisticated approaches reduce the storage consumption by up to 99%.

1 INTRODUCTION

Today, deep learning (DL) is widely used in research and production software. In both cases, it is necessary to manage the DL models, especially when they fulfill safety-critical tasks [8, 9].

A DL model usually goes through two phases: (1) the *model development phase*, where the initial model is developed and trained, and (2) the *model deployment phase*, where the model is deployed to devices and locally updated. The model development phase starts with researchers exploring the capabilities of new DL model architectures. They evaluate the models' prediction performance on a fixed or standardized dataset using loss- and accuracy-based metrics. The models are often large, require resources-intense training, and are highly optimized to achieve very good performance on the evaluation dataset. Industry adapts the findings to develop initial use-case-specific models on real-life data. However, due to resource constraints on the target devices, the models are often optimized for inference and are smaller than the ones developed in research.

In the model deployment phase, companies deploy their best-performing model to production and continuously update it based on local data. This leads to thousands of frequently updated models sharing the same architecture. Examples are mobile phone applications where individual models are adapted to user requirements, smart devices in health applications where models are adjusted to patients' behavior, or recommendation systems where models are adjusted to usage characteristics.

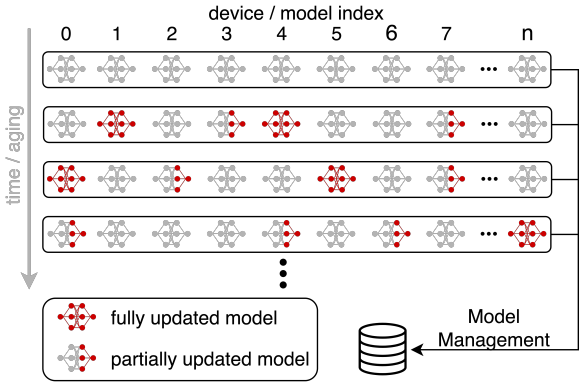


Figure 1: Multi-Model Management

One DL model performs one well-defined task, for example, to predict the behavior of a specific real-world entity. In the case of multiple equivalent entities in one device or location, multiple models with the same architecture run on the same device. One practical example is the modeling of electric car battery systems. Electric car batteries can consist of thousands of individual cells, each possibly being associated with its own DL model that has thousands of parameters and predicts the cell's behavior. DL models are used because they can be evaluated and continuously updated with a comparably low computational cost [4, 14]. It is beneficial to use one model per battery cell instead of one single model for the entire battery because individual models provide a spatial resolution regarding, for instance, temperature evolution, cell aging, or current distribution.

Regardless of the phase and use case, all DL models need to be managed. We distinguish between single-model management, where we save single models at a time, and multi-model management, where we save sets of models. In the model development phase, we run individual experiments and manage the associated models, which requires single-model management. In the model deployment phase, in many real-life use cases, we manage sets of different and frequently updated models sharing the same architecture. Here we apply multi-model management.

Multi-model management is a common and very relevant model management scenario because (1) multi-model management is critical during the deployment phase, which is the phase a model is primarily used in, and (2) multi-model management is resource intense because of the large number of models and the high update frequency.

In this paper, we focus on multi-model management in the deployment phase, as presented in Figure 1. We have n ($n >> 1000$) individual models sharing the same architecture. Over time the model performance decreases, and the models are partially or fully updated on locally collected data. We save every model ever generated for analytical and archival purposes but only recover a selected number of models, for example, after an accident.

Existing model management approaches [6, 11, 13, 15] do not cover multi-model management or perform poorly in this

scenario. They focus on experiment management for the model development phase or save single models with snapshot sizes dominated by millions of parameters.

