DeepEverest: Accelerating Declarative Top-K Queries for Deep Neural Network Interpretation

Technical Report

Dong He
Paul G. Allen School of Computer Science & Engineering
University of Washington
donghe@cs.washington.edu

Maureen Daum
Paul G. Allen School of Computer Science & Engineering
University of Washington
mdaum@cs.washington.edu

Magdalena Balazinska
Paul G. Allen School of Computer Science & Engineering
University of Washington
magda@cs.washington.edu

arXiv:2104.02234v1 [cs.DB] 6 Apr 2021

ABSTRACT

We develop, implement, and evaluate DeepEverest, a system for the efficient execution of *interpretation by example* queries over the activation values of a deep neural network. DeepEverest consists of an efficient indexing technique and a query execution algorithm with various optimizations. Experiments with our prototype implementation show that DeepEverest, using less than 20% of the storage of full materialization, significantly accelerates individual queries by up to 62× and consistently outperforms other methods on multi-query workloads that simulate DNN interpretation processes.

1 INTRODUCTION

Deep neural networks (DNNs) are increasingly used by machine learning (ML) applications. When training and deploying DNNs, interpretation and debugging are important for researchers and data scientists to understand what their models are learning. DNN interpretation is a relatively new field of research, and techniques are evolving. While many new approaches are being developed, they often do not scale with the size of the datasets and models [47]. The problem that we address in this paper is the efficient execution of a common class of DNN interpretation queries.

The fundamental building blocks of DNN interpretation are neurons and groups of neurons. As described in Section 2, each neuron outputs an activation value as the input is propagated through the network. Existing DNN interpretation techniques often perform analysis on the activation values of neurons [6, 7, 14, 41, 61, 63]. When DNNs are trained on tasks such as image classification or scene synthesis, there emerge individual neurons and groups of neurons that match specific human-interpretable concepts [7, 14, 62], such as "palaces" and "trees".

To understand what individual neurons and groups of neurons learn and detect, researchers often ask *interpretation by example* queries [29]. These queries help with understanding the functionality of neurons and neuron groups by tying that functionality to the input examples in the dataset. A widely used query is, *"find the top-k inputs that produce the highest activation values for an individual neuron or group of neurons"* [3, 13, 15, 22, 34, 51, 58, 59, 62]. Another common query is, *"for any input, find the k-nearest neighbors in the dataset using the activation values of a group of neurons based on the proximity in the latent space learned by the DNN"* [3, 10, 24, 37, 39, 44, 55]. As a concrete example, consider a DNN trained to classify images. A user may be interested in understanding what parts of an image of a dog caused the model to correctly predict its class. The user may inspect the maximally activated neurons of different layers in the network based on the conjecture that groups of maximally activated neurons act as semantic detectors of features in the image (e.g., floppy ears). To investigate whether these neurons exhibit similar behavior for other images of dogs, the user may then ask for the most similar images to the sample image based on the activation values of a group of neurons.

This paper presents a system called DeepEverest that focuses on accelerating the aforementioned two kinds of queries: (1) find top-k inputs that produce the highest activation for a user-specified group of neurons, and (2) find the top-k most similar inputs based on a given input's activation values for a user-specified neuron group. A group of neurons consists of one or more neurons within a layer of the DNN. We call the first type of query a *top-k highest* query and the second type of query a *top-k most-similar* query.

Executing these types of *interpretation by example* queries efficiently with low storage overhead is challenging. One baseline approach is to materialize the activation values for all inputs and all neurons. However, this approach requires too much storage space. For example, storing all the activation values uncompressed of a ResNet50 network on a dataset of 10,000 images occupies 1.35 TB of disk storage. As another example, storing all the activations for ten epochs of a VGG16 network on a dataset of 50,000 images requires 350 GB of compressed storage [53]. At the other extreme, recomputing all activation values at query time imposes no storage overhead, but is compute-intensive and extremely slow because it requires DNN inference to compute the activation values on the entire dataset at query time. For instance, answering a *top-k most-similar* query that targets a relatively late layer of ResNet50 on a dataset of 10,000 images takes more than 120 seconds, which renders the DNN interpretation process tedious.

