Complaint-Driven Training Data Debugging at Interactive Speeds

ABSTRACT

Modern databases support queries that perform model inference (inference queries). Although powerful and widely used, inference queries are susceptible to incorrect results if the model is biased due to training data errors. Recently, Rain [40] proposed *complaint-driven data debugging* which uses user-specified errors in the *output* of inference queries (*Complaints*) to rank erroneous training examples that most likely caused the complaint. This can help users better interpret results and debug training sets. Rain combined influence analysis from the ML literature with relaxed query provenance polynomials from the DB literature to approximate the derivative of complaints w.r.t. training examples. Although effective, the runtime is O(|T|d), where T and d are the training set and model sizes, due to its reliance on the model's second order derivatives (the Hessian). On a Wide Resnet Network (WRN) model with 1.5 million parameters, it takes >1 minute to debug a complaint.

We observe that most complaint debugging costs are independent of the complaint, and that modern models are overparameterized. In response, Rain++ uses precomputation techniques, based on non-trivial insights unique to data debugging, to reduce debugging latencies to a constant factor independent of model size. We also develop optimizations when the queried database is known apriori, and for standing queries over streaming databases. Combining these optimizations in Rain++ ensures interactive debugging latencies (~10ms) on models with millions of parameters.

ACM Reference Format:
. 2022. Complaint-Driven Training Data Debugging at Interactive Speeds. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD'22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Modern database systems provide first-class support for ML model inference, so that SQL queries can easily perform model inference as part of analytics queries [17, 27, 33]. For instance, Google BigQuery integrates native TensorFlow support [27], SQLServer supports ONNX models [9], and MadLib extends PostgreSQL using user-defined functions and types [17]. Despite the increasing availability and use of these *inference queries*, they are also more challenging to debug as compared to traditional relational queries. This can be exemplified by the following use case.

SIGMOD'22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

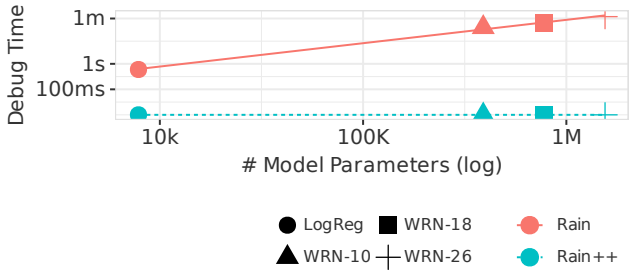

Figure 1: Rain++ (this paper) reduces the time for complaint-driven data debugging by over 7000× to support ~10ms interactive debugging. This complaint is over a join-count query that where the join condition is M.predict(left) = M.predict(right)**.**

Example 1 (Image Classification Inference Query). *Consider an online clothing retailer, where individual sellers upload images and metadata about items to sell. For quality assurance purposes, the retailer wants to cross check the metadata provided by the sellers. One check verifies that the type of clothing in the uploaded image (e.g. trousers or shirt) matches the information in the metadata.*

Elliot, a data scientist, is tasked with creating and monitoring an image recognition model to predict clothing type, and flagging potential metadata errors using the model. Elliot monitors the number of flagged entries with the streaming query:

```
SELECT hour(S.time_added), count(*)
FROM sellerdata AS S
WHERE clothingtype(S.image) != S.clothingtype
GROUP BY hour(S.time_added)
```

where `clothingtype(·)` *is a model that takes an image as input and outputs clothing type. As a sanity check, Elliot sets an alert that triggers when the number of flagged items exceeds 500 within an hour.*

However, should Elliot even believe the alerts? The unique challenge that inference queries introduce is that, *even if the query formulation and the streaming data are correct, the query output can be still be wrong due to errors in the training data.* Mislabeled training examples, or noisy images, or incorrect training metadata can all cause the model to become biased and mispredict, ultimately affecting the inference query output.

Although many systems to debug non-inference queries [1, 28, 38, 39] and analytic workflows [36] exist, there are fewer options for debugging training data when errors are identified in the outputs of inference queries. Approaches such as data 'unit tests' attempt to identify training data errors before training [3, 31]. Influence analysis techniques [21, 30, 46] use labelled mispredictions to identify the training records that most contributed to the misprediction. They do so for differentiable models by estimating the sensitivity (the gradient) of the prediction with respect to each training example. However, neither of these approaches account for how

the predictions are used as part of downstream analytics. Further, end users do not have visibility into the dataflow process to even provide label mispredictions. Ultimately, existing approaches do not account for how the model predictions are used as part of the query, nor their effects on the errors that the user identifies.

The recent Rain [40] system proposed *complaint-driven training data debugging*. Given a user's specification of errors in the inference query result (a complaint), Rain ranks training examples based on their effect on the complaint if deleted. This functionality can be used for model interpretation, by returning relevant training examples that affected a query result, or for training set debugging, by identifying erroneous training examples based on query errors. For example, if Elliot complains that the flagged count in the current hour is too high, Rain ranks training examples on their tendency to reduce the count if removed from the training set.

It is impractical to individually remove each training example, retrain the model, and rerun the query. Thus, Rain leverages influence functions, a form of influence analysis, to approximate the effects of retraining on differentiable ML models (e.g., logistic regression, neural networks). To support SQL queries, which are inherently not differentiable, Rain proposes a provenance-based relaxation of SPJA queries [13]. In short, Rain combines the sensitivity of model predictions to the training examples with the sensitivity of the query result to model prediction probabilities to construct an end-to-end differentiable pipeline. Rain showed that these techniques can identify erroneous training examples far more accurately than prior approaches. Unfortunately, it has performance and scalability challenges that limit its use to small models and training datasets.

Complaint-driven debugging is typically initiated from a data visualization, where the user can easily see anomalies and interactively annotate and specify them as complaints. In this context, it is crucial that the debugging system responds at interactive timescales to not impede the user's analysis flow [25]. However, as shown in Figure 1, Rain's responsiveness quickly degrades beyond models containing a few thousand parameters. Unfortunately, modern deep neural network (DNN) models, such as Wide Residual Networks (WRN) used in image classification, can have hundreds of thousands, or millions or parameters. At such scales, Rain takes minutes to identify and rank erroneous training records for a complaint, whereas our goal is to debug at interactive time scales (< 100ms [12, 26]).

Rain's poor scalability arises from the cost of estimating two types of model sensitivities. The first type is the sensitivity of the model parameters to removing examples from the training set. Instead of analyzing the sensitivity of the loss function directly, which requires retraining the model and rerunning the query, Rain uses a quadratic Taylor approximation of the loss function that is faster. However, it still requires computing the second order derivative (the Hessian) which is expensive. The second type is the sensitivity of model predictions (and the inference query) to changes in the model parameters, whose computation cost increases with the model size. Rain also proposed optimizations to approximate the effects of training set deletions on the relaxation of the query without materializing the quadratic approximation of the loss function. Unfortunately, for a training set of size $|T|$ and d model parameters, Rain still costs $O(d|T|)$, which is untenable for non-trivial models.

Our work builds on three insights. First, a significant amount of computation can be pushed offline by precomputing the quadratic approximation of the loss function (Section 2.4). However, a naive approach requires considerable space and only reduces latency by a constant factor. Thus, our second insight is to build space-efficient approximations of the model loss function that only rely on a tiny subset of the model parameters (Section 3). Although conventional wisdom towards loss function approximation is to choose parameters that are most sensitive to training set perturbations, we show that the exact opposite is true for influence-based complaint debugging. Namely, that the more sensitive a model parameter is, the less the model relies on it when making predictions—in other words, the less it contributes to debugging! In fact, including sensitive parameters introduces numerical instabilities that *degrade* debugging quality. Finally, we observe that, rather than compress the model parameters directly, it is even more effective to directly compress the quadraic approximation of Rain (Section 3).

Rain++ uses these insights to precompute a small number of eigenvalues and eigenvectors of the loss function's inverse Hessian. While matrix compression traditionally computes the largest eigenvalues, we show counter-intuitively that the *smallest* eigenvalues are most appropriate for training data debugging (Section 3). We further develop optimizations when the inference database or the inference query is known apriori (as in Example 1). The former precomputes gradients for inference DB tuples that accelerate *any* future inference query, while the latter incrementally maintains the query gradient as new batches of records are inserted.

These optimizations reduce complaint debugging latency by >7000× from over 1 minute to ∼10ms (Figure 1). The precomputation costs are modest: less than 30 minutes for a WRN-26 neural network model with 1.5M parameters. Beyond scalability, Rain++ addresses two additional limitations in Rain. First, Rain relies on access to the model training infrastructure (training dataset, model definition, and parameters) in order to compute the above derivaties and gradients. However, model *users* rarely have access to this infrastructure. Complaint-driven debugging can be solely performed on the data structures that we propose to precompute, which obviates the need for this access. Second, Rain assumes that deleting errors is always appropriate. However, this is both undesirable when training records are sparse, and incorrect if the errors cannot be fixed by deletions. We propose extensions to support updated-based interventions and illustrate in the experiments how the correct intervention choice is crucial for training data debugging.
To summarize, our contributions include:

- Offline precomputation techniques that both speed up complaint-driven debugging and improve numerical stability.
- Offline precomputation techniques when the inference database is know a priori (e.g., a published dashboard).
- Maintenance-based optimizations for streaming queries where new batches of data are inserted into the inference database.
- Extension of the Rain problem formulation to support interventions that fix, rather than delete, erroneous training examples.
- Extensive evaluations using image (MNIST, FASHION-MNIST, CIFAR10), text (SST2), and tabular (ADULT) datasets, and a variety of linear and neural network models (CNNs, Feed Forward Nets, Logistic Regression, WRNs, LSTMs). Rain++ has comparable or better debugging accuracy, >7000× lower latency, and supports non-deletion interventions.

- In-depth analysis of the conditions when complaint-based debugging can be expected to be effective. We find evidence that complaints based on queries whose outputs significantly incorrect are more likely to accurately identify training errors, which matches the settings when an end-user will identify and submit a complaint. **Critically, we show that knowing how a model is used in the downstream application is critical to accurate and efficient data debugging.** This is leveraged in other problem areas such as domain adaptation [47] in the ML literature, but rarely applied in the data cleaning literature.