In this paper, we make the following contributions:

(1) We define and analyze the multi-model management use case to identify optimization opportunities not utilized by related model/experiment management approaches.
(2) Based on our analysis, we develop three approaches that are optimized for multi-model management: the *Baseline* approach that minimizes the overhead of redundantly saved information, the *Update* approach that only saves updated parameters, and the *Provenance* approach that saves model provenance information instead of model parameters.
(3) We evaluate all approaches for the use cases of managing deep learning-based car battery models and image classification in terms of storage consumption, time-to-save, and time-to-recover. We find that our *Baseline* approach outperforms existing baselines significantly in all evaluated metrics and that the *Update* approach and the *Provenance* approach additionally reduce the storage consumption by up to 86% and 99%, respectively but come with an increased time-to-recover. Based on the results, we discuss the tradeoffs of every approach and recommend what approach to choose under what conditions.

The rest of the paper is structured as follows. In Section 2, we provide some background on DL and afterward present relevant related approaches. In Section 3, we present three different approaches optimized for multi-model management. We evaluate the approaches for the example use cases of managing battery cell models and image classification in Section 4 before we conclude our work in Section 5.

2 BACKGROUND

In this section, we will first introduce the foundations of DL that are essential to understand the presented approaches. Afterward, we give an overview of related work.

2.1 Deep Learning

On a very high level, a DL model takes a given input, performs computations, and produces an output. For example an image classification model, takes a picture's pixel matrix as input and predicts the probability for each predefined image class. For the example of modeling car battery cells, we employ electrical DL models that take the current, temperature, and charge information as inputs and predict the voltage response.

The behavior of a DL model is defined by the model architecture and the model parameters. The model architecture represents the computational structure of the model and defines what type of computation is performed. It consists of different layers, each defining a part of the overall architecture. The model parameters consist of weights and biases that parameterize the computation defined by the architecture [3, 11]. To make a model learn a given task, we adjust its parameters based on training data using a variation of a stochastic gradient descent algorithm. The model architecture remains unchanged.

Throughout the lifecycle of a DL model, its performance degrades. In the car battery use case, for example, this is because the battery cell that it represents ages. To prevent the model's performance from degrading, it is common practice to take the current state of the model as a starting point and: (1) retrain the entire model, which adjusts all model parameters, or (2) retrain single layers of the model which adjusts only a subset of the model. We follow previous work's definitions [11] and name the model before adjustment *base model*, a model that was retrained entirely *fully updated model version*, and a model that was retrained only partially *partially updated version*.

2.2 Related Approaches

Looking at related approaches in the domain of model management, we find that none of them target the multi-model management use case. Most related work focuses on experiment management and either (1) saves model metadata, but no representation that allows us to recover the model from it, or (2) saves model snapshots in a naive way. Runway [12] and ModelKB [2] belong to category (1), ModelDB [13] and MLflow [15] belong to category (2).

The approaches closest to the multi-model management use case are ModelHub [6] and MMlib [11]. ModelHub's parameter archival storage (PAS) is designed to optimize the storage footprint and to provide short model query latency with a minimal loss of accuracy. The authors do not consider the multi-model management use case and the proposed algorithms to save models have worse than quadratic run time, which makes ModelHub's approach orthogonal to our work.

The model management library MMlib includes three approaches: a baseline approach, a model provenance approach, and a parameter update approach. The baseline approach saves models as full snapshots, while the parameter update approach and the model provenance approach optimize storage consumption by considering that derived models are related. The model provenance approach is based on the idea that one can exactly reproduce model training by tracking model provenance information and avoiding non-deterministic computations. The provenance information consists, for example, of seeds, detailed soft and hardware information, and the source code of the training pipeline. The model provenance approach saves models by saving its provenance information instead of the model parameters and retrieves models by repeating its training based on the provenance information starting from the last fully saved model.

The parameter update approach compares related models on a layer granularity and only saves the delta between models' parameters. For the parameter update approach, the authors consider the immediate base model for computing the delta and only save the very first model as a full model snapshot. This leads to recursively increasing recovery times that can be prevented by saving intermediate model snapshots using the baseline approach. Bhattacherjee et al. [1] discuss more advanced delta-based algorithms in the related field of dataset versioning. They present multiple algorithms to address different constraints for the storage-recreation trade-off and mainly focus on their runtime complexity.