Further, although the target query is a k-nearest neighbor (KNN) search, existing approaches to accelerate KNN queries are not applicable. KNN methods rely on building efficient data structures such as trees [8, 32, 42] or hash tables [4, 11] in advance for faster query execution later. One could try to build a single, large, multidimensional data structure for all neurons in each layer. However, such an index would not perform well because of its very large dimensionality. DNNs frequently have layers with multiple thousands of neurons, thus dimensions. One could build data structures for all possible neuron groups that a user could query. However, this would either limit the user to a small set of possible queries or would be prohibitively expensive both in time and storage because the number of possible neuron groups grows exponentially

with the number of neurons in each layer. Additionally, in all cases, precomputing and storing all activation values in such data structures would add a prohibitive storage overhead.

While many systems have recently been developed to enable various forms of DNN interpretation [3, 22, 23, 26, 45, 47, 53], as we discuss further in Section 3, none supports flexible and efficient *interpretation by example* queries. In prior work [35], we investigated the use of sampling for model diagnosis. That work, however, focused only on aggregate queries and the use of approximate query processing. The closest work to DeepEverest is MISTIQUE [53]. MISTIQUE introduced storage techniques such as compression and quantization. Those techniques are orthogonal to DeepEverest and could complement our approach. It is, however, possible to use some of MISTIQUE's techniques as a caching algorithm, which we compare against in our experiments.

In DeepEverest, we design and implement an indexing technique called Neural Partition Index, and an efficient query execution algorithm, called Neural Threshold Algorithm, which has low storage overhead, reduces the number of activation values that must be computed at query time, and guarantees the correctness of top-k results. DeepEverest builds on the classic threshold algorithm [12], which can support top-k queries that target arbitrary neuron groups. The classic threshold algorithm requires full materialization of the activation values of the dataset. Because the bottleneck of query execution is DNN inference (not the calculation of the top-k group), the classic threshold algorithm would not accelerate our target queries. We argue that for any algorithm to improve query time, it must reduce the number of inputs on which DNN inference is run at query time. DeepEverest achieves this reduction of DNN inference at query time while keeping the storage overhead low by building the Neural Partition Index and using this index in the Neural Threshold Algorithm which is a modified threshold algorithm. Rather than store the activation values for all neurons, the Neural Partition Index partitions the inputs and stores a small amount of information per-partition that is useful when deciding which activation values to recompute at query time. The Neural Threshold Algorithm then uses insights from the classic threshold algorithm to decide when to terminate as it incrementally recomputes activation values using DNN inference only for small subsets of inputs as needed to answer the query.

In addition to its fundamental approach, DeepEverest also includes several important optimizations: (1) incremental indexing to avoid a large computation overhead in advance of query execution; (2) an extra Maximum Activation Index to accelerate *top-k most-similar* queries that specifically target maximally activated neurons and *top-k highest* queries; (3) automated configuration tuning; (4) and inter-query acceleration, which further speeds up sequences of related queries.

In summary, the contributions of this paper are:

- We propose, design, and implement a system called DeepEverest that includes an efficient index structure and query execution algorithm that accelerates *interpretation by example* queries for DNN interpretation while keeping the storage overhead low (Section 4.2, Section 4.3, Section 4.4, Section 4.5).
- We develop multiple additional optimizations for DeepEverest that accelerate individual queries (Section 4.6.1), auto-tune parameters (Section 4.6.2), and accelerate sequences of related queries (Section 4.6.3).
- We implement a DeepEverest prototype and evaluate it on benchmark datasets and models (Section 5). We demonstrate that DeepEverest, using less than 20% of the storage of full materialization, significantly accelerates individual *interpretation by example* queries by up to 62.7× and consistently outperforms other methods on multi-query workloads that simulate DNN interpretation processes.

2 PROBLEM FORMULATION

Modern machine learning applications commonly utilize deep neural networks (DNNs). A DNN consists of layers composed of units, called neurons, connected by edges with associated weights. Inputs to the DNN are propagated through the layers. The output of a single neuron is a linear combination of its inputs and their associated edge weights that is optionally transformed by a nonlinear activation function. For example, an activation function may output only positive values by mapping negative values to 0 [38], or scale inputs to values in the range $(0, 1)$.

The output of each neuron for a given input is called its *activation value* (or *activation*). DNN interpretation involves the study of these activation values [6, 7, 14, 41, 61, 63]. Typical questions that a researcher or scientist may ask include *"For any input, which are the most highly activated neurons in a layer?"* or *"What are the nearest neighbors for an input in the dataset based on the proximity in the latent space defined by these highly activated neurons?"* These types of queries enable researchers and scientists to reason about what the DNN learns and identify how groups of neurons match human-interpretable concepts (e.g., a wheel in a picture of a bicycle).