2 BACKGROUND AND CHALLENGES

In this section we first present the debugging problem solved by the authors of [40], then we describe how they use influence functions and query gradients to address it. We then highlight the performance bottlenecks addressed in the subsequent sections.

2.1 Problem Overview

An inference query Q is a Select-Project-Join-Aggregation (SPJA) query whose expressions may include model inference calls inside the **SELECT**, **WHERE**, **GROUP BY** clauses, or within inside aggregation functions (e.g., **SUM**, **COUNT**, and **AVG**). The model $\mathcal{M}$ has already been trained on a training set $\mathcal{T}$. Given the inference database $\mathcal{D}$, the database where $\mathcal{M}$ is applied on for prediction, $Q(\mathcal{D}, \mathcal{T})$ executes the query over $\mathcal{D}$.

The user can complain about errors he or she observes by defining violated constraints over the the output of $Q(\mathcal{D}, \mathcal{T})$. In Example 1, Elliot, seeing the number of flagged errors increase, can indicate that the aggregation count of the current hour should be lower than its current value. Rain supports many types of complaints like specifying that a tuple should not exist in $Q(\mathcal{D}, \mathcal{T})$ or that attribute of a tuple in the output, like our aggregation count in Example 1, should be higher, lower or equal to a target value. In all cases we can think of user complaints as boolean functions on the query output that return true if and only if the complaint is resolved. We will use $C(Q(\mathcal{D}, \mathcal{T}))$ to denote the boolean output of a complaint C for the output $Q(\mathcal{D}, \mathcal{T})$.

Given a complaint C, Rain aims to help the user identify the smallest set of modifications to the training set $\mathcal{T}$ so that the complaint on Q is resolved. Focusing on deletions of training examples, the authors of [40] define the problem as follows.

Problem 1. *(Complaint-driven Training Data Debugging) Given a training set $\mathcal{T}$, a inference database $\mathcal{D}$ and a query Q and a complaint C, the goal is to identify the minimum set of training records such that if they were deleted, the complaint would be resolved:*

$$\min_{\Delta \subseteq \mathcal{T}} |\Delta| : C(Q(\mathcal{D}, \mathcal{T} - \Delta)) = \mathit{True}$$

While Rain [40] focused on deletions, Section 5.3 extends the formulation to support a library of custom interventions (e.g., image denoising).

Rain proposed a heuristic solution to this generally intractable problem. Instead of returning a candidate set for deletion, Rain returns a ranked list of training examples of $\mathcal{T}$. Training examples at the top, if removed from $\mathcal{T}$ one by one should push the output of Q towards satisfying C. In Example 1, Rain highly ranks training examples of $\mathcal{M}$ that most reduce the flagged count if removed from the training set.

Even this more tractable version of the problem remains prohibitive as it requires training $|\mathcal{T}|$ models. To sidestep this, the authors of [40] focus on differentiable models where tools from influence functions [21, 37] allows one to approximate retraining. We will discuss next the fundamentals of these techniques. Our work on Rain++ operates on the same principles albeit with a different implementation as we shall see in Section 4.

2.2 Influence Functions

Given a differentiable model like a neural network, there is no closed form solution for the optimal parameters of the training loss functions and thus we cannot just incrementally update them in response to a deletion of a training example. The influence functions framework [37] works around this limitation by constructing a surrogate loss function for which a closed form solution exists and then uses it to derive an approximate solution. In particular, quadratic functions $h(\vartheta)$ are useful surrogate functions because they have closed form solutions to compute the minimizers ϑ^*_h

$$h(\vartheta) = a\vartheta^2 + b\vartheta + \gamma \qquad \vartheta^*_h = -\frac{b}{2a}.$$

Modifications to h like adding a linear function $g(\vartheta) = r\vartheta + s$ can be easily handled as well with an incremental formula

$$\vartheta^*_{h+g} = \vartheta^*_h - \frac{g'(\vartheta^*_h)}{h''(\vartheta^*_h)} = \vartheta^*_h - \frac{r}{2a} = -\frac{b+r}{2a}. \tag{1}$$

Now let us turn to how we can reduce complex optimization problems to the easy cases above. Let $z_i = (x_i, y_i)$ be the i-th training example of $\mathcal{T}$, composed of pair of a feature vector x_i and its corresponding label y_i. Let $\ell(\vartheta, z)$ return the loss of a training example z for a model with parameters ϑ. We define our loss function and its minimizer

$$L(\vartheta) = \sum_{i=1}^{|\mathcal{T}|} \ell(\vartheta, z_i) \qquad \vartheta^* = \arg\min_{\vartheta} L(\vartheta).$$

Let us suppose that we want to estimate the effects of adding a training example $z = (x, y)$. We need to compute

$$\vartheta^*_{new} = \arg\min_{\vartheta} \{L(\vartheta) + \ell(\vartheta, z)\}.$$

The influence function framework reduces $L(\vartheta)$ and $\ell(\vartheta, z)$ to the quadratic surrogate function above by computing their Taylor series approximation

$$L(\vartheta) \approx L(\vartheta^*) + \langle \nabla_\vartheta L(\vartheta^*), \vartheta - \vartheta^* \rangle + \frac{1}{2}(\vartheta - \vartheta^*)^T H_{\vartheta^*}(\vartheta - \vartheta^*)$$
$$\ell(\vartheta, z) \approx \ell(\vartheta^*, z) + \langle \nabla_\vartheta \ell(\vartheta^*, z), \vartheta - \vartheta^* \rangle$$

where the Hessian matrix H_{ϑ^*} is the second derivative of $L(\vartheta)$. Now applying the multivariate version of Equation (1) we get

$$\vartheta^*_{new} \approx \vartheta^* - H^{-1}_{\vartheta^*} \nabla_\vartheta \ell(\vartheta^*, z). \tag{2}$$

For our setting we are not interested in ϑ^*_{new} itself but in functions computed over its output predictions (such as the output of Q that the complaint is specified over). Rain reduces the complaints to a

differentiable function of the model parameters $q(\vartheta)$ by computing the first order Taylor approximation of q

$$q(\vartheta^*_{\text{new}}) \approx q(\vartheta^*) - \nabla_\vartheta q(\vartheta^*) H^{-1}_{\vartheta^*} \nabla_\vartheta \ell(\vartheta^*, z). \quad (3)$$

Naively evaluating the formula is still infeasible. Given a model with d parameters ($\vartheta \in \mathbb{R}^d$), simply inverting the Hessian already takes $O(d^3)$ time and $O(d^2)$ space where d can be 10^6 or more for state of the art neural network architectures.

The good news is that even though we need to evaluate Equation (3) once for every training example intervention $\ell(\vartheta^*, z)$, we can share the computation of $\nabla_\vartheta q(\vartheta^*) H^{-1}_{\vartheta^*}$ across all interventions. The problem thus boils down to solving a single system of linear equations over the unknown vector $w \in \mathbb{R}^d$

$$H_{\vartheta^*} w = \nabla_\vartheta q(\vartheta^*). \quad (4)$$

Yet again, this linear system is still impractical to solve exactly. [21] observes that approximately solving Equation (4) amounts to find an approximate minimizer of the following function

$$\mu(w) = w^T H_{\vartheta^*} w - \langle \nabla_\vartheta q(\vartheta^*), w \rangle.$$

The benefit of this formulation is that $\nabla_w \mu(w) = H_{\vartheta^*} w - \nabla_\vartheta q(\vartheta^*)$ can be easily computed without materializing H_{ϑ^*}. Automatic differentiation frameworks (e.g. Tensorflow) can compute $H_{\vartheta^*} w$ given a vector w in $O(|\mathcal{T}|d)$ time. The Conjugate Gradient (CG) algorithm [18] can then solve the problem exactly using d calls to $\nabla_w \mu(w)$. In practice [21] finds that a constant number of evaluations yields an empirically sufficient approximation. Despite all these improvements we find in our experiments that this step remains the key bottleneck of the approach of [21] and thus Rain, taking more than a minute for a 26 layer Wide Residual Network.

2.3 Query Gradients via Provenance

In this subsection we will outline how Rain translates the complaint C and query Q into a differentiable function $q(\vartheta)$ of the model parameters as required by Equation (3). Notice that this step is required because the inference query Q depends on the discrete, also known as hard, predictions of the model $\mathcal{M}$ which are not differentiable. Rain analyzes Q using provenance polynomials [2, 13] to construct a symbolic representation of the query results. Specifically for an aggregation result this analysis returns a formula that takes the model predictions as input and returns the aggregation result as output. For the case of Example 1, for a tuple i we denote S[i] the clothing type registered in table S, $\mathcal{M}(i)$ the model prediction and hour_added(i) the hour added it was added. Then for the aggregation result for hour h the formula is

$$\sum_{\text{hour\_added}(i)=h} \sum_{j \neq S[i]} \mathbb{1}_{\mathcal{M}(i)=j}.$$

where $\mathbb{1}$ is the indicator function. Similar analyses can be performed for non aggregation results as well. Unfortunately this formula is still not differentiable because it still depends on the hard predictions. To side step this Rain replaces the hard predictions with the probabilities of each prediction in the inference data estimated by $\mathcal{M}$, and replaces boolean operators (AND, OR and NOT) with continuous alternatives. Since the model now emits probabilities for all classes, rather than for the single predicted class, the differentiable function is over the space of all prediction probabilities $P(\vartheta) \in \mathbb{R}^{V \times R}$, where V is the total number of model predictions and R is the number of model classes. This process transforms C to a differentiable function f of $P(\vartheta)$:

$$C(Q(\mathcal{D}, \mathcal{T})) \rightarrow q(\vartheta) = f(P(\vartheta)). \quad (5)$$

Revisiting Example 1, Elliot's potential complaint that the output for hour h should be smaller is translated to the complaint that the following differentiable function should have a smaller value

$$q(\vartheta) = \sum_{\text{hour\_added}(i)=h} \sum_{j \neq S[i]} p_{ij}(\vartheta).$$

2.4 Limitations

To put the pieces together, Rain computes the relaxed provenance polynomial $q(\vartheta)$, and then uses conjugate gradients algorithm to estimate $\nabla_\vartheta q(\vartheta) H^{-1}_\vartheta$. The result is then multiplied with $\nabla_\vartheta l(\vartheta, z)$ for every training record, to compute the ranking criteria.