The main contribution of the MMlib paper is that the authors evaluate every approach's performance in terms of storage consumption, time-to-save, and time-to-recover for differently sized models. This allows them to discuss the trade-offs in the field of model management and to give recommendations on when to use which approach. But, only MMlib's baseline approach can be used for multi-model management, and since all of MMlib's approaches are designed and optimized for single-model management, the baseline approach performs poorly for multi-model management.

3 APPROACH

In this section, we first present opportunities for optimization in the multi-model management scenario. Based on this, we develop three approaches to efficiently manage thousands of DL models with the same architecture but different parameters, as it is the case for multi-model management in the deployment phase. We assume that all trained models are saved but only occasionally recovered for purposes such as post-accident analysis. We focus on reducing storage consumption and time-to-save and assume the time-to-recover as least important metric.

3.1 Optimization Opportunities

Assume the following generic multi-model management scenario with n models that share the same architecture and – except for the training data – the same training pipeline, but have different parameters. Existing approaches are not optimized for this and save every model individually. We see the following opportunities for improvement:

(O1) Redundant Model Data. When saving n models individually, we save all data that is not dependent on the model parameters redundantly. This includes, for example, the model architecture, parameter dictionary keys, and metadata. This overhead is significant when saving thousands of small models, which gives us the opportunity to optimize the storage consumption for multi-model management.

(O2) Redundant Provenance Data. MMlib's provenance approach saves models individually and includes, next to the training pipeline information, a snapshot of the training dataset for the update. This means that when we save a set of models using the provenance approach, we save all pipeline information redundantly. Additionally, the training data is often saved regardless of the model management (either by the manufacturer for analytical or by the user for backup purposes). Here we see an opportunity to optimize storage consumption further by saving the provenance information only once and by referencing the used dataset instead of saving a duplicate of it together with the model.

(O3) Write Overhead. Existing model management approaches save metadata, parameters, and code in different services or locations. Saving n models results in n individual writes to every service and a long time-to-save. By saving the models as a set, we can significantly reduce the time-to-save, depending on the target services and location.

3.2 Baseline

Our *Baseline* approach (*Baseline*) is our first approach optimized for multi-model management and represents a set of models by three types of data: metadata, model architecture, and parameters. With *Baseline*, we address the optimization opportunities *O1* (redundant model data) and *O3* (write overhead) while being able to recover every set of models independently.

When saving the model set, we only save the metadata and the model architecture for the first model. This reduces the storage consumption and is possible because all models in the set have the same architecture. For the model parameters, we iterate over all models, concatenate the floating-point numbers representing the parameters, and save them to one binary file. This reduces the overhead of saving the layer names for every model and the overhead that comes with serializing and saving every model's parameter dictionary separately. Representing the set of models by three artifacts instead of thousands – as it would be when saving all models individually – reduces the the number of writes to metadata and file stores.

To recover the model set, we first load the model metadata and architecture, which defines how many parameters each model and layer has. According to that information, we read the parameters sequentially from the parameter file to fully recover all models.

3.3 Update Approach

Our *Update* approach (*Update*) is an improved version of our *Baseline* and, thus, also addresses O1 (redundant model data) and O3 (write overhead). To further optimize *Baseline*, we make use of the fact that per update cycle (1) not all models are updated, and (2) some of the updated models are only partially updated.

Using *Baseline* in this scenario results in saving parameters that have not changed and, thus, in redundantly saved data. We improve this with *Update*, by only saving the updated parameters of the updated models in a binary format instead of full model snapshots for all models.

For an initial model set, we use *Baseline*'s logic and additionally save the hash of every model's layers. We save all subsequent model sets by performing the following steps: (1) We save a reference to the base model set and other metadata. (2) We calculate the parameter hashes for every model and layer and save them. (3) We identify all changed parameters based on the hash information of the previous model set and document the changes in a list. (4) We concatenate all changed parameters and save them to a single binary file.

To recover a set of models, we recursively load the base model set and apply the parameter updates based on the diff list and the binary file representing the model parameters.

