MGit: A Model Versioning and Management System

Wei Hao [*1 2] **Daniel Mendoza** [*1 3] **Rafael da Silva** [4] **Deepak Narayanan** [1 5]
Amar Phanishayee [4] **Asaf Cidon** [2] **Junfeng Yang** [2]

Abstract

New ML models are often derived from existing ones (e.g., through fine-tuning, quantization or distillation), forming an ecosystem where models are *related* to each other and can share structure or even parameter values. Managing such a large and evolving ecosystem of model derivatives is challenging. For instance, the overhead of storing all such models is high, and models may inherit bugs from related models, complicating error attribution and debugging. In this paper, we propose a model versioning and management system called MGit that makes it easier to store, test, update, and collaborate on related models. MGit introduces a lineage graph that records the relationships between models, optimizations to efficiently store model parameters, and abstractions over this lineage graph that facilitate model testing, updating and collaboration. We find that MGit works well in practice: MGit is able to reduce model storage footprint by up to 7×. Additionally, in a user study with 20 ML practitioners, users complete a model updating task 3× faster on average with MGit.

1. Introduction

ML models are deployed across a wide set of tasks, spanning different target hardware, data and label availability regimes. In all of these disparate use cases, it has became increasingly common for models to be generated from existing ones. For instance, when a large amount of supervised data might not be available for a particular task, pretrained models created by self-supervised pretraining on a large unlabeled dataset can be fine-tuned on a smaller labeled dataset (Pratt, 1992; Torrey & Shavlik, 2010; Weiss et al., 2016; Bommasani et al., 2021). For efficient execution on low-powered devices like mobile phones or embedded devices, full-precision models trained on datacenter accelerators are often quantized, pruned, and distilled (Ba & Caruana, 2014; Polino et al., 2018; Guo et al., 2021; Hao et al., 2022). In regimes where new data is constantly streaming in (e.g., in recommendation systems), models are continuously trained (Baylor et al., 2019; Liu, 2017). In complex deployments like self-driving cars, an individual model might be composed of several related sub-models built collaboratively (e.g., Tesla's Autopilot software system (Karpathy, 2019) in 2019 had multiple components using the same shared backbone models to detect key visual elements like traffic lights and human beings).

Unfortunately, no existing system allows for the easy management of these related (or *derived*) models. Existing widely-used frameworks like PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016) and Hugging Face `Transformers` (Wolf et al., 2019) support the development and management of single models at a time, but do not store dependence information across different models.

In this paper, we show that this is an untapped opportunity. Without automated lineage tracking, various tasks in the model management life cycle are hard and inefficient:

- **Model storage and memory redundancy.** Many models share parameter values, or have minor deviations, leading to redundancy.
- **Model debugging.** It is hard to collectively debug models that are themselves derived from other models. Does an undesired behavior originate in a given model or in an upstream model?
- **Model updating.** It is hard to update related models and keep them in sync. If a certain model is updated, how should dependent models be updated?
- **Model collaboration.** It is hard for multiple users to *collaboratively* develop models and determine if concurrent changes made to the model conflict.

To leverage the opportunity presented by lineage tracking, we design and build a system called MGit. MGit makes the following contributions.

Lineage graph. MGit proposes a lineage graph data struc-

[*]Equal contribution [1]Work done while authors were at Microsoft Research [2]Columbia University [3]Stanford University [4]Microsoft Research [5]NVIDIA. Correspondence to: Wei Hao, Daniel Mendoza <`wh2473@columbia.edu`, `dmendo@stanford.edu`>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

ture to track provenance across ML models through *dependency edges*, and uses *creation functions* to optionally record how models are derived from their parents. The lineage graph also stores other metadata like test functions that can be used for model monitoring. A lineage graph can be created automatically from existing model checkpoints, or manually through a Python or command-line interface.

Storage and memory deduplication. MGit incorporates optimizations to more efficiently store model parameters in the lineage graph: it uses content-based hashing and indirection to store parameters shared across models efficiently, and can compress the deltas between non-shared parameters of parent and child models efficiently with no change in underlying model accuracy. MGit derives deltas between models of different architectures by using a `diff` primitive to automatically match the layers of the same dimension between models. MGit's storage optimizations are able to compress model checkpoints by up to 7× relative to storing each model separately. These optimizations can also be used to conduct memory-efficient collocated inference of related models with little change in accuracy.

Support for disparate applications. In our evaluation, we show that MGit can automatically track dependencies across fine-tuned models, models created using federated learning, and also models specialized for edge devices. Once constructed, the lineage graph can be used to test models and perform diagnostics using a `traversal` primitive. This primitive can also be used to automatically update models given upstream updates. We also provide a `merge` primitive that supports collaboration use cases.

We have open sourced our implementation of MGit at `https://github.com/msr-fiddle/mgit`.

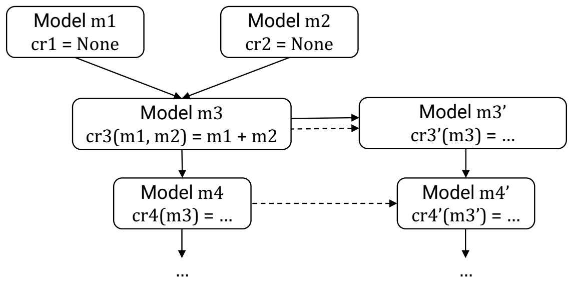


Figure 1: Example lineage graph. Nodes in the graph are models; direct relationships between models are tracked through edges. Each model node can be associated with an optional creation function `cr` that specifies how the model can be created from its parent nodes. For example, model `m3` in this example is created by summing `m1` and `m2`. Provenance edges are shown as solid lines and versioning edges are shown as dashed lines. Models are created by following solid edges only. Two models can have both provenance and versioning edges between them.

2. Target Settings

MGit is useful in various target settings where models are derived from other models. We describe some of these settings below; this list is not intended to be exhaustive.

Adaptation. Transfer learning (Pratt, 1992; Torrey & Shavlik, 2010; Weiss et al., 2016) has been widely adopted to specialize models to downstream tasks, especially in settings where a large *labeled* dataset might not be available. For example, a model trained with a masked language modeling objective (MLM) using self-supervision can then be fine-tuned on various text classification tasks with small labeled datasets; the Hugging Face model repository (Wolf et al., 2019) has more than 27,000 BERT model derivatives.

Model versioning. Model updates (e.g., to fix an undesired behavior or to train a model on new training data) can create new versions of a model. In such cases, users often do not just value the latest model version: for instance, a new model version might suffer from a functionality regression discovered later, forcing a rollback to an older version.

Federated learning (FL). Federated learning (Konečnỳ et al., 2016; Li et al., 2020) makes it possible to train models in a decentralized way, ensuring that the entire training dataset does not need to be available in a single central location. This is advantageous in application settings where privacy concerns might preclude data being uploaded to a central repository or network bandwidth is limited. In FL, multiple copies of a model are updated independently, inducing new model derivatives, and then coalesced periodically to obtain the global shared model.

Specialization to edge devices. Dense full-precision models often can be too memory- or compute-intensive to run efficiently on edge devices like mobile phones, necessitating edge-adaptation techniques like quantization and pruning (Hao et al., 2022). When employed, such techniques create new derived models.

Multi-task learning (MTL). In MTL, a single model is trained on multiple tasks, resulting in per-task "backbones" and a few task-specific parameters (Ruder, 2017). This has been shown to improve model generalization.

3. Lineage Graph

To automatically track model derivatives, MGit proposes a lineage graph data structure. Individual models are represented as nodes in the lineage graph. A given lineage graph is only intended to keep track of *related* models; a company's image classification and language models would be stored in separate lineage graphs for instance.

Component	Description
Node	ML models are represented in the lineage graph as nodes. Each node has an optional creation function `cr` that tracks how the model can be created from its parents. Nodes are also associated with other relevant metadata like the model type and a unique name.
Provenance edge	Edge between a model and model(s) derived from it. Tracks how models are created. Can be followed to determine the root cause of a model regression, or to update models when an upstream model is modified.
Versioning edge	Edge between two consecutive versions of the same model. Used to track versions of a model, and can be queried (e.g., to run tests on all versions of a given model).

Table 1: Components of MGit's lineage graph.

