Inspecting Neural Networks with Queries

Dipta Chandra Paul, Hashirul Quadir, Nicholas Anderson, and Carlos Ordonez

Department of Computer Science
University of Houston, USA

Abstract. Understanding the dynamics of neural network (NN) initialization and weight optimization is a subject of significant research interest, yet diagnostic practices are frequently limited to monitoring loss metrics. While specialized tools allow inspection of model weights, these methods typically rely on custom Python scripts and often face limitations in interpretability and substantial computational overhead. This paper introduces a prototype that instead monitors a training neural network through SQL queries over a relational database. The prototype stores only binary neuron activity, biases, and predictions in relational tables, while excluding weight matrices, to keep the cost of high-frequency parameter transfer low. Analysts can then issue standard SQL queries to study the decision boundary, the distribution of biases across hidden layers, and convergence behavior during training. An experimental evaluation on public datasets illustrates the usefulness of three representative queries and measures the database I/O overhead required to transfer this internal state during training.

1 Introduction

Deep Neural Networks (DNNs) are often criticized as opaque "black boxes," making it difficult to understand their internal logic and behavior based on input data. Explainable AI requires tools to peer into these hidden layers, but traditional auditing methods—transferring raw tensors and full weights to disk—create unsustainable I/O bottlenecks. Storing the entire network state takes more time than the PyTorch computations themselves, and analyzing it requires custom, low-level Python scripts that must be rewritten for every new diagnostic question. To bridge the gap between deep learning complexity and actionable interpretability, we introduce a lightweight relational database monitoring system with very low overhead.

While specialized ML observability tools and raw tensor storage systems exist, we use a standard relational database because it offers a proven, declarative mechanism to query network state through SQL rather than fixed Python instrumentation. This decouples the analytical layer from the training loop, lets analysts ask new questions on the fly without modifying the training code, and makes monitoring queries portable across any system that speaks SQL. The cost of this choice is that the network state must be reshaped into relational tables rather than tensors; our schema in Section 3.2 is designed precisely to make that reshaping cheap.

Rather than relying on heavy, complex monitoring, our system extracts only two critical primitives into relational tables, which power our core contributions:

- **Neuron Activation:** We store a simple binary representation (1 or 0) signifying whether a neuron fired. This metric is fast to transfer, can be quantified vector by vector, and is summarized at the iteration level for the entire dataset.
- **Bias Histograms:** Rather than tracking full weights, we extract floating-point biases to create histograms. Because a positive bias indicates a neuron activates more often and a negative bias indicates it activates less, this allows us to quickly identify regions of high network activity and see exactly where the network is most active.
- **SQL Queries for Neural Network Inspection:** We program several specific, efficient SQL queries designed to inspect and tune the network. These allow analysts to observe oscillating versus stable neurons and trace false positive and false negative classifications back to specific features in the input layers.

The remainder of this paper is organized as follows. Section 2 establishes the foundational definitions for the multilayer perceptron and relational database concepts. Section 3 details our core contribution, describing parameter selection, the database schema, the lightweight transfer pipeline, monitoring queries, and complexity analysis. Section 4 presents the experimental evaluation, including analytical insights and system overhead. Section 5 reviews related work, and Section 6 concludes the paper with directions for future research.

2 Definitions

This section establishes the two pieces of background that the rest of the paper builds on: the neural network we monitor, and the storage mechanism we use to store its internal state.

2.1 The Multilayer Perceptron for Binary Classification

The Multilayer Perceptron (MLP) is the foundational building block of modern neural networks. In practice, virtually every deep learning architecture uses an MLP as a subroutine, from the fully-connected classification heads of convolutional networks to the position-wise feed-forward sublayers of transformers. We therefore present our monitoring approach in terms of a standard classification MLP designed for binary classification, with the understanding that the same primitives extend to richer architectures.

The input matrix $X \in \mathbb{R}^{d \times n}$ has d rows and n columns. Each column represents a training sample; each row stores a feature (e.g., height, age, salary). The input layer has one neuron for each feature and only takes numerical data; there are no weights or activations for input neurons, since input data is preprocessed into numerical form before being sent through the input layer.