3.4 Provenance Approach

The overall idea for the *Provenance* approach (*Provenance*) is to save detailed provenance information that is sufficient to recover a set of models by retraining them instead of model parameters. *Provenance* is based on MMlib's provenance approach but we addresses O2 (redundant provenance data) and O3 (write overhead).

For the initial model set, we save complete model representations using *Baseline*'s logic. For derived model sets, we save the provenance for the set of models. In detail, we save the model metadata, training info, and the environment only once. For the training data, we save one reference per model. These data are sufficient to represent all models because of two implicit assumptions derived from our scenario: (1) the training procedure for updating the models differs only by the used data, and (2) the training data are saved regardless of the model management.

To recover a given set of models, we recover the training information and update every model by deterministically repeating its training on the associated dataset.

4 EVALUATION

In this section, we evaluate our approaches. We first describe our evaluation setup and afterward evaluate the approaches' performance on storage consumption in Section 4.2, on time-to-save (TTS) in Section 4.3, and on time-to-recover (TTR) in Section 4.4. Finally, we discuss the results in Section 4.5.

4.1 Setup

We evaluate our approaches on the metrics of storage consumption, time-to-save (TTS), and time-to-recover (TTR). The storage consumption measures the amount of storage needed to save a set of models; it does not include the storage consumption of referenced models. The TTS is the time it takes to collect and prepare the data to represent a set of models and persist it. The TTR represents the time to load all data and to recover a set of models from it. We use *MMlib's baseline approach* (*MMlib-base*) as a reference point and integrate our approaches into MMlib to guarantee a fair comparison[1]. For the TTS and TTR, we present the median times of five individual runs; the storage consumption is constant.

For all experiments, we use PyTorch version 1.7.1 and run one of two setups. Unless stated otherwise, we use the *server* setup, which is a server with an AMD Threadripper PRO 3995WX 64 Core CPU and 64 GB of RAM. To show the performance on a second, less powerful setup, we also use the *M1* setup, which consists of a machine with the Apple M1 Pro processor (10-Core CPU, 16-Core GPU), 32 GB of RAM, and built-in SSD storage.

As the default scenario, we follow our running example of car battery cell models as described in Section 1. We evaluate all approaches on a fixed sequence of use cases shown in Figure 2. We start with one iteration of U_1, in which the initial models are managed. We assume new batteries with 5000 individual cells, each associated with a small DL model. Following U_1, we have multiple iterations of U_3[2] where the models are fully or partially updated. We do not update all models within one iteration of U_3 but assume that only a subset of models has diverged significantly from their expected behavior and needs updating. We assume that for 5% of all models, a partial update of the parameters is necessary, and for another 5%, a full update.

By default, we run our experiments using one of the best-performing model architectures out of a study on battery electric modeling conducted by Heinrich et al. from Volkswagen [4]. The architecture consists of four fully connected layers and a total of 4,993 parameters. Throughout the paper, we refer to this architecture as *FFNN-48*.

Our initial training data consists of 342M training samples containing information on 352 dynamic discharge cycles. We generate the data using a second-order equivalent circuit model of a 18650 battery cell, which maps an input current to the voltage response, cell temperature, and cell charge [7]. For input currents, we use records of real-world driving discharge cycles provided by Steinstraeter et al. [10]. To increase the data diversity, we generate each cycle with slightly altered model parameters. To create the training data for the use cases, we decrement the state of health (SoH) of the batteries every update cycle. This leads to different aging trends from the initial SoH until the battery's end-of-life. Additionally, we corrupt the data by adding measurement noise to prevent models from training with equal data. Before training, we normalize the data to provide an equal feature scale.

If not stated otherwise, the experiments are conducted as described above. To gain additional insights, we vary the experiments in the following ways: (1) To see the effect of different model update rates, we change the percentage of updated models from 10% to 20% and 30%. (2) To observe the effect of a larger model, we execute our experiments using a model (*FFNN-69*) with 10,075 parameters, that is, except for the number of parameters per layer, identical to *FFNN-48*. (3) To evaluate the effect of data and model from a different domain, we also evaluate our approaches using a convolutional model (*CIFAR*) performing image classification on CIFAR-10 [5] with 6,882 parameters. (4) To investigate the effects of different hardware, we use the *M1* instead of the *server* setup.

