MetaStore: Analyzing Deep Learning Meta-Data at Scale

Huayi Zhang
WPI, Data Science
Worcester, MA
zhanghuayi01@gmail.com

Binwei Yan
MIT
Cambridge, MA
bineva@mit.edu

Lei Cao
U of Arizona, CS; MIT, CSAIL
Cambridge, MA
lcao@csail.mit.edu

Samuel Madden
MIT, CSAIL
Cambridge, MA
madden@csail.mit.edu

Elke Rundensteiner
WPI, Computer Science
Worcester, MA
rundenst@cs.wpi.edu

ABSTRACT

The process of training deep learning models produces a huge amount of meta-data, including but not limited to losses, hidden feature embeddings, and gradients. Model diagnosis tools have been developed to analyze losses and feature embeddings with the aim to improve the performance of these models. However, gradients, despite carrying rich information that is potentially relevant for model interpretation and data debugging, have yet to be fully explored due to their size and complexity. Each single gradient has a size as large as the number of parameters of the neural net – often measured in the tens of millions. This makes it extremely challenging to efficiently collect, store, and analyze large numbers of gradients in these models. In this work, we develop MetaStore to fill this gap. MetaStore leverages our observation that storing certain compact intermediate results produced in the back propagation process, namely, the prefix and suffix gradients, is sufficient for the exact restoration of the original gradient. These prefix and suffix gradients are much more compact than the original gradients, thus allowing us to address the gradient collection and storage challenges. Furthermore, MetaStore features a rich set of analytics operators that allow the users to analyze the gradients for data debugging or model interpretation. Rather than first having to restore the original gradients and then run analytics on top of this decompressed view, MetaStore directly executes these operators on the compact prefix and suffix structures, making gradient-based analytics efficient and scalable. Our experiments on popular deep learning models such as VGG, BERT, and ResNet and benchmark image and text datasets demonstrate that MetaStore outperforms strong baseline methods from 4 to 678x in storage costs and from 2 to 1000x in running time.

PVLDB Reference Format:
Huayi Zhang, Binwei Yan, Lei Cao, Samuel Madden, and Elke Rundensteiner. MetaStore: Analyzing Deep Learning Meta-Data at Scale. PVLDB, 17(6): 1446 - 1459, 2024.
doi:10.14778/3648160.3648182

Proceedings of the VLDB Endowment, Vol. 17, No. 6 ISSN 2150-8097.
doi:10.14778/3648160.3648182

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Mazic4/MetaStore/.

1 INTRODUCTION

Background and Motivation. The training process of deep neural networks (DNNs) produces a massive amount of meta-data, including feature embeddings [39], losses [34], and gradients [31]. This meta-data holds significant value that can be leveraged for many tasks critical to achieving superior model performance. These tasks include but are not limited to cleaning noise in the training data, explaining the behavior of trained models, or reusing and fine-tuning models. For example, research [8, 20, 29, 39, 52] has shown that analyzing training loss and feature embeddings can explain inference results and help debug DNN models. To address this need, we develop a system called **MetaStore**, that collects, stores, and analyzes such meta-data at scale.

Promise of Gradient Meta-Data. In this paper, we focus on one particular type of meta-data: gradients. DNNs train models using a sequence of *gradient descent* steps which gradually fit the model parameters to the training data. Thus, as the bridge between the data and the model, these gradients can be used to effectively estimate the influence of training samples or hyper-parameters on the learned model parameters. For example, in the machine learning literature, many robust deep learning techniques use *gradients* [7, 23, 30, 31, 45–47, 53, 55] during DNN training to for example mitigate the impact of potential noise in the training examples and dynamically adjust the hyper-parameters such as learning rate.

Similarly, in our setting of offline meta-data analytics, if these gradients can be appropriately analyzed, they are of great value to solve the data issues in deep learning and explain the behavior of the models. In particular, we observe that *meta gradient* – the inner product between the gradients of a training sample and a set of testing samples – effectively measures how a training sample contributes to the model performance. A positive meta gradient indicates that it impacts the model in a positive way, and vice versa. With this meta gradient, we are able to discover the mislabeled samples in the training data, as they tend to contribute negatively to the performance of the model. Moreover, given a testing sample, we could explain why the model predicts it in the identified manner by finding a small number of training samples whose gradients have the largest inner product with that of the testing sample. Further, the training samples that contribute the most to the model could guide the collection of new training data to improve the performance of the model.

Performance Challenges. However, it is challenging to effectively collect, store, or analyze gradients, because the size of the gradient tends to be huge. DNNs are typically composed of many layers, including convolutional layers, linear layers, batch normalization, etc. A DNN computes gradients w.r.t. the training examples layer-by-layer. At each layer, the dimensionality of the gradient is equivalent to the number of trainable parameters in that layer. Many modern DNN models are huge, with up to billions of parameters [12].

As an example, in the well-known DNN models, such as ResNet [17] or VGG [36], a single linear layer can have 4096 × 4096 parameters. Given a CIFAR-10 dataset with 50,000 training samples, it would take about 3 TB of disk space to store the gradients produced in just one *single* linear layer. Worst yet, deep learning trains models epoch by epoch and thus produces this amount of gradients per epoch.

Therefore, merely *storing* this volume of gradients w.r.t. one (or worse yet all) layers quickly becomes infeasible. Even if we had sufficient (near infinite) storage resources for keeping all such gradient data for each round of training, the mere task of just *collecting* these gradients would be challenging itself. Directly logging the gradients produced by the DNN training in an online process would dramatically slow down the already exceedingly expensive training process. Moreover, loading a large amount of meta-data into memory for *analytics* would introduce *exorbitant I/O costs* during query execution. On the other hand, if we instead were to re-compute the gradients on-the-fly whenever a gradient analytics query is issued, this would cause *prohibitive query execution costs.* This is because computing a gradient from scratch effectively requires re-execution of the NN training pipeline.

Proposed Solution. By exploiting the properties of popular DNN models and their gradient computation methodology, our **MetaStore** effectively addresses the above challenges.

MetaStore Compact Data Storage. First, our analysis of the back-propagation process of DNN training reveals that the huge gradient of a training sample can be decomposed into 2 small gradients, namely, *prefix* and *suffix* gradients, from which the gradient can be *exactly* re-constructed via a matrix product operation. These two partial-gradients are typically several orders of magnitude smaller than the original gradient especially when produced in layers with a huge number of parameters.

MetaStore Lightweight Data Collection. Instead of first computing the full gradient and then manually decomposing it, we observe that both the small prefix and suffix gradients correspond to intermediate data that could naturally be produced during the back-propagation step when computing the gradient. Their collection can thus be done via a very lightweight process.

MetaStore Efficient Analytics. MetaStore is the first system to provide *a rich set of operators* that allow users to conduct many gradient-based analytics on the stored meta-data from discovering erroneous training samples to interpreting model behavior. These operators often involve computing the inner product similarity of two gradients (meta gradient). This inner product operation is computational expensive [47, 55] due to the high dimensionality of the gradients. We design an efficient strategy to exactly compute the inner product of two gradients directly on their respective prefix and suffix gradients. With the prefix and suffix gradients much smaller than the gradient itself, this speeds up the inner product operation by several orders of magnitude.

Contributions. In summary, our key contributions include:

- We design MetaStore, a system that enables a novel class of gradient-based analytics for model interpretation, data debugging, and data valuation.
- Leveraging the prefix and suffix gradients decomposition observation, MetaStore overcomes the critical data volume bottleneck in storing and analyzing gradients.
- We design efficient execution strategies to compute the inner-product similarity between gradients directly on top of the compact dual prefix/suffix gradient structures.
- Our experiments on popular benchmark datasets and a variety of pre-trained DNN model architectures demonstrate that MetaStore speeds up the query execution from 2 to 1000 fold and reduces the storage costs from 4 to 678 fold.

2 PRELIMINARIES

Here, we review the forward and backward propagation processes in DNN training to understand MetaStore's methodology. A DNN model $\phi(x;\theta)$ is formed by a stack of layers, with x being the input data sample, and θ being the parameters of the model.

Forward Propagation Process. During the inference time, also called forward propagation, the data samples are fed into the first layer. Then each layer takes the previous layer's outputs as its input and transforms the input features into new representations. Finally, the output of the last layer is considered the DNN model's output. The transformation process inside each layer typically is to multiply the input features with a set of parameters, called neurons. Each layer thus can be regarded as a function of the input features and its parameters, i.e.,$z^{l+1} = f^l(z^l, \theta^l)$, where z^l denotes the inputs (output) of the lth ($l-1$th) layer, and θ^l the parameters of the l^{th} layer. The overall DNN model corresponds to a function composition:

$$\hat{y} = \phi(x;\theta) = f^L(f^{L-1}\cdots(f^1(x;\theta^1)\cdots),\theta^L). \tag{1}$$

Backward Propagation Process. Deep learning uses back-propagation to train a DNN model. For this, the machine learning practitioners provide the expected outputs of each data sample, such as a label. They also define a loss function that produces a loss value based on the difference between the expected and actual outputs of the DNN model. Then, the gradients of each parameter with respect to the loss value are calculated to update the parameters in the DNN model. More specifically, the gradients of the parameters in each layer are calculated with the chain rule below:

$$\nabla_{\theta^l} C = \frac{dC}{d\theta^l} = \frac{dC}{dz^{l+1}} \cdot \frac{dz^{l+1}}{d\theta^l} = \frac{dC}{dz^L} \cdot \frac{dz^L}{dz^{L-1}} \cdots \frac{dz^{l+1}}{dz^l} \cdot \frac{dz^l}{d\theta^l} \tag{2}$$

where C is the loss value, $C = Loss(\hat{y}, y)$. An optimization method, typically Stochastic Gradient Descent (SGD), updates the parameters θ^l by taking one step of gradient descent:

$$\theta^l = \theta^l - \alpha \cdot \nabla_{\theta^l} C \tag{3}$$

where α is a predefined learning rate that controls the learning speed of the DNN model.

3 GRADIENT-BASED DNN ANALYTICS

In this section, we first discuss **meta gradient**, the foundation that most gradient-based DNN analytics techniques are built upon, and then introduce our core operators for gradient-based analytics.