In this paper, we address the problem of enabling fast queries over activation values in a neural network. Conceptually, a DNN and input dataset can be described by the following two relations:

- `Neuron(neuronID, layerID, ...)`
- `Artifact(inputID, neuronID, activation)`

DeepEverest supports two fundamental classes of queries over activation values: *top-k highest* queries that find the top-k inputs that produce the highest activation values for a user-specified group of neurons and *top-k most-similar* queries that find the top-k inputs that are most similar to a user-specified sample input based on the activation values of a user-selected group of neurons. The rank of an input is decided by a user-specified distance function (or the system default function), DIST. Based on the user-selected group of neurons, for *top-k highest* queries, DIST measures the magnitude of the input's activations, and it takes as input a set of activation values; for *top-k most-similar* queries, DIST measures the distance between the input and the sample input, and it takes as input a set of absolute differences between the input's activations and the sample input's activations. This distance function DIST must be monotonic, i.e., $\text{DIST}(x_1, x_2, \ldots, x_n) \leq \text{DIST}(x'_1, x'_2, \ldots, x'_n)$ whenever $x_i \leq x'_i$ for each i. The monotonicity is satisfied by common distance functions, such as $l1$-distance, $l2$-distance, cosine distance (once transformed to normalized $l2$-distance), and weighted distances like Mahalanobis distance, among others. The default DIST in DeepEverest is $l2$-distance.

3 RELATED WORK

DNN interpretation. Many approaches have been proposed to interpret the internals of deep neural networks [3, 6, 7, 10, 13–15, 22, 24, 34, 39, 41, 44, 51, 55, 58, 59, 61–63]. These approaches ask *interpretation by example* queries that return most similar inputs with respect to the learned representations (activations of a group of neurons) of a given input or inputs that maximally activate a group of neurons. They motivate the design of DeepEverest. DeepEverest does not invent new interpretation methods, but it instead builds novel indexes and algorithms that accelerate the query execution of these commonly asked queries for DNN interpretation.

System for Machine Learning. Many systems have been proposed to support efficient machine learning such as [36, 49, 54, 57]. DeepEverest falls into the groups of systems that support model diagnosis and interpretation [3, 9, 22, 23, 25, 28, 30, 33, 47, 53, 58]. A number of systems like ModelTracker [3], VisTrails [9], Prospector [25], CNNVis [30], and others [22, 23, 28, 33, 58] support visual inspection of machine learning models, workflows, and features. These systems could utilize DeepEverest to accelerate some of the queries used to build the visualizations. DeepBase [47] abstracts model diagnostic queries as hypotheses verification tasks and lets users identify neurons that have statistical dependencies with user-specified hypotheses. However, it does not support *interpretation by example* queries. MISTIQUE [53] accelerates diagnostic queries that examine the activations of neurons by focusing on storage techniques such as quantization while sacrificing some query accuracy to reduce the storage overhead, which is orthogonal to DeepEverest, and DeepEverest could incorporate these techniques to further reduce the storage overhead. MISTIQUE also proposes a storage cost model that captures the trade-off between materialization and recomputation of the activations for different layers and makes materialization decisions accordingly. These systems have not addressed the problem of accelerating *interpretation by example* queries well because they do not reduce the number of activation values computed or loaded from disk during query execution as DeepEverest does.

Nearest Neighbor Search. The target query in this paper is a k-nearest neighbor (KNN) search. While there exist many methods for exact nearest neighbors [8, 16, 32, 42] as well as for approximate nearest neighbors [4, 11, 20, 21, 56], the challenge in this paper is fundamentally different from the KNN search. These KNN search methods need to know what dimensions will be queried ahead of time and construct data structures in that space. In our problem, the dimensions of the KNN search are defined by the neuron group specified only at query time.

Top-K Query Processing. Top-k query processing is formalized by the seminal work on the threshold algorithm [12]. The threshold algorithm scans multiple sorted lists and maintains an upper bound for the aggregate score of unseen objects. Each newly seen object is accessed (by random access) in every other list and the aggregate score is computed by applying the scoring function to the object's value in every list. The algorithm terminates after k objects are seen with scores greater than or equal to the upper bound. Many follow-up approaches propose approximation, optimizations, and extensions [2, 5, 17, 19, 43, 52, 60]. These top-k query processing techniques assume that accesses (either sorted or random access) are available to the underlying data sources. However, this assumption does not hold in our problem setting. The activation values needed for the top-k query cannot be stored on disk because the storage overhead is too high. They also cannot be computed at query time because of the high computation overhead. What is novel in DeepEverest is that it avoids computing as many activation values as possible at query time, while keeping storage overheads low, by building the indexes we design and using those indexes in a modified threshold algorithm during query execution.