[1] you can find the source code under https://github.com/hpides/mmlib-multi
[2] naming to be consistent with MMlib

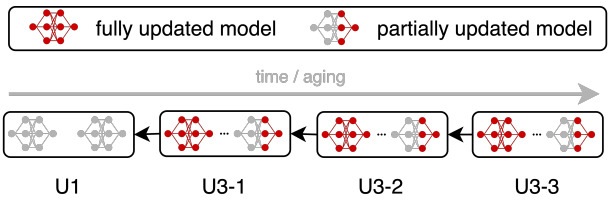


Figure 2: Sequence of use cases used for evaluation

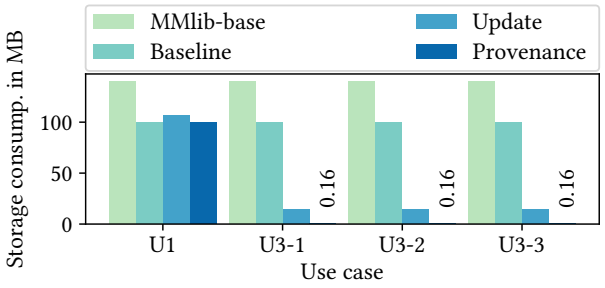


Figure 3: Storage consumption per use case

4.2 Storage Consumption

In this section, we evaluate the amount of storage used to save a set of 5000 models following the *FFNN-48* architecture. Figure 3 shows the storage consumption across the use cases in MB. For U_1, we see that *Baseline* and *Provenance* consume the least storage. All approaches save all 4,993 parameters per model represented by 4 Byte floats which add up to approximately 99.9 MB. This shows that *Baseline* and *Provenance* come with a storage overhead for model architecture and metadata of approximately 4 KB. *MMlib-base* additionally saves the model architecture, the layer names, the model code, and the environment information for every model accumulating to an overhead of approximately 8 KB per model and 40.4 MB in total. Thus, *Baseline* and *Provenance* outperform *MMlib-base* by 29%. Due to slightly different environment information, the improvements are on the *M1* setup with 33% even larger. *Update* uses the same logic as *Baseline* and *Provenance* for U_1. Its increased storage consumption is caused by saving the parameter hash information used to detect changes without having to load the full representation of the previous model.

For U_3, *Update* and *Provenance* outperform both baseline approaches, whose storage consumptions do not change compared to U_1. *Provenance* shows a 99.89% lower storage consumption than *MMlib-base* and an improvement of 99.84% over *Baseline*. It is because *Provenance* saves only the references to the training data and the provenance information for one model. For *Update*, we find a 90% lower storage consummation compared to *MMlib-base* and 86% compared to *Baseline*. *Update* only saves the updated model parameters and the models' hash info. In every iteration

of U_3, we update 250 (5%) models fully and 250 (5%) models partially. Additionally, we save hash info which adds up to a storage consumption of approximately 14 MB for all U_3s. For *MMlib-base* and *Baseline*, the storage consumption does not change compared to U_1 because both approaches always save complete representations of all models and do not take similarities between derived models into account.

In addition to the 10% update rate from above, we also run experiments for 20% and 30%, respectively. Comparing the results to the 10% update rate, we see that only the performance of *Update* changes noticeably and correlates to the update rate. This is expected, because with a higher update rate, there are more model changes and more updated parameters to save. *MMlib-base*'s and *Baseline*'s performance does not change because they always save entire snapshots of all models. The storage consumption of *Provenance* does not change significantly because the additional information we have to save per iteration is 500 (for 20%) or 1000 (for 30%) additional references to datasets.