3.1 Meta Gradient

In deep learning, optimization methods such as SGD directly use gradients to update the parameters of the DNN models.

Observation. *Meta gradient* – the inner product between the gradients of a training sample and a set of validation samples – effectively estimates how a training sample contributes to the model performance. Below, we theoretically show why this important observation is true.

Intuitively, the contribution of a training sample can be measured by how differently the model would perform if the target sample was not in the training set [30, 31, 45]. Let's consider a standard classification task. The DNN model $\phi(x;\theta)$ is evaluated on a set of validation samples $\{(x_j^v, y_j^v)\}_{j=1}^{N^v}$ that are not in the training set with y_j^v the label of x_j^v. We denote the validation loss as $\mathcal{L}^v(\theta) = \frac{1}{N^v}\sum_{j=1}^{N^v} l(x_j^v, y_j^v;\theta)$. Let θ_t be the parameters of the model that was trained *with* the target training sample x_t and θ be the parameters trained *without* using the target training sample x_t. Then the contribution of the training sample x_t corresponds to the difference between the validation losses of $\phi(x^v;\theta)$ and $\phi(x^v;\theta_t)$, i.e., $L^v(\theta_t) - L^v(\theta)$. With Taylor Expansion, this becomes:

$$L^v(\theta_t) - L^v(\theta) =< \nabla_\theta L^v(\theta), \theta_t - \theta > \quad (4)$$

By Eq. 3, $\theta_t - \theta = \alpha \cdot \nabla_\theta L(\theta)$. Substituting this in, we get:

$$L^v(\theta^t) - L^v(\theta) \propto< \nabla_\theta L^v(\theta), \nabla_\theta L(\theta) > \quad (5)$$

In Eq. 5, $\nabla_\theta L^\nu(\theta) \cdot \nabla_\theta L(\theta)$ represents the inner product between the training example's gradient and the *average gradient* of the validation samples. This is the meta gradient.

Therefore, Eq. 5 substantiates our claim that the meta gradient effectively estimates to what degree a training sample contributes to the model's performance. A positive meta gradient indicates that the training sample impacts the model in a positive way.

3.2 MetaStore Gradient-based Analytics

Leveraging the principles of meta gradients, MetaStore provides 4 core operators for gradient-based analytics:

- *Point-to-point (P2P)*: given a training sample and a validation (or, testing) example, estimate the contribution of the training sample to the prediction result of the validation (testing) example.
- *Point-to-batch (P2B)*: given a training sample and a batch of validation (testing) examples, estimate the contribution of the training sample to the prediction results of the batch of validation (testing) examples.
- *Batch-to-point (B2P)*: given a batch of training samples and a validation (testing) example, estimate the contribution of the batch of training samples to the prediction result of the validation (testing) example.
- *Batch-to-batch (B2B)*: given a batch of training samples and a batch of validation (testing) examples, estimate the contribution of the batch of training samples to the prediction results of the batch of validation (testing) examples.

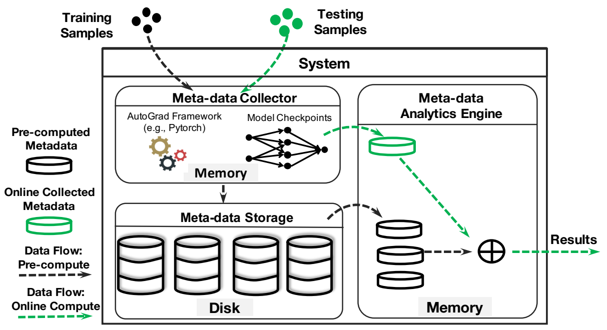


Figure 1: System Overview

Using these operators as building blocks, for the first time users could easily develop gradient-based analytics techniques to interpret the model prediction by examples [16, 38, 44], debug data issues [8, 29], or valuate the training samples [22, 49], etc. These tasks are critical for deep learning to achieve superior performance. Below are some intuitive examples.

Interpreting Model Prediction By Examples. Users could use the P2P operator to first compute the contribution of each training sample to the prediction of one testing sample and then select the top k training samples shown to have the most significant contribution to explain why the model predicts the given testing sample in the identified manner.

Data Debugging. Users could use P2B operator to determine how each specific training sample contributes to the prediction of a set of testing samples. If the P2B operator returns a negative value, it indicates this training sample could jeopardize the overall performance of the model. The users thus could identify this sample as a potential outlier or as mislabeled.

Data Valuation. Accordingly, using the P2B operator, the users could evaluate the training samples based on their contribution to the model. The more the training samples contribute, the more valuable they are. Potentially, the valuation results could guide the users to determine what new training samples they should collect to best improve the model performance.

Similarly, the B2P and B2B operators allow the users to evaluate how a batch of training samples as a whole impacts either the prediction of one testing sample or the overall performance of the model, thus interpreting model prediction or debugging data issues. As deep learning typically updates the model batch by batch using the average gradient of a batch of training samples, these operators mimic the training process of deep learning, thus meaningful.

4 SYSTEM OVERVIEW

In this section we overview MetaStore (Fig. 1), which consists of three key components: (1) a Meta-data Collector, (2) Meta-data Storage, and (3) a Meta-data Analytics Engine.

Meta-data Collector: MetaStore collects meta-data in a way non-intrusive to the DNN training process. MetaStore achieves so by storing a set of model checkpoints during DNN training. Each model checkpoint records the model parameters at a certain DNN training step. MetaStore then collects the gradients of the data samples at a model checkpoint by using the **model replay** feature. Model replay is a process that is **independent** from the model training process. Therefore, in MetaStore, collecting meta-data does not intervene with model training.

More specifically, given a data sample x_i and a model checkpoint $\phi(x;\theta)$, model replay first performs a forward propagation process to get the prediction $\hat{y}_i$ of x_i by $\phi(.)$, where $\hat{y}_i = \phi(x_i, \theta)$. It then calculates the loss value $C_i = L(\hat{y}_i, y)$ and performs backward propagation as described in Sec. 2 to obtain the gradient-related meta-data. But it does not update the model parameters. By replaying models, MetaStore is able to collect and materialize meta-data for the training samples in the offline pre-processing stage.

Our meta-data collector is **compatible** with existing deep learning frameworks. For example, we can integrate MetaStore with Pytorch by using its forward/backward hook function and with Tensorflow by using its custom gradient function. This is because the $<$ *prefix*, *suffix* $>$ pairs that MetaStore leverages are naturally produced during the backpropagation process, while backpropagation is used by all deep learning frameworks. Therefore, as long as the deep learning framework provides the interfaces to access intermediate data during backpropagation, MetaStore is able to collect the $<$ *prefix*, *suffix* $>$ pairs.

Meta-data Storage: The meta-data is maintained on disk (Sec. 5). With a DNN model composed of a series of layers (Sec. 2), a DNN model's gradient equals the concatenation of each layer's gradient. Thus, in MetaStore, the minimal unit of storage encapsulates the meta data of a specific layer in the DNN, which then is typically stored in a file. If the training set is large, MetaStore may further divide the entire data set into small batches. In this case, each file only contains the meta data corresponding to a small batch of data samples. MetaStore also maintains a **directory index** that indicates what data samples are stored in which file. It thus minimizes the disk I/O costs at online query time by only loading into memory the meta-data required by the query.

By decomposing the gradient into two partial gradients, namely the *prefix and suffix* gradients, MetaStore's storage strategies eliminate the storage bottleneck caused by the size of the gradients. The details are discussed in Sec. 5.

Meta-data Analytics Engine: This component provides efficient execution strategies for the 4 core operators discussed in Sec. 3.2. The input to each operator is the training and testing samples specified by the users. Because MetaStore already collects the meta-data of all training samples and maintains them in storage, the engine directly loads the requested gradients of the training samples from storage into GPU memory. However, unlike the training samples, MetaStore had not seen the testing samples in the training process. Therefore, it will compute their gradients on the fly by calling the *model replay* function. The engine then efficiently executes these operators using the optimized strategies discussed in Sec. 6 and Sec. 7. In addition, the engine uses **caching** to maintain the meta-data in GPU memory whenever possible and thus reduces I/O costs. It uses the standard LRU cache replacement policy to evict meta-data when memory overflows [16, 39].

5 SPACE-EFFICIENT GRADIENT STORAGE

MetaStore leverages our **prefix/suffix observation** to compactly store the gradients meta-data. In Sec. 5.1, we introduce the key idea using linear layers as an example layer type. Thereafter, we illustrate how to extend these principles to other types of DNN layers, including convolution and self-attention.

5.1 Gradient Storage: Linear Layers

Given a DNN model, assume its lth layer is a linear layer that applies a linear transformation to the input feature vector x: $y = \theta x + b$.

Suppose the input feature vector x and the output feature vector y have D^{in} and D^{out} dimensions, respectively. Then $\boldsymbol{\theta}$ contains $D^{in} \times D^{out}$ parameters. Let $\frac{dC}{d\theta}$ or $\nabla_\theta C$ denote the gradient of this layer. By Eq. 2, $\frac{dC}{d\theta} = \frac{dC}{dy} \cdot \frac{dy}{d\theta}$.

Prefix Gradient. The first matrix $\frac{dC}{dy}$ corresponds to the gradient of the output feature vector with respect to the loss value, called *prefix gradient.* Since the loss value C is calculated based on the output of the final layer, calculating the matrix $\frac{dC}{dy}$ requires backpropagation from previous layers. Because the linear layer is the lth layer, then $\frac{dC}{dy} = \frac{dC}{dz^L} \cdot \frac{dz^L}{dz^{L-1}} \cdots \frac{dz^{l+1}}{dz^l}$. Although calculating the prefix gradient through backpropagation is expensive, its size is only D^{out}. That is, it is identical to the size of the output feature vector y, being much smaller than the size ($D^{in} \times D^{out}$) of the final gradient $\frac{dC}{d\theta}$ we are interested in.