The primary bottleneck is computing the first and second order model derivatives, especially when d is large, since they are needed when calculating $\nabla_\vartheta q(\vartheta^*)$, the $H_{\vartheta^*} w$ computations for the Conjugate Gradient algorithm, as well as $\nabla_\vartheta l(\vartheta^*, z)$ for all training examples in $\mathcal{T}$. The resulting complexity of the algorithm, even if we ignore the calculation of $\nabla_\vartheta q(\vartheta^*)$, is $O(|\mathcal{T}|d)$. Thus, Rain will not remain interactive when used for models with many parameters, or trained on large training sets.

Fundamentally, a complexity of $O(|\mathcal{T}|d)$ should be expected for any solution to Problem 1 – a solution must, at minimum, evaluate $\mathcal{M}$ over all training examples in order to return a ranking. To go beyond constant factor improvements over Rain, a significant portion of the computation workload needs to be made *complaint independent*, meaning it is independent of the user's query Q and can thus be pushed offline.

Unfortunately naively doing so ends up bringing constant factor improvements at best. Computing H_{ϑ^*} or its inverse offline would end up hurting Rain's performance: Even reading the Hessian or its inverse takes $O(d^2)$ which is slower than $O(|\mathcal{T}|d)$ for most state of the art neural net architectures. Another approach would be to calculate offline and store $H^{-1}_{\vartheta^*} \nabla_\vartheta \ell(\vartheta^*, z)$ for every training example z in $\mathcal{T}$. While we avoid a significant amount of first and second order derivatives at online time, this amounts to only a constant factor improvement since the online complexity remains $O(|\mathcal{T}|d)$. On top of that, the offline cost can be prohibitive, requiring $O(d|\mathcal{T}|^2)$ time and $O(|\mathcal{T}|d)$ space. Clearly neither approach is practical.

3 INSIGHTS FROM OPTIMIZATION

The critical challenge in this paper is computing $H^{-1}_{\vartheta^*} \nabla_\vartheta \ell(\vartheta^*, z)$ for all z in $\mathcal{T}$ using less than $O(d|\mathcal{T}|^2)$ time and $O(d|\mathcal{T}|)$ space. It is also unrealistic to solve one linear system at a time within interactive latencies, our goal is to compress $H^{-1}_{\vartheta^*}$ and quickly process each z.

Given a matrix operation $A \cdot b$ where A is a square matrix, the predominant way to compress A is low rank factorization [8]. This keeps the top eigenvalues and eigenvectors of A, which captures the set of vectors b where $A \cdot b$ most sensitive. This ensures that the accuracy along those sensitive directions will be high.

Despite these guarantees, low rank factorization is *not* appropriate due to interactions between $A = H^{-1}_{\vartheta^*}$ and $b = \nabla_\vartheta \ell(\vartheta^*, z)$ unique

to our problem. This section provides intuition for why Rain++ instead compresses $H_{\vartheta^*}^{-1}$ using its *smallest* eigenvalues and their eigenvectors.

First, the smallest eigenvalues are most accurate for representing the effects of training set changes on the model predictions. In fact, recent work in deep learning optimization suggests that state-of-the-art neural networks training loss gradients are concentrated on the subspace spanned by the smallest eigenvectors of $H_{\vartheta^*}^{-1}$ [14]. We illustrate this using a simple example based on a linear regression model with four training examples. The features X and targets y of the model are the following

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 10^{-3} \\ 0 & -10^{-3} \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 1 \\ 3 \end{bmatrix}$$

If $\vartheta = (\vartheta_1, \vartheta_2)$ are the two feature weights of the model, then training the model amounts to minimizing the squared loss

$$\begin{aligned} L(\vartheta) &= \sum_{i=1}^{4} \ell(\vartheta, z_i) = \sum_{i=1}^{4} (\langle x_i, \vartheta \rangle - y_i)^2 \\ &= (\vartheta_1 - 1)^2 + (-\vartheta_1 - 3)^2 + (10^{-3}\vartheta_2 - 1)^2 + (-10^{-3}\vartheta_2 - 3)^2 \\ &= 2\vartheta_1^2 + 4\vartheta_1 + 10 + 2 \cdot 10^{-6}\vartheta_2^2 + 4 \cdot 10^{-3}\vartheta_1 + 10 \end{aligned}$$

The optimal parameters are $\vartheta^* = (-1, -10^3)$, and inverse Hessian is

$$H_{\vartheta^*}^{-1} = \begin{bmatrix} \frac{\partial^2 L(\vartheta)}{\partial^2 \vartheta_1} & \frac{\partial^2 L(\vartheta)}{\partial \vartheta_1 \partial \vartheta_2} \\ \frac{\partial^2 L(\vartheta)}{\partial \vartheta_2 \partial \vartheta_1} & \frac{\partial^2 L(\vartheta)}{\partial^2 \vartheta_2} \end{bmatrix}^{-1} = \begin{bmatrix} 4 & 0 \\ 0 & 4 \cdot 10^{-6} \end{bmatrix}^{-1} = \frac{1}{4} \begin{bmatrix} 1 & 0 \\ 0 & 10^6 \end{bmatrix}.$$

The top eigenvector of $H_{\vartheta^*}^{-1}$ is $(0, 1)$ with eigenvalue $0.25 \cdot 10^6$. For training example $z_i = (x_i, y_i)$, its training gradient is $\nabla_\vartheta \ell(\vartheta^*, z_i) = 2(\langle x_i, \vartheta^* \rangle - y_i)x_i$. Thus for z_1 and z_3 we have

$$\nabla_\vartheta \ell(\vartheta^*, z_1) = \begin{bmatrix} -4 \\ 0 \end{bmatrix} \quad \nabla_\vartheta \ell(\vartheta^*, z_3) = \begin{bmatrix} 0 \\ -4 \cdot 10^{-3} \end{bmatrix}.$$

Examples z_2 and z_4 have the same gradients as z_1 and z_3 albeit with opposite signs. Note that the gradients of all of these examples are concentrated around the inverse Hessian's bottom eigenvector $(1, 0)$ direction, rather than the top eigenvector. Thus, multiplying the smallest eigenvalues of $H_{\vartheta^*}^{-1}$ with the loss gradients will more accurately approximate $H_{\vartheta^*}^{-1} \nabla_\vartheta \ell(\vartheta^*, z)$.

In general, it's a problem if model predictions rely heavily on model parameters that are overly sensitive to small training set changes, which are precisely captured by the directions of the largest eigenvectors of $H_{\vartheta^*}^{-1}$.

Second, the largest eigenvalues greatly *reduce*, rather than improve, the accuracy of influence function approximations. Suppose we add a new training example z_5 with $x_5 = (0, 1)$ and $y_5 = 1$. The new loss term $(\vartheta_2 - 1)^2$ in $L(\vartheta)$ dominates all existing ϑ_2 terms. The new optimal parameters should be $\approx (-1, 1)$, yet our approximation using Equation (2) is wildly off

$$\vartheta^*_{\text{new}} \approx \vartheta^* - H_{\vartheta^*}^{-1} \nabla_\vartheta \ell(\vartheta^*, z_5) \approx \begin{bmatrix} -1 \\ 5 \cdot 10^8 \end{bmatrix}.$$

The reason is that influence functions employ a Taylor approximation that is accurate only in a small neighborhood of the original solution ϑ^*. Sensitive parameters like ϑ_2 can change considerably when the training set is perturbed, and render the Taylor approximations highly inaccurate.

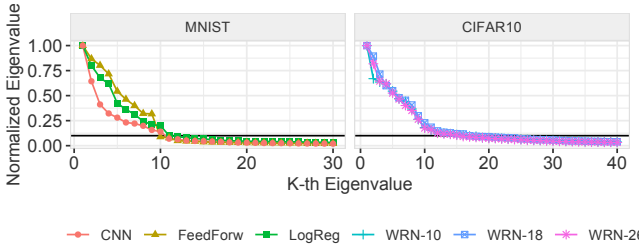

Figure 2: Caching the top-k Hessian eigenvalues for MNIST and CIFAR10 (with $K \approx 10$) is sufficient for influence functions.

Third, accurately estimating the top eigenvalues themselves suffers from numerical stability issues. Modern NN architectures, such as computer vision models, are low rank, and 99.99% of the eigenvalues are near-zero [11]. Since the eigenvalues of the inverse Hessian are reciprocal to the ones of the Hessian, calculating the top eigenvalues of the inverse suffers from numerical stability issues.

Looking ahead: The above points highlight that large eigenvalues of the $H_{\vartheta^*}^{-1}$ should not be used for influence analysis, and that the smallest eigenvalues should instead be used. Further, Figure 2 shows that 10-15 eigenvalues are sufficient to store for even million-parameter models. Surprisingly, in addition to its performance benefits, this also improves the approximation accuracy and robustness compared to computing and using the full inverse Hessian matrix.

4 OUR APPROACH

As we discussed in Section 3, Rain++ computes $H_{\vartheta^*}^{-1} \nabla_\vartheta \ell(\vartheta^*, z)$ for all z in T offline to accelerate the online evaluation of Equation (3). The basis of our implementation is the approximation of $H_{\vartheta^*}^{-1}$ through the top eigenevectors of H_{ϑ^*}. Let v_i and λ_i be the eigenvectors of H_{ϑ^*} in descending order. Rain++ computes only the top-k v_i and λ_i to replace H_{ϑ^*} in Equation (3) with the following surrogate

$$\tilde{H}_{\vartheta^*} = \sum_{i=1}^{k} \lambda_i v_i v_i^T.$$

The matrix above would coincide with the exact Hessian for the choice of $k = d$. The crux of our implementation is to approximate the $H_{\vartheta^*}^{-1} \nabla_\vartheta \ell(\vartheta^*, z)$ while materializing as few of its intermediates of the computation as possible. Rain++ computes the top-k v_i and λ_i without first materializing the uncompressed H_{ϑ^*}. $H_{\vartheta^*}^{-1} \nabla_\vartheta \ell(\vartheta^*, z)$ is then computed directly in a compressed format without needing to materialize $\nabla_\vartheta \ell(\vartheta^*, z)$ first. Further, we will remark on heuristics to estimate k – in our experiments, k is on the order of 10 or 15, as compared to the $>10^6$ model parameters.