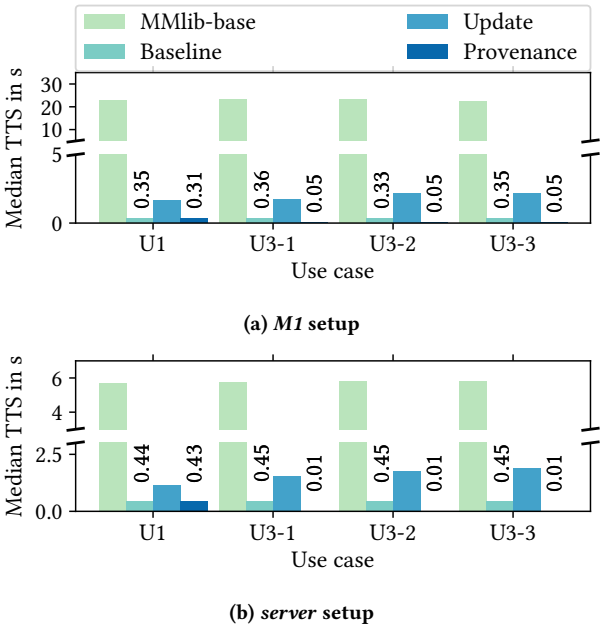
To analyze the effects of saving larger models, we also ran the 10% update rate experiment using *FFNN-69*. The higher number of parameters increases the storage consumption for *MMlib-base*, *Baseline*, and *Update* while the storage consumption for *Provenance* is not affected. *MMlib-base*'s storage consumption increases by 1.7×. The reason for the increase not being exactly 2.0× is that *MMlib-base* saves a lot of redundant metadata, whose size is independent of the number of model parameters. For example, the amount of storage consumed for saving the keys of the parameter dictionaries is only dependent on the number of layers, but not on the number of parameters per layer. *Baseline* and *Update* minimize the amount of redundant metadata, which is why, for these approaches the storage consumption increases by approximately 2.0×. The storage consumption for *Provenance* is not affected by the larger model because the training pipeline for both models is independent of the number of model parameters which leads to almost identical provenance data.

Comparing the storage consumption for *CIFAR* to the one for *FFNN-48*, we find the same trends to the comparison of *FFNN-48* and *FFNN-69* but scaled to the difference in number of parameters between *FFNN-48* and *CIFAR*. The reason is that the storage consumption for all approaches except for *Provenance* depends almost exclusively on the number of model parameters, but not on the type of model architecture or training data.

4.3 Time to Save

Figure 4 shows the TTS across approaches and use cases. In Figure 4a, we see that *MMlib-base* has the highest and *Baseline* the lowest TTS across all use cases. The reason for *MMlib-base*'s long TTS is that it saves all models of a given set individually leading to a high number of writes to the disk and to the document store. In contrast, all other approaches save all models' metadata and parameters bundled. *Update* uses the same logic to save models as *Baseline* but additionally generates and saves parameter hashes which leads to an overhead in TTS. In U_1, *Provenance* saves the model using *Baseline*'s logic. The shorter TTS in U_3 is caused by the drastically reduced storage consumption and the comparably low overhead of saving the provenance information.

Looking at Figure 4b, we also see significant speedups for all our approaches compared to *MMlib-base*. The main differences we notice for the *server* setup compared to the *M1* setup is a significantly reduced TTS for *MMlib-base* in all use cases. The reason is the faster connections to the document store on the



(a) *M1* setup

(b) *server* setup

Figure 4: Median time-to-save per use case

server setup, which significantly reduces the overhead of saving individual models. Analyzing the TTS for the larger models *FFNN-69* and *CIFAR*, we find the same trends but slightly increased numbers for *FFNN-69* because of the larger amount of data to persist.

4.4 Time to Recover

Figure 5 shows the TTR across approaches and use cases. To reduce the training time for the recovery process of *Provenance*, we – exclusively for this approach – only train one model with reduced data per iteration of U_3. This leads to the same trends for the TTR but enables us to run multiple runs of the same experiment in a reasonable amount of time. At the end of this section, we provide the reader with some intuition on numbers for a more realistic training.