4 DEEPEVEREST

In this section, we first consider baseline approaches, then describe how DeepEverest improves upon those baselines to accelerate query execution while keeping storage costs low.

4.1 Baselines

In this section, we discuss baseline approaches and explain why applying the classic threshold algorithm (or any KNN algorithm) would not improve the query time.

PreprocessAll. The first baseline, *PreprocessAll*, has a high storage cost. It performs DNN inference on the entire dataset and stores all the activations ahead of time. It executes queries by loading the previously-stored activation values of the neuron group for all inputs from disk and maintaining a top-k result set.

ReprocessAll. The second baseline, *ReprocessAll*, has a high computation cost. It has no storage overhead and performs no preprocessing. It executes queries by computing the activation values of the layer being queried by DNN inference on all inputs and maintaining a top-k result set as it performs the recomputation.

LRU Cache. The third baseline, *LRU Cache*, is a disk cache that has a fixed storage budget with a least-recently-used (LRU) replacement policy. This strategy strikes a balance between the storage overhead of *PreprocessAll* and the computation overhead of *ReprocessAll*. *LRU Cache* maintains a fixed-sized disk cache that stores the activation values for queried layers. A query is executed as in *PreprocessAll* if the activations of the queried layer are present in the disk cache. Otherwise, it is executed as in *ReprocessAll*. After that, the activations of the queried layer are persisted to the disk cache. When the size of the disk cache exceeds the storage budget, the cache evicts the activations of the least recently used layer.

Priority Cache. The final baseline, *Priority Cache*, is a technique adapted from MISTIQUE [53]. *Priority Cache* has a fixed-sized disk cache to store the activation values for some layers. As a preprocessing step, it uses the storage cost model from [53] to pick which layers to store on disk, assuming each layer will be queried the same number of times. Under the storage budget, this storage cost model prioritizes the layers that save the most query time per GB of data stored. It performs DNN inference on every input and stores the activation values for the layers selected ahead of time. A query is executed as in *PreprocessAll* if the activations of the queried layer are present in the disk cache. Otherwise, the query is executed as in *ReprocessAll*.

The classic threshold algorithm could be applied to each of these baselines by first using the materialized or recomputed activations to construct the `Artifact` table (defined in Section 2). `Artifact` is then used to construct a relation in which each row represents an input, and each column represents a neuron and contains the absolute difference between the activation of that row's input and the activation of the target input on the column's neuron. The classic threshold algorithm can be applied after sorting the absolute differences in ascending order, using any monotonic norm of the absolute differences as the aggregation function.

However, applying the classic threshold algorithm (or any KNN algorithm) on top of each of the baselines would not improve query times. Using it along with *ReprocessAll* would not help because *ReprocessAll* requires running DNN inference on the entire dataset to compute `Artifact` at query time, which is the bottleneck of query execution. Similarly, applying it on *PreprocessAll* would not improve query times because generating the relation of absolute differences requires a full scan over `Artifact` before the threshold algorithm can be applied. However, the top-k result set could already be computed during the full scan. Applying it on top of *LRU Cache* and *Priority Cache* would not improve the query times for the same reasons as *PreprocessAll* (for layers in the cache) and *ReprocessAll* (for layers not in the cache).

4.2 Overview of DeepEverest

As described, directly applying the classic threshold algorithm does not improve query time because `Artifact` must be fully computed at query time, which requires DNN inference on all inputs. Query execution can be significantly accelerated by avoiding running the DNN on inputs that will not be one of the top-k results.

We design and build a novel index, which we call the Neural Partition Index (discussed in Section 4.3), and a query execution algorithm, which we call the Neural Threshold Algorithm (discussed in Section 4.4). The Neural Threshold Algorithm is a modified threshold algorithm. In contrast to the classic threshold algorithm, it does not require all activation values of all inputs before it starts. It utilizes the Neural Partition Index to progressively access the inputs that are theoretically possible to be in the top-k results, and only performs DNN inference on these inputs. This algorithm overcomes the bottleneck of query execution, DNN inference, by reducing the number of inputs on which DNN inference is performed at query time, while guaranteeing the precision of the top-k results returned and introducing only tolerable storage overhead.