For a hidden layer l, the pre-activation vector $\hat{z}^{(l)}$ is calculated by computing a matrix multiplication between the weight matrix $\hat{w}^{(l)}$ and the activation vector of the previous layer $\hat{x}^{(l-1)}$, after which the bias vector $\hat{b}^{(l)}$ is added. The pre-activation $\hat{z}^{(l)}$ is then passed through a non-linear activation function σ to compute the post-activation vector $\hat{x}^{(l)}$. In this work we use the Rectified Linear Unit (ReLU) for all hidden layers and the sigmoid function for the output layer. The output layer follows the same mathematical structure as the hidden layers, but the sigmoid activation allows the MLP to have a single output neuron, converting the output weight matrix to a vector $\hat{w}^{(l)}$ and reducing the pre-activation and bias vectors to scalars. A *prediction cutoff* is supplied as a hyperparameter; all post-activation values less than this cutoff are labeled as 0, and the remaining values are labeled as 1, corresponding to a binary prediction.

2.2 Storing Neural Network State in a Relational Database

The internal state of a neural network can in principle be stored in many ways: as multi-dimensional arrays in Python, as matrices in MATLAB, as on-disk tensors written by a deep learning library, or as records in a database. We choose a relational database. The motivation is not the storage representation itself but the access mechanism it enables: a relational database lets us express each monitoring question as a declarative SQL query rather than as imperative Python or C code, so that a new diagnostic question becomes a new query, not a new instrumentation script. These queries run over the network's state, which we organize as a small set of relational tables kept in normalized form (3NF, BCNF); we refer the reader to a standard database textbook [3] for the formal definition.

3 Storing and Querying Neural Networks

A neural network learns by repeatedly transforming its internal parameters in response to training data. At the start of training, the network is given an initial assignment of weights and biases. It then enters an iterative loop that runs over multiple epochs. During an epoch, the training data is divided into mini-batches. For each mini-batch iteration, the network runs a forward pass to compute predictions, compares the predictions to the ground truth to quantify the error, and runs a backward pass that computes the gradients of the error with respect to the parameters, updating the weights and biases to minimize this error. The loop continues until the error converges or a fixed number of epochs is reached. The behavior of this loop—which neurons activate on which inputs, how biases drift, and where misclassifications come from—is what we want to observe.

3.1 Neural Network State and Parameter Selection

Observing all of this at full fidelity is not feasible. A single training iteration produces a large amount of internal state: for every layer, a weight matrix and a

bias vector; for every data point, a pre-activation and a post-activation at every neuron; for every weight and bias, a gradient produced by the backward pass; and, for the iteration as a whole, a scalar loss. Persisting all of this to a database at the rate the training loop generates it would easily cost more time and storage than the training itself, defeating the purpose of a lightweight monitor. We therefore store only a subset of this state, chosen to be small enough to transfer at high frequency and rich enough to answer useful monitoring questions.

We exclude the weight matrices and their gradients. The weights are by far the largest component of the network state, they do not change drastically between consecutive iterations — so logging them at every iteration produces mostly redundant data — and they are hard to interpret in isolation, since reading a single weight requires understanding the local shape and slope of the manifold the network has learned, as well as its non-linear interaction with every other weight in the layer. The gradients are the same size as the weights and are even more transient, and they describe how the optimizer is updating the network rather than what the network currently represents, so we do not transfer them. We also exclude the continuous pre- and post-activations: for the questions our queries ask, what matters is whether a neuron fired, not the magnitude of its response, so a single bit per neuron is sufficient.

We store three things: a binary activation per neuron per data point, the floating-point biases, and the predictions. Biases are small — one number per neuron — and are directly interpretable: a positive bias indicates a neuron activates more often and a negative bias indicates it activates less, so the distribution of biases across a layer tells us where the network is most active and where it is struggling to place a decision boundary. The binary activations record, at one bit each, whether a neuron fired. The predictions are needed to trace misclassifications back to the neurons and input features responsible for them. The per-iteration loss and accuracy are single scalars and are recorded as well, at negligible cost. We treat this combination as an initial approximation of network state, and our experiments in Section 4 show that it is rich enough to support useful monitoring queries.

We store only a binary activation (1 or 0) in place of the full continuous post-activation for three reasons:

- **Storage and transfer cost.** A single-precision floating-point number requires four bytes, whereas a binary activation requires only one bit. Storing one bit instead of a 32-bit float for each (neuron, data point) entry reduces the activation footprint by a factor of 32, which is what keeps high-frequency transfers feasible without stalling the training loop.
- **Query sufficiency.** The monitoring queries we present in Section 3.4 depend only on whether a neuron fired, not on the magnitude of its response. This aligns with weight-pruning research [7], which shows that highly compact network representations can still preserve task-relevant signal.
- **Compatibility with the pipeline.** The binary representation packs cleanly into our buffered, bulk-insert pipeline: binary states can be packed into compact bitfields and inserted in a single SQL transaction, whereas continuous

activations would require more frequent flushes that would inflate the I/O overhead reported in Table 2.