Lineage graphs have two different types of edges: provenance and versioning, shown as solid and dashed edges in Figure 1. Provenance edges are used to track which models are derived from another. A provenance edge from model A to model B indicates that B is derived from A using some function of model A's parameters, which is (optionally) defined by an MGit creation function (§3.1.2). Examples of "derived" relationships in the lineage graphs considered in our evaluation include: B is derived from A after removing / adding certain layers, B (e.g., a model trained for sentiment analysis) is derived from A (e.g., a model trained for masked language modeling) after changing a few layers and fine-tuning, and B is derived from A through parameter pruning. A model with no provenance relation with another indicates that the model is not derived from an already existing model in the lineage graph. In other words, the model is created through some black-box process that MGit does not have visibility into. Provenance edges are useful to monitor the behavior of models as they are derived from each other (e.g., track if an undesired behavior originates in a given model or in an upstream model), and is used for other MGit functionality like `run_update_cascade`, which is described in §5.

Versioning edges are used to subjectively track versions of a given model, and thus require user annotation (in MGit, these are specified using the `add_version_edge` API). They are useful for tracking how models change with time independent of provenance. Versioning edges capture cases where a model A for a target application is replaced by a new model A'. A versioning edge does not imply a provenance edge: if A' is trained from scratch, then there is no provenance between A and A'. For example, when a model service for image classification replaces a ResNet-50 model with a ResNet-101 model trained from scratch, the ResNet-101 model becomes the next version of the ResNet-50 model though it is not derived directly from the ResNet-50 model (i.e., the weights of the ResNet-50 model are not used to create the ResNet-101 model). In this case, there is a version edge between the ResNet-50 and ResNet-101 models, but no provenance edges. Traversal functions can visit nodes through these versioning edges (e.g., we can compute how "robust" to perturbations different versions of the same model are). Two models can be connected by both provenance and versioning edges (e.g., `m3` and `m3'` in Figure 1 are connected by a solid provenance edge and a dashed versioning edge; `m3'` is created from `m3` and is also annotated as the next version of `m3`). Models connected by provenance edges are not necessarily different versions. In Figure 1, `m4'` is not the next version of `m3'`.

The lineage graph also tracks other metadata, such as the model type and a unique name, which is useful for testing models, updating them, and mutating the graph. An example is shown in Figure 1.

3.1. Interface

MGit can be used from both command line and Python. Its command-line interface is analogous to that of `git`, and enables users to effortlessly view the lineage graph, mutate it, run registered tests, etc. The Python interface also allows specification of a model's creation function. To facilitate both interfaces, changes to metadata are serialized to disk at the end of every operation, and de-serialized at the start of every operation. MGit's full API is shown in Table 4 in the Appendix §A.1.

3.1.1. Node and Edge Addition

MGit's API supports adding or removing edges and nodes. Node and edge addition can be directly integrated into larger applications. For example, a federated learning controller (Konečnỳ et al., 2016) can create new nodes (and corresponding edges) in the corresponding lineage graph directly in code using the Python API.

3.1.2. Creation Function

Each node is associated with an optional *creation* function `cr` that specifies how the model is created from its parents. Creation functions follow the same code-style as typical ML programs (i.e., a main training loop, data loading, model initialization, definition of a loss function, etc.). The creation function facilitates automated model updating if an upstream model in the lineage is updated. Creating a new model involves calling the function `cr`, and then calling `add_edge` and/or `add_version_edge`. A model's creation function has one argument for each of the node's "provenance parents".

Fine-tuning and adaptation. Fine-tuning and other lightweight adaptation techniques (e.g., adapters (Rebuffi et al., 2017), BitFit (Zaken et al., 2021)) involve initializing a new model from the parent's checkpoint (full checkpoint or partial with parameters for a subset of layers copied over), and then running training iterations:

```
class CreationFunctionFineTuning:
  def __init__(self):
    self.lg, self.data_loader = mg.LineageGraph(<
        filepath>), torch.DataLoader(<filepath>)

  def initialize_model(self, parent_list):
    self.child_model, parent_model = Model(),
        parent_list[0].get_model()
    copy_parameters(parent_model, self.child_model)
        // Copy parameters from parent_model.
    self.child_model.head = initialization(
        child_head_dimensions) // Initialize head.

  def run_iteration(self):
    batch = next(self.data_loader)  // Iterate
        through data using DataLoader.
    loss = cross_entropy_loss_fn(self.child_model(
        batch)) // Forward pass.
    loss.backward(); optimizer.step() // Backward
        pass -> Step optimizer.

  def __call__(self, parent_list):
    self.initialize_model(parent_list)
    while self.data_loader.has_next():
      self.run_iteration()
    return self.child_model
```

Edge device specialization. Quantization and pruning also fit into this framework. For example, a simple form of quantization can just downcast each parameter tensor, which is easy to encode in a `cr` function. Distillation is similar, but with a more complex creation function.

Multi-task learning. Typically, some parameters of a multi-task model are shared across tasks, while some parameters are not. MGit facilitates multi-task learning by automatically synchronizing updates of shared parameters across models. We can also use creation functions to train models in an MTL fashion by specifying the shared parameters in the creation function. Pseudocode is in Appendix §A.2.

3.1.3. TRAVERSALS

Traversals are specified as an iterator over the lineage graph nodes. Nodes can be visited in arbitrary orders. Simple example traversals are BFS and DFS. Traversals can also specify the types of edges that should be traversed. For example, to test all versions of a particular model, we could use a traversal that starts at the first version of the given model and then only follows versioning edges. More complex traversals like binary search (for test bisections) are also possible using Python generators. We detail how we leverage traversals over MGit's lineage graph to automate model updating and testing in §5.

3.2. Graph Construction

Graphs can be constructed manually or automatically.

Manual construction. Users can manually add nodes to the lineage graph and specify their provenance, using the provided `add_node`, `add_edge`, `add_version_edge` and `register_creation_function` APIs. These are available both in the command-line interface or in Python. For example, an FL controller implemented in Python can register nodes and edges in code. Similarly, fine-tuning code can directly register edges between the parent model and new child model.

Automated construction. To alleviate manual effort, MGit can automatically infer provenance edges between models constructed outside MGit's API based on structural and parameter similarity.

We use a custom `diff` primitive to compute the differences between two models, both structural (connectivity between layers of the model) and contextual (values of parameters in the model). `diff` makes no assumptions on the model's architecture, and can also be used for dynamic models like MoEs (Fedus et al., 2022; Du et al., 2022) that use routing layers with learned parameters, since `diff` only looks at layer parameters and layer connectivity which is agnostic to dynamic control flow. After obtaining both models' DAG representations (Reed et al., 2022) where layers and their connectivities are represented as nodes and edges in the DAGs, `diff` runs a hash-table-based graph matching algorithm (Appendix §A.4.) that returns the layers and edges to add and remove to produce model B from model A. MGit calculates two scores $d^{\text{structural}}$ and $d^{\text{contextual}}$ based on the number of edges in the `diff` output:

$$d^{\text{structural/contextual}} = \frac{|\text{edges}_{\text{diff}}^{\text{structural/contextual}}|}{|\text{edges}_A| + |\text{edges}_B|} \quad (1)$$

For a model x, MGit locates the model in the graph that has the smallest contextual and then structural divergence score with x; this node is chosen as the parent of x. The automated algorithm only adds provenance edges; versioning edges require user annotation. If no model is sufficiently contextually or structurally similar, `x` is added as a root (node with no parents). We show the runtime scaling of this algorithm in Figure 5 for large lineage graphs in Appendix §C.1.

4. Storage and Memory Optimizations

In this section, we describe how MGit leverages the lineage graph to minimize storage and memory footprint for model parameters.

Content-based hashing. MGit uses content-based hashing to encode parameters, allowing it to avoid storing redundant copies of parameters. MGit manages a global hash table that stores the parameters of all models in a lineage graph. The `SHA-256` hash of each parameter tensor (using both tensor value and its shape) is used as the hash key.

Delta compression. The non-identical parameters of parent and child models might only differ slightly. This motivates the use of compression and decompression of *parameter deltas*, which can be sparse for similar models. Previous work (Hu et al., 2020) explored various lossy and lossless compression methods for delta compression and concluded that combining quantization, which converts the delta from a float array to an integer array (lossy), with lossless compression of the subsequent quantized delta works well for many models. MGit extends this approach for delta compression between models in the lineage graph.

One challenge in compressing deltas across models is the fact that parent and child models in the lineage graph might not have identical architectures. To circumvent this, MGit employs a longest common subsequence algorithm to compute a mapping between parameters of the model that *have the same shape*. For models with the same architecture, this creates a mapping between corresponding parameters of the same layer.