4.3 Neural Partition Index

The activation values in a DNN can be conceptually represented by `Artifact` introduced in Section 2. Conceptually, DeepEverest builds an index on the search key `(neuronID, activation)` and supports queries that return the `inputIDs` for a given `neuronID` and range of `activation` values. An important goal of DeepEverest is to avoid materializing as many activation values as possible. For this purpose, DeepEverest builds an index on `(neuronID, partitionID)` instead, where `partitionID` (`PID`) is the identifier of a range-partition over activation values. DeepEverest builds equi-sized partitions, and partition 0 contains the largest activation values. The index then supports efficient

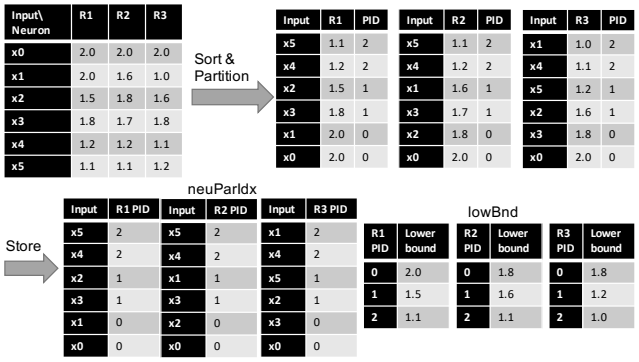


Figure 1: An example of building the Neural Partition Index of three neurons, $R1, R2, R3$, for six inputs, $x0, \ldots, x5$.

lookups for a given `(neuronID, partitionID)` combination. Specifically, the index returns the set of `inputIDs` whose activation values for the given `neuronID` belong to the partition identified with `partitionID`. Moreover, the index also supports queries that return the `partitionID` for a given `(neuronID, inputID)` combination. We return to this type of query later. We call this structure the Neural Partition Index and denote the queries with `neuParIdx(neuronID, partitionID)` and `neuParIdx(neuronID, inputID)`. Additionally, for each partition, DeepEverest also stores the *lower bound* of the activation values in that partition and supports queries that ask for that lower bound. We denote this query with `lowBnd(neuronID, partitionID)`. The number of partitions, $nPartitions$, is a configurable parameter that is discussed further in 4.6.2.

There are two approaches to implementing the index. The first approach would be to maintain a set of buckets, each identified with a unique `(neuronID, partitionID)` combination as the key, and, for each bucket, maintain a list of `inputIDs`. The second approach, which DeepEverest uses, is to maintain a list of `(neuronID, inputID)` pairs as keys, and, for each entry, store the `partitionID`. `neuronID` and `inputID` are integers, so rather than building a B-tree or a hash index over the keys `((neuronID, partitionID)` in the first approach, and `(neuronID, inputID)` in the second approach), we create an optimized index structure using an array where the `neuronID` and `inputID` act as offsets for lookups in the array. This enables us to only store the values and therefore avoid the cost of storing the keys. Figure 1 illustrates the Neural Partition Index for an example dataset with three partitions.

During pre-processing, DeepEverest runs DNN inference on all inputs once to build the Neural Partition Index for every neuron. The pre-processing time is reported in Section 5. The method that DeepEverest adopts is more space-efficient than building an index over `(neuronID, partitionID)` pairs because it costs $nNeurons \cdot nInputs \cdot \log_2(nPartitions)$ bits rather than $nNeurons \cdot nInputs \cdot \log_2(nInputs)$ bits, where $nPartitions << nInputs$. The Neural Partition Index also has a much smaller storage overhead compared to fully materializing all activation values. A `partitionID` takes less storage than an activation value because a `partitionID` only costs $log_2(nPartitions)$ bits, while an activation value is usually a 32-bit floating point number that costs 32 bits. For example, if

dPar

PID\Neuron	R1	R2	R3
0	0.9	0.7	0.6
1	0.4	0.5	0.0
2	0.0	0.0	0.2

ord

c\Neuron	R1	R2	R3
0	2	2	1
1	1	1	2
2	0	0	0

Figure 2: *dPar* and *ord* for the execution of the Neural Threshold Algorithm for the example query of finding the most similar inputs to $x5$.

toRun (c = 0)

Neuron	R1	R2	R3
toRun	{x4, x5}	{x4, x5}	{x2, x5}

toRun (c = 1)

Neuron	R1	R2	R3
toRun	{x2, x3}	{x1, x3}	{x1, x4}

(a) Build *toRun*

dist (c = 0)

Input newly computed	x2	x4
dist	1.5	0.3

dist (c = 1)