3.2 Database Schema

We now make concrete the relational tables introduced in Section 2.2. Normalization eliminates redundancy and allows the database to absorb high-frequency updates from the training loop without inflating storage; the trade-off is that questions touching several relations must be answered with joins at query time, which our schema is designed to keep cheap. The schema has five tables organized hierarchically by experiment, iteration, neuron, and data point. Primary keys are underlined; foreign keys are inherited from the shared attributes (e.g., `experiment_id` in every table references **Experiment**; `(experiment_id, iteration_id)` in **Bias** and **Activation** references **Iteration**; `(experiment_id, point_id)` in **Activation** references **Prediction**).

- **Experiment**(experiment_id, experiment_name, dataset_name, hyperparameters)
- **Iteration**(experiment_id, iteration_id, loss, val_loss, accuracy)
- **Prediction**(experiment_id, point_id, actual_class, predicted_class)
- **Bias**(experiment_id, iteration_id, layer_id, neuron_id, bias)
- **Activation**(experiment_id, iteration_id, point_id, layer_id, neuron_id, is_triggered)

The **Activation** table is the largest by several orders of magnitude, since it contains one row per (iteration, neuron, data point) triple. We analyze its size and its impact on query cost in Section 3.5.

3.3 Lightweight Transfer Pipeline

The dominant cost of monitoring a neural network at fine granularity is not query evaluation but parameter transfer: moving snapshots of internal state from the training process into the database. To keep this cost low, activations and biases are transferred in a buffered, periodic fashion rather than written one value at a time. The user defines a buffer size and a transfer frequency f. Binary activations are buffered and transferred to the database in bulk only when the buffer is full, which sharply reduces transfer overhead. Biases are transferred every f iterations: they do not change drastically between consecutive iterations, so storing every iteration would create unnecessary overhead and redundant entries. Figure 1 illustrates this extraction and logging architecture.

3.4 Important Queries to Inspect the Neural Net

We program three SQL queries, one for each of the primitives the system monitors. The queries are written against the schema in Section 3.2 and use only standard SQL supported by PostgreSQL and every major commercial relational engine.

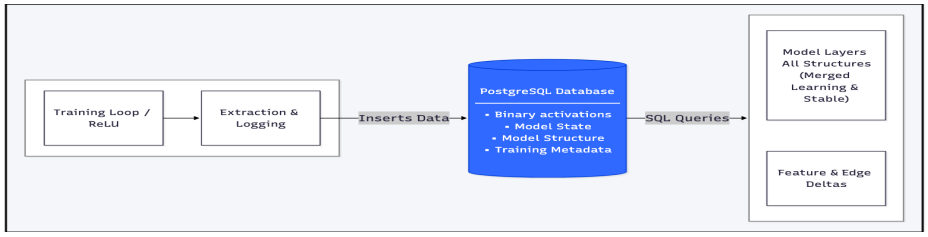

Fig. 1. Lightweight transfer of binary (1/0) neuron activations and floating-point biases from PyTorch into the relational database.

– **Neuron Activation.** We use the activation table to identify oscillating neurons whose firing state constantly flips across iterations — units that are struggling to find and establish a stable decision boundary in some regions of the input space. The query uses the OLAP window function `LAG` to compare each row with the same neuron's activation in the previous iteration, then counts the number of times the activation flipped.

```
WITH Flips AS (
  SELECT
    layer_id, neuron_id, point_id, is_triggered,
    LAG(is_triggered) OVER (
      PARTITION BY layer_id, neuron_id, point_id
      ORDER BY iteration_id
    ) AS prev_triggered
  FROM activation
  WHERE experiment_id = %d
)
SELECT
  layer_id,
  neuron_id,
  COUNT(*) AS total_flips
FROM Flips
WHERE prev_triggered IS NOT NULL
  AND is_triggered != prev_triggered
GROUP BY layer_id, neuron_id
ORDER BY total_flips DESC;
```