Suffix Gradient. The other matrix $\frac{dy}{d\theta}$, also called Jacobian matrix, corresponds to the *suffix gradient.* It indicates the expected update on parameters θ that will produce a better output feature embedding. Even though its general formulation is very complex and large, the Jacobian Matrix in a linear layer is simple:

$$\frac{dy}{d\theta} = \frac{d(\theta x + b)}{d\theta} = x \tag{6}$$

By Eq. 6, the suffix gradient in the linear layers is in fact identical to its input feature vector x. The size, D_{in}, is much smaller than the size of the parameters θ.

Prefix/Suffix Observation. Naturally, extracting out and maintaining the pair of small prefix and suffix gradients is **sufficient to reconstruct the original gradient** of a linear layer as follows:

$$(\nabla_\theta C)_{r,s} = (\frac{dC}{dy})_r \cdot x_s \tag{7}$$

where $\nabla_\theta C_{r,s}$ represents the rth row and sth column of θ and x_s represents the sth column of input x.

Space Complexity. The space complexity of storing the prefix and suffix gradients is O(D^{out}+D^{in}), while storing the full gradient takes $D^{out} \times D^{in}$ space. Thus leveraging this prefix/suffix observation, MetaStore drives down the storage costs by $\frac{D^{out} \times D^{in}}{D^{out} + D^{in}}$.

General Outlook. Next, we show that the principle of decomposing gradients into prefix and suffix gradients is also applicable to other typical DNN layers, including those that tend to have a large number of parameters and thus produce huge gradients. This is because: (1) all these layers use the chain rule to compute gradients during backpropagation, and (2) they can each be decomposed into a set of linear layers.

In this paper, we use the convolutional (Sec. 5.2) and self-attention layers (Sec. 5.2) as examples. Other similar layers include normalization layers [21], embedding layers, long short term memory (LSTM) layers [13], and gated recurrent units (GRU) [11], to just name a few. In addition we will briefly discuss how MetaStore supports complex blocks that contain multiple such layers.

5.2 Gradient Storage: Convolutional Layers

For the ease of understanding, we use the standard 1D convolutional layer as an example to illustrate the idea. Same as with the linear layer, we denote the parameters of the convolutional layer as θ. The input data sample x corresponds to a tensor in the shape (C_{in}, S), where C_{in} represents the number of input channels and S the number of features in each channel. For example, if the input data is an RGB image with 32×32 resolutions, C_{in} is equal to 3 and S equal to 32×32. Similarly, its output is a tensor with a shape of $(C_{out}, S-K)$, where C_{out} represents the number of output channels and K the number of the dimensions of one kernel. As a 1D matrix, a kernel $\mathcal{K}$ performs the convolution operation on the features of an input channel as follows: $y_s = \sum_i^K \mathcal{K}_i \cdot x_{s+i}$.

The convolution operation produces the output features with $S-K$ dimensions for each individual input channel. Aggregating these output features produces the final features of one output channel m:

$$y_{m,s} = \sum_i^{C_{in}} \sum_j^{K} \theta_{m,i,j} \cdot x_{C_{in},s+j} \tag{8}$$

Repeating this process C_{out} times produces an output with C_{out} channels. Thus, there are $C_{out} \times C_{in}$ kernels in a convolutional layer. In the training process, DNN learns these kernels to produce good output features. Thus, in the convolutional layer, parameters θ is a tensor with the shape of (C_{out}, C_{in}, K). The final output y is a tensor that contains C_{out} channels, with each channel composed of $S-K$ features.

Similar as with linear layers, the gradients $\frac{dC}{d\theta}$ of the convolutional layer can be decomposed into the prefix gradient $\frac{dC}{dy}$ and suffix gradient $\frac{dy}{d\theta}$, i.e., $\frac{dC}{d\theta} = \frac{dC}{dy} \cdot \frac{dy}{d\theta}$. This is because all layers in DNN use the same chain rule (Eq. 2) to compute gradients during back-propagation.

The Storage Strategy. Because C is a scalar, the size of the prefix gradient $\frac{dC}{dy}$ is equal to the number of output features. Next, we analyze the suffix gradient $\frac{dy}{d\theta}$. For this, we establish the *connection between the convolutional layer and the linear layer*, so that MetaStore will be able to adapt the storage strategy for the linear layer to the convolutional layer.

Recall that θ is a tensor in the shape of (C_{out}, C_{in}, K). It can thus be regarded as an aggregation of K linear sub-layers, with the shape of each sub-layer being (C_{out}, C_{in}). For the ease of presentation, we denote the ith linear sub-layer $\theta_{(\cdot,\cdot,i)}$ as θ_i, and similarly $x_{(\cdot,s)}$ as x_s, and $y_{(\cdot,s)}$ as y_s. Then, we have:

$$\frac{dC}{d\theta_i} = [\frac{dC}{dy_0} \cdots \frac{dC}{dy_{s-K}}][\frac{dy_0}{d\theta_i} \cdots \frac{dy_{s-K}}{d\theta_i}]^T = \sum_{s=0}^{s-K} \frac{dC}{dy_s} \cdot \frac{dy_s}{d\theta_i} \tag{9}$$

From Eq. 8, we have:

$$\frac{dy_s}{d\theta_i} = \frac{d(\sum_{\tilde{i}}^K \theta_{\tilde{i}} \cdot x_{s+\tilde{i}})}{d\theta_i} \tag{10}$$

Since $\frac{d(\theta_{\tilde{i}} \cdot x_{s+\tilde{i}})}{d\theta_i} = 0$ if $\tilde{i} \neq i$, while $\frac{d(\theta_{\tilde{i}} \cdot x_{s+\tilde{i}})}{d\theta_i} = x_{s+i}$ if $\tilde{i} = i$. Finally, we have:

$$\frac{dC}{d\theta_i} = \sum_s^{S-K} [\frac{dC}{dy_s} \cdot x_{s+i}] \tag{11}$$

Eq. 11 shows that MetaStore is able to reconstruct the gradients of the convolutional layers *in a similar way* to those of the linear layers. Therefore, MetaStore only needs to store *the prefix gradient and the features of the input samples*, where the size of the prefix gradient is the same as that of the output features.

Space Complexity. Storing the gradients as described above, the space complexity of MetaStore is determined by the size of the input samples and the size of the gradient of the output samples, that is, $S \times (C_{in} + C_{out})$. Storing the original gradient takes $K \times C_{out} \times C_{in}$ space. Therefore, when $S \times (C_{in} + C_{out}) < K \times C_{out} \times C_{in}$, MetaStore saves space. This is often true. For example, the last layer of the VGG16 model contains $9 \times 512 \times 512$ parameters, while its input and output features are only $512 \times 1 \times 1$ when training a VGG16 model on CIFAR-10 dataset. In this case, the saving is 4068x. In Sec. 8.2, we verify this with experiments.

5.3 Gradient Storage: Self-Attention Layers

Here, we use the sentence classification task as an example to show our gradient storage strategy on the self-attention layers (SAL). The input sample x of a SAL is a tensor with the shape of (S, H), where S denotes the length of the sentence and H the number of hidden features of each word. SAL uses Key-Query-Value to produce attention scores and update feature embeddings, accordingly. More specifically, SAL consists of three sub-layers, the key sub-layer θ^k, the query sub-layer θ^q, and the value sub-layer θ^v. Each sub-layer is a *linear layer*. Given an input sample, each sub-layer performs a linear transformation on all word representations in the sentence and generates three representations for each word, namely z_k, z_q, z_v, using the following equation: $z_{ks} = \theta^k \cdot x_s$, $z_{qs} = \theta^q \cdot x_s$, $z_{vs} = \theta^v \cdot x_s$. Then the final output is $y_s = softmax(z_{ks} \cdot z_{qs}/\sqrt{H}) \cdot z_{vs}$.

Storage Strategy. The three sub-layers perform the linear transformation on each word in the sentence. The shape of x is (S, H), while the shapes of θ^k, θ^q, and θ^v are all (H, H). Therefore, the shapes of z_k, z_q and z_v are (S, H). This is equivalent to linearly transforming a batch of S samples, where S is the length of the sentence. Because only the three sub-layers contain parameters, MetaStore handles each sub-layers separately. It then concatenates the gradients of each sub-layer to obtain the final gradient of the SAL.

Each input sequence can be modeled as a batch of words. Then given a sub-layer, its gradient is equivalent to the sum of the gradients with respect to a batch of data samples, where a sample corresponds to one word. Then given one data sample x_s, because the sub-layer is linear, its gradient with respect to this sub-layer can be decomposed the same way as done by the linear layer (Eq. 7), that is, decomposed to a *prefix gradient* and *input features* x_s. Finally, the gradient of each sub-layer can be computed with Eq 12:

$$\frac{dC}{d\theta^k} = \sum_s^S \frac{dC}{dz_l^k} \cdot x_s, \quad \frac{dC}{d\theta^q} = \sum_s^S \frac{dC}{dz_l^q} \cdot x_s, \quad \frac{dC}{d\theta^v} = \sum_s^S \frac{dC}{dz_l^v} \cdot x_s \tag{12}$$

Handling each sub-layer separately, MetaStore only needs to store the prefix gradient per layer and the input features, where the size of the prefix gradient corresponds to the size of output features. MetaStore then is able to restore the original gradients using Eq. 12.

Space Complexity. The space complexity of MetaStore is $(3H + H) \times S$. Storing the full gradients takes $3 \times H \times H \times$ S space. So

MetaStore drives down the storage costs by $O(\frac{3\times H}{4})$x. Given a SAL which produces 128 dimensional feature embeddings (H = 128), the saving would be 96 fold.