Input newly computed	x1	x3
dist	1.6	1.9

(b) Perform inference to compute *dist*

top (c = 0)

(x, dist(x))	(x4, 0.3)	(x2, 1.5)

top (c = 1)

(x, dist(x))	(x4, 0.3)	(x2, 1.5)

minBoundary, maxBoundary (c = 0)

Neuron	R1	R2	R3
(minA, maxA)	(1.1, 1.2)	(1.1, 1.2)	(1.2, 1.6)

minBoundary, maxBoundary (c = 1)

Neuron	R1	R2	R3
(minA, maxA)	(1.1, 1.8)	(1.1, 1.7)	(1.0, 1.6)

F, G (c = 0)

Neuron	R1	R2	R3
(F, G)	(∞, 1)	(∞, 1)	(1, 1)

F, G (c = 1)

Neuron	R1	R2	R3
(F, G)	(∞, 1)	(∞, 1)	(∞, 1)

minDist (c = 0)

Neuron	R1	R2	R3
minDist	0.1	0.1	0.0

t = 0.1 + 0.1 + 0.0 = 0.2 (c = 0)

minDist (c = 1)

Neuron	R1	R2	R3
minDist	0.7	0.6	0.4

t = 0.7 + 0.6 + 0.4 = 1.7 (c = 1)

(c) Check termination

Figure 3: Intermediate variables for the execution of the Neural Threshold Algorithm for the example query. The values when c=0 are shown on the left, and the values when c=1 are shown on the right.

DeepEverest has 8 partitions for each neuron, representing a `partitionID` costs 3 bits, which is less than 10% of the storage cost of full materialization. The cost of storing the lower bounds is negligible compared to that of storing `partitionID`s.

4.4 Neural Threshold Algorithm

Notation. We denote with N the set of all neurons in the DNN and with D the database of all inputs. A neuron is denoted with $n \in N$ and an input with $x \in D$. The user issues a query: $\texttt{topk}(s, g, k, \text{DIST})$, where $s \in D$ is the sample input, or sometimes called target input, that the user is focusing on. $g \subseteq N$ is a set of neurons from N. Neurons in g are from the same layer. k is the desired number of results, and DIST is the distance function. This function computes the distance between the set of activation values of s and x looking only at the neurons in g. Users can specify their desired distance function or use the system default, $l2$-distance. Table 1 lists our frequently used notation.

The Neural Threshold Algorithm returns the set of top-k inputs that are closest to the sample input when considering only the neurons in g. This set can be defined as set $top \subseteq D$ of k inputs. top is initially empty and is conceptually built incrementally by identifying and adding to top the next input that satisfies:

$$\arg \min_{x \in D \setminus top} dist(s, x, g) \tag{1}$$

We further denote with $g(i)$ (or i when clear in context) the i-th neuron in set g, and with $\texttt{act}(g(i), x)$ (or when clear $\texttt{act}(i, x)$), the activation value of neuron $g(i)$ on input x. For each neuron, $n \in N$, the Neural Partition Index includes the set of partitions, $P(n) \in P$. We denote a single partition for neuron n with $p \in P(n)$. We denote the lower bound of this partition $p \in P(n)$ with $\texttt{lowBnd}(n, p)$. The Neural Threshold Algorithm proceeds as follows,

Step 1: Load indexes. We assume that the Neural Partition Index is on disk when the query arrives. The first step reads the Neural Partition Index for neurons in the layer to which g belongs from disk. Only the Neural Partition Index for the neurons $g(i) \in g$ will be used. The index holds the set of partitions, their lower bounds, and the `partitionID` of each input for each $g(i) \in g$.

Step 2: Compute target activations. For each $g(i) \in g$, compute $\texttt{act}(i, s)$, the activation value for input s and neuron $g(i)$ by running DNN inference on input s. A single inference pass is sufficient to compute the activation values for all neurons in g.

Step 3: Order partitions. This step computes the order by which the partitions are accessed by the algorithm for each neuron. For each neuron $g(i) \in g$ and partition $p \in P(i)$, compute $dPar(i, p)$ as,

$$dPar(i, p) = \begin{cases} |\texttt{lowBnd}(i, p) - \texttt{act}(i, s)|, & s \notin p \\ 0, & s \in p \end{cases} \tag{2}$$

which is the distance between the target input's activation value for neuron i and the smallest activation value in partition p. For each neuron i, sort the partitions on their $dPar(i, p)$ values and put them in a list, denoted with $ord(i)$.