Given a mapping of parameters (p_1, p_2) of two different models, MGit first computes the delta Δp between each pair of parameters and then quantizes Δp (Hu et al., 2020):

$$\Delta p = p_1 - p_2, \Delta p^{\text{quantized}} = \left\lfloor \frac{\Delta p}{2 \cdot \log(1+\epsilon)} + 0.5 \right\rfloor \quad (2)$$

MGit then uses `compressor` and `decompressor` modules to losslessly compress $\Delta p^{\text{quantized}}$. Different lossless compression techniques can be used like RLE (Robinson & Cherry, 1967) and LZMA (Igor, 1998); each of these provide different tradeoffs between compression ratio and runtime (§6.2).

ϵ is a configurable error bound. Larger ϵ leads to more values in $\Delta p^{\text{quantized}}$ being driven to 0, contributing to a higher compression ratio after lossless compression. However, increasing ϵ also reduces the faithfulness of $\Delta p^{\text{quantized}}$ to Δp and may cause a larger deviation in model accuracy. By default, we set $\epsilon = 10^{-4}$.

MGit only *accepts* the delta compression if the compression results in storage saving and an accuracy drop within a configurable threshold (if tests are registered). Each delta-compressed parameter will be stored on disk as the compressed delta along with a pointer to the parent layer to facilitate future decompression. If not, compression is rejected and the uncompressed model is persisted (Algorithm 3 in Appendix). This procedure can be applied recursively. That is, the delta can be computed between the layers of a child model and a parent model that is itself delta compressed. Loading a model instance then involves recursively decompressing up the chain until the first ancestor node that is not delta compressed.

Memory optimization during inference. These techniques can be extended to perform memory-efficient collocated inference of multiple related models. Similar to related work (Li et al., 2022; Padmanabhan et al., 2023; Zhou et al., 2022), MGit can load unique weight tensors into GPU memory by leveraging the same content-based hashing techniques described above; this not only decreases memory footprint of inference, but can also increase inference throughput by increasing the arithmetic efficiency of the underlying operators. MGit can also more aggressively reduce memory consumption *even in settings where model parameters are different* by identifying small parameter deltas and squashing them to zero; only a single copy of the parameters is then loaded during collocated model inference. We provide more details in Appendix §C.2.

5. Higher-Level Functions on Lineage Graph

Various higher-level workflows around testing, updating and collaboration can now be built on top of the lineage graph and the abstractions it exposes.

Testing. Given a lineage graph, MGit exposes APIs to examine models in the graph. Testing can be thought of as executing per-node test functions `t` as part of a graph traversal. MGit provides a way to register functions both for individual models and for all models of a specific type. Users can specify a regex `re`; for every node encountered in the traversal, all registered tests whose names match `re` are run. Running the same test for multiple related models allows users to track model regressions more easily (e.g., all descendants of a particular model might show poor accuracy on a particular test) and correlate dependency information with model accuracy on various tasks. Users can also use this workflow to search for the upstream model where a certain unintended behavior originated.

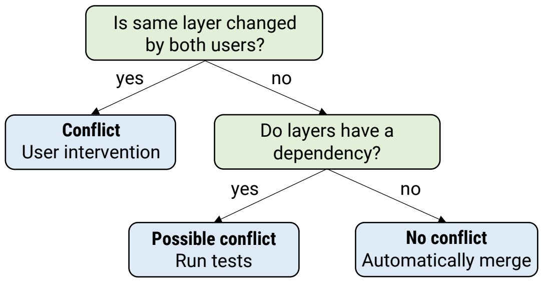


Figure 2: Decision tree for merging changes. If a layer is changed by both users, manual merging is required because of the conflict. Otherwise, if changed layers have a dependency, a conflict is possible and tests are required to verify whether this is the case. If neither of the above conditions is true, the merge can be done automatically.

We can also execute other per-model functions as diagnostics. For example, we could compute the deltas between every model and its parent(s), or measure the sparsity levels of models.

Model updating. When users update a model, they need to create a new node in the lineage graph by using MGit's `add` function; MGit does not support in-place updates to models. This will then provide users the option to automatically trigger the update cascade on downstream dependent models. In the most basic form of model updating, when a new version of a model is created, provenance edges in the lineage graph are followed to produce a new set of model versions for all of the model's descendants (`run_update_cascade` API). We use a modified BFS traversal, where a node is visited only once all of its parents have been visited, to ensure that the creation function is called only when all required upstream models are available. A *new version* of the model is computed using the node's registered `cr` function; MGit never overwrites existing models with its automated functionality since users might want to vet new models. MGit's storage optimizations ensures that keeping all model versions is cost effective. Full pseudocode is in Algorithm 1 in the Appendix.

We can use MTL to continuously share parameters across models even across updates by using an appropriate creation function (§3.1.2). The traversal to re-train models needs to ensure that full MTL groups are executed only once all MTL groups on which they depend complete. Additionally, individual creation functions `cr` are not called. Instead, we take all desired functions `cr_1`, `cr_2`, ..., `cr_n` and merge them into `cr'` that returns n new models. Internally, this merged creation function `cr'` ensures that weights are shared, appropriate loss functions are used, etc.

Collaboration. MGit also supports collaboration workflows through a `merge` primitive. The objective of this primitive is to identify if concurrent changes made to a given model are "compatible" or not. Changes made to the same layers of a given model need to be manually merged (akin to a manual merge for a merge conflict in `git`). Changes made to different layers of a model are also not necessarily compatible: in cases of a dependency between two layers (i.e., one layer consumes the output of the other eventually or a downstream layer consumes the outputs of both layers) changed by different users, additional tests are needed to verify that the concurrent changes did not result in a model regression. We provide more details in Appendix §A.4.

Name	Description	# Nodes / # Edges
$G1$	NLP models downloaded from HuggingFace.	23 / 21
$G2$	BERT-style models specialized for NLP tasks using fine-tuning and other lightweight adaptation techniques. Some models are trained using different datasets, creating multiple versions.	91 / 171
$G3$	Vision models trained in a decentralized fashion using Federated Learing.	60 / 95
$G4$	Vision models with pruned model weights for edge devices.	22 / 19
$G5$	BERT-style models specialized for NLP tasks with Muti-Task Learning to enforce parameter sharing.	10 / 9
$G6$	Pythia-6.9B checkpoints from EleutherAI (sampled every 1000 steps).	10 / 9

Table 2: Lineage graphs considered in evaluation.

6. Evaluation

In this section, we evaluate MGit's storage optimizations, and its utility when developing and debugging models. Unless otherwise noted, experiments were run on a workstation with 4 NVIDIA RTX A6000 GPUs with CUDA 11.7.

6.1. Lineage Graphs

Table 2 shows the lineage graphs considered in this evaluation, reflecting various applications that create ML model derivatives (§2). $G1$ was automatically constructed using the algorithm outlined in §3.2. 22 out of 23 nodes were correctly inserted, while one node was misidentified by MGit (`bert-base-uncased`). MGit's API allows errors made by the automated algorithm to be corrected manually by users (using the `remove` functions in the API). The automated graph construction function is able to correctly insert models that have frozen weights inherited from their parent model by computing structural and contextual divergence scores between model pairs. Graphs $G2$ through $G6$ were manually created using the `add` functions, in conjunction with the training APIs used to create the models.

Graph	Compression technique	Comp. ratio ↑	Accuracy Δ ↓ Max.	Avg.	Per-model ↓ compression time
$G1$	`MGit (LZMA + Hash)`	**2.14**	0.09	0.01	35.7 mins
	`MGit (RLE + Hash)`	1.13	1.02	0.08	30.9 mins
	`MGit (Hash)`	1.05	**0.00**	**0.00**	**12.0 mins**
	`Full`	1.83	0.08	0.00	36.5 mins
	`Full w/o quantization`	0.87	0.00	0.00	29.8 mins
$G2$	`MGit (LZMA + Hash)`	**5.35**	0.01	0.00	7.4 mins
	`MGit (RLE + Hash)`	1.84	0.01	0.00	4.1 mins
	`MGit (Hash)`	1.01	**0.00**	**0.00**	**0.1 min**
	`Full`	1.85	0.00	0.00	14.6 mins
	`Full w/o quantization`	0.78	0.00	0.00	3.8 mins
$G3$	`MGit (LZMA + Hash)`	**6.96**	0.11	0.01	2.5 mins
	`MGit (RLE + Hash)`	3.11	0.49	0.03	2.4 mins
	`MGit (Hash)`	1.00	**0.00**	**0.00**	**1.1 mins**
	`Full`	2.29	0.25	0.06	4.0 mins
	`Full w/o quantization`	0.72	0.06	0.01	2.8 mins
$G4$	`MGit (LZMA + Hash)`	**2.57**	0.35	0.07	2.5 mins
	`MGit (RLE + Hash)`	2.04	0.35	0.07	2.5 mins
	`MGit (Hash)`	1.00	**0.00**	**0.00**	**1.1 mins**
	`Full`	2.57	0.37	0.07	3.0 mins
	`Full w/o quantization`	1.47	0.07	0.01	2.6 mins
$G5$	`MGit (Hash)`	4.93	0.00	0.00	**0.1 min**
$G6$	`MGit (LZMA + Hash)`	2.64	0.02	0.01	**2.7 mins**

Table 3: Compression ratio, maximum / average accuracy delta across models in lineage graph, and per-model compression time of delta compression techniques for various lineage graphs. `Full` is the approach of using quantization and LZMA on full models instead of the deltas.