– **Misclassification.** By joining the activation table with the prediction table at the final iteration and restricting to misclassified points, we find which neurons in the last hidden layer fire systematically during false positives and false negatives. This isolates the neurons most responsible for the network's misclassifications.

```
SELECT
  a.layer_id,
  a.neuron_id,
  CASE
```

```
    WHEN p.actual_class = 0 AND p.predicted_class = 1
      THEN 'False Positive'
    WHEN p.actual_class = 1 AND p.predicted_class = 0
      THEN 'False Negative'
  END AS mistake_type,
  SUM(a.is_triggered)          AS fired_count,
  ROUND(AVG(a.is_triggered) * 100, 2) AS fire_rate_pct
FROM activation a
JOIN prediction p
  ON a.point_id = p.point_id
 AND a.experiment_id = p.experiment_id
WHERE a.experiment_id = %d
  AND p.actual_class != p.predicted_class
  AND a.iteration_id = (
    SELECT MAX(iteration_id) FROM activation
    WHERE experiment_id = %d
  )
  AND a.layer_id = (
    SELECT MAX(layer_id) - 1 FROM activation
    WHERE experiment_id = %d
  )
GROUP BY a.layer_id, a.neuron_id, p.actual_class, p.
    predicted_class
HAVING AVG(a.is_triggered) > 0.5
ORDER BY mistake_type ASC, fired_count DESC;
```

– **Bias.** We bucket biases at a chosen iteration into an equi-depth histogram based on quantiles (using `NTILE`), which is robust to diverse data distributions. For each bucket, we report what fraction of neurons in that bias range are active, inactive, or oscillating. This connects bias distribution to neuron behavior and exposes where the network places its decision boundary.

```
WITH NeuronStats AS (
  SELECT
    b.layer_id,
    b.neuron_id,
    b.bias,
    NTILE(4) OVER (ORDER BY b.bias ASC) AS bucket_number,
    CASE
      WHEN SUM(a.is_triggered) <  COUNT(a.point_id) * 0.10
           THEN 'Inactive'
      WHEN SUM(a.is_triggered) >= COUNT(a.point_id) * 0.90
           THEN 'Active'
      ELSE 'Oscillating'
    END AS behavior
  FROM bias b
  JOIN activation a
    ON  a.layer_id   = b.layer_id
    AND a.neuron_id  = b.neuron_id
    AND a.experiment_id = b.experiment_id
```

```
    AND a.iteration_id  = b.iteration_id
  WHERE b.experiment_id = %d AND b.iteration_id = %d
  GROUP BY b.layer_id, b.neuron_id, b.bias
)
SELECT
  bucket_number,
  MIN(bias) AS lowest_bias,
  MAX(bias) AS highest_bias,
  COUNT(*)  AS total_neurons,
  ROUND(100.0 * COUNT(*) FILTER (WHERE behavior = 'Active'
      )      / COUNT(*), 2) AS pct_active,
  ROUND(100.0 * COUNT(*) FILTER (WHERE behavior = '
      Inactive')    / COUNT(*), 2) AS pct_inactive,
  ROUND(100.0 * COUNT(*) FILTER (WHERE behavior = '
      Oscillating') / COUNT(*), 2) AS pct_oscillating
FROM NeuronStats
GROUP BY bucket_number
ORDER BY bucket_number;
```

3.5 Time and Space Complexity Analysis

We analyze the space and time complexity of the monitoring system. We describe the network by its depth L (the number of hidden layers) and its average layer width W, so the network has $L \cdot W$ hidden neurons in total. The number of recorded iterations is governed by convergence rather than by the algorithm, so we report the cost of a single recorded snapshot. Because the input layer computes no activations or biases, the feature dimensionality d does not factor into the overhead. We evaluate the system under two scaling paradigms: varying the network architecture over a fixed data set, and varying the data set size over a fixed architecture.

Scaling with Network Architecture (Fixed Dataset). Here the data set size n is held constant. A bias snapshot stores one float per neuron, so it scales as $O(L \cdot W)$ floats. An activation snapshot stores one bit per (neuron, data point) pair, so it occupies $O(L \cdot W \cdot n)$ bits; with n fixed this grows as $O(L \cdot W)$ in the architecture, and excluding the weight matrices removes what would otherwise be the dominant storage term. At query time, the oscillation and bias-histogram queries both scan the activation state of the network and run in $O(L \cdot W \cdot n)$, i.e. $O(L \cdot W)$ for fixed n. The misclassification query is the exception: it inspects only the final hidden layer, so with an index on the layer attribute its cost is bounded by that layer's width, $O(W)$, and is independent of the depth L. This decoupling lets analysts audit misclassifications in very deep networks without paying a depth penalty.