Example: To illustrate the first three steps, consider a query $\texttt{topk}(x5, \{R1, R2, R3\}, 2, l1\text{-distance})$ *that finds the top-2 most similar inputs based on* $x5$*'s activations on the neurons shown in*

Table 1: Summary of frequently used notation.

Symbol	Meaning
N	Set of neurons in the DNN
D	Database of all inputs
x	Arbitrary input from D
s	Sample input (or target input) from D
g	Group of neurons from N
k	Number of results to return
$\texttt{topk}(s, g, k, \text{DIST})$	*Top-k most-similar* query
DIST	Function to compute distances between inputs
$g(i)$	The i-th neuron in g
i	The i-th neuron in g (if clear in context)
$\texttt{act}(i, x)$	Activation value of $g(i)$ for input x
$dist(s, x, g)$	Distance between s and x based on g
top	Set of top-k inputs that are closest to the target input
$P(n)$	Partitions for a neuron n
`neuParIdx`	Neural Partition Index
$\texttt{lowBnd}(n, p)$	Lower bound of a single partition $p \in P(n)$
`maxActIdx`	Maximum Activation Index

Figure 1. In this example, s=$x5$, g=$\{R1, R2, R3\}$. Step 1 reads from disk the Neural Partition Index, denoted with neuParIdx and lowBnd in the figure. Step 2 runs inference to compute the activation values for $x5$ (i.e., $(\texttt{act}(R1, x5), \texttt{act}(R2, x5), \texttt{act}(R3, x5))$=$(1.1, 1.1, 1.2)$). In step 3, for each $i \in \{R1, R2, R3\}$, $ord(i)$ is computed as shown in Figure 2.

Step 4: Find top-k. This step runs the modified threshold algorithm. It starts with the partitions to which the target input belongs, and it expands its search from there. Unlike the classic threshold algorithm, this step incrementally computes the activation values for candidate inputs and does so in batches to get good GPU performance. This step proceeds as follows:

Starting with an index $c = 0$,

Step 4 (a): For each neuron $g(i)$, maintain a set $toRun_i$ that contains the inputs whose activation values should be computed. Access $ord(i, c)$ to get the partition that contains the next most similar inputs. Query the Neural Partition Index to get the `inputIDs` that belong to this partition $ord(i, c)$ and add them to $toRun_i$, i.e., $toRun_i \leftarrow \texttt{neuParIdx}(i, ord(i, c))$.

In the example, when c=0, $toRun_{R1,0}$=$\{x4, x5\}$ because $ord(R1, 0)$=2, as shown in Figure 3a

Step 4 (b): For each neuron i, compute the activation values for the inputs in $toRun_i$ (excluding those that have already been computed) by running DNN inference in batches. $toRun_i$ is cleared after DNN inference. Note that this inference step computes the activation values for *all* neurons in the neuron group being queried. Compute the distance between each newly computed input x and the sample input s as $dist(s, x, g) = \textsc{dist}(|\texttt{act}(0, x) - \texttt{act}(0, s)|, \ldots, |\texttt{act}(|g| - 1, x) - \texttt{act}(|g| - 1, s)|)$. Update top if $dist(s, x, g)$ is one of the k-lowest the algorithm has seen so far, i.e., input x is one of the k-most similar seen so far. Ties are broken arbitrarily.

In the example shown in Figure 3b, when c=0, the activation values of inputs $x2, x4$ are computed ($x5$ was computed in Step 1). The distances from $x2$ and $x4$ to $x5$ are 1.5 and 0.3, respectively.

Step 4 (c): Maintain a range of seen activation values for inputs from $toRun_i$ for each neuron i, which is the range of activation values such that the algorithm has seen every input with an activation value in the open interval of this range. Note that it is possible that the algorithm has seen one or more inputs from other neurons' $toRun$ sets with activation values outside of this range. However, the open interval of this range denoted by $(minBoundary_i, maxBoundary_i)$ only contains the values for which we are guaranteed to have seen every input.