We provide more details about these lineage graphs in Appendix §B; we are also open sourcing code to re-create these lineage graphs on Github.

6.2. Storage Optimization

We now evaluate MGit's storage optimizations. Table 3 shows the results for various MGit configurations, combining the content-based hashing and delta compression techniques described in §4. We show results for two versions of the delta compression algorithm: one that uses `LZMA` for its lossless **`compressor`** / **`decompressor`**, and another that uses `RLE` instead. We also show the content-based hashing technique alone (`Hash`). For $G4$, we quantize parameters before calculating deltas so that the sparsity is preserved in each model. Additionally, we implemented two baselines that run LZMA on either a quantized version or the original full model (`Full` and `Full w/o quantization`). We show three metrics: compression ratio (larger is better), maximum accuracy delta between original uncompressed models and models in the compressed lineage graph (smaller is better), and average compression + testing runtime per model (smaller is better).

As a lossless storage method, content-based hashing shows storage savings proportional to the number of parameters duplicated across models in the lineage graph. We observe that these numbers are 9.4%, 16.5% and 79.6% for $G1$, $G2$ and $G5$ respectively. $G5$ has more duplicate parameters since its models were explicitly trained to share parameter values using MTL. LZMA shows the best compression ratio across all graphs for delta compression methods which are lossy due to the quantization step.

Quantization and LZMA applied to the full models results in worse compression ratios than the default MGit approach (compressing deltas) except for $G4$. There are three reasons for this: first, the fraction of compressed parameters in $G4$ is lower compared with $G2$ and $G3$ due to accuracy check failures. Second, deltas between corresponding layers of (parent, child) model pairs in $G4$ is larger than the deltas between models in other graphs as a result of how the models were derived from each other (L1 pruning instead of fine-tuning). Third, the three roots models in $G4$ were not compressed in MGit whereas they were in the `Full` baseline. This is an optimization that can be possibly added to MGit.

Methods with larger compression ratios take longer to compress. We believe an average runtime of even 10-15 minutes per model is reasonable given long model training times ($G1$ takes particularly long because we ran tests on CPUs instead of GPUs). Compression time can be further reduced by compressing layers concurrently. For example, in $G6$, MGit conducts compression in parallel and reduces the per-model compression time from 44.4 to 2.7 minutes.

6.3. Memory Optimization

We ran experiments to demonstrate how MGit can optimize memory by colocating models during inference, thereby reducing GPU memory consumption. We collocate `Llama-2-7b` with `Llama-2-7b-chat` (Touvron et al., 2023), which share the same architecture but have different parameter values. We evaluate the memory savings when 30%, 60% and 90% of corresponding layers are merged by MGit, and the corresponding set of shared weights is loaded from `Llama-2-7b-chat`. Our results show that collocation saves 0.5%, 16.2% and 39.3% of GPU memory in these three setting, compared with loading two models independently. However, due to the merged weights, we observe a decrease in efficacy of the `Llama-2-7b` model (measured in terms of accuracy norm, multiple-choice scores and bits-per-byte), ranging from -0.6% to +12.2% compared with conducting inference on the original `Llama-2-7b` model. Moreover, we observe a speed up in inference by up to $1.5\times$ since model collocation and merging involves batching of the merged layers, increasing the arithmetic efficiency of the computation. We provide more details in Appendix §C.2.

6.4. Functionality

MGit enables functionality that is hard to perform without a model management system.

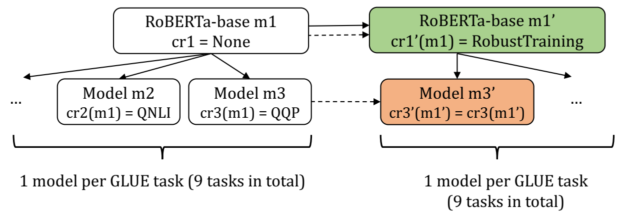
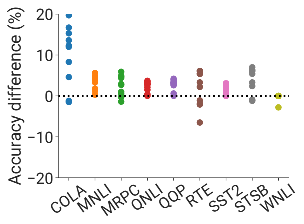


Figure 3: Lineage graph $G2$, along with the new models auto-generated using MGit's `run_update_cascade` functionality. The parent MLM model initially has 9 children (one corresponding to each GLUE task), connected by provenance edges. When the parent MLM model is updated (green box) to be robust to input data perturbations (e.g., text with various types of typos), MGit's `run_update_cascade` function is triggered, which automatically generates new versions of the GLUE models (orange boxes) that are also robust to perturbations.

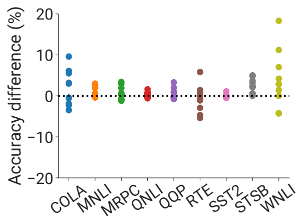
Testing. We found MGit to be useful in testing models by providing a way to combine dependency information with testing functions. For example, MGit facilitates running test bisections, searching for the first model in a lineage chain which fails a particular test. In the best case, we found that failing models can be found as much as $1.5\times$ faster using test bisections. We expect this improvement to be even greater for deeper lineage chains where asymptotic improvements matter more.

Model training and updating. MGit's lineage graph and creation functions can also be leveraged to train models that share state. $G5$ was trained using MTL (§3.1.2) to create RoBERTa models for 9 different GLUE tasks (Wang et al., 2018); the models in $G5$ shared 98% of their parameters (only parameters in the model heads were not shared).

We also evaluate MGit's automated model updating functionality (`run_update_cascade` API). For $G2$ and $G5$, we try to more efficiently build task-specific models robust to various perturbations by fine-tuning the parent MLM model (`m1`) with perturbed data, generating a new model `m1'`. We then run `run_update_cascade` to automatically generate new children `m2'`, `m3'`, . . ., `m10'` from `m1'` by reusing the creation functions that facilitated the creation of `m2`, etc. from `m1`. This process is visualized in Figure 3. These creation functions do not use perturbed data at all; any ability of `m2'`, `m3'`, . . ., `m10'` to be more robust on the perturbed GLUE tasks is passed down from its parent model `m1'`. The perturbations inject common misspellings and grammatical errors often found in real-world natural language processing data including random character or word deletion, swapping, and insertion (Moradi & Samwald, 2021). Figure 4 shows the accuracy differences between the new models `m2'`, `m3'`, . . ., `m10'` and the original models `m2`, `m3`, . . ., `m10` for all 9 GLUE tasks with 9 unique data perturbations per task (each data point represents a unique task and perturbation). For most



(a) $G2$.



(b) $G5$.

Figure 4: Accuracy difference between models produced by MGit's automated model updating feature and base models for various GLUE tasks.

perturbations and GLUE tasks, the new models are more robust (accuracy difference > 0).

6.5. User Study

We conducted an IRB-approved user study with 20 participants (Computer Science graduate students), to test whether users are more productive debugging and updating models using MGit compared with using regular model APIs such as Hugging Face's `Transformers` library. Users were randomly and evenly assigned to two groups: a control group that uses `Transformers` APIs and a test group that uses MGit. We tried to ensure that proficiency in the `Transformers` library, age and gender were equally distributed between the two groups (shown in Table 6 in Appendix §D). Both groups were first provided a tutorial on using APIs for loading, training and inference NLP models, and were then asked to perform model debugging and updating tasks. At the end of the study, both groups were introduced to the concepts underlying MGit, and were asked to rate its (estimated) helpfulness in solving these tasks (on a scale from 0 to 10).

Model debugging. In this task, users were first shown a bug in a language model. We used a *real-world bug* taken from 66 related issues in the `Transformers` repository (Hugging Face `Transformers`, d), where a particular sub-group of models return `NaN` in their output when loaded in `fp16` precision. A detailed discussion of this bug is in Appendix §D. The participants were then asked to find as many buggy models as possible from a pool of 91 models in 15 minutes. MGit users were able to zero in on buggy models that share common structure quickly by leveraging the lineage graph, while control users were forced to rely on random sampling. As a result, the average numbers of buggy models found by MGit users and `Transformers` users (out of 8) was 5.3 and 0.5, respectively. We ran a t-test between the numbers of buggy models found by each group and the p-value is $3.6 \times 10^{-5} (< 0.05)$ which demonstrates statistical significance. The average helpfulness rating of MGit and its traversal API was 8.6 and 9.2 in the control and test group respectively.