5.4 Gradient Storage: Complex Blocks

Similar to the Convolutional layer and the Self-attention layers, most of the complex blocks in popular deep learning model architectures, such as residual blocks, can be decomposed into simple linear sub-layers [18]. Because the key insight of MetaStore, i.e., collecting and operating on the small $<$ *prefix*, *suffix* $>$ pairs, works effectively on the linear layers, MetaStore can handle complex blocks like residual connections by decomposing these blocks into a series of simple linear sub-layers. For example, consider a simple residual layer,

$$y = x + F(x;\theta) \tag{13}$$

where θ represents the parameters of the residual layer, and x, y correspond to the model input and output respectively. Then based on the chain rule (Eq. 2):

$$\nabla_\theta C = \frac{dC}{dy}\frac{dy}{d\theta} = \frac{dC}{dy}\frac{d(x+F(x;\theta))}{d\theta} = \frac{dC}{dy}\frac{dF(x;\theta)}{d\theta} \tag{14}$$

As shown above, we can observe that the gradients of the residual layer parameters $\frac{dy}{d\theta}$ are independent of the input tensor x. The prefix gradient $\frac{dC}{dy}$ and the suffix gradient $\frac{dF(x;\theta)}{d\theta}$ are equivalent to the normal non-residual layers.

6 META-DATA ANALYTICSS: P2P

Next, we describe MetaStore's strategies that efficiently realize the gradient-based analytics operators described in Sec. 3. We introduce the execution strategy for the P2P operator below, while the P2B operator is covered in Sec. 7. Due to space limitation, we only briefly sketch the B2P and B2B operators in Sec. 7.2.

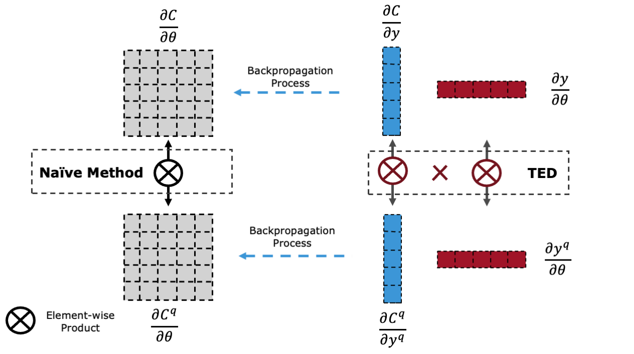
6.1 P2P Operator: Linear Layers

Because MetaStore stores the compact prefix and suffix gradients instead of the original (often huge) gradients, a straightforward solution to compute the inner product similarity between the gradients of two data samples would be to restore the gradients first and then to compute the inner product. Obviously, this would introduce extra overhead due to having to perform the restore operation.

MetaStore succeeds to compute the inner product of two gradients exactly *without having to restore* them first. More specifically, MetaStore could compute the exact inner product of two gradients by first in parallel computing the inner product on the prefix gradient $\frac{dC}{dy}$ and on the suffix gradient x, and thereafter multiplying these two results. Because it directly operates on the small prefix and suffix gradients, it is orders of magnitude faster than storing the original gradients before and then directly computing the inner product. Lemma 1 proves the correctness of this optimized method.

Lemma 1. *Given two data samples x_1 and x_2, denote their corresponding outputs of a linear layer θ as y_1 and y_2, their loss values as C_1 and C_2, and the gradients as $\nabla_\theta C_1$ and $\cdot\nabla_\theta C_2$. Then Eq. 15 holds.*

$$<\nabla_\theta C_1, \nabla_\theta C_2> = <\frac{dC_1}{dy_1}, \frac{dC_2}{dy_2}> \cdot <x_1, x_2> . \tag{15}$$



$\frac{\partial C}{\partial \theta}$
$\frac{\partial C}{\partial y}$
Backpropagation Process
$\frac{\partial y}{\partial \theta}$
Naïve Method
TED
Backpropagation Process
$\frac{\partial y^q}{\partial \theta}$
Element-wise Product
$\frac{\partial C^q}{\partial \theta}$
$\frac{\partial C^q}{\partial y^q}$

Figure 2: Naive method VS MetaStore.

Proof. From Eq. 7, we have,

$$\begin{aligned}
<\nabla_\theta C_1, \nabla_\theta C_2> &= \sum_{r=0}^{D^{out}}\sum_{s=0}^{D^{in}}(\nabla_\theta C_1)_{r,s}\cdot(\nabla_\theta C_2)_{r,s} \\
&= \sum_{r=0}^{D^{out}}\sum_{s=0}^{D^{in}}(\frac{dC}{dy_1})_r\cdot(x_1)_s\cdot(\frac{dC}{dy_2})_r\cdot(x_2)_s \\
&= <\frac{dC_1}{dy_1}, \frac{dC_2}{dy_2}> \cdot <x_1, x_2>
\end{aligned} \tag{16}$$

Therefore, Eq. 15 holds. □

Time Complexity. As discussed in Sec. 5.1, a prefix gradient has D^{out} dimensions, while a suffix gradient has D^{in} dimensions. Therefore, the time complexity of MetaStore is $O(D^{in} + D^{out})$. Because the size of the original gradients is $D^{out} \times D^{in}$, the time complexity of computing the inner product directly on the gradients is $O(D^{out} \times D^{in})$. Theoretically, MetaStore speeds up the P2P operation by $O(\frac{D^{out}\times D^{in}}{D^{in}+D^{out}})$.

6.2 P2P Operator: Convolutional Layers

As discussed in Sec. 5.2, given a convolutional layer whose parameters form a tensor θ with a shape of (C^{in}, C^{out}, K), because the K is often very small (e.g., K=9 in the VGG16 model), we could decompose θ into K linear sub-layers θ^i.

Therefore, when computing the inner product of two gradients produced in a convolutional layer, intuitively we could leverage the P2P operator designed for the linear layers to compute the inner product between the gradients with respect to θ^i and then sum up all the results. Given two data samples x_1 and x_2, we denote the corresponding outputs of a convolutional layer θ as y_1 and y_2, their loss values as C_1 and C_2, and thus the gradients as $\nabla_\theta C_1$ and $\nabla_\theta C_2$. Lemma 2 shows how to use the prefix and suffix gradients to directly compute $\nabla_\theta C_1 \cdot \nabla_\theta C_2$ in a convolutional layer.

Lemma 2. $<\nabla_{\theta^k} C_1, \nabla_{\theta^k} C_2> = \sum_s^{S-K}\sum_{\tilde{s}}^{S-K} <\frac{dC_1}{dy_1^{s-K}}, \frac{dC_2}{dy_2^{\tilde{s}-K}}> \cdot <x_1^s, x_2^{\tilde{s}}>$

Proof. From Eq. 11 and Lemma 1, we have,

$$\begin{aligned}
<\nabla_{\theta^k} C_1, \nabla_{\theta^k} C_2> &= <\sum_s^{S-K}\frac{dC_1}{dy_1^{s-K}}\cdot x_1^s, \sum_{\tilde{s}}^{S-K}\frac{dC_2}{dy_2^{\tilde{s}-K}}\cdot x_2^{\tilde{s}}> \\
&= \sum_s^{S-K}\sum_{\tilde{s}}^{S-K} <\frac{dC_1}{dy_1^{s-K}}, \frac{dC_2}{dy_2^{\tilde{s}-K}}> \cdot <x_1^s, x_2^{\tilde{s}}>
\end{aligned} \tag{17}$$

□

By Lemma 2 MetaStore could use Eq. 17 to compute $< \nabla_{\theta^k} C_1, \nabla_{\theta^k} C_2 >$.

Time Complexity. By Eq. 17 the time complexity of computing $< \nabla_{\theta^k} C_1, \nabla_{\theta^k} C_2 >$ is $(S-K)^2 \times (C_{out} + C_{in})$. Because $\nabla_\theta C_1 \cdot \nabla_\theta C_2 = \sum_k \nabla_{\theta^k} C_1 \cdot \nabla_{\theta^k} C_2$, the total time complexity of MetaStore calculating the gradient inner product on a CNN layer is $K \times (S-K)^2 \times (C_{in} + C_{out})$. The time complexity of directly using the original gradients to compute the inner product would be $K \times C_{in} \times C_{out}$ – which is identical to the number of the parameters. The potential speedup is thus $\frac{C_{in} \times C_{out}}{(S-K)^2 \times (C_{in}+C_{out})}$.

Therefore, the performance of MetaStore will depend on the number of features (S) of the input samples and the number of input and output channels (C_{in} and C_{out}). For most of the popular models, S decreases with the number of layers due to the convolution operation, while C_{in} and C_{out} increase. Therefore, the number of parameters in the later convolutional layers of a DNN model is often much larger than its early layers. Thus, MetaStore tends to significantly outperform the naive method on the later convolutional layers, while it can be slower on the earlier layers.

6.3 P2P Operator: Self-Attention Layers

As discussed in Sec. 5.3, a self-attention layer is formed by three linear sub-layer θ^k, θ^q and θ^v. Therefore, MetaStore can directly leverage the strategy designed for the linear layer to compute the inner product between the gradients with respect to each sub-layer and then at the end multiply the results.

Time Complexity. Let's denote two data samples as x_1 and x_2, the corresponding output of a CNN layer θ is y_1 and y_2. The time complexity of MetaStore is $S^2 \times (H + H)$, where S is the length of the input sequence and H represents the number of the dimensions of the hidden vector for each word. The time complexity of the naive solution that pre-computes and stores the original gradients is $H \times H$. So the potential speedup of MetaStore is $\frac{H}{2 \times S^2}$.

In a standard BERT model, $H = 768$. As long as the length of each sequence S is smaller than $\sqrt{384}$, MetaStore will win. We find this often holds on most of the popular benchmark NLP datasets [56].

6.4 Discussion: General to Other Scenarios

In addition to meta-gradient based analytics, MetaStore can provide benefit to other applications. As an example application, in adversarial attacks on DNN models, given a training sample x, one of the most popular gradient-based adversarial attack techniques, Fast gradient sign method (FGSM) [15], generates adversarial data samples based on the gradient of a training sample on the feature space $\frac{dC}{dx}$, where C is the loss value of x. Computing the value of $\frac{dC}{dx}$ normally requires a full pass of forward and backward propagation. However, with the chain rule, one is able to calculate $\frac{dC}{dx}$ directly from the first layers of the meta-data (e.g., the < *prefix*, *suffix* > pairs) that MetaStore materializes. For example, given a linear layer, $\frac{dC}{dx} = \frac{dC}{dy}\frac{dy}{dx} = \frac{dC}{dy}\frac{Wx+b}{x} = \frac{dC}{dy}W$, where $\frac{dC}{dy}$ corresponds to the *prefix artifacts* that MetaStore stores.