4.1 The Lanczos Algorithm

Given H_{ϑ^*}, computing its top-k eigenvalues and eigenvectors would be a straightforward task using any linear algebra library. As we have noted before though, even computing the full Hessian would take $O(|\mathcal{T}|d^2)$ time which in practice would be prohibitive.

The Lanczos Algorithm [23] is a generalization of the CG algorithm that allows us to compute eigenvalues and eigenvectors

of H_{ϑ^*} without requiring access to H_{ϑ^*} (recall that CG was used to simply compute $\nabla_\vartheta q(\vartheta^*)H_{\vartheta^*}^{-1}$). Similar to CG, the Lanczos Algorithm only requires access to an oracle that, given a v, computes $H_{\vartheta^*}v$. This again can be done using backpropogation in an auto-differentiation framework such as TensorFlow. For $k \ll d, |\mathcal{T}|$ as we discussed, the complexity of this algorithm is $O(k|\mathcal{T}|d)$.

4.2 Gradient Compression

Replacing H_{ϑ^*} with $\tilde{H}_{\vartheta^*}$ in Equation (3) we get

$$q(\vartheta^*_{new}) \approx q(\vartheta^*) - \nabla_\vartheta q(\vartheta^*) \sum_{i=1}^{k} \frac{1}{\lambda_i} v_i v_i^T \nabla_\vartheta \ell(\vartheta^*, z).$$

We can reorganize the expression using vector notation to highlight the opportunity to compress $H_{\vartheta^*}^{-1}\ \nabla_\vartheta \ell(\vartheta^*, z)$:

$$q(\vartheta^*_{new}) \approx q(\vartheta^*) - \sum_{i=1}^{k} \langle \nabla_\vartheta q(\vartheta^*), v_i \rangle \frac{1}{\lambda_i} \langle v_i, \nabla_\vartheta \ell(\vartheta^*, z) \rangle. \quad (6)$$

The compressed version of $H_{\vartheta^*}^{-1}\ \nabla_\vartheta \ell(\vartheta^*, z)$ that Rain++ stores is thus

$$b_z \longleftarrow \left[\frac{1}{\lambda_1} \langle v_1, \nabla_\vartheta \ell(\vartheta^*, z) \rangle, \cdots, \frac{1}{\lambda_k} \langle v_k, \nabla_\vartheta \ell(\vartheta^*, z) \rangle \right].$$

Observe that b_z has size k much smaller than the uncompressed d.

Computing all the b_z gives rise to a time-space trade-off. Computing large batches of $\nabla_\vartheta \ell(\vartheta^*, z)$ to leverage GPU parallelism requires large amounts of GPU memory, a limited resource. The situation is more dire compared to model training since we compute one d dimensional gradient for each training example in the batch and not one gradient for the whole batch as in stochastic gradient descent.

Rain++ computes the $\langle v_i, \nabla_\vartheta \ell(\vartheta^*, z) \rangle$ directly without materializing $\nabla_\vartheta \ell(\vartheta^*, z)$. To reduce the number of variables during gradient computation, Rain++ views the calculation of each of the k projections as the derivative of a function of one scalar variable h

$$\langle v_i, \nabla_\vartheta \ell(\vartheta^*, z) \rangle = \frac{\partial \ell(\vartheta^* + h v_i, z)}{\partial h}\Big|_{h=0}.$$

Backpropagation here ends up calculating $\nabla_\vartheta \ell(\vartheta^*, z)$ and projecting it to v_i which does not help. Forward mode differentiation [5], available in frameworks like Tensorflow, can calculate $\langle v_i, \nabla_\vartheta \ell(\vartheta^*, z) \rangle$ on the fly while evaluating $\ell(\vartheta^*, z)$. As a result $\nabla_\vartheta \ell(\vartheta^*, z)$ is never materialized. The dramatic memory reduction allows for significantly larger batch sizes that more than make up the cost of the $k \ll d$ passes needed, one for each v_i.

Storing the k vectors v_i, and b_z for each training example z, reduces space complexity from $O(|\mathcal{T}|d)$ to $O(kd + k|\mathcal{T}|)$. Given $\nabla_\vartheta q(\vartheta^*)$, they also reduce the online computation of Equation (6) for all training examples from $O(|\mathcal{T}|d)$ in Rain to $O(kd + k|\mathcal{T}|)$. In our experiments, for a WRN-26 model of 1.5M parameters and 50K training examples, setting $k = 20$ our technique reduces the cost by one order of magnitude. For this setting, our forward mode based gradient compression is 4 times faster than computing the uncompressed gradients with backpropagation.

4.3 Choosing the number of eigenvalues

It is clear that the choosing the right number of eigenvalues k is critical for Rain++. Given that there is no uniform choice of k that works for all datasets, it is important to design heuristics to identify an appropriate setting. Prior work [14] already suggests that for many deep learning architectures the number of dominant eigenvalues of the Hessian is a small multiple of the number of classes of the classification task at hand. To refine this rough estimate, we need to determine the location of the last dominant eigenvalue. As we can see in Figure 2, at the beginning of the spectrum of the Hessian eigenvalues decrease rapidly with λ_{i+1}/λ_i being significantly lower than one. This behaviour continuous until we reach an inflection point after which the drop-off stops and λ_{i+1}/λ_i spikes to a value close to one. Our heuristic chooses initializes k to the number of classes, which is 10 for the case of Figure 2 and then scans the spectrum to identify the first inflection point. In our experiments, we observe that the chosen k consistently achieves performance that is close to the optimal choice.

5 OPTIMIZATIONS AND EXTENSIONS

The previous sections focused on pushing the computations of first and second order derivatives of $\mathcal{M}$ over the training set offline. In our discussions we have ignored the cost of computing $\nabla_\vartheta q(\vartheta^*)$ which can become the new bottleneck given our optimizations. In this section we discuss two optimizations to address this.

5.1 Known inference database

One of the key cases where Rain++ can accelerate the computation of $\nabla_\vartheta q(\vartheta^*)$ is when the inference database $\mathcal{D}$ is known offline. As we saw in Equation (5), Rain relaxes the complaints C in differentiable functions q that operate on top of the $V \times R$ matrix of prediction probabilities $P(\vartheta)$ of each inference of $\mathcal{M}$ over $\mathcal{D}$. An element $p_{ij}(\vartheta)$ of $P(\vartheta)$ corresponds to the probability of class j assigned to the i-th inference example of $\mathcal{D}$. Using the multi-variate chain rule on the equality of Equation (5) we get

$$\nabla_\vartheta q(\vartheta^*) = \sum_{i=1}^{V} \sum_{j=1}^{S} \frac{\partial f(P(\vartheta^*))}{\partial p_{ij}} \nabla_\vartheta p_{ij}(\vartheta^*). \quad (7)$$

Equation (7) decomposes the sensitivity of the relaxed complaint q in two distinct factors. The first one is the sensitivity of q to the changes of the probabilities in $P(\vartheta)$ expressed by $\partial f(P(\vartheta^*))/\partial p_{ij}$. The second is the sensitivity of the prediction probabilities to model parameter changes $\nabla_\vartheta p_{ij}(\vartheta^*)$. Equation (7) also shows that despite the fact that there is an infinite number of potential complaints, all complaint gradients can be expressed as a linear combination of a finite base of the VR gradients $\nabla_\vartheta p_{ij}(\vartheta^*)$.

This gives a concrete approach towards accelerating the computation of $\nabla_\vartheta q(\vartheta^*)$. We can compute offline all the VR gradients $\nabla_\vartheta p_{ij}(\vartheta^*)$ as well as the matrix $P(\vartheta^*)$. During the online computation, we can construct the function f that corresponds to the user's complaint. Since $\partial f(P(\vartheta^*))/\partial p_{ij}$ depends only on $P(\vartheta^*)$ and the complaint and $\nabla_\vartheta p_{ij}(\vartheta^*)$ is already computed, we can compute $\nabla_\vartheta q(\vartheta^*)$ without requiring any additional model inference or derivative.

Unfortunately it is prohibitive to store $\nabla_\vartheta p_{ij}(\vartheta^*)$ for large models as it takes $O(VRd)$ space. However the computation of Equation (6) requires only the k projections $\langle \nabla_\vartheta q(\vartheta^*), v_i \rangle$. Applying this projection to Equation (7) we get

$$\langle \nabla_\vartheta q(\vartheta^*), v_i \rangle = \sum_{i=1}^{V} \sum_{j=1}^{S} \frac{\partial f(P(\vartheta^*))}{\partial p_{ij}} \langle \nabla_\vartheta p_{ij}(\vartheta^*), v_i \rangle. \quad (8)$$

Thus storing only $\langle \nabla_\vartheta p_{ij}(\vartheta^*), v_i \rangle$ is sufficient, reducing space to $O(VRk)$. The eigenvectors v_i are also no longer required for the online computation so only $O(VRk + k|\mathcal{T}|)$ space is needed which is independent of d. This applies to the online time complexity as well where given $\partial f(P(\vartheta^*))/\partial p_{ij}$, we only require $O(VRk + k|\mathcal{T}|)$. In Figure 7 we show that this optimization can reduce the cost of computing $\nabla_\vartheta q(\vartheta^*)$ by three orders of magnitude. For WRN-26 model of 1.5M parameters and V = 10K, R = 10 and k = 20 using forward mode gradient compression is 12 times faster than just calculating the gradients with backpropagation. The increased speed up compared to Section 4.2 is because for each example i backpropation does a single forward pass to compute all class probabilities $p_{ij}(\vartheta^*)$ but needs to do one backward pass for each class to compute all $\nabla_\vartheta p_{ij}(\vartheta^*)$. In contrast, forward mode calculates $\langle \nabla_\vartheta p_{ij}(\vartheta^*), v_i \rangle$ for all j in a single forward pass.