Model updating. In this task, users were first shown how various perturbations can decrease the evaluation accuracy of two models (a parent MLM model `m1` and a sequence classification model `m2`). Users were also shown how to build a model `m1'` robust to various perturbations by fine-tuning `m1` with perturbed data. Finally, users were asked to build a robust `m2'`, similar to the setup in $G2$ in §6.4. We observe that all MGit users used the `run_update_cascade` API to efficiently produce `m2'`. On the other hand, only four control users accomplished the task in the time constraint of 40 minutes. These users used one of three different methods: (1) manually adapting `m2'` from `m1'`, (2) manually constructing a perturbed dataset and fine-tuning `m2` on it, and (3) further tuning `m2` on the unperturbed dataset it was trained with. We measured the average time taken by two groups of users on this task (we conservatively mark incompletes with a time "score" of 40 minutes). The control group took 35.7 minutes on average, while the test group took 11.3 minutes; MGit users completed the task 3× faster on average. This has a p-value of $5.8 \times 10^{-7} (< 0.05)$. The average helpfulness rating of MGit and its `run_update_cascade` API was 9.2 and 9.9 in the control and test group respectively.

7. Related Work

We now briefly discuss other work related to MGit and the model management problem it tackles.

Model repositories and versioning systems. Similar to MGit, ModelHub (Miao et al., 2016) provides a `git`-like interface for managing models. However, ModelHub is not intended for derived ML models, and also does not present solutions for automated model updating, testing, and collaboration. Hugging Face Model Hub (Hugging Face, b; Wolf et al., 2019) is a widely-used model repository where users can upload their trained models; however, it does not record provenance information. MLflow (Zaharia et al., 2018), ModelDB (Vartak et al., 2016) and DVC (Barrak et al., 2021) provide ways to understand ML model performance (by tracking the code, data and hyperparameters used to create a model), and also deploy models in downstream applications. These systems also keep track of models' version IDs. However, beyond adding version IDs, these systems do not track relationships between models.

Git-Theta (Kandpal et al., 2023) is a Git extension designed for multi-user model collaboration. While Git-Theta partially supports the collaboration use case (i.e., `merge`) considered in our paper by providing a Git-like interface, and has some supports for efficient storage (only using locality-sensitive hashing, not with additional delta compression like in MGit), there are some key differences between the systems. Git-Theta focuses more on the versioning of individual models, optimizing for detailed parameter-level tracking and management of changes within a single model. In contrast, MGit's scope is broader, aiming to manage a network of models. MGit emphasizes the relationships and lineage among different models, making it more suitable for scenarios where multiple models are derived from each other or share components. MGit proposes a unifying set of abstractions, most importantly the lineage graph and methods to traverse, mutate and access it, that can be used by ML developers to efficiently debug and update models that are related to each other; Git-Theta cannot perform these important functions.

Debugging in ML. Testing and debugging is crucial for deploying ML models. Checklist (Ribeiro et al., 2020) observes many state-of-the-art models often exhibit bugs and offers test templates for NLP models. AdaTest (Ribeiro & Lundberg, 2022) recommends test cases using a ML model, and MLEXray (Qiu et al., 2022) helps monitor and debug edge-deployed models. Model assertions (Kang et al., 2020) identify invariants that should hold for model outputs corresponding to related inputs and correct discrepancies.

After finding undesired behaviors in models, ideally we would like to make *scoped* changes, updating only what's necessary without altering other already desired behaviors, without having to retrain from scratch. AdaTest (Ribeiro & Lundberg, 2022) aids in generating new training data with a human-in-the-loop for NLP model bug fixing. External learned editors (Cao et al., 2021; Mitchell et al., 2021; Hase et al., 2021) can adjust raw fine-tuning gradients to scope changes. MGit provides a framework to automatically identify and update models that might depend on a buggy model, obviating the need to manually update them.

Data provenance. MGit does not explicitly track *changes* in datasets (Wonsil et al., 2023). Fine-grained tracking of datasets and their effect on the quality of models in the lineage graph is exciting future work.

8. Conclusion

ML models are increasingly derived from prior models. This greatly complicates modern ML workflows: diagnosing and updating models is more challenging than ever on account of these dependencies. In this paper, we propose a system called MGit that tries to ease this burden using a lineage graph that records dependencies between models and abstractions over the lineage graph that facilitate better testing, updating, collaboration and inference. MGit's storage optimizations reduce the model storage footprint by up to 7×. We also found MGit improves user productivity: in a user study, users were able to patch a buggy model on average 3× faster with MGit. Our implementation is open sourced at `https://github.com/msr-fiddle/mgit`.

Impact Statement

This paper presents a new model versioning and management system. MGit can improve the reliability of machine learning by facilitating robust version control and model management, ensuring accurate tracking of updates and modifications, potentially leading to more stable and trustworthy AI systems.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 265–283, 2016.

Ba, J. and Caruana, R. Do Deep Nets Really Need to be Deep? *Advances in Neural Information Processing Systems*, 27, 2014.

Barrak, A., Eghan, E. E., and Adams, B. On the Co-evolution of ML Pipelines and Source Code - Empirical Study of DVC Projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 422–433, 2021.

Baylor, D., Haas, K., Katsiapis, K., Leong, S., Liu, R., Menwald, C., Miao, H., Polyzotis, N., Trott, M., and Zinkevich, M. Continuous Training for Production ML in the TensorFlow Extended (TFX) Platform. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, pp. 51–53, 2019.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. On the Opportunities and Risks of Foundation Models. *arXiv preprint arXiv:2108.07258*, 2021.

Cao, N. D., Aziz, W., and Titov, I. Editing factual knowledge in language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP 2021)*, pp. 6491–6506, 2021.

Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabharwal, A., Schoenick, C., and Tafjord, O. Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

Du, N., Huang, Y., Dai, A. M., Tong, S., Lepikhin, D., Xu, Y., Krikun, M., Zhou, Y., Yu, A. W., Firat, O., et al. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. In *International Conference on Machine Learning*, pp. 5547–5569. PMLR, 2022.

Fedus, W., Zoph, B., and Shazeer, N. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., and Leahy, C. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv:2101.00027*, 2021.

Google. The `bfloat16` Numerical Format. https://cloud.google.com/tpu/docs/bfloat16.

Guo, P., Hu, B., and Hu, W. Mistify: Automating DNN Model Porting for On-Device Inference at the Edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 705–719, 2021.

Hao, W., Awatramani, A., Hu, J., Mao, C., Chen, P.-C., Cidon, E., Cidon, A., and Yang, J. A Tale of Two Models: Constructing Evasive Attacks on Edge Models. *Proceedings of Machine Learning and Systems*, 4:414–429, 2022.

Hase, P., Diab, M., Celikyilmaz, A., Li, X., Kozareva, Z., Stoyanov, V., Bansal, M., and Iyer, S. Do Language Models Have Beliefs? Methods for Detecting, Updating, and Visualizing Model Beliefs, 2021.

He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017.

Hu, Z., Zou, X., Xia, W., Jin, S., Tao, D., Liu, Y., Zhang, W., and Zhang, Z. Delta-DNN: Efficiently Compressing Deep Neural Networks via Exploiting Floats Similarity. In *49th International Conference on Parallel Processing-ICPP*, pp. 1–12, 2020.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely Connected Convolutional Networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Hugging Face. Mixed Precision for `bfloat16`-pretrained Models. https://discuss.huggingface.co/t/

mixed-precision-for-bfloat16-pretrained-models/5315, a.

Hugging Face. Hugging Face Model Hub. https://huggingface.co/models, b.

Hugging Face. T5 fp16 forward yields NaN. https://discuss.huggingface.co/t/mixed-precision-for-bfloat16-pretrained-models/5315, c.

Hugging Face Transformers. Clamping hidden state values to allow fp16. https://github.com/huggingface/transformers/pull/19229, a.

Hugging Face Transformers. Fix T5 inference in fp16 + bnb Error. https://github.com/huggingface/transformers/pull/21281, b.

Hugging Face Transformers. Enable T5 fp16. https://github.com/huggingface/transformers/pull/9487, c.