Moreover, MetaStore remains beneficial even when gradient reconstruction is eventually needed. Specifically, when the original gradient is needed, MetaStore will load the compact < *prefix*, *suffix* > pairs into memory and then reconstruct the gradient in memory. This greatly reduces the disk I/O costs compared to loading the huge original gradients, no matter whether the analytics request includes inner product operations or not. In fact, in our experiments, we find that the I/O costs are the main bottleneck. They take around 95% of the overall query execution time.

7 META-DATA ANALYTICS: BATCH OPERATORS

Next, we discuss our strategy to efficiently support the P2B operator in Sec. 7.1. Then in Sec. 7.2 we show how to leverage the efficient P2B execution strategy to support B2P and B2B operators.

7.1 P2B Operator: No Gradient Restore

The point-to-batch (P2B) operator estimates the contribution of one training sample x_i on the prediction results of a batch of testing samples B_j. By the concept of the meta-gradient introduced in Sec. 3.1, MetaStore measures this as the average inner product similarity between the gradients of the training sample and any testing sample in the batch. We denote this as $\mathcal{I}^{avg}(x, B)$.

Because MetaStore has already preprocessed and stored the gradient of each training sample beforehand as a < *prefix*, *suffix* > pair, this pair can be directly fetched. However, MetaStore has to obtain the gradient of the unseen testing samples on-the-fly using model replay. Given a batch of testing samples, MetaStore can get their gradients in two ways: (1) for each sample, we get its gradient in the format of < *prefix*, *suffix* > pair; or, (2) we directly get the average gradient of this batch. The existing deep learning infrastructures such as Pytorch readily provide this interface, because deep learning typically updates the model parameters based on the average gradient of a batch of training samples.

The advantage of the first approach is that MetaStore can directly call the efficient P2P operators introduced in Sec. 6 to compute $\mathcal{I}^{avg}(x, B)$. However, it has to iteratively compute the inner product for each pair of training and testing samples. When the testing batch is large, this will become expensive.

On the other hand, $\mathcal{I}^{avg}(x, B)$ is equivalent to the inner product between the gradient of x_i and the *average gradient* of the testing batch. Therefore, if MetaStore restores the full gradient of the training sample from the < *prefix*, *suffix* > pair and extracts the average gradient of the testing batch using the second approach, then it would be able to compute $\mathcal{I}^{avg}(x, B)$ with *one single* inner product operation. However, restoring the training sample from the < *prefix*, *suffix* > pair tends to be expensive.

We design an efficient P2P execution strategy which uses *one single* inner product operation to compute $\mathcal{I}^{avg}(x, B)$, while *not restoring* the full gradient of the training sample. This strategy is built on Lemma 3.

Lemma 3. *Let $\nabla_\theta C$ denote the gradient of training sample x and $\bar{G}^t$ the average gradient of the testing batch, given a linear layer with the shape of (D^{in}, D^{out}), Eq. 18 holds.*

$$\mathcal{I}^{avg}(x, B) = < \nabla_\theta C, \bar{G}^t > = < x^T, \bar{G}^t \frac{dC}{dy} > \quad (18)$$

Proof.

$$\mathcal{I}^{avg}(x, B) = < \nabla_\theta C, \bar{G}^t > = \sum_{i=0}^{D^{in}} \sum_{j=0}^{D^{out}} [(\nabla_\theta C)_{i,j} \cdot \bar{G}^t_{i,j}] \quad (19)$$

From Eq. 7, we have $(\nabla_\theta C)_{i,j} = \boldsymbol{x}_i \cdot (\frac{dC}{dy})_j$. Then:

$$\begin{aligned} \mathcal{I}^{avg}(x, B) &= \sum_{i=0}^{D^{in}} \sum_{j=0}^{D^{out}} [x_i \cdot \bar{G}^t_{i,j} \cdot (\frac{dC}{dy})_j] \\ &= x^T \sum_{j=0}^{D^{out}} [\bar{G}^t_{i,j} \cdot (\frac{dC}{dy})_j] = < x^T, \bar{G}^t \frac{dC}{dy} > \end{aligned} \tag{20}$$

where $\boldsymbol{x}$ and $\frac{dC}{dy}$ correspond to the <prefix, suffix> pair of the training sample. This concludes the proof of Lemma 3. □

Time Complexity. The time complexity is $D^{in} \times D^{out}$, which does not rely on the size of the batch. Therefore, this strategy scales to large testing batches.

Extending this method to the convolutional and self-attention layers is straightforward. MetaStore decomposes their parameters into a set of linear sub-layers. Using the average gradient of the testing batch w.r.t. each linear sub-layer and the pre-stored < *prefix*, *suffix* > pair, it first calculates the partial inner product as described above and then aggregates up the partial results.

7.2 Other Operators: B2P and B2B

Unlike the P2P and P2B operators, the B2P and B2B operators involve a batch of training samples which have their prefix and suffix gradients already maintained in MetaStore storage. An intuitive method to execute these two types of operators would thus be to first restore the original gradient from the prefix and suffix gradients for each training sample, compute the average gradient for this batch, then use model replay to extract the gradient or the average gradient for the testing samples, and finally compute the inner product. This method only needs to compute the inner product once. However, as we have discussed in Sec. 7.1 and confirmed in the experiments (Sec. 8.4), restoring the original gradient from the prefix and suffix gradient typically is even more expensive than the inner product operation itself, thus not acceptable.

After ruling out restoring the gradients as an option, we are left with having to iterate over each training sample in the batch, then call the P2P or P2B operator to compute the inner product, and lastly, to take the average. To mimic the DNN training process, users might set the size of the training batch according to the *batch size* hyper-parameter (typically 64 or 128). Thus, the cost of iterating over each training sample in a batch tends to be acceptable.

8 EXPERIMENTS

Our experimental study focuses on the following questions:

- Storage: Does MetaStore reduce the storage footprint of gradient meta-data and thereby offer practical feasibility?
- Execution Time: Does MetaStore speed up the execution of gradient-based analytics compared to unoptimized methods?
- Preprocessing: Is MetaStore efficient at collecting meta-data?
- How useful are our analytics interfaces in applications?

8.1 Experimental Setup

Settings. All the experiments are implemented in Python3.7 on Pytorch. We conduct all experiments on a virtual cloud instance with Intel Xeron G6248 CPU, 0.5 TB Memory, an SSD storage disk with 2TB space, and one V100 GPU with 32G memory.

Datasets. We evaluate our method with three benchmark datasets, namely, **ImageNet** (image), **CIFAR10** (image) and **AGNews** [54] (text) dataset. ImageNet contains 1,200,000 images from 1000 classes. Each image has the dimensions of $3 \times 512 \times 512$. CIFAR10 contains 50,000 images from 10 classes. Each image has the dimension of $3 \times 32 \times 32$. AGNews contains 30,000 sentences from four classes, where each sentence contains 6 to 89 words.

Baseline Methods. In the experiments we only measure the efficiency of the point-to-point (P2P) operators and the point-to-batch (P2B) operators, because the B2P and B2B operators simply leverage the P2P and P2B operators, as discussed in Sec. 7.2.

For the P2P operators, we compare MetaStore with the following baseline methods:

1) **Pre-compute**: We pre-compute the full gradient on the queried layers for all training samples and store them in disk. Once an analytics query is submitted, we retrieve the gradient of the indicated training sample from the disk into GPU memory, extract the gradient for the indicated testing sample in the <prefix,suffix> pair format, and run the corresponding analytics operators.

2) **Re-compute**: After an analytics query is submitted, it computes the gradient of the training sample on the fly through model replay using the model maintained in GPU.

For the point-to-batch (P2B) operators, we evaluate the *Iterate* and *Reconstruction* methods discussed in Sec. 7. Both methods leverage our compact <prefix, suffix> storage structure to reduce the I/O costs when collecting the gradients of training samples.

1) **Iterate**: This method extracts the gradients for each indicated testing sample in the <prefix,suffix> pair format as described in Sec.4, and then calls our *optimized* P2P operator to compute the inner product between the training samples and each testing sample in the query batch and then compute the average.

2) **Reconstruction:** This method extracts the average gradient for the testing batch through model replay, and then reconstructs the gradients of training samples from the <prefix, suffix> pair. Finally, it directly calculates the similarity between the gradient of a training sample and the average gradient of the testing batch. Therefore, *Reconstruction* only computes the inner product once.

Unlike the P2P operators experiments, in the P2B experiments we don't compare against *Pre-compute* and *Re-compute* baselines, because *Iterate* and *Reconstruction* leverage our compact <prefix,suffix> storage structure and optimized P2P operator. Thus, they are clearly more efficient than *Pre-compute* and *Re-compute*. Similarly, although *Reconstruction* could work for P2B operator, we don't compare against it in the P2P experiments. This is because for the P2P operator, *Reconstruction* does not reduce the number of inner product computations, while introducing extra computation costs to reconstruct the original gradient from the <prefix, suffix> structure. It is thus guaranteed to be worse than *Pre-compute.*

Data Compression. We evaluate how quantization improves the performance of the above baseline methods as well as our MetaStore. More specifically, we apply the *Lower precision float representation* quantization [39] to reduce the size of meta-data.

Usefulness: We compare MetaStore with two baseline methods (Sec. 8.7), namely *Gradient-shapely* [14] and *Small-loss* [29] also discussed in related work, in studies assessing their relative utility.

DNN Models. We evaluate our method on three popular deep neural network architectures, namely ResNet-50 [18], VGG16 [36]

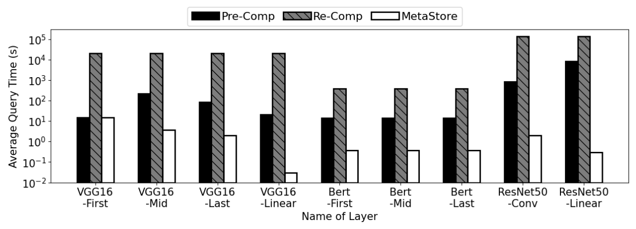


Figure 3: The End-to-End Query Execution Time of P2P Operator using the VGG16, BERT and ResNet50 Models.

and BERT [12]. ResNet-50 contains 49 convolutional layers and one linear layer. VGG16 consists of 13 Convolutional layers and three linear layers. The BERT Model, popular in natural language processing, contains 12 Attention layers and one linear layer. We trained a ResNet-50 on ImageNet, a VGG16 model on CIFAR10, and a BERT model on AGNews through finetuning.