5.2 Streaming queries

In Section 1, we discussed another important setting where interactive response times are critical, the case where there is an incoming stream of inference examples. Although Equation (8) is in theory always applicable, as the stream increases in size, computing the projections of $\nabla_\vartheta q(\vartheta^*)$ from scratch becomes increasingly more costly. In this section we will discuss how we can incrementally update $\nabla_\vartheta q(\vartheta^*)$ for complaints over streaming queries.

For simplicity, we focus on a streaming aggregation query Q. Let us assume that we start with an inference database $\mathcal{D}$ and after a single tuple insert we get a database $\mathcal{D}'$. Since SPJA queries are incrementally maintainable, we know that there is a query ΔQ, a delta query as it is usually called, that efficiently computes the difference in the value of Q

$$\Delta Q = Q(\mathcal{D}'_{\mathcal{M}}) - Q(\mathcal{D}_{\mathcal{M}}).$$

ΔQ is itself an SPJA query that Rain can analyze and relax just like it would do for the original query Q. Let $h(\vartheta)$ be the relaxation of $Q(\mathcal{D}_{\mathcal{M}})$, $\Delta h(\vartheta)$ be the relaxation of ΔQ and $h'(\vartheta)$ be the relaxation of $Q(\mathcal{D}'_{\mathcal{M}})$. The relaxation of Rain preserves addition so we have

$$\nabla_\vartheta h'(\vartheta^*) = \nabla_\vartheta h(\vartheta^*) + \nabla_\vartheta \Delta h(\vartheta^*).$$

Given $\nabla_\vartheta h'(\vartheta^*)$ and $h'(\vartheta^*)$, which we can compute by a similar addition rule, we can compute any complaint gradient on top of $Q(\mathcal{D}'_{\mathcal{M}})$ via the chain rule. Thus to the extent that ΔQ depends only on a small number of model inferences, we can incrementally compute the complaint gradient $\nabla_\vartheta q(\vartheta^*)$ efficiently. As a canonical example, in Figure 8 we will study the case of streaming class frequency counts. For this case, the update cost depends on the only on the size of the incremental update which allows Rain++ to scale to very large databases $\mathcal{D}$.

5.3 Non-Deletion Interventions

The above discussion is focused on the context where query complaints (and the model mispredictions) can be fixed by deleting corrupted training examples. However, there may be other valid interventions. For instance, the user may wish to apply a low-pass filter to fix images with random or salt-and-pepper noise, or to set erroneous numerical attributes to a default or median value. In addition, when the set of relevant training records is limited (e.g., there are few examples for a given class), deleting the corrupted records is undesirable as it reduces the effective number of samples that are available for training.

We now describe a simple extension to the problem formulation to support interventions that update a training example z to z′. Such interventions can change the features, the label, or both. We can model this as deleting z and adding z′

$$\vartheta^*_{new} = \arg\min_\vartheta \{L(\vartheta) - \ell(\vartheta, z) + \ell(\vartheta, z')\}$$

Following the similar derivation steps as in Section 2.2 produces the following approximation

$$q(\vartheta^*_{new}) \approx q(\vartheta^*) - \nabla_\vartheta q(\vartheta^*) H^{-1}_{\vartheta^*} \left(\nabla_\vartheta \ell(\vartheta^*, z) - \nabla_\vartheta \ell(\vartheta^*, z')\right).$$

Thus, Rain can use the optimizations described in this paper to approximate the effects of any per-record intervention, and rank them based on how well they address the query complaint. In our experiments, we show the importance of corruption-relevant interventions on query complaints. We implement this by precomputing the interventions on all training records, along with their corresponding offline data structures. We leave policy decisions, such as how to choose interventions to consider, as well as techniques that avoid applying and materializing all possible interventions, to future work.

6 EXPERIMENTS

Our experiments seek to understand how the number of materialized eigenvalues affects Rain++'s debugging quality, offline, and online runtimes. Our comparisons against the baseline Rain system finds that Rain++ maintains or improves debugging quality and reduces online runtimes by orders of magnitude, while required modest amounts of offline precomputation times. We further study the characteristics of complaints, as well as types of intervention, that affect debugging quality.

6.1 Experimental Settings

The optimal number of eigenvectors k depends on the hessian's spectral properties, which varies based on datasets, tasks and model architectures. Thus we vary these three dimensions.

6.1.1 Datasets & Models. Scalability becomes a major factor as the number of model parameters increases. Thus we focus on settings that use deep neural networks (DNNs). We use 3 object classification image datasets, and a sentiment analysis NLP dataset. We also use a tabular dataset to show that Rain++ is competitive even on models with fewer parameters that are not overparameterized.

- **MNIST** [24] contains 70k gray scale 28×28 pixel images of handwritten digits 0-9. 60k are used for training and 10k for testing. The model classifies each image with the depicted diget. We traied three models: a logistic regression model with 7850 parameters, a two layer feed-forward network with 1.8M parameters, and a three layer CNN with 1.2M parameters.
- **Fashion-MNIST** [43] is a harder version of **MNIST**. It has the same number of image dimensions, but the images are of clothing from 10 clothing classes. We use the same models as MNIST.
- **CIFAR-10** [22] contains 60k 32×32 color images of 10 different object classes; 50k are used for training. For this classification task, we use three Wide Residual Network (WRN) models [45] with 10,

18 and 26 layers (390K, 778K and 1.55M parameters respectively). This is a harder task than **MNIST** and **Fashion-MNIST**.
- **SST-2** [32], or the Stanford Sentiment Treebank v2, is a binary sentiment analysis dataset. The training set contains 67349 sentence fragments labelled as positive or negative sentiment. For this binary classification task, we use a LSTM based classifier with 3.4M parameters.
- **ADULT** [7] is a tabular dataset that predicts whether a person makes more or less than 50K$ per year, given their census information. DNNs often fail to offer competitive performance on tabular datasets as compared to simpler alternatives like linear models or decision trees [29]. We thus use a logistic regression classifier with 50 parameters.

6.1.2 Training Set Errors. Training examples can have errors in the features, labels, or both; the errors can be random or systematic over the training set. We generate systematic corruptions by choosing a subset of the training set that satisfies a feature or label-based predicate, and adding errors to a random subset of those examples. Random corruptions are uniformly distributed in the dataset. Table 2 and Table 3 summarize the corruption and rates we use for label and feature-based corruptions.

- **Class-conditional Label Error** chooses a class from the training set, and flips a percentage (the corruption rate) of those labels to another class. For example, we flip 40% of **MNIST** '1' labels to '7' ultimately corrupts 4% of the total training set. We vary the corruption rates in 10% increments.
- **Feature Noise** adds Salt & Pepper and gaussian blur to a random or systematic subset of the image training examples. Salt & Pepper randomly sets 30% of the image pixels to either 0 or 1 with equal probability. Gaussian blur convolves the image using a Gaussian kernel of σ = 2px, resulting in a blurred image. Systematic corruption is done by corrupting a subset of examples from one class.

Note that the random corruption rates are over the *full training set*, whereas class-conditional rates are with respect to the *subset* of the training set with the corrupted label value. We also use Salt & Pepper noise to evaluate non-deletion interventions in Section 6.9

6.1.3 Complaints. We evaluate complaints over three types of aggregation queries shown in Table 1. The complaint specifies that the aggregation output is either too high or too low, depending on its value as compared to the ground truth.

6.1.4 Measures. We evaluate debugging quality by considering the precision and recall of each ranked training point. Following Rain [40], we summarize the quality of the top-k results using AUC_R. Let r_i be the percentage of correctly identified corrupted training examples in the top-i ranked points. AUC_R computes the average r_i up to the true number of corruptions N, i.e. $\frac{1}{N}\sum_{i=1}^{N} r_i$. The result is divided by its maximum value to derive a normalized score in $[0, 1]$. We also evaluate offline and online runtimes.

6.1.5 Implementation. Rain and Rain++ are implemented in JAX [20], an automatic differentiation framework on top of XLA [35]. All experiments are run on a Google Cloud **n1-standard-8** machine with one NVIDIA V100 GPU. Runtimes assume that the all code required for the GPU acceleration is precompiled. This is possible for the gradients needed for hessian vector products, gradients for each training example, and for streaming queries, because they are known in advance. In general however, query gradients depend on the user complaint and add additional overhead (8sec on unoptimized code)—deeper integration between Rain++ and NN compilers like XLA to reduce this overhead is promising for future work.

Q_1 SELECT COUNT(*) FROM LEFT L, RIGHT R WHERE predict(L) = predict(R)
Q_2 SELECT COUNT(*) FROM D WHERE predict(*)= {class}
Q_3 SELECT AVG(predict(*)) FROM D

Table 1: Summary of query templates used in the experiments.

Dataset	Corruption	Rate
MNIST	1 → 7	10 - 40%
Fashion-MNIST	pants → sneakers	10 - 40%
CIFAR-10	automobile → horse	10 - 40%
SST2	negative → positive	10 - 40%
ADULT	<50k$ → ≥50k$	10 - 40%

Table 2: Summary of label corruptions.

Type	Affected Classes	Rate
Gauss. Blur (σ = 2)	1 (MNIST), pants (Fashion)	10 - 100%
Salt & Pepper (30%)	auto (CIFAR)	70 - 100%
Gauss. Blur (σ = 2) Salt & Pepper (30%)	All classes	10 - 40%

Table 3: Summary of feature corruptions to image datasets.

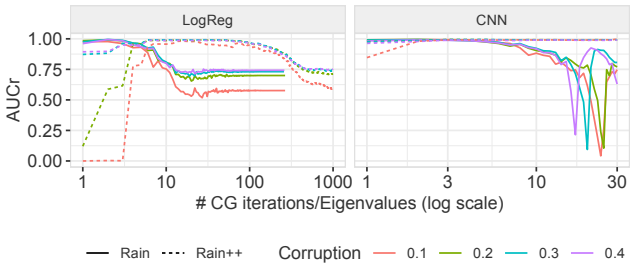

Figure 3: Q_1 AUC_R varying the number of CG iterations and eigenvectors for corruption rates 0.1-0.4.

6.2 Effects of small eigenvalues

Our first experiments illustrate how small eigenvalues of H_{θ^*} degrade complaint-based influence analysis.