Hugging Face Transformers. ALL issues related to T5 outputing NaN. https://github.com/huggingface/transformers/issues?q=is%3Aissue+T5+NAN, d.

Igor, P. The Algorithm: Lempel-Ziv-Markov Chain. 1998.

Kandpal, N., Lester, B., Muqeeth, M., Mascarenhas, A., Evans, M., Baskaran, V., Huang, T., Liu, H., and Raffel, C. Git-Theta: A Git Extension for Collaborative Development of Machine Learning Models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Kang, D., Raghavan, D., Bailis, P., and Zaharia, M. Model Assertions for Monitoring and Improving ML Models. *Proceedings of Machine Learning and Systems*, 2:481–496, 2020.

Karpathy, A. PyTorch at Tesla. https://www.youtube.com/watch?v=oBklltKXtDE, 2019.

Konečnỳ, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., and Bacon, D. Federated Learning: Strategies for Improving Communication Efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

Li, J., Zhao, L., Yang, Y., Zhan, K., and Li, K. Tetris: Memory-Efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

Li, T., Sahu, A. K., Talwalkar, A., and Smith, V. Federated Learning: Challenges, Methods, and Future Directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

Lin, S. C., Hilton, J., and Evans, O. TruthfulQA: Measuring How Models Mimic Human Falsehoods. In *Annual Meeting of the Association for Computational Linguistics*, 2021.

Liu, B. Lifelong Machine Learning: A Paradigm for Continuous Learning. *Frontiers of Computer Science*, 11(3): 359–361, 2017.

Miao, H., Li, A., Davis, L. S., and Deshpande, A. ModelHub: Towards Unified Data and Lifecycle Management for Deep Learning. *arXiv preprint arXiv:1611.06224*, 2016.

Mitchell, E., Lin, C., Bosselut, A., Finn, C., and Manning, C. D. Fast Model Editing at Scale. *arXiv preprint arXiv:2110.11309*, 2021.

Moradi, M. and Samwald, M. Evaluating the Robustness of Neural Language Models to Input Perturbations. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 1558–1570. Association for Computational Linguistics, 2021.

Padmanabhan, A., Agarwal, N., Iyer, A., Ananthanarayanan, G., Shu, Y., Karianakis, N., Xu, G. H., and Netravali, R. Gemel: Model Merging for Memory-Efficient, Real-Time Video Analytics at the Edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 973–994, 2023.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 32, 2019.

Polino, A., Pascanu, R., and Alistarh, D. Model Compression via Distillation and Quantization. *arXiv preprint arXiv:1802.05668*, 2018.

Pratt, L. Y. Discriminability-based Transfer between Neural Networks. *Advances in Neural Information Processing Systems*, 5, 1992.

Qiu, H., Vavelidou, I., Li, J., Pergament, E., Warden, P., Chinchali, S., Asgar, Z., and Katti, S. ML-EXray: Visibility into ML Deployment on the Edge. *Proceedings of Machine Learning and Systems*, 4:337–351, 2022.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21 (140):1–67, 2020.

Rebuffi, S.-A., Bilen, H., and Vedaldi, A. Learning Multiple Visual Domains with Residual Adapters. *Advances in Neural Information Processing Systems*, 30, 2017.

Reed, J., DeVito, Z., He, H., Ussery, A., and Ansel, J. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. *Proceedings of Machine Learning and Systems*, 4:638–651, 2022.

Ribeiro, M. T. and Lundberg, S. Adaptive Testing and Debugging of NLP Models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3253–3267, Dublin, Ireland, 2022. Association for Computational Linguistics.

Ribeiro, M. T., Wu, T., Guestrin, C., and Singh, S. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4902–4912, Online, 2020. Association for Computational Linguistics.

Robinson, A. and Cherry, C. Results of a Prototype Television Bandwidth Compression Scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967.

Ruder, S. An Overview of Multi-Task Learning in Deep Neural Networks. *arXiv preprint arXiv:1706.05098*, 2017.

Torrey, L. and Shavlik, J. Transfer Learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pp. 242–264. IGI global, 2010.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open Foundation and Fine-tuned Chat Models. *arXiv preprint arXiv:2307.09288*, 2023.

Vartak, M., Subramanyam, H., Lee, W.-E., Viswanathan, S., Husnoo, S., Madden, S., and Zaharia, M. A. ModelDB: A System for Machine Learning Model Management. In *HILDA '16*, 2016.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. *arXiv preprint arXiv:1804.07461*, 2018.

Weiss, K., Khoshgoftaar, T. M., and Wang, D. A Survey of Transfer Learning. *Journal of Big Data*, 3(1):1–40, 2016.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Huggingface's Transformers: State-of-the-Art Natural Language Processing. *arXiv preprint arXiv:1910.03771*, 2019.

Wonsil, J., Sullivan, J., Seltzer, M., and Pocock, A. Integrated Reproducibility with Self-describing Machine Learning Models. In *Proceedings of the 2023 ACM Conference on Reproducibility and Replicability*, pp. 1–14, New York, NY, USA, 2023.

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., et al. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

Zaken, E. B., Ravfogel, S., and Goldberg, Y. BitFit: Simple Parameter-Efficient Fine-Tuning for Transformer-Based Masked Language Models. *arXiv preprint arXiv:2106.10199*, 2021.

Zhou, Z., Wei, X., Zhang, J., and Sun, G. PetS: A Unified Framework for Parameter-Efficient Transformers Serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 489–504, 2022.

A. Additional Details on MGit's Design and Abstractions

In this section, we provide additional details on MGit's design and the abstractions it exposes to applications.

A.1. Components and API

Table 4 shows MGit's API. This API allows both mutation of the lineage graph and traversals over the models in it (with functionality that supports running tests and updating models as a part of a traversal).

A.2. Creation Functions with Multi-Task Learning

MGit supports the creation of models with shared parameters through a custom creation function:

```
class CreationFunctionMultiTaskLearning:
  def __init__(self):
    self.lg = mg.LineageGraph(<filepath>)
    self.data_loader = torch.DataLoader(<filepath>)

  def initialize_model(self, parent_list):
    self.child_model = Model()
    parent = parent_list[0]
    sibling = parent.get_children()[0]
    if sibling != self:
      // Share parameters with siblings.
      self.lg.share_parameters(sibling.get_model(),
           self.child_model)
    else:
      // Copy parameters from parent_model.
      self.lg.copy_parameters(parent.get_model(),
           self.child_model)
    // Randomly initialize head.
    self.child_model.head = random_initialization(
         child_head_dimensions)
```

A.3. Model Updating

Algorithm 1 shows pseudocode for the MGit's cascading model update functionality, where new versions of models can be created in response to the creation of a new version of an upstream model.

Algorithm 1 Pseudocode for model updating.

```
def run_update_cascade(m, m', skip_fn, terminate_fn):
  // Create new versions of all children of m, given
  // update of m to m'.

  // First, create (empty) next versions of models.
  skip_fn2 = lambda x: skip_fn(x) or x == m
  for x in BFS(m, skip_fn2, terminate_fn):
    // Get next version of each parent of x if it
    // exists, otherwise get current version.
    ps' = [get_next_version(p) for p in x.parents]
    x' = x.cr.initialize(ps')

    // Add provenance and version edges, and copy
    // creation function.
    add_edge(p', x') for p' in ps'
    add_version_edge(x, x')
    x'.cr = x.cr

  // Next, start traversal at children of m', and
  // train models by calling creation function.
  // traversal_all_parents_first returns an iterator
  // over nodes (or group of nodes if using MTL) such
  // that a node is visited only once _all_ of its
  // parents (parent MTL groups if using MTL) are
  // visited.
  skip_fn2 = lambda x: skip_fn(x) or x == m'
  for xs' in traversal_all_parents_first(m', skip_fn2
      , term_fn):
    if isinstance(xs', list):
      // Run MTL using combined creation function.
      merged_cr(xs', xs'.parents)
    else:
      // Otherwise, call individual model node's
      // creation function.
      [x'] = xs'
      x'.cr(x'.parents)
```