8.2 Storage Costs

Table 1: Storage Costs: MetaStore vs Full Gradient.

Storage Cost (MB)				
Layers	Shape	MetaStore	Full Gradient	Disk Space Saving
VGG16-Conv1	$9 \times 3 \times 64$	2744	69	0.025×
VGG16-Conv7	$9 \times 128 \times 256$	1310	23593	18.0×
VGG16-Conv13	$9 \times 512 \times 512$	163	94371	578×
VGG16-Linear1	512×10	21	205	9.76×
BERT-SAL1	$3 \times 768 \times 768$	2949	70779	24.00×
BERT-SAL6	$3 \times 768 \times 768$	2949	70779	24.00×
BERT-SAL11	$3 \times 768 \times 768$	2949	70779	24.00×
BERT-Linear1	768×4	31	122	3.93×
ResNet50-Conv48	$9 \times 512 \times 512$	157	90100	573.88×
ResNet50-Linear	2048×1000	118	80100	678.81×

In this set of experiments, we evaluate the storage costs and savings of the MetaStore's prefix/suffix gradient strategy of storing decomposed gradients. We evaluate the storage costs for 10,000 training samples randomly sampled from the training set because the baseline cannot handle the whole training set. Following previous work [16], for the VGG16 model, we report the storage costs of the first, mid, and last convolutional layers and the linear layer, while for the BERT model, we report the storage costs of the first, mid and last self-attention layer and the last linear layer. For the ResNet50 model, we report the storage costs of the last linear layer and the 48th Convolutional layer as it contains the most number of parameters in the ResNet50 model.

Table 1 shows these storage costs. We see that compared to storing the original gradients, MetaStore reduces the storage costs by up to x578 for the VGG16 model and by up to x678 for the ResNet50 model. The only exception is the first convolutional layer that features only a few parameters, has a small gradient, and thus not much disk space is saved in this case.

Similarly, for the BERT model, MetaStore reduces the storage costs by 24x. However, for this model, both methods need more disk space than the ResNet50 and VGG16 models, because the BERT-AGNews model contains many more parameters than the VGG16-CIFAR10 model. Also, each layer in the BERT-AGNews model generates a larger number of input and output features for each training sample compared to the ResNet50 and VGG16-CIFAR10 model.

8.3 P2P Operator: End-to-End Execution Time

In this experiment, we evaluate the end-to-end execution time of the P2P operator which computes the inner product between the gradients of two data samples. This execution time includes the times for calculating the gradients of the testing samples by model replay, loading the gradients of the training samples into GPU memory, and running the corresponding analytics operators.

8.3.1 Execution Times for Different DNN Layers.

First, we compare MetaStore against the *Pre-compute* and *Re-compute* methods (Sec. 8.1) on the VGG16-CIFAR10, ResNet50-ImageNet and BERT-AGNews models. Similar as with the above storage experiments, we evaluate the first, middle and the last convolutional layer of the VGG16 model, the 48th Convolutional layer and the last linear layer of the ResNet50 model, and the first, middle and the last self-attention layer of the BERT Model. We randomly select one testing sample from the testing set. For each pair of training sample and this chosen testing sample, we run the P2P operator. We use 10,000 training samples and thus call the P2P operator 10,000 times. We repeat the experiment 10 times and report the average execution time.

Fig. 3 (in **log scale**) shows that for the VGG16-CIFAR10 model, MetaStore is up to 1,000 times faster than *Pre-compute*, and 7 orders of magnitude faster than *Re-compute*. In particular, *Pre-compute* is slower on the later convolutional layers, while MetaStore improves speed there. This is because that the complexity of *Pre-compute* increases linearly with the number of parameters, and the later convolutional layers have more parameters. On the other hand, the complexity of MetaStore increases linearly with the size of the input features, which is smaller in the later layers compared to the earlier layers. This is common for CNN networks, since the convolution operation naturally shrinks the size of the features.

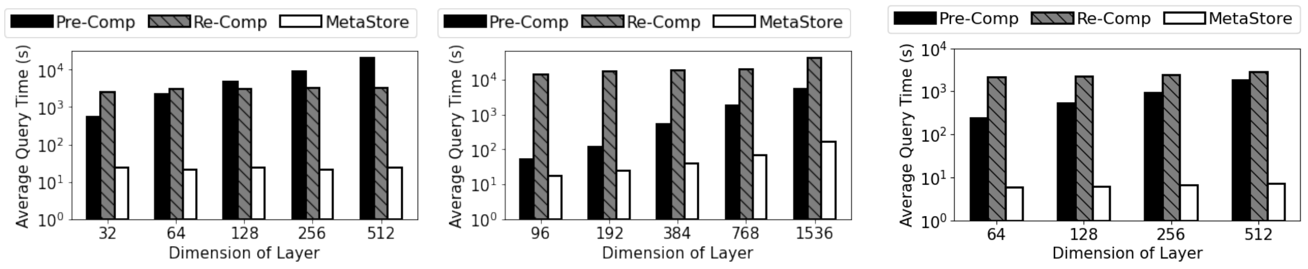
For the ResNet50 model, all three methods are slower on the ResNet50 model than on the VGG16 model. This is expected, because ResNet50 has many more parameters than the VGG16 model. However, MetaStore is still up to 3 orders of magnitude faster than *Pre-compute* and 5 orders of magnitude faster than Re-compute.

For BERT, MetaStore is about 10 to 100 times faster than Pre-compute and 100 to 1000 times faster than Re-compute. Because different self-attention layers in BERT have the same architecture, their performance does not vary much across different layers.

8.3.2 Varying Number of Dimensions of Layers.

In this set of experiments, we evaluate MetaStore's performance on DNN layers with a varying number of dimensions. To achieve this, for the linear layer, we append one additional linear layer before the last layer in ResNet50. Similarly, for the convolutional layer, we append one additional convolutional layer after the last convolutional layer in VGG16. We refer to these two "extended" models as ResNet50-Linear and VGG16-Conv, respectively. We then vary the number of dimensions of these new layers. For the self-attention layer, we directly vary the input and output dimension of each self-attention layers. We name this model BERT-Att.

For the ResNet50-Linear model, to ensure the appended layer is aligned with the previous layers, we keep the input dimensions fixed and only vary the output dimensions from 32 to 512. Similarly, for the VGG16-Conv model, we fix the number of input channels



(a) **VGG16-Conv** (b) **BERT-Att** (c) **ResNet50-Linear**

Figure 4: The End-to-End Query Execution Time of P2P Operator: Varying Num. of Dimensions of Different Layers.

and vary the output channels from 32 to 512. Thereafter, we focus on comparing the end-to-end execution time on the new layer of each model. For the BERT-Attention model, we vary the input and output dimensions of each Self-Attention layers from 96 to 768. We report the execution time of the last self-attention layer.

As depicted in Fig. 4, MetaStore is up to 1000× faster than both baseline methods in all experiments. For the VGG16-Conv and the ResNet50-Linear models, as shown in Fig. 4(a) and Fig. 4(c), respectively, for all three types of layers, the execution time of Pre-compute increases quickly as the output dimensions get larger, while the query time of MetaStore does not increase significantly. This can be explained by the time complexity of *Pre-compute* which equals the input dimensions multiplied by the output dimensions, while the time complexity of MetaStore equals the input dimensions plus the output dimensions, as discussed in Sec. 5. For the BERT-Attention model, MetaStore is up to 1,000× faster than both baseline methods. In all experiments, *Re-compute* is much slower than the other two methods in most cases, because calculating the gradient of a single layer on the fly is expensive.

8.3.3 Vary the Number of Training Samples.

We vary the number of training samples for each query from 500 to 8,000 and compare MetaStore against the Pre-Compute and Re-Compute methods. We measure the cumulative total time of running 100 queries on the last convolutional and the last linear layer in the VGG16-CIFAR10 model, the 48th convolutional and the last linear layer in the ResNet50-ImageNet model, and the last self-attention layer in the BERT-AGNews model. We cache the gradients in the memory, when possible, using LRU as cache replacement policy. As shown in Fig. 5, MetaStore only gets about 5 times slower when increasing the number of training samples from 500 to 8000 on the VGG16-CIFAR10, ResNet50-ImageNet, and BERT-AGNews models, while the execution time of *Pre-Compute* and *Re-Compute* increase 12-15 times in both cases. This is because MetaStore can cache more data samples in memory due to its efficient storage strategy, therefore significantly reducing the I/O costs. The query execution time of *Pre-compute* increases fast as the number of analyzed samples increases. Eventually, when the number of training samples increases to 8,000, it becomes as slow as the *Re-compute* method which computes the gradients on the fly. This is because the gradients of the ResNet50-ImageNet model are very large. It is thus not able to cache the gradients of many training samples in memory, hence suffering from high disk I/O costs. In contrast, when the number of analyzed samples increases from 500 to 8,000, the execution time of MetaStore only increases around 10 times.

8.4 P2B Operator: Execution Time

We evaluate the performance of our optimized method (Sec. 7.1) for the P2B operator. In this experiment, we compare *Iterate* and *Reconstruction* as baseline methods. The reconstruction method leverages our prefix/suffix gradients insights, thus significantly reducing its I/O costs.

As shown in Fig. 6, our method is at least 2 times faster than the baseline methods in all experiments. Compared with the reconstruction method, our method speeds up the execution by up to 10x, because it directly computes the results on the <prefix, suffix> pairs of training samples, and thus avoids reconstructing large gradients for the training samples.