Figure 5 shows the TTR across approaches and use cases. For the *M1* setup, we see in Figure 5a that *Baseline*'s TTR outperforms all other approaches and is constant across all use cases. *MMlib-base*'s TTR is also close to constant across the use cases but is significantly higher than all other approaches' TTR.

The constant TTR for *Baseline* and *MMlib-base* is because both approaches save the models in one use case independently of those models saved in other use cases. This implies that we can recover models independently of the models in other use cases. The reason for *MMlib-base*'s long TTR is the same as for its long TTS. Every model is saved separately, significantly increasing the overhead of disk and document store accesses.

For U_1, *Update* and *Provenance* have a TTR close to *Baseline* that increases with every iteration of the U_3 following a staircase pattern. For *Update*, this is because to recover a given model set saved in iteration i of U_3, we have to recover the model saved in the previous iteration to apply the saved differences in parameters. But, the model from the previous iteration may also be dependent on a previous model, which makes recovering models a recursive process. For *Provenance*, recovering a model is also a

recursive process that is similar to the one for *Update*. The difference is that instead of applying the updates in every iteration, we reproduce the training of all the models that were updated.

For the *server* setup and the *FFNN-48* model, the *FFNN-69* model, and the *CIFAR* model, we find the same overall trends as for the *M1* setup. In Figure 5b, we see for the server setup that both *MMlib-base* and *Baseline* show close to constant numbers while *Update* and *Provenance* show a staircase pattern because of their recursive recovery process. The TTS for all approaches is lower than on the *M1* setup, which is mostly due to faster connections to the metadata store. *MMlib-base* is the approach most benefiting from this, which is why the improvements over *MMlib-base* are smaller but still significant.

To get an intuition for the duration of the recovery process, we executed one run of all use cases for *Provenance* with an extensive training considering over 90,000 training samples and ten epochs. For the first iteration of U_3, we measured a TTR of approximately six hours. For further iterations of U_3, the numbers follow the discussed staircase pattern leading to approximately 12h for iteration two and 18h for iteration three.

Compared to the TTR of the other approaches, these numbers are orders of magnitude higher. But, we have to consider two aspects: (1) The pipeline we executed for model training is not optimized in any way. For example, the data pre-processing is done online on a single thread and not parallelized. In addition, we execute the pipeline on a CPU instead of a GPU. But, the only effect of the non-optimized pipeline are higher inference and training times which means that the presented numbers for the provenance approach are conservative and that the statistical relevance of the results is not influenced. By fully optimizing the pipeline for the given workload, we can significantly reduce the training time and, with this, the TTR.

(2) As mentioned above, we train for ten epochs and over 90,000 data points. Performing additional experiments, we saw that using fewer data and a lower number of epochs leads to similar model performance.

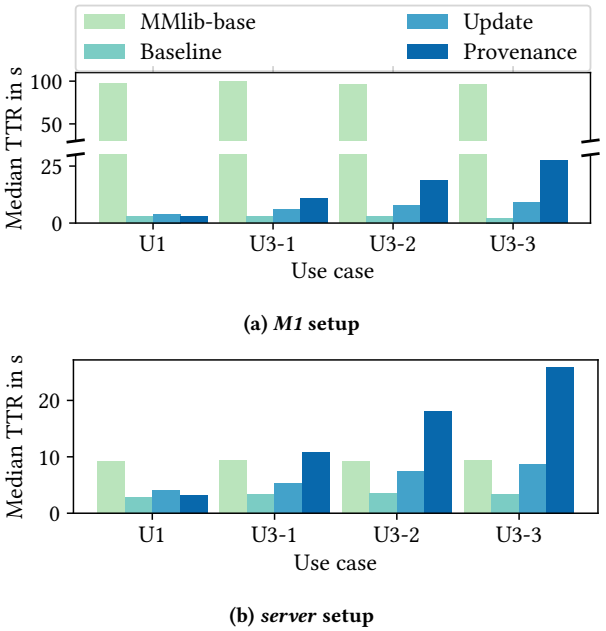
Since the actual run time of a model training is use case, implementation, and hardware dependent, we leave an in-depth analysis of different configurations as future work. Nevertheless, both aspects discussed above put the high TTR into perspective and show that *Provenance* is a reasonable choice.