Scaling with Dataset Size (Fixed Architecture). Here the architecture (L and W) is held constant. As n grows, the Prediction table grows as $O(n)$ rows and an activation snapshot as $O(L \cdot W \cdot n) = O(n)$ bits; storing a single bit rather than a 32-bit float for each activation is what keeps this linear growth from

becoming an I/O bottleneck. The bias *storage* is the one quantity independent of the data: a bias snapshot is $O(L \cdot W)$ floats, i.e. $O(1)$ with respect to n, because biases are architectural parameters. Query time, however, scales with the data for all three queries, since each must scan the activation table: the oscillation, misclassification, and bias-histogram queries all run in $O(n)$. In particular, the bias-histogram query is $O(n)$, not $O(1)$, because it joins the bias table with the activation table to classify each neuron as active, inactive, or oscillating, which requires reading that neuron's activations across all n data points.

4 Experimental Evaluation

In this section, we evaluate our prototype from a database performance and data mining perspective. Our experiments aim to answer two key questions: (1) can the SQL queries expose useful information about the network's internal state — its convergence behavior, its misclassifications, and where its neurons are active — and (2) can this be done efficiently, adding only modest overhead to the training loop?

4.1 Experimental Setup

To ensure reproducibility, all experiments were performed on a single server equipped with an Intel I7-4770 with 32 GB of main memory utilizing locally attached solid-state storage to minimize network latency. We used Python 3.13 as the primary adapter language, PyTorch 2.9.1 for constructing and training the neural network, and PostgreSQL 17.9 as the relational database engine. For data manipulation and model evaluation we used Pandas 2.3.3, NumPy 2.3.5, and Scikit-learn 1.8.0.

We evaluated the system using three diverse, publicly available datasets. To demonstrate that the system generalizes across domains, we selected two human-centric datasets: Adult Income (sourced from the UCI Machine Learning Repository) and Pima Indians Diabetes (sourced from Kaggle), alongside one network traffic dataset, KDD (also from the UCI repository).

Table 1. Characteristics of the evaluation datasets.

Dataset	Rows (n)	Dimensions (d)
KDD (network traffic)	494,021	42
Adult Income (financial)	32,561	14
Pima Indians Diabetes (medical)	768	8

4.2 Analytical Insights and Decision Support

We now show how the three queries from Section 3.4 surface useful information about a training network. Each query targets one of the three primitives the

system monitors—neuron activation, bias distribution, and misclassification—and we report what each query revealed across the three datasets in Table 1.

Neuron Activation. The neuron activation query identifies oscillating neurons: hidden units whose firing state flips repeatedly across consecutive iterations, signalling that the network has not converged in those regions of the input space. On the Adult Income dataset, the query exposed a major training issue. Out of the 224 hidden neurons, more than 91% oscillated, with some groups of neurons oscillating at 100%. On Pima Indians Diabetes, oscillation was also widespread, suggesting the smaller dataset destabilized the network's learning process. On KDD, with substantially more training data, oscillation was lower across the network, consistent with the intuition that more data stabilizes activation patterns. For the analyst, a high oscillation rate is an immediate signal to intervene—typically by lowering the learning rate—without having to write any custom Python instrumentation.

Bias. The bias query distributes the per-neuron biases into an equi-depth histogram and reports, for each bias bucket, the fraction of neurons that are active, inactive, or oscillating. On Adult Income, the histogram exposed an asymmetric pattern: fully inactive neurons appeared only in the negative middle range (bucket 2 contained the entire 8.93% inactive share), while fully active neurons appeared only in the highest positive bucket (3.57% of bucket 4). This concentration tells the analyst that the network is using only a narrow slice of its bias space and that broad regions of the input feature space have no decision boundary placed in them. On Pima Indians Diabetes, the bias distribution was more uniform across buckets, while on KDD the bias buckets were more populated overall, reflecting the larger feature dimensionality. In all three cases, isolating which feature ranges correspond to under-served bias buckets points directly at where additional feature engineering or training data would help, rather than having the analyst guess.