8.5 Meta-data Collection and Storage Times

We evaluate the time of extracting and storing the gradient of 10,000 training samples. We compare MetaStore against computing and storing the full gradients. Again, we measure the collection time on the first, mid, and last convolutional layers and the linear layer in the VGG16-CIFAR10 model, the 48th convolutional layer and the last linear layer of the ResNet50-ImageNet model, and the first, mid, and last self-attention layers and the last linear layer in the BERT-AGNews model. Fig. 7 shows that MetaStore is up to 1,000 times faster than the baseline. This is because although both methods use the same forward and backward propagation process to extract meta-data, MetaStore only needs to log the small prefix and suffix matrices into the storage.

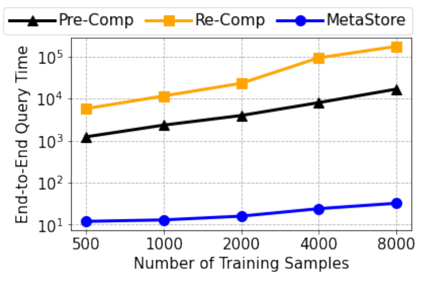
Similar to the trend in the storage cost experiments, the baseline takes more time to collect meta-data on the later convolutional layer in the VGG16 model in comparison to MetaStore. Again, this is because the later convolutional layers in the VGG16 model have more parameters than the earlier convolutional layers.

We also evaluate the meta-data collection time by varying the number of dimensions of the target layers. As shown in Fig. 8, MetaStore consistently outperforms the baseline. As the number of dimensions increases, the collection time of the baseline increases linearly, while MetaStore only becomes slightly slower.

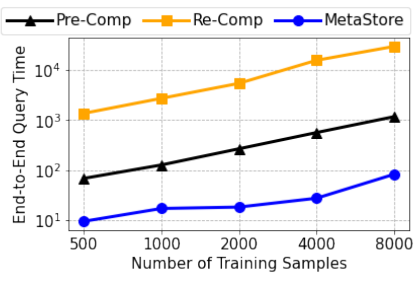
8.6 Augmented with Data Compression

Our technique is *orthogonal* to the choice of the compression methods, including quantization. For this reason, we were able to apply quantization to both MetaStore and to the pre-compute baseline.

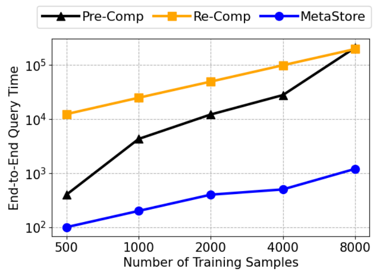
We evaluate this addition of compression by varying the precision of quantized meta-data from 8 digits to 32 digits, and then



(a) **VGG16-CIFAR10**

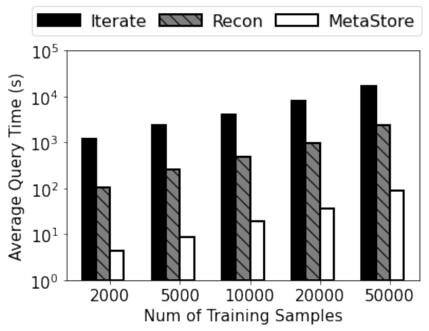


(b) **BERT-AGNews**

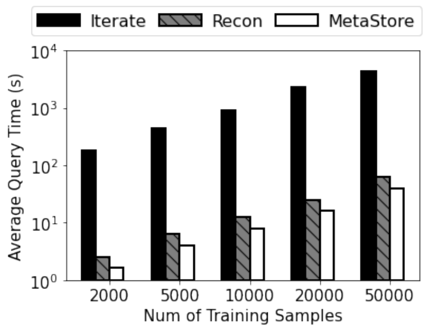


(c) **ResNet50-ImageNet**

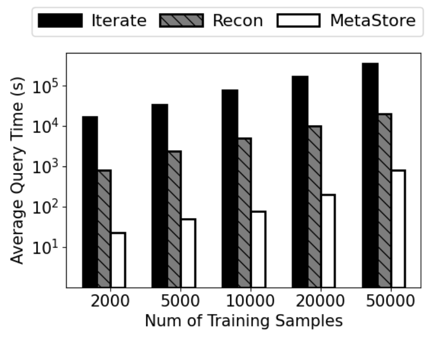
Figure 5: The End-to-End Query Execution Time of P2P Operator: Varying the Number of Training Samples.



(a) **VGG16-CIFAR10**



(b) **BERT-AGNews**



(c) **ResNet50-ImageNet**

Figure 6: The End-to-End Query Execution Time of P2B Operator: Varying the Number of Training Samples.

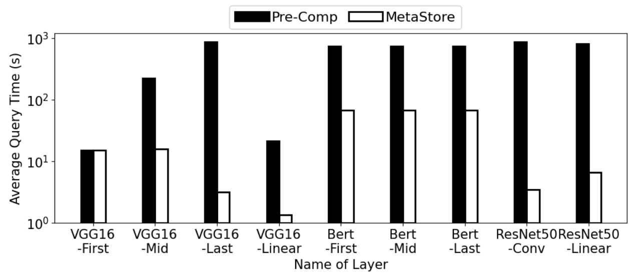


Figure 7: Pre-processing Time

measure both the end-to-end query execution time and the storage costs. As shown in Fig. 10, quantization indeed reduces the storage costs of both MetaStore and the pre-compute methods by up to 4×. However, MetaStore is still up to three orders of magnitude more efficient than the pre-compute method in terms of both storage costs and query execution time. In particular, for query execution time, the pre-compute method with quantized tensors is up to 10 × faster than the original method using the standard precision tensors. Because MetaStore is already very efficient due to our $<$ *prefix*, *suffix* $>$ insight, quantization is not able to speed up MetaStore that much. Most importantly, in all scenarios, MetaStore remains up to 1000× faster than the baselines even with the addition of quantization.

8.7 Utility of Gradient-based Analytics

We use data debugging as an example to showcase that the gradient-based analytics enabled by MetaStore are indeed useful.

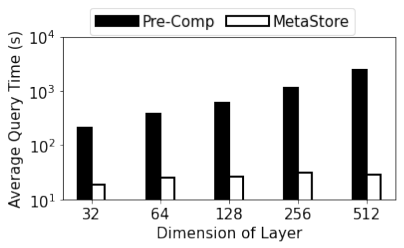
As discussed in Sec. 3.2, users can use the P2B operator to discover mislabeled objects (data debugging). The k training samples (where k is a user defined input parameter) that have the smallest meta-gradient with a batch of testing samples are the least influential samples, and therefore the most likely to be mislabeled.

We train a VGG16 model using CIFAR10. We randomly flip the labels of 1000 samples from class 0 to class 1 and select 1000 testing samples to form the batch. We gradually add the layers of the DNN model, starting with only the last linear layer and then adding the last, middle, and first convolutional layers step by step.

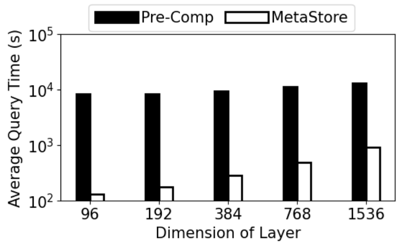
As shown in Fig. 9, our MetaStore achieves higher noisy label detection precision and query efficiency than the small-loss method because the later method requires one pass of forward propagation to calculate the loss value.

In Fig. 9, we also observe that our MetaStore solution achieves similar precision on this noisy labeling task as Gradient-shapely [14] – yet in addition it is up to *3 orders of magnitude* faster. This is because Gradient-shapely requires iteratively calculating the original gradients of training samples at query time, while our MetaStore operates on the compact $<$ *prefix*, *suffix* $>$ pairs.

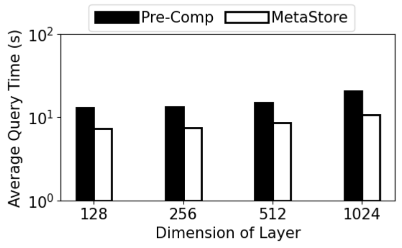
Fig. 9 shows that by increasingly analyzing more layers, the precision of MetaStore and Gradient-shapely increases from 0.1 to 0.6. Moreover, although the query execution time of Gradient-shapely increases significantly as more layers are analyzed, the query execution time of MetaStore remains relatively stable. This shows MetaStore allows users to analyze more layers and thus is able to get better results with much less computing resources.



(a) **VGG16-Conv**

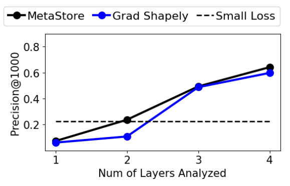


(b) **BERT-Attention**

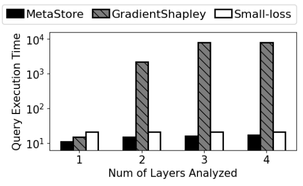


(c) **ResNet50-Linear**

Figure 8: Fig.(a)-(c):The Meta-data Collection Time: Varying the # of Dimensions of Different Types of Layers.

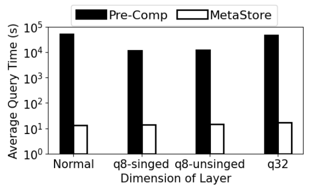


(a) **Precision of Detected Accuracy**

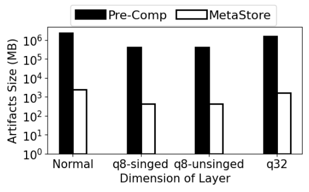


(b) **End-to-End Query Time**

Figure 9: Mislabel Detection: Precision and Query Time



(a) **End-to-End Query Times**



(b) **Storage Costs**

Figure 10: Quantization: Query Time and Storage Costs.

9 RELATED WORK

DNN Diagnosis Tools. A plethora of developed tools have been researched for diagnosing DNN models, some of which involve meta-data. Among them, MISTIQUE [39, 40] compactly stores the meta-data, namely, feature embeddings and losses. DeepEverest [16] speeds up the model diagnosis queries on storage. However, none of them support gradient-based diagnostic queries.

Several works [4, 24, 24, 28, 33, 35, 50] target the visualization of meta-data. However, none of them use gradients.