4.5 Discussion

Overall, our evaluation shows that our *Baseline* approach outperforms non-multi-model management approaches in terms of storage consumption, TTS, and TTR. The approaches *Update* and *Provenance* further reduce storage consumption but come with an increased TTR.

Considering that our highest priority is storage consumption and we assume model recoveries to happen rarely, *Provenance* is the best approach. However, *Provenance* comes with the drawback of a long and compute-intensive model recovery process. If this is not acceptable, *Update* is the next best approach; it has a lower storage consumption but only slightly increases the TTR. If the storage consumption is not important and TTR has the highest priority, *Baseline* is the best approach.

Discussing the approaches' performance, we find that there is no single best choice. Thus, we have to carefully consider the tradeoffs when choosing the best approach for a given scenario. Currently, this is a manual choice, but as part of future work, we plan to develop heuristic-based approaches that dynamically choose the most suitable strategy for a given scenario. As part of this, we also plan to include other promising related approaches that come with different specific trade-offs.



(a) *M1* setup

(b) *server* setup

Figure 5: Median time-to-recover per use case

Another direction of future work is to evaluate if it is beneficial to integrate compression techniques into our approaches and with what trade-offs different algorithms come. With *Update*, we deduplicate exact same parameters and represent each by 4 Bytes. Related work [6] shows that the storage consumption can be reduced using delta encoding and other compression techniques.

5 CONCLUSION

In this paper, we define and analyze the problem of managing multiple different models in the model deployment phase.

We discuss why existing approaches perform poorly in this scenario and present three optimized approaches: (1) *Baseline* saving full model representations using a binary format and reducing the amount of saved metadata, (2) *Update* additionally saving only changed parameters, and (3) *Provenance* saving model provenance data instead of model parameters to recover the model by reproducing its training.

We evaluate all our approaches for the two use cases of managing deep learning based car battery models and image classification models in terms of storage consumption, time-to-save, and time-to-recover. Next to the use case, we vary the size of the models, the percentage of updated models, and the hardware setup. We find that *Baseline* outperforms existing approaches for TTS and TTR by more than one order of magnitude while reducing the storage consumption by up to 29%. *Update* and *Provenance* reduce the storage consumption additionally by up to 86% and 99%, respectively.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of dataset versioning: Exploring the recreation/storage tradeoff. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 8. 1346.

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

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

[4] Felix Heinrich, Patrick Klapper, and Marco Pruckner. 2021. A comprehensive study on battery electric modeling approaches based on machine learning. *Energy Informatics* 4, 3 (2021), 1–17.

[5] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009).

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

[7] Steven Neupert and Julia Kowal. 2018. Inhomogeneities in Battery Packs. *World Electric Vehicle Journal* 9, 2 (Aug. 2018), 20. https://doi.org/10.3390/wevj9020020 Number: 2 Publisher: Multidisciplinary Digital Publishing Institute.

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

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

[10] Matthias Steinstraeter. 2020. Battery and Heating Data in Real Driving Cycles. (Oct. 2020). https://ieee-dataport.org/open-access/battery-and-heating-data-real-driving-cycles Publisher: IEEE Type: dataset.

[11] Nils Strassenburg, Ilin Tolovski, and Tilmann Rabl. 2022. Efficiently Managing Deep Learning Models in a Distributed Environment. In *Proceedings of the 25th International Conference on Extending Database Technology (EDBT 2022) Edinburgh, UK, March 29 - April 1*. OpenProceedings.org, 234–246. https://doi.org/10.48786/edbt.2022.12

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

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

[14] Billy Wu, W. Dhammika Widanage, Shichun Yang, and Xinhua Liu. 2020. Battery digital twins: Perspectives on the fusion of models, data and artificial intelligence for smart battery management systems. *Energy and AI* 1 (Aug. 2020), 100016. https://doi.org/10.1016/j.egyai.2020.100016

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