We use the join-count query Q_1 on MNIST, where digits 0 – 4 (LEFT) are joined with digits 5 – 9 (RIGHT). The ground truth query should return 0. We evaluate Logistic Regression and CNN models over class conditional label noise. The compaint specifies that the output should be lower.

Figure 3 shows how the number of CG iterations (for Rain, solid lines) and eigenvalues (for Rain++, dashed lines) affect debugging quality; line colors depict corruption rate. Since the debugging

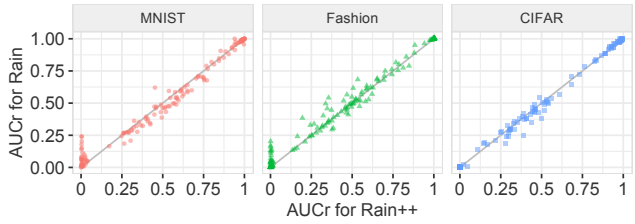

Figure 4: Peak AUC_R Rain vs Rain++

quality quickly reaches a peak AUC_R (nearly 1 in all cases) we plot the x-axis in log scale.

The location of the peak matches the eigenvalue spectra in Figure 2(left), where the normalized eigenvalues with respect to the maximum eigenvalue is near-zero after 10 eigenvalues. Rain converges to its peak more quickly because CG solves for any orthogonal vectors to minimize the objective function, whereas Lanczos used in Rain++ is restricted to eigenvectors of the Hessian. Increasing the number of iterations/eigenvalues beyond the peak ultimately degrades AUC_R due to numerical instability, as small eigenvalues dominate the gradient analysis. As the number of iterations/eigenvalues converges to the total number of model parameters d, we expect both approaches to be equivalent.

Non-convex models such as the CNN are typically trained to reach an approximate local minima because it is faster to compute and reduces the risk of overfitting. As a result, the Hessian can potentially have small negative eigenvalues whose eigenvectors correspond to directions that increase the loss. This is why Rain's AUC_R fluctuates widely beyond the peak. In contrast, Rain++ uses positive eigenvalues, and does not suffer from this instability.

Takeaway: Small and negative eigenvalues of the Hessian degrade AUC_R. Rain++ avoids these issues by only using the top k *eigenvalues.*

6.3 Baseline Comparison: Debugging Quality

Figure 4 compares the peak AUC_R for Rain and Rain++ across all models, image datasets, corruptions and their rates, and queries Q_1 and Q_2. For Q_1, the join condition is over two disjoint subsets of the inference dataset, so the aggregation is expected to be 0. For Q_2, we filter on '0' digit, pants, and automobiles for the three datasets. We vary the number of CG iterations/eigenvalues and report peak AUC_R. Each point compares the peak AUC_R for both approaches.

The vast majority of points are near the gray y = x line, and shows that the debugging qualities are comparable. Additionally, the peak AUC_R for both approaches is interspersed across $[0, 1]$ indicating that not all complaints are effective for all settings. We study the conditions when a complaint can be expected to be effective for debugging in Section 6.8.

Takeaway: Rain and Rain++ report comparable peak AUC_R.

6.4 Number of eigenvalues

How does the number of eigenvalues affect AUC_R, and how closely does our heuristic for picking k in Section 4.3 get to the peak AUC_R? Figure 5 focuses on Rain++ and studies how AUC_R varies with the number of eigenvalues. We report the percentage of the peak AUC_R, averaged over all corruptions and queries Q_1 and Q_2. We exclude

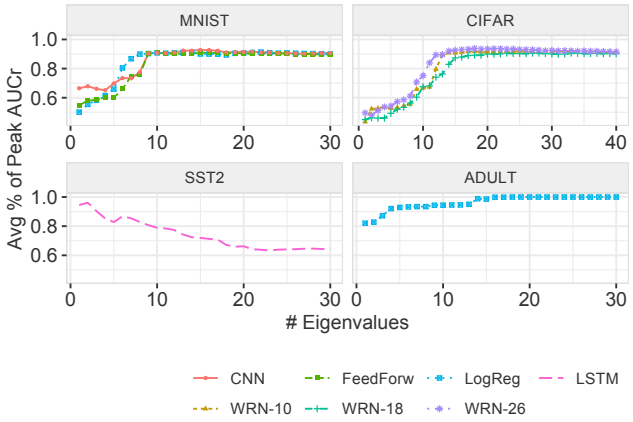

Figure 5: Percentage of peak AUC_R for varying eigenvalues, averaged over all corruption types and rates.

results when Rain++ is ineffective (peak $AUC_R \leq 0.1$) but the results do not change if they are included; for **SST2**, we report results for Q_2 and for the **ADULT** on Q_3.

Across different models for each dataset, the best choice for k does not change significantly. Furthermore, our heuristic for choosing k would select 10, 13, 2 and 6 for the four datasets, which are near optimal. Interestingly, even though the LSTM model for SST2 has the most parameters (3.4M), only two eigenvalues are needed for complaint-based debugging.

Takeaway: Number of eigenvalues for peak AUC_R is empirically robust to model size for the same dataset, and is close to the number of classes.

6.5 Baseline Comparison: Online Runtime

Figure 6 reports the end-to-end online runtimes to compute influence scores for all training examples in the MNIST and CIFAR10 datasets, for a Q_1 complaint. We run Rain, and run Rain++ without the query-gradient optimizations in Section 5. We use a single corruption setting, since it does not affect runtime performance.

Rain++ reduces runtimes by over an order of magnitude even when compared to a single CG iteration. Interestingly, runtimes for **CIFAR-10** are longer than for **MNIST** despite fewer model parameters. CNNs and WRNs reuse the parameters for many operations in their convolutional layers and thus their gradient computations is more costly. Further, sequential layer operations in deeper models like WRNs are more expensive because they are not parallelizable.

In this section we will compare the online time required by Rain and Rain++ to return the scores for all training set interventions given a complaint on the query output . This includes computing $\nabla_\vartheta q(\vartheta^*)$ and using it as a part of the influence calculations for Rain and Rain++. Here we will focus on the performance improvements of Rain++ without the use of the optimizations of Section 5.

Table 4 breaks down the runtimes into individual steps. Computing $\nabla_\vartheta q(\vartheta^*)$ is common to both approaches, however Rain must also computer CG, multiply $\nabla_\vartheta q(\vartheta^*)H_{\vartheta^*}^{-1}$ with each $\nabla_\vartheta \ell(\vartheta^*, z)$ to compute each training example's score (Rain score). We report Rain end-to-end time for 2 CG iterations for **MNIST** and 4 iterations

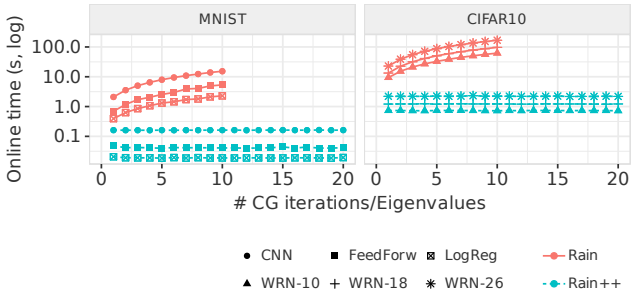

Figure 6: Online complexity varying eigenvalues.

		Rain-only			
Model	$\nabla_\vartheta q(\vartheta^*)$	**CG iter**	**Rain score**	**Rain**	**Rain++**
LogReg	0.02	0.23	0.16	0.61	0.02
CNN	0.16	1.46	0.48	3.50	0.16
FeedForw	0.04	0.46	0.16	1.14	0.04
WRN-10	0.73	5.9	2.92	26.91	0.73
WRN-18	1.21	9.26	2.91	41.02	1.22
WRN-26	2.20	15.86	4.63	71.08	2.20

Table 4: Breakdown of online runtime (sec) for Rain and Rain++ ($\nabla_\vartheta q(\vartheta^*)$ is shared). Rows 1 – 3 are for MNIST, 4 – 6 for CIFAR10.

for **CIFAR-10**, which typically achieves close to peak AUC_R in our experiments, and Rain++ using 20 eigenvalues. CG is the bottleneck for Rain, whereas computing query gradients ($\nabla_\vartheta q(\vartheta^*)$) is the bottleneck for Rain++ on complex models like WRNs. We will evaluate query gradient optimizations next.

Takeaway: Rain++ reduces runtimes by orders of magnitude, but is bottlenecked by computing the query gradient $\nabla_\vartheta q(\vartheta^*)$.

6.6 Query Gradient Optimizations

Figure 7 reports the runtime optimization benefits when the inference database is known apriori (Section 5.1). We focus on **CIFAR-10** since its results are representative. The horizontal line corresponds to the unoptimized cost to compute the query gradient $\nabla_\vartheta q(\vartheta^*)$. Computing the gradient for each test example dominates the query gradient runtime, so precomputing them reduces the runtime by over an order of magnitude, and in effect, eliminates the computational bottleneck. As a result, Rain++ can compute influence scores for all experimental settings in interactive time. **For WRN-26 on CIFAR-10, our suite of optimizations reduces the end-to-end debugging time from over 1.18 minutes using Rain, to less than 10ms using Rain++: a 7000× reduction.**

Figure 8 reports the incremental maintenance cost of ΔQ_2 is run over a streaming database that updates in varying update sizes. We set k = 20. This is akin to the fashion monitoring use case described in the introduction. We see that the incremental update cost varies with the update size and is independent of the test database size. In fact, updates sizes of up to one thousand records can update in under 500ms because the records can be computed on the GPU in one batch. Larger update sizes must be split and run on the GPU in serial order. Smaller update sizes underutilize the GPU, which

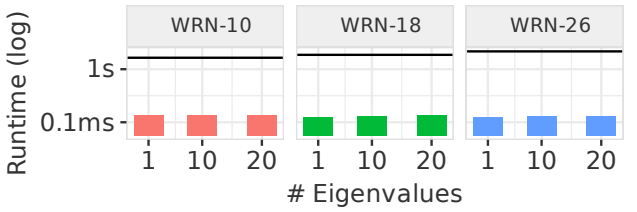

Figure 7: Query gradient optimization for CIFAR-10. Horizontal line is unoptimized time.

is why the curve is flat. The interesting trade-off between latency and throughput to best utilize the hardware is left for future work.