Gradient-shapely [14] evaluates the contribution of each training sample to model predictions by estimating the shapely value [22]. Given a training sample, Gradient-Shapely calculates its gradient and updates the model parameters. It compares the validation loss on the models before and after the update and uses the decrease of the validation loss as the training sample's shapely value. Gradient-shapely confirms that gradients are indeed effective in model interpretation, debugging, and data valuation. However, unlike our work, Gradient-shapely [14] does not address the scalability and efficiency issues of gradient analytics.

Robust Deep Learning with Meta-data. Researchers have used meta-data in the training process to make DNN models robust to noisy data and adversarial attack [9, 19, 29, 34, 37, 39]. In particular, Small-loss [29] uses training losses to detect mislabeled training samples. It is based on a simple and common observation that correctly training samples tend to have smaller training loss values than incorrectly labeled samples.

People have used gradients to perform adversarial attacks on DNN models [15, 32, 51]. Some methods [1, 9] leverage statistics of gradients to identify potential data leakage of DNN models. Some other methods use the gradients to modify the training process and search for hyper-parameters [7, 23, 27, 48] to improve DNN models' performance. However, all above works do not tackle the problem of compactly storing and efficiently analyzing gradients.

Gradient Compression. Federated learning may need to transfer gradients from the clients to the servers. To reduce the communication costs, researchers have proposed techniques [3, 5, 6, 10, 26, 43] to compress the gradients by approximation. Some works [2, 3, 5, 6, 43] use quantization and sparsification techniques to compress the gradients by preserving the large gradient values while discarding the small ones. Because our $< prefix, suffix >$ based optimization is *orthogonal* to the choice of the compression methods, including quantization, we are able to seamlessly apply the quantization technique to our MetaStore.

Some other works [25, 41, 42] use matrix factorization to decompose big gradients. However, performing matrix factorization on each training sample will introduce prohibitive overhead. Furthermore, the original gradients have to be reconstructed when computing the meta gradient at the online query stage, while reconstructing gradients is slow as shown in our experiments (Sec. 8.4). Our MetaStore instead efficiently analyzes the gradients without conducting any extra operations such as matrix factorization and gradient reconstruction.

10 CONCLUSION

We propose MetaStore to efficiently collect, store, and analyze meta-data produced by DNN training. The key techniques of MetaStore address the challenges caused by the size of the gradients and thus enable gradient-based analytics for data debugging and model interpretation. Our experiments show that MetaStore significantly reduces storage costs and query execution times by orders of magnitude compared to baseline solutions.

ACKNOWLEDGMENTS

This research was supported in part by NSF under grants IIS-1910880, CSSI-2103832, CSSI-2103799, CNS-1852498, NRT-HDR-1815866, NRT-HDR-2021871, DBI-2327954, and by the U.S. Dept. of Education under grant P200A180088.

REFERENCES

[1] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 308–318, 2016.

[2] A. F. Aji and K. Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

[3] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in neural information processing systems*, 30, 2017.

[4] S. Amershi, M. Chickering, S. M. Drucker, B. Lee, P. Simard, and J. Suh. Modeltracker: Redesigning performance analysis tools for machine learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 337–346, 2015.

[5] D. Basu, D. Data, C. Karakus, and S. Diggavi. Qsparse-local-sgd: Distributed sgd with quantization, sparsification and local computations. *Advances in Neural Information Processing Systems*, 32, 2019.

[6] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pages 560–569. PMLR, 2018.

[7] O. Bohdal, Y. Yang, and T. Hospedales. Evograd: Efficient gradient-based meta-learning and hyperparameter optimization. *Advances in Neural Information Processing Systems*, 34:22234–22246, 2021.

[8] L. Cao, Y. Yan, Y. Wang, S. Madden, and E. A. Rundensteiner. Autood: Automatic outlier detection. In *SIGMOD*.

[9] N. Carlini, U. Erlingsson, and N. Papernot. Distribution density, tails, and outliers in machine learning: Metrics and applications. *arXiv preprint arXiv:1910.13427*, 2019.

[10] C.-Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, and K. Gopalakrishnan. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[11] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[12] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[13] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[14] A. Ghorbani and J. Zou. Data shapley: Equitable valuation of data for machine learning. In *International conference on machine learning*, pages 2242–2251. PMLR, 2019.

[15] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[16] D. He, M. Daum, W. Cai, and M. Balazinska. Deepeverest: Accelerating declarative top-k queries for deep neural network interpretation. *Proc. VLDB Endow.*, 15(1):98–111, 2021.

[17] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[19] D. Hofmann, P. VanNostrand, H. Zhang, Y. Yan, L. Cao, S. Madden, and E. Rundensteiner. A demonstration of autood: a self-tuning anomaly detection system. *Proceedings of the VLDB Endowment*, 15(12):3706–3709, 2022.

[20] R. Hu, D. Zhang, D. Tao, H. Zhang, H. Feng, and E. Rundensteiner. Uce-fid: Using large unlabeled, medium crowdsourced-labeled, and small expert-labeled tweets for foodborne illness detection. In *2023 IEEE International Conference on Big Data (BigData)*, pages 5250–5259. IEEE, 2023.

[21] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[22] R. Jia, D. Dao, B. Wang, F. A. Hubis, N. Hynes, N. M. Gürel, B. Li, C. Zhang, D. Song, and C. J. Spanos. Towards efficient data valuation based on the shapley value. In K. Chaudhuri and M. Sugiyama, editors, *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*, volume 89 of *Proceedings of Machine Learning Research*, pages 1167–1176. PMLR, 2019.

[23] Y. Jin, T. Zhou, L. Zhao, Y. Zhu, C. Guo, M. Canini, and A. Krishnamurthy. Autolrs: Automatic learning-rate schedule by bayesian optimization on the fly. *arXiv preprint arXiv:2105.10762*, 2021.

[24] M. Kahng, D. Fang, and D. H. Chau. Visual exploration of machine learning results using data cube analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–6, 2016.

[25] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[26] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[27] M. Liu, L. Chen, X. Du, L. Jin, and M. Shang. Activated gradients for deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[28] D. Matthew Zeiler and F. Rob. Visualizing and understanding convolutional neural networks. ECCV, 2014.

[29] G. Pleiss, T. Zhang, E. Elenberg, and K. Q. Weinberger. Identifying mislabeled data using the area under the margin ranking. *Advances in Neural Information Processing Systems*, 33:17044–17056, 2020.

[30] G. Pruthi, F. Liu, S. Kale, and M. Sundararajan. Estimating training data influence by tracing gradient descent. *Advances in Neural Information Processing Systems*, 33:19920–19930, 2020.

[31] M. Ren, W. Zeng, B. Yang, and R. Urtasun. Learning to reweight examples for robust deep learning. In *International conference on machine learning*, pages 4334–4343. PMLR, 2018.

[32] J. Rony, L. G. Hafemann, L. S. Oliveira, I. B. Ayed, R. Sabourin, and E. Granger. Decoupling direction and norm for efficient gradient-based l2 adversarial attacks and defenses. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4322–4330, 2019.

[33] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[34] Y. Shen and S. Sanghavi. Learning with bad training data via iterative trimmed loss minimization. In *International Conference on Machine Learning*, pages 5739–5748. PMLR, 2019.

[35] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[36] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[37] S. Swayamdipta, R. Schwartz, N. Lourie, Y. Wang, H. Hajishirzi, N. A. Smith, and Y. Choi. Dataset cartography: Mapping and diagnosing datasets with training dynamics. *arXiv preprint arXiv:2009.10795*, 2020.

[38] P. M. VanNostrand, H. Zhang, D. M. Hofmann, and E. A. Rundensteiner. Facet: Robust counterfactual explanation analytics. *Proceedings of the ACM on Management of Data*, 1(4):1–27, 2023.

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

[40] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Modeldb: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–3, 2016.

[41] T. Vogels, S. P. Karimireddy, and M. Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

[42] H. Wang, S. Sievert, S. Liu, Z. Charles, D. Papailiopoulos, and S. Wright. Atomo: Communication-efficient learning via atomic sparsification. *Advances in Neural Information Processing Systems*, 31, 2018.

[43] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *Advances in neural information processing systems*, 30, 2017.

[44] W. Wu, L. Flokas, E. Wu, and J. Wang. Complaint-driven training data debugging for query 2.0. In *SIGMOD*, pages 1317–1334, 2020.

[45] M. Xia, S. Malladi, S. Gururangan, S. Arora, and D. Chen. Less: Selecting influential data for targeted instruction tuning. *arXiv preprint arXiv:2402.04333*, 2024.

[46] T. Xiao, X.-Y. Zhang, H. Jia, M.-M. Cheng, and M.-H. Yang. Semi-supervised learning with meta-gradient. In *International Conference on Artificial Intelligence and Statistics*, pages 73–81. PMLR, 2021.

[47] Y. Xu, L. Zhu, L. Jiang, and Y. Yang. Faster meta update strategy for noise-robust deep learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 144–153, 2021.

[48] H. Yong, J. Huang, X. Hua, and L. Zhang. Gradient centralization: A new optimization technique for deep neural networks. In *European Conference on Computer Vision*, pages 635–652. Springer, 2020.

[49] J. Yoon, S. Arik, and T. Pfister. Data valuation using reinforcement learning. In *International Conference on Machine Learning*, pages 10842–10851. PMLR, 2020.

[50] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[51] Z. Yuan, J. Zhang, Y. Jia, C. Tan, T. Xue, and S. Shan. Meta gradient adversarial attack. In *Proceedings of the IEEE/CVF International Conference on Computer*

Vision, pages 7748–7757, 2021.

[52] H. Zhang, L. Cao, S. Madden, and E. Rundensteiner. Lancet: labeling complex data at scale. *Proceedings of the VLDB Endowment*, 14(11), 2021.

[53] H. Zhang, L. Cao, P. VanNostrand, S. Madden, and E. A. Rundensteiner. Elite: Robust deep anomaly detection with meta gradient. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2174–2182, 2021.

[54] X. Zhang, J. J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *NIPS*, 2015.

[55] Z. Zhang and T. Pfister. Learning fast sample re-weighting without reward data. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 725–734, 2021.

[56] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.