Misclassification. The misclassification query joins the activation table with the prediction table at the final iteration and isolates the neurons in the last hidden layer that fire systematically during false positives and during false negatives. On Adult Income, the query identified a small group of last-layer neurons whose activation rate during misclassified samples was substantially higher than during correct samples, suggesting those neurons were memorizing a spurious shortcut in the training set. On KDD, false positives and false negatives were driven by different neuron sets, indicating the network learned distinct (and partly conflicting) patterns for the two error types. On Pima Indians Diabetes, the limited dataset size made the misclassification pattern noisier, but the query still highlighted a small handful of neurons whose firing pattern was most correlated with errors. This level of granular auditing points the analyst at specific neurons to investigate; furthermore, a high concentration of misclassification neurons may strongly indicate that the network lacks capacity, and that wider or additional layers are needed to resolve the overlapping boundaries.

4.3 Time Efficiency and Database Overhead

Recall from Section 3.5 that the Activation table dominates storage, with the footprint of each recorded snapshot scaling jointly with network depth, width, and data set size as $O(L \cdot W \cdot n)$ bits, and that the per-iteration transfer cost is dominated by buffered binary writes. We now report the measured overhead against these bounds.

The primary criticism of in-database machine learning is the I/O bottleneck. If extracting data takes longer than the matrix multiplications, the training loop will stall.

Table 2. Database I/O overhead for pipeline extraction (all times in seconds).

Dataset	PyTorch Time	PostgreSQL Transfer Time	Overhead
KDD	1329.0	208.0	16%
Adult Income	100.0	22.0	22%
Pima Diabetes	2.7	0.6	22%

As shown in Table 2, our prototype introduces approximately 16–22% time overhead. This low overhead is achieved because we store only bias and binary activation information instead of extracting full weight matrices, which keeps the extraction cost well below the cost of the PyTorch computations themselves.

4.4 Limitations

We acknowledge several limitations of the current prototype, which we view as motivating directions rather than fundamental obstacles. First, the system is currently designed for multilayer perceptrons performing binary classification; supporting multi-class outputs requires extending the Prediction schema, and convolutional, attention, or recurrent layers require additional attributes to capture spatial, sequential, or attentional structure. Second, the binary representation of neuron activity discards magnitude information, so questions that require continuous-valued activations—such as saturation analysis or gradient-flow inspection—fall outside the current schema. Third, the storage cost analyzed in Section 3.5 still scales jointly with the network depth L, layer width W, and the number of data points n, and our experiments cover three tabular datasets on a single-node PostgreSQL deployment; behavior on image, text, or graph benchmarks and on distributed backends remains to be characterized. Fourth, choosing the refresh frequency f in a principled way is an open problem. Because SGD converges non-linearly, neither refreshing after every iteration nor refreshing after a fixed number of iterations is likely to be the optimal policy; deciding when to refresh requires understanding the convergence behavior of training, which is currently beyond the scope of this prototype. Finally, our analytical results are primarily qualitative: the queries surface candidate problems such as oscillation and bias clustering, but we do not yet provide controlled experiments showing

that acting on these signals improves convergence, and the system should not be confused with full explainable-AI methodologies such as SHAP or LIME.

5 Related Work

General ML observability and interpretability. Industrial machine learning workflows commonly rely on observability platforms such as MLflow and Weights & Biases, which track experiments, metrics, and artifacts at the run level. These platforms address a coarser granularity than our work: they record summary information about a training run, whereas our schema records per-iteration, per-neuron, per-data-point activations and per-iteration biases. A complementary research thread targets the interpretability of trained networks rather than runtime monitoring. Network Dissection [1] automatically labels hidden units by aligning them with a database of semantic concepts, providing a foundation for linking hidden features to input semantics. Visualization of loss landscapes [8] supplies a mathematical basis for comparing the sharpness of minima, a metric that a monitoring database could in principle compute to suggest learning-rate adjustments. The broader question of when an opaque deep model is preferable to an interpretable one—and how the two compare on practical tasks—has been studied for medical diagnosis by Ordonez et al. [13], who contrast association rules with deep neural networks for heart-disease prediction and quantify the interpretability–accuracy trade-off that motivates inspection-oriented work like ours. Compact-representation approaches reduce the same trade-off from the model side: weight-pruning methods such as those of Han et al. [7] retain only critical weights to reduce storage and accelerate inference, and Erhan et al. [4] show that high-sparsity training strategies can preserve binary-classification accuracy on convolutional models, supporting the broader principle on which our binary-activation design relies. These directions are orthogonal to our contribution but together suggest that runtime SQL-based monitoring, post-hoc interpretability, and weight-level compression can be combined to give a more complete picture of neural network behavior.