API Name	Description
`add_node(x, xn, [optional] cr)`	Adds a model `x` as a node to the lineage graph with name `xn`. A creation function `cr` can be optionally specified.
`add_edge(x, y)`	Adds a provenance edge between nodes `x` and `y`. Calls `add_node(x)` and `add_node(y)` if nodes `x` and `y` do not already exist.
`add_version_edge(x, y)`	Adds a versioning edge between nodes `x` and `y`. `x` and `y` must have the same model type. Calls `add_node` if `x` and `y` do not exist.
`remove_edge(x, y, type)`	Removes provenance or versioning (specified by `type`) between node `x` and `y`.
`remove_node(x)`	Removes node `x` and its sub-tree from the lineage graph. Calls `remove_edge` on `x` and all of its parents and all edge types.
`register_creation_function(x, cr)`	Registers a creation function `cr` for node `x`. The creation function specifies how the model `x` should be created from its parents. The creation function can also be used to specify MTL groups.
`register_test_function(t, tn, [optional] x, [optional] mt)`	Registers a test `t` with name `tn` either for a specific model `x` or for all models of type `mt` (only one of `x` or `mt` should be specified).
`deregister_test_function(tn, [optional] x, [optional] mt)`	De-registers a test with name `tn` either for a specific model `x` or for all models of type `mt` (only one of `x` or `mt` should be specified).
`traversal()`	Returns an iterator of individual nodes or a group of nodes encountered in a traversal. An example traversal is `BFS`.
`get_next_version(x)`	Returns the next version of model `x` if it exists.
`merge(x1, x2)`	Try to automatically merge the models pointed to by `x1` and `x2`; if not possible, manually request conflict resolution from user.
`run_tests(i, [optional] re)`	Runs all registered tests matching the specified optional regex `re` on all nodes returned by the iterator `i`.
`run_function(i, f)`	Runs function `f` (e.g., compute the parameter norm of the model) on all nodes returned by the iterator `i`.
`run_update_cascade(m, m', skip_fn, terminate_fn)`	Trigger update cascade as a result of the model update $m \rightarrow m'$. Nodes are visited once all their parents are visited, starting from `m` with provided skip and termination functions. A new version of a model is created if it has a registered creation function `cr`.

Table 4: MGit API. We show both the lower-level API that can be used to access and mutate the lineage graph directly, as well as higher-level methods that provide more sophisticated functionality.

A.4. Implementation of Key Primitives

`diff` and `merge` are two key MGit primitives that power its automated graph construction algorithm and its collaboration functionality.

Diff primitive. Algorithm 2 shows full pseudocode for MGit's `diff` primitive.

Merge primitive. MGit's `merge` primitive helps support collaboration use cases, where multiple users might make "edits" to the same model concurrently. `merge` is given two models (models created by concurrent edits) and their closest common ancestor (the original model on which changes were made concurrently) in the lineage graph as input. It returns three possible results:

- **Conflict.** At least one common layer is updated by both changes. In this case, manual intervention is required.
- **Possible conflict.** Two layers changed by different users have a "dependency". Consequently, additional tests are needed to verify that the concurrent changes did not result in a model regression.
- **No conflict.** No common layer updated by both users, and no dependency between the any two of the users' changes. In this case, the merge can be processed automatically.

The decision tree shown in Figure 2 summarizes the conflict detection approach implemented in the `merge` primitive. Let `m1` and `m2` be the models created by different users starting from model `m`. Then the above checks can be performed by first computing `d1 = diff(m, m1)` and `d2 = diff(m, m2)`, and then performing a DFS through the models to check for dependencies between the changed layers.

Algorithm 2 Peudocode for `diff` between two models $m1$ and $m2$.

```
def module_diff(m1, m2):
  // m1 and m2 are DAG representations of the input
  // models. DAG nodes are torch.nn.module layers
  // (e.g.,Linear, Conv2D). An edge between two nodes
  // indicates dataflow.  We want to compute the diff
  // (the nodes and  edges to remove and add to
  // convert m1 to m2).

  // Compute hash tables of nodes/ edges for m1 and
  // m2 where values are node / edge lists sorted in
  // topological order. The hash of an edge is the
  // hash of its end points.
  N1, E1 = generate_hash_table(m1)
  N2, E2 = generate_hash_table(m2)

  // Iterate over E1: if a hash exists in E2,
  // greedily match each edge in two edge lists.
  // Before deciding on a matching, check the nodes
  // in these edges and only commit when
  // corresponding nodes have the same matched
  // status. Matching a node in m1 with more than one
  // node in m2 is not allowed.
  Matches_N, Matches_E = {}, {}
  for hash in E1:
    es1 = E1[hash], es2 = E2[hash]
    for e1 in es1:
      for e2 in es2:
        if check(e1, e2):
          e1[0].matched, e1[1].matched = True, True
          e2[0].matched, e2[1].matched = True, True
          Matches_N.add((e1[0],e2[0]), (e1[1],e2[1]))
          Matches_E.add((e1,e2))
          E2[hash].drop(e2)
      es2 = E2[hash]

  // Match nodes that do not belong to common edges.
  for hash in N1:
    ns1 = [n1 in N1[hash] if n1.matched = False]
    ns2 = [n2 in N2[hash] if n2.matched = False]
    for i in range(min(len(ns1, ns2))):
        ns1[i].matched, ns2[i].matched = True, True
        Matches_N.add((ns1[i], ns2[i]))

  // Sort Matches_N/E by topological order of nodes /
  // edges in m1 and remove inverse matches.
  // E.g., A-B-A-C and A-B-C-A should have a node
  // matching of only {A, B, C (or A)}.
  Matches_N = filter(sort(Matches_N))
  Matches_E = filter(sort(Matches_E))

  // Add_N/E are the unmatched nodes / edges in m2.
  // Del_N/E are the unmatched nodes / edges in m1.
  Add_E = E2.difference(e2 in Matches_E)
  Del_E = E1.difference(e1 in Matches_E)
  Add_N = N2.difference(n2 in Matches_N)
  Del_N = N1.difference(n1 in Matches_N)

  return Add_E, Add_N, Del_E, Del_N
```

Algorithm 3 Pseudocode for delta compression.

```
def delta_compression(m2, m1, t_thr):
  // m1 and m2 are the parent and child models. We
  // want to compress m2 - m1.
  // t_thr is a user-configurable test accuracy
  // threshold. If the model m2' after  compression
  // has an accuracy difference larger than t_thr,
  // model compression is rejected.

  // First, run LCS to find a mapping between
  // parameters of the same shape.
  (P1, P2) = lcs(m1, m2)

  // Calculate quantized deltas between parameter
      sets.
  D = quantize(P1, P2)

  // Compressor performs lossless compression.
  // Possible options are RLE, LZMA, etc.
  CD, storage_saving = compressor(D)
  if storage_saving < 1:
    return False, None, m2
  else:
    P2' = dequantize(D, P1)

    // Restore parameters that are not compressed.
    m2' = m2.difference(P2).union(P2')
    if (run_tests(m2) - run_tests(m2')) / run_tests(
        m2) > t_thr:
      return False, None, m2
    else:
      return True, CD, m2'
```

A.5. Delta Compression

Algorithm 3 shows full pseudocode for MGit's delta compression algorithm, used to optimize the storage footprint of the parameters of models in a lineage graph.

B. Descriptions of Lineage Graphs

In this section, we describe the lineage graphs used in our evaluation in more detail.

G1. $G1$ is a lineage graph created from NLP models downloaded directly from the HuggingFace model hub (Hugging Face, b). The full list of models used in $G1$ is:

- `bert-base-cased`
- `bert-base-uncased`
- `aloxatel/bert-base-mnli`
- `ericRosello/bert-base-uncased-fine-tuned-squad-frozen-v2`
- `deepset/bert-base-uncased-squad2`
- `bert-large-uncased`
- `bert-large-cased`
- `TehranNLP-org/bert-large-mnli`
- `roberta-base`
- `deepset/roberta-base-squad2`
- `textattack/roberta-base-MNLI`
- `roberta-large`
- `roberta-large-mnli`
- `deepset/roberta-large-squad2`
- `albert-base-v2`
- `twmkn9/albert-base-v2-squad2`
- `prajjwal1/albert-base-v2-mnli`
- `distilbert-base-uncased`
- `distilbert-base-cased`
- `twmkn9/distilbert-base-uncased-squad2`
- `ericRosello/distilbert-base-uncased-fine-tuned-squad-frozen-v2`
- `google/electra-small-generator`
- `howey/electra-small-mnli`

We then ran MGit's automated graph construction method on these models to create a lineage graph. As noted earlier, 22 out of 23 nodes are correctly inserted relative to a "gold" lineage graph. The only mis-inserted model is `bert-base-uncased`. The automated graph construction function is able to correctly insert models, including some that have frozen weights inherited from their parent model, by computing structural and contextual divergence scores between model pairs.

G2. We started with a vanilla RoBERTa model trained on the standard masked language modeling (MLM) objective, and then fine-tuned task-specific models for each of the GLUE tasks (Wang et al., 2018). We created 10 versions of each task-specific model by fine-tuning on additional perturbed data (Moradi & Samwald, 2021).