Takeaway: the query gradient optimizations leverage the known inference database or query to ensure interactive debugging times.

6.7 Offline Precomputation Time

Figure 9 reports the offline costs to precompute the gradients for the training set (Section 4) as well as the query gradients when the inference database is known (Section 5.1). We vary the number of eigenvalues to precompute, and mark k = 10 with a vertical line. The overall costs are quite reasonable—for instance, at k = 20, it takes 20 minutes to precompute gradients for the 26 layer WRN model. This corresponds to the time for Rain to answer 15 Q_1 complaints using 2 CG iterations (see Table 4).

Takeaway: Offline preprocessing times are comparable to running Rain for a dozen complaints. We believe it is reasonable enough to perform as a preprocessing step before releasing a model.

6.8 When Are Complaints Useful?

Section 6.3 showed that correctly expressed complaints can still be ineffective at training set debugging. In response, we seek to understand the properties of a query complaint that affect AUC_R. Intuitively, we should expect that it depends on the relationship between the corruption and how it affects the query. At the extreme,

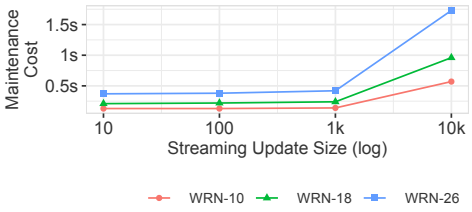

Figure 8: Maintenance cost for varying stream update sizes.

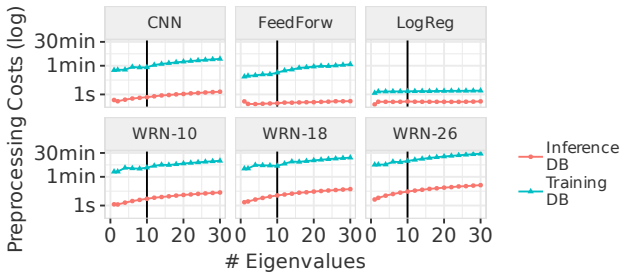

Figure 9: Offline precomputation costs.

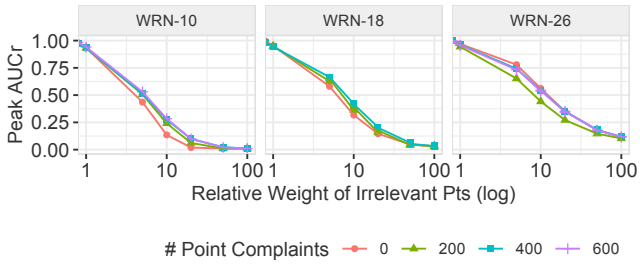

Figure 10: Effects of varying the number of relevant point complaints, and the overall weight of irrelevant point complaints.

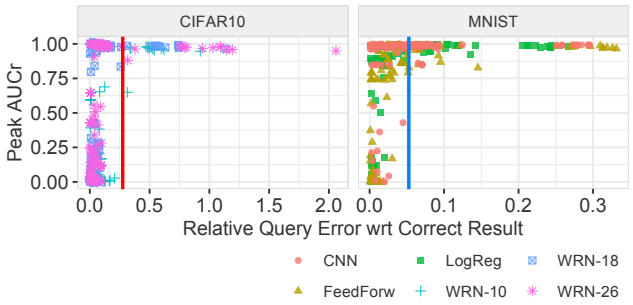

Figure 11: Relationship between the query output's relative error and debugging quality.

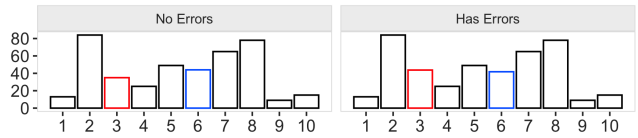

Figure 12: Small relative errors are difficult to visually detect. The height of bars 3 and 6 are changed by 25% and 5%, respectively.

if the training corruptions only cause an $\varepsilon \approx 0$ to the query's result value, then we should not expect it to be effective.

6.8.1 Initial Point Complaint Analysis. Our analysis will be based on Equation (7), which decomposes $q(\vartheta)$ into a linear combination of individual prediction probabilities $p_{ij}(\vartheta)$ for each inference record i and class j. We can view p_{ij} as a *point complaint* that record i should have label j. Intuitively a point complaint is likely to be effective for debugging if the model mispredicts i *and* if removing the training errors would lead to a correct prediction. To check this intuition, we add class conditional label noise in **CIFAR-10** (40% rate), and point complaints where the corrupted model prediction differs from the clean model's prediction. These complaints indeed have a high peak AUC_R of 86%, agreeing with our intuition.

6.8.2 Revelant and Adversarial Point Complaints. We now use point complaints to study query complaints. Query complaints are modeled as linear combinations of point complaints that differ in terms of their weights $\partial f(P(\vartheta^*))/\partial p_{ij}$. We expect that a query complaint that assigns high weights to the "relevant" point complaints and low weights to the "adversarial" point complaints should be effective at debugging training errors.

Our experiment varies the weights that we assign to relevant and adversarial point complaints. This is akin to a SUM aggregation with a predicate where tuples that satisfy the predicate increase the sum by an amount based on their attributes. Join aggregations have this property as well. We define relevant complaints test points as those whose true label is *automobile*, are mislabeled by the model but are correctly predicted when the training errors are removed. We define adversarial point complaints as test examples whose true predicted labels are *horse*. The two types of point complaints push the model in different directions. We use an equal number of relevant and adversarial points, but give adversarial points Y× the weight as relevant points, where $Y \in [0, 100]$.

Figure 10 shows that debugging quality is insensitive to the number of point complaints, but is highly sensitive to the *ratio* of weights. When the ratio is $\leq 5\times$, Rain++ remains effective, however the quality quickly degrades. This suggests that query complaints are most effective when adversarial complaints do not dominate. Note that an irrelevant point complaint p_{ij}—for instance, a test point predicted with high confidence as *bird* for a query that filters on *horse*—has negligible effects on debugging quality since their contribution to the query gradient is 0.

6.8.3 Magnitude of Query Errors. Although the above analysis sheds light on when Rain++ can be effective, it relies on apriori knowledge of relevant and irrelevant point complaints. However, assuming this puts the cart before the horse, as the user only has visibility of the query results. Thus, this experiment studies the relationship between the magnitude of the query's output error wrt the correct query output, and debugging effectiveness. The intuition is that larger query errors may be more likely to be due to training example corruptions, rather than for spurious reasons.

We use all models, the **CIFAR-10** and **MNIST** datasets, and vary the rate of class conditional label noise from 10% to 40%. We sweep the possible queries that can be generated using Q_1 and Q_2 templates. For Q_1, we set the left and right sides of the join to subsets of the test database with different true labels (e.g., LEFT is digit '5', RIGHT is digit '9' for **MNIST**). We use this procedure to generate 20 random query complaints. For Q_2, we vary the filter condition over all 10 classes for each dataset, resulting in 10 complaints per dataset. This results in 300 complaints per dataset.

Figure 11 shows that, irrespective of the model architecture, the peak AUC_R improves as the relative query error increases. When the query result increases beyond a threshold (red (25%) and blue (5%) vertical lines), the peak AUC_R tends to be near-1. This is an encouraging result, because small relative differences are difficult to see [4, 16, 34], and users are most likely to submit complaints when errors are noticable. Figure 12 illustrates that it is difficult to tell, even side by side, that the heights of bars 3 and 6 have been respectively changed by 25% and 5%.

Among the corruptions that did not affect the query results, we found cases where heavy corruptions did not have a significant effect on model accuracy. For example, Salt & Pepper noise on even 50% of 1 digits of **MNIST** reduced test set accuracy by less than 1%. This strongly indicates that data debugging approaches that are unaware of how the model is used downstream may spend significant amounts of time cleaning training examples that do not

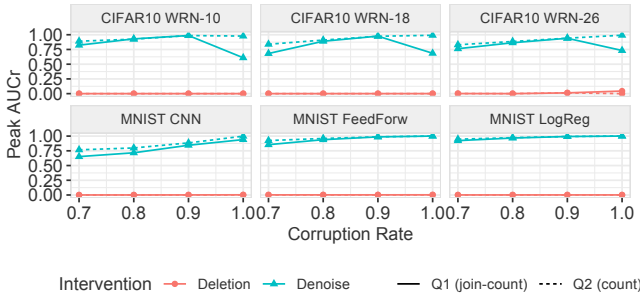

Figure 13: Deletion is an ineffective intervention for addressing Salt & Pepper noise; denoising via median filter is effective.

end up affecting model accuracy. The complaint driven approach of Rain and Rain++ clearly avoids that.

Takeaway: Debugging quality is related to the weights of relevant point complaints as compared to adversarial point complaints, further larger query output differences correlate with higher peak AUC_R. Complaint driven debugging can reduce debugging effort by prioritizing training set bugs that actually affect model accuracy.

6.9 Intervention Effectiveness

So far we have assumed that deleting the corrupted training examples is sufficient to resolve the user's complaint. However, this is not always the case, because feature errors such as Salt & Pepper noise on images can also affect query results. For example, when we corrupt 90% of the automobile training examples with Salt & Pepper on **CIFAR-10**, WRN-26 predicts 847 automobiles in the test database when the true count is 1000. However, deleting these corrupted training examples will simply worsen the query output. In addition, using a poor intervention also affects the effectiveness of influence analysis. If deleting the corrupted training examples does not change the model in a way that resolves the complaint, then Rain and Rain++ will not rank those examples highly.

To illustrate this, our experiment corrupts the **CIFAR-10** and **MNIST** training sets with Salt & Pepper, and evaluates Rain++ using the deletion intervetion that we have used so far, and a denoise intervention. The latter assigns each pixel the median value of its neighboring pixels; this is effective for Salt & Pepper noise. We run Q_1 and Q_2 using their default configurations. Figure 13 shows that across all models, datasets, and queries, the deletion intervention is completely ineffective. Although the query complaint specifies that the query result should be higher, deleting the corrupted training examples actually *reduces* the query results further. In contrast, denoise has a peak AUC_R consistently above 0.65 and converges to 1 as the corruption rate (of the corrupted class) increases to 100%.