Database systems for neural networks. A second line of work proposes *systems* that treat neural networks as queryable or storable objects. The Neural Query System (NQS) of Lytton [10] pioneered the use of a relational database to manage and analyze neuron simulations, providing SQL-based access to parameters and connectivity. ModelDB [15] extended these ideas to machine learning workflows by indexing model metadata, hyperparameters, and performance. DeepBase [14] is the closest prior work to ours: it provides a system for deep inspection of trained neural networks via declarative queries over learned representations. More recently, TRAILS [17] integrates model selection and internal activation filtering as a declarative query process under service-level objectives, NeurDB [18] proposes a tensor-native database design for exploratory queries, and MorphingDB [16] presents a task-centric AI-native DBMS for unified model management and inference. Parallel to these system-building efforts, recent studies have

explored the underlying algorithmic and architectural questions: Ordonez [12] develops sparse-matrix algorithms for evolving neural networks based on database-style coordinate-tuple storage, which directly motivates our choice to view the network as a data source for a relational engine; Bouhatous et al. [2] target the power-efficiency dimension of AI-optimized databases; Monir and Ghinita [11] study differentially-private neural network training, addressing a privacy-aware version of the same training-loop monitoring concern that we address with a focus on overhead; and Loizou and Tsoumakos [9] model analytics operators over multiple datasets using vector embeddings, an orthogonal use of learned representations inside a database setting. These are all full systems or system-flavored studies that store a substantial portion of the network state (often including weights, gradients, or learned representations) or that build custom engines on top of relational ideas. In contrast, our system deliberately stores only binary activations and floating-point biases on an off-the-shelf PostgreSQL backend, sacrificing the breadth of those systems to obtain very low monitoring overhead.

Formal query languages for neural networks. A third, more recent line of work studies neural network querying as a *theoretical* problem rather than as a systems problem. Grohe et al. [6] introduce a family of query languages for neural networks at ICDT, establishing formal foundations for declarative access to model parameters and outputs. SQL4NN, by Gerarts et al. [5], demonstrates how validation and expressive querying of models as data can be expressed in SQL, providing a concrete bridge between traditional relational engines and modern neural network workloads. This direction has emerged only in the past year or so, as is typical: the theory of a problem tends to be studied once the problem itself becomes popular. Our contribution is complementary: rather than proposing a new query language, we show that existing SQL is sufficient to express a useful class of monitoring queries, provided the underlying schema is designed to capture compact runtime primitives.

6 Conclusions

In this paper we introduced a lightweight relational database system to monitor and analyze the internal behavior of neural networks during training. To avoid the I/O bottlenecks associated with extracting full weight matrices, the system selectively captures only binary neuron activations, floating-point biases, and predictions, leaving the weights out of the database. By treating the neural network as a data source and using standard SQL queries, we showed how analysts can trace misclassifications, study the distribution of biases across hidden layers, and identify unstable oscillating neurons without writing low-level instrumentation code. The experimental evaluation across three datasets of different scale and domain confirmed that this approach keeps the parameter-transfer cost a small fraction of the cost of the PyTorch computations themselves, and that the three monitoring queries return informative signals about a training run.

For future work, we plan to expand the system along several directions. First, we aim to scale our approach to evaluate larger and more complex deep

learning architectures, including Convolutional Neural Networks (CNNs) and transformer-based models. Second, we will explore extraction techniques that capture continuous activation behavior and inter-neuron interactions while preserving the low I/O overhead of our current binary pipeline. Additionally, future work may extend the schema to log weight matrices at a much lower frequency than activations and biases, since they do not change drastically and do not need to be synchronized at every iteration. Third, we plan to move from passive monitoring to active optimization, with controlled experiments where SQL query insights—such as detecting high oscillation rates—trigger automated training interventions, allowing us to directly measure improvements in convergence. We also intend to benchmark our database-driven approach against established machine learning observability platforms in larger-scale deployment settings. Finally, building on the preliminary analysis in Section 3.5, we plan a deeper formal study of these monitoring queries, including I/O cost on disk-resident tables, parallel and distributed query evaluation, satisfiability of compound monitoring predicates, and the connection between the convergence dynamics of training and the differential equations underlying backpropagation.

Acknowledgments. We thank Divija Amaram and Janet Y. Anagli for their initial work that served as the starting point for this project; we simplified and streamlined their early prototype into the system described here.