G3. We trained a ResNet-50 image classification model (He et al., 2016) on the ImageNet-1K dataset (Deng et al., 2009) using federated learning. Each worker operates on a data silo with a subset of the 1000 labels in the ImageNet-1K dataset. We ran experiments with 40 workers (data silos), and 10 rounds of federated averaging. In each round, 5 of 40 workers are randomly sampled.

G4. To create models that can be deployed on the edge, we pruned three image classification models to varying degrees: ResNet-50, DenseNet121 (Huang et al., 2017) and MobileNet-v3 (Howard et al., 2017). For each model architecture, we create models progressively greater sparsities in a two-step process. In the first step, a model with sparsity s_i is created by masking out the s_i fraction of its non-zero parameters with lowest magnitude. We then check if the resulting model is accurate enough, and if not, we fine-tune the model on ImageNet-1K to further improve accuracy while preserving its sparsity.

G5. We use MTL to create RoBERTa models for GLUE tasks with shared weights. This is similar to $G2$.

G6. We use the last 10 Pythia check-

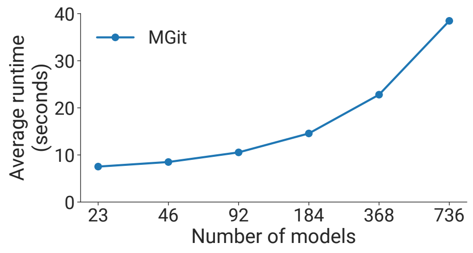


Figure 5: Average per-model insertion time for lineage graphs of different sizes.

points released from EleutherAI, i.e., `EleutherAI/pythia-6.9b/revision/step134000` through `EleutherAI/pythia-6.9b/revision/step144000`.

C. Additional Results

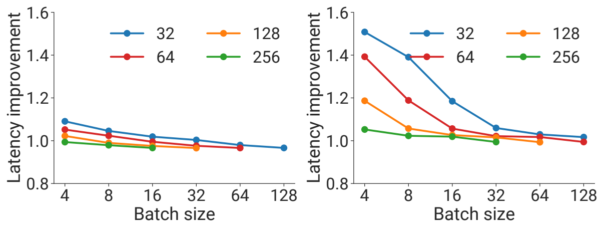
In this section, we show results on the scalability of the lineage graph auto-insertion algorithm, and the impact of merging similar layers on inference latency and accuracy in the collocated model inference setting.

C.1. Scalability Evaluation of Auto-Insertion

Figure 5 shows the average insertion time per-model for lineage graphs of different sizes when using the auto-insertion algorithm described in §3.2. We create larger graphs by scaling up $G1$ (§6.1) by a desired factor: for example, our graph with 92 models or nodes is created by replicating each model in $G1$'s model pool 4 times. "Auto-inserting" a model into the lineage graph involves a pair-wise comparison with all other models already in the lineage graph followed by `add_node` and `add_edge` operations (both time complexity $O(1)$); consequently, the average per-model runtime increases quadratically with the size of the graph. We believe that with large lineage graphs with hundreds of models, average insertion times of 40 seconds / model are reasonable, especially when compared to model training time which is often many hours or days.

C.2. Collocated Model Inference

When serving multiple model derivatives on the same host (i.e., model collocation), MGit's `diff` primitive can be used to automatically identify shared parameters between models. At inference, inputs to the shared layers can be batched, which increases arithmetic efficiency and thus also throughput. The higher throughput can result in lower latencies compared to sequential execution. For sets of models which have no shared parameters, MGit can identify pairs of layers with the same structure and small L2 distance; these layers can then be merged into a single layer. Merging layers may lower inference accuracy.



(a) 30% collocation. (b) 90% collocation.

Figure 6: Improvement in latency of generating next token when using collocated models compared with sequential execution. Different sequence lengths are shown in different colors.

To illustrate the trade-off between accuracy and efficiency with model collocation, we show the accuracy versus percentage of merged layers when collocating `Llama-2-7b` with `Llama-2-chat-7b` (Touvron et al., 2023) in Table 5 on the ARC (Clark et al., 2018), TruthfulQA (Lin et al., 2021), and Pile (Gao et al., 2021) datasets. As the percentage of merged layers increases, the accuracy degrades. Without collocation, `Llama-2-7b` performs better on the Pile dataset but worse on the ARC and TruthfulQA dataset compared to `Llama2-chat-7b`. With 90% model collocation, `Llama-2-7b` performs better on the ARC and the Pile dataset but worse on the TruthfulQA dataset compared with `Llama-2-chat-7b`.

To show the benefit of merging similar layers on collocated inference, Figure 6 plots the average speedup (computed as ratio of latencies) versus batch size for different percentages of merged layers and input sequence lengths when collocating the same two models. Model collocation achieves a latency improvement of up to 1.5× when the percentage of merged layers is 90% and the input sequence length is 32 with a batch size of 4. The speedup plateaus to 1 as the percentage of merged layers decreases and the input sequence length and the aggregated batch size increases. This is because the baseline of running models sequentially becomes more efficient as the problem size along various dimensions increases. The memory consumption of model merging saves 0.5%, 16.2% and 39.3% GPU memory when the percentage of shared layers are 30%, 60% and 90% respectively.

D. Additional Details on the User Study

Table 6 has statistics on the users who participated in our user study.

The bug we asked users to find in the first task of the user study is commonly found in `T5` models (Raffel et al., 2020) but rarely in others; these `T5` models were pre-trained with a mixed-precision recipe of `bf16` (Google) and `fp32` but are commonly loaded for inference in `fp16` which can cause

Metric	Dataset	30% collocation (%)	60% collocation (%)	90% collocation(%)
ac_norm ↑	ARC	+0.4	+2	+2.9
mc ↑	TruthfulQA	-0.6	+1	+5.2
bpb ↓	Pile (Arxiv)	+0.9	+2.3	+12.2

Table 5: Change in model accuracy on three datasets as the collocation percentage increases.

overflow (Hugging Face, a). This bug was first reported from a `T5` model user on 11/5/2020 (Hugging Face, c) and was fixed multiple times between 2020 and 2023 (Hugging Face `Transformers`, b). During these three years, the bug was fixed in multiple `T5` derivative models like `T5`, `T5v1.1` and `longT5`. However, since model lineage was not tracked, the exact same patches needed to be applied multiple times on the other `T5` models: a PR patching `T5-base` models was merged on 1/8/2021 (Hugging Face `Transformers`, c) and a PR patching `longT5` was merged on 9/28/2022 (Hugging Face `Transformers`, a) due to the same issue. Given an example of one buggy model, users should be able to quickly identify other buggy models and apply patches efficiently. However, for the simplicity of the user study, we only ask the users to find as many buggy models as possible without constructing patches.

When selecting the models in this task, we randomly picked eight `T5` models from HuggingFace that show this buggy behaviour. We pick 83 other non-buggy models from $G1$ randomly to try to keep the proportion of `T5` models in the pool roughly the same as in the Hugging Face model repository.

ID	Age	Gender	Proficiency with `Transformers`	Number of found buggy models	Time to patch model (mins)	Helpfulness rating
$A0$	24	Female	Intermediate	1	40	8 / 9
$A1$	22	Female	Intermediate	1	40	9 / 8
$A2$	25	Female	Intermediate	0	33.2	9 / 10
$A3$	22	Male	Intermediate	0	38.9	7 / 8
$A4$	25	Male	Proficient	1	22.2	10 / 10
$A5$	22	Male	Intermediate	1	40	9 / 9
$A6$	24	Male	Intermediate	0	40	8 / 9
$A7$	25	Male	Proficient	1	40	8 / 9
$A8$	24	Male	Proficient	0	40	8 / 10
$A9$	24	Male	Intermediate	0	22.7	10 / 10
$B0$	26	Female	Intermediate	7	7.1	8 / 9
$B1$	26	Female	Intermediate	3	8.8	10 / 10
$B2$	25	Female	Proficient	7	15.9	10 / 10
$B3$	23	Male	Intermediate	4	11	9 / 10
$B4$	23	Male	Intermediate	5	12.6	10 / 10
$B5$	22	Male	Proficient	7	15	8 / 10
$B6$	24	Male	Intermediate	7	9.8	10 / 10
$B7$	25	Male	Proficient	7	11.3	9 / 10
$B8$	24	Male	Intermediate	1	11.3	9 / 10
$B9$	25	Male	Proficient	5	9.8	9 / 10

Table 6: Statistics of users who participated in our user study. Users $A0$ through $A9$ were in the control group, while users $B0$ through $B9$ were in the test group.