Takeaway: Effective training example debugging relies on using the appropriate intervention, and deletion is not always the most effective. Further studies are needed to better understand the interaction between data corruption, interventions, and complaints.

7 RELATED WORK

Approximate Retraining: Approximate retraining has recently attracted a lot of interest [15, 19, 41, 42]. While the alternative approaches to influence analysis can sometimes provide more accurate estimates, they are either significantly more expensive to run or they are limited to convex models or even both. To the best of our knowledge our work is the first one to study the problem of accelerating approximate retraining based on offline computation and to enable its interactive use for large neural networks.

Model compression and simplification: Model compression (ala [10]) selects a subnetwork that may be up to 90% smaller than the overparameterized originial. A similar area is model quantization, which uses the trace of the hessian (average of eigenvalues) to determine layer sensitivity and thus bound errors introduced due to numerical quantization [6]. The speedups obtained by Rain++ go beyond running Rain on a compressed or quantized model because while $k \approx 20$ eigenvectors are enough for debugging, 20 parameters are not sufficient to classify **MNIST** or **CIFAR-10**.

Gradient compression for distributed training: In distributed training, the communication of gradients between workers can easily dominate the training time. To reduce the communication cost, prior work [44] proposed to compress gradient vectors using PCA. Rain++ is calculating the eigenvectors of the Hessian and not the principal components of the gradients. Observe however that on Section 3, these two sets of vectors are closely related since the training loss gradients span the top eigenvectors of the Hessian.

8 CONCLUSION

End-users and practitioners increasingly use models through inference queries, and interact with inference queries through interactive visualization interfaces. Complaint-driven debugging allows users to identify and understand how the model training data affected the visualized data. To support user-facing interactivity, Rain++ develops a novel set of precomputation techniques that reduces the online debugging latency by multiple orders of magnitude as compared to the prior Rain system, while also scaling to models with millions of parameters. Our analysis of when complaints are effective finds evidence that Rain++ is more accurate when the query output's error is larger, which matches the settings when users will identify errors in a visual interface.

One point we wish to emphasize is that training data debugging can benefit considerably by taking the downstream uses of the model into account—particularly query errors are not the primary source of result errors. Complaint-driven debugging helps prioritize erroneous training errors that directly affect downstream results that matter to the user (or application), and helps avoid wasted cleaning efforts on training data that does not affect the application. While these ideas are well established in the area of domain adaptation, which studies adapting model training to the test distribution of interest, integrating downstream knowledge into training data cleaning is still nascent.

REFERENCES

[1] Firas Abuzaid, Peter Kraft, Sahaana Suri, Edward Gan, Eric Xu, Atul Shenoy, Asvin Ananthanarayan, John Sheu, Erik Meijer, Xi Wu, Jeff Naughton, Peter Bailis, and Matei Zaharia. 2018. DIFF: A Relational Interface for Large-Scale Data Explanation. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 419–432. https://doi.org/10.14778/3297753.3297761

[2] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for aggregate queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011,*

Athens, Greece, Maurizio Lenzerini and Thomas Schwentick (Eds.). ACM, 153–164. https://doi.org/10.1145/1989284.1989302

[3] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. https://mlsys.org/Conferences/2019/doc/2019/167.pdf

[4] W. Cleveland and R. McGill. 1984. Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods. *J. Amer. Statist. Assoc.* 79 (1984), 531–554.

[5] George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann (Eds.). 2002. *Differentiation Methods for Industrial Strength Problems*. Springer New York, New York, NY. https://doi.org/10.1007/978-1-4613-0075-5_1

[6] Zhen Dong, Zhewei Yao, Yaohui Cai, Daiyaan Arfeen, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2019. HAWQ-V2: Hessian Aware trace-Weighted Quantization of Neural Networks. *CoRR* abs/1911.03852 (2019). arXiv:1911.03852 http://arxiv.org/abs/1911.03852

[7] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[8] Carl Eckart and Gale Young. 1936. The approximation of one matrix by another of lower rank. *Psychometrika* 1, 3 (1936), 211–218.

[9] Open Neural Network Exchange. 2019. ONNX. https://onnx.ai/. [Online; accessed 1-December-2020].

[10] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=rJl-b3RcF7

[11] Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. 2019. An Investigation into Neural Net Optimization via Hessian Eigenvalue Density. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 2232–2241. http://proceedings.mlr.press/v97/ghorbani19b.html

[12] W. D. Gray and D. Boehm-Davis. 2000. Milliseconds matter: an introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of experimental psychology. Applied* 6 4 (2000), 322–35.

[13] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Beijing, China) *(PODS '07)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1265530.1265535

[14] Guy Gur-Ari, Daniel A. Roberts, and Ethan Dyer. 2018. Gradient Descent Happens in a Tiny Subspace. *CoRR* abs/1812.04754 (2018). arXiv:1812.04754 http://arxiv.org/abs/1812.04754

[15] Satoshi Hara, Atsushi Nitanda, and Takanori Maehara. 2019. Data Cleansing for Models Trained with SGD. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 4215–4224. http://papers.nips.cc/paper/8674-data-cleansing-for-models-trained-with-sgd

[16] J. Heer and M. Bostock. 2010. Crowdsourcing graphical perception: using mechanical turk to assess visualization design. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2010).

[17] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

[18] Magnus R Hestenes and Eduard Stiefel. 1952. Methods of Conjugate Gradients for Solving Linear Systems1. *J. Res. Nat. Bur. Standards* 49, 6 (1952).

[19] Zachary Izzo, Mary Anne Smart, Kamalika Chaudhuri, and James Y. Zou. 2020. Approximate Data Deletion from Machine Learning Models: Algorithms and Evaluations. *CoRR* abs/2002.10077 (2020). arXiv:2002.10077 https://arxiv.org/abs/2002.10077

[20] JAX. 2020. JAX reference documentation — JAX documentation. https://jax.readthedocs.io/en/latest/. [Online; accessed 1-December-2020].

[21] Pang Wei Koh and Percy Liang. 2017. Understanding Black-box Predictions via Influence Functions. In *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 1885–1894. http://proceedings.mlr.press/v70/koh17a.html

[22] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. (2009).

[23] Cornelius Lanczos. 1950. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA.

[24] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. http://yann.lecun.com/exdb/mnist/

[25] Z. Liu and J. Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics* 20 (2014), 2122–2131.

[26] Z. Liu and J. Stasko. 2010. Mental Models, Visual Reasoning and Interaction in Information Visualization: A Top-down Perspective. *IEEE Transactions on Visualization and Computer Graphics* 16 (2010), 999–1008.

[27] Google LLC. 2019. Introduction to BigQuery ML. https://cloud.google.com/bigquery-ml/docs/bigqueryml-intro. [Online; accessed 10-October-2019].

[28] Alexandra Meliou and Dan Suciu. 2012. Tiresias: The Database Oracle for How-to Queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2213836.2213875

[29] OpenML. 2020. OpenML Supervised Classification on adult. https://www.openml.org/t/7592. [Online; accessed 1-December-2020].

[30] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1144. https://doi.org/10.1145/2939672.2939778

[31] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Bießmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *Proc. VLDB Endow.* 11, 12 (2018), 1781–1794. https://doi.org/10.14778/3229863.3229867

[32] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, 1631–1642. https://www.aclweb.org/anthology/D13-1170/

[33] SQLFlow. 2019. SQLFlow: Bridging Data and AI. https://sqlflow.org. [Online; accessed 1-December-2020].

[34] Justin Talbot, V. Setlur, and A. Anand. 2014. Four Experiments on the Perception of Bar Charts. *IEEE Transactions on Visualization and Computer Graphics* 20 (2014), 2152–2160.

[35] Tensorflow. 2020. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla. [Online; accessed 1-December-2020].

[36] Jason Teoh, Muhammad Ali Gulzar, and Miryung Kim. 2020. Influence-based provenance for dataflow applications with taint propagation. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 372–386. https://doi.org/10.1145/3419111.3421292

[37] Aad W Van der Vaart. 2000. *Asymptotic statistics*. Vol. 3. Cambridge university press.

[38] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data X-Ray: A Diagnostic Tool for Data Errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1231–1245. https://doi.org/10.1145/2723372.2750549

[39] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *Proc. VLDB Endow.* 6, 8 (June 2013), 553–564. https://doi.org/10.14778/2536354.2536356

[40] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. 2020. Complaint-driven Training Data Debugging for Query 2.0. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1317–1334. https://doi.org/10.1145/3318464.3389696

[41] Yinjun Wu, Edgar Dobriban, and Susan Davidson. 2020. DeltaGrad: Rapid retraining of machine learning models. In *International Conference on Machine Learning*. PMLR, 10355–10366.

[42] Yinjun Wu, Val Tannen, and Susan B. Davidson. 2020. PrIU: A Provenance-Based Approach for Incrementally Updating Regression Models. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 447–462. https://doi.org/10.1145/3318464.3380571

[43] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* abs/1708.07747 (2017). arXiv:1708.07747 http://arxiv.org/abs/1708.07747

[44] Mingchao Yu, Zhifeng Lin, Krishna Narra, Songze Li, Youjie Li, Nam Sung Kim, Alexander G. Schwing, Murali Annavaram, and Salman Avestimehr. 2018. GradiVeQ: Vector Quantization for Bandwidth-Efficient Gradient Aggregation in Distributed CNN Training. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo

Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 5129–5139. http://papers.nips.cc/paper/7759-gradiveq-vector-quantization-for-bandwidth-efficient-gradient-aggregation-in-distributed-cnn-training

[45] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide Residual Networks. *CoRR* abs/1605.07146 (2016). arXiv:1605.07146 http://arxiv.org/abs/1605.07146

[46] Xuezhou Zhang, Xiaojin Zhu, and Stephen J. Wright. 2018. Training Set Debugging Using Trusted Items. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*. AAAI Press, 4482–4489. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16155

[47] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43–76.