Use of AI Tools. The authors used Claude (Anthropic) and Gemini (Google) for language editing, restructuring of arguments, LaTeX formatting, and generating the SQL queries during manuscript preparation. All research contributions and experimental work are the responsibility of the authors.

References

1. Bau, D., Zhou, B., Khosla, A., Oliva, A., Torralba, A.: Network dissection: Quantifying interpretability of deep visual representations. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017. pp. 3319–3327. IEEE Computer Society (2017)
2. Bouhatous, A., Bellatreche, L., Abdelwahed, E.H., Ordonez, C.: PAID: Power-efficient ai-optimized databases. In: Big Data Analytics and Knowledge Discovery - 27th International Conference, DaWaK 2025. pp. 244–250. Lecture Notes in Computer Science, Springer (2025)
3. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems. Pearson, 7th edn. (2016)
4. Erhan, L., Cavallaro, L., Antinori, M.A., Liotta, A.: Evaluation of high sparsity strategies for efficient binary classification. In: Big Data Analytics and Knowledge Discovery - 26th International Conference, DaWaK 2024. pp. 106–111. Lecture Notes in Computer Science, Springer (2024)
5. Gerarts, M., Steegmans, J., den Bussche, J.V.: SQL4NN: validation and expressive querying of models as data. In: Proceedings of the Workshop on Data Management for End-to-End Machine Learning, DEEM 2025, Berlin, Germany, June 22-27, 2025. pp. 10:1–10:5. ACM (2025)

6. Grohe, M., Standke, C., Steegmans, J., den Bussche, J.V.: Query languages for neural networks. In: 28th International Conference on Database Theory, ICDT 2025, Barcelona, Spain, March 25-28, 2025. pp. 9:1–9:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025)
7. Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both weights and connections for efficient neural network. In: Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada. pp. 1135–1143 (2015)
8. Li, H., Xu, Z., Taylor, G., Studer, C., Goldstein, T.: Visualizing the loss landscape of neural nets. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada. pp. 6391–6401 (2018)
9. Loizou, A., Tsoumakos, D.: Analytics modelling over multiple datasets using vector embeddings. In: Database and Expert Systems Applications - 36th International Conference, DEXA 2025. pp. 237–253. Lecture Notes in Computer Science, Springer (2025)
10. Lytton, W.W.: Neural query system - data-mining from within the NEURON simulator. Neuroinformatics **4**(2), 163–175 (2006)
11. Monir, I.A., Ghinita, G.: Differentially-private neural network training with private features and public labels. In: Big Data Analytics and Knowledge Discovery - 26th International Conference, DaWaK 2024. pp. 208–222. Lecture Notes in Computer Science, Springer (2024)
12. Ordonez, C.: Sparse matrix algorithms for evolving neural networks. In: Big Data Analytics and Knowledge Discovery - 27th International Conference, DaWaK 2025. pp. 3–17. Lecture Notes in Computer Science, Springer (2025)
13. Ordonez, C., Fund, I., Bellatreche, L.: Comparing association rules and deep neural networks for heart disease prediction. In: IEEE International Conference on Big Data, BigData 2022. pp. 4416–4424. IEEE (2022)
14. Sellam, T., Lin, K., Huang, I.Y., Yang, M., Vondrick, C., Wu, E.: Deepbase: Deep inspection of neural networks. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019. pp. 1117–1134. ACM (2019)
15. Vartak, M., Subramanyam, H., Lee, W., Viswanathan, S., Husnoo, S., Madden, S., Zaharia, M.: Modeldb: a system for machine learning model management. In: Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016. p. 14. ACM (2016)
16. Wu, S., Xia, R., Yang, D., Wang, R., Lai, H., Guan, J., Bai, J., Zhang, D., Tang, X., Xie, Z., Lu, P., Chen, G.: Morphingdb: A task-centric ai-native DBMS for model management and inference. Proc. ACM Manag. Data **3**(6), 1–26 (2025)
17. Xing, N., Cai, S., Chen, G., Luo, Z., Ooi, B.C., Pei, J.: Database native model selection: Harnessing deep neural networks in database systems. Proc. VLDB Endow. **17**(5), 1020–1033 (2024)
18. Zhao, Z., Cai, S., Gao, H., Pan, H., Xiang, S., Xing, N., Chen, G., Ooi, B.C., Shen, Y., Wu, Y., Zhang, M.: Neurdb: On the design and implementation of an ai-powered autonomous database. In: 15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025. www.cidrdb.org (2025)