Let $minDist_i$ be the shorter distance from the boundaries of this range to the sample input for each neuron i: $minDist_i = \min\{F_i \cdot |minBoundary_i - \texttt{act}(i, s)|, G_i \cdot |maxBoundary_i - \texttt{act}(i, s)|\}$, where F_i is an indicator function that indicates whether the algorithm has seen the last partition (inputs with lowest activations) of neuron i, and G_i is another indicator function that indicates whether the 0-th partition (inputs with highest activations) of neuron i has been seen. Specifically, F_i=∞ when the last partition of neuron i has been seen; F_i=1 otherwise. G_i=∞ when the first partition of neuron i has been seen; G_i=1 otherwise. Define the threshold to be,

$$t = \textsc{dist}(minDist_0, minDist_1, \ldots, minDist_{|g|-1}) \tag{3}$$

The threshold, t, represents the smallest possible distance to s from any unseen input. The termination condition is,

$$\max_{(x, dist(s,x,g)) \in top} \{dist(s, x, g)\} \leq t \tag{4}$$

The left-hand side of this inequality represents the maximum distance to s in the current top-k result set. As soon as this inequality holds, halt and return top as the query results.

In the example, as shown in Figure 3c, $minBoundary_i$, $maxBoundary_i$ and $minDist_i$ are maintained and calculated for $\{R1, R2, R3\}$. For example, when c=0, $minBoundary_{R1}$=1.1, $maxBoundary_{R1}$=1.2. Since the algorithm has seen the last partition (2) and has not seen the first partition (0), F_{R1}=∞, G_{R1}=1. Therefore, $minDist_{R1}$=$|maxBoundary_{R1} - act_{R1,x5}|$=$|1.2-1.1|$=0.1.

When $c = 0$, $t = 0.2 < 1.5 = \max_{(x,dist(s,x,g)) \in top} \{dist(s, x, g)\}$, so the algorithm does not halt. When $c = 1$, $t = 1.7 \geq 1.5 = \max_{(x,dist(s,x,g)) \in top} \{dist(s, x, g)\}$, so the algorithm halts, and returns top as the query results. It is worth noting that the cost of DNN inference on $x0$ is saved because its activations are not needed.

Step 4 (d): Increment c by 1. Repeat Step 4 (a) - (d) until all partitions have been seen or the halting condition is satisfied in Step 4 (c).

The pseudocode is shown in Algorithm 1.

The key innovation of the Neural Threshold Algorithm compared to the classic threshold algorithm is in its processing of the inputs partition-by-partition using the pre-constructed Neural Partition Index until the termination condition is met. The Neural Threshold Algorithm runs DNN inference on only the necessary partitions of inputs for it to be certain that it has the precise top-k results when it terminates. This approach significantly reduces the number of inputs that DNN inference is performed on at query time compared to computing the activation values for all inputs at query time. The algorithm also has a much smaller storage overhead compared to fully materializing the activation values for all inputs. It is guaranteed to return the precise top-k results, and the correctness and instance optimality can be proved following the same sketch in Fagin's paper[12]. The algorithm further improves performance by utilizing batch processing on GPUs [40]. All inputs that share a partition are sent to the DNN for inference at once.

4.5 Incremental Indexing

As we show in the evaluation, the DeepEverest approach described so far achieves excellent query execution times with only a small storage overhead. The approach, however, incurs a potentially high preprocessing cost, especially for large datasets and large models. Before executing any query, DeepEverest needs to compute the activation values for all neurons and all inputs by running DNN inference. It then needs to construct the indexes for all layers and persist those indexes to disk.

To address this challenge, we propose to build the indexes incrementally as queries execute. With this approach, DeepEverest performs no preprocessing ahead of time. When the user submits a query, if the Neural Partition Index and Maximum Activation Index of the queried layer are available on disk, DeepEverest proceeds as described in Section 4.4 and Section 4.6.1. Otherwise, DeepEverest computes the activation values of the queried layer by running DNN inference on all inputs. While doing so, it computes the query answer and returns it to the user. DeepEverest then constructs the

Algorithm 1 The Neural Threshold Algorithm for *top-k most-similar* queries.

function ANSWERQUERY($model, D, s, g, k,$ DIST) ▷ *model*: the DNN, *D*: dataset, *s*: sample image, *g*: neuron group, *k*: number of results to return, DIST: function to compute distances between inputs
  $layer \leftarrow$ GETLAYER(g)
  neuParIdx, lowBnd $\leftarrow$ LOADINDEX($layer$) ▷ Load indexes
  for all $g(i) \in g$ **do** ▷ $P(i)$ contains the partitions for neuron $g(i)$
    $P(i) \leftarrow$ GETPARTITIONS(neuParIdx, $g(i)$)
  $sampleAct \leftarrow$ MODELINFERENCE($model, layer, s$) ▷ Compute the activations for s by DNN inference
  Initialize act to an empty map that contains the activations of the neuron group for accessed inputs
  for all $g(i) \in g$ **do**
    act$(i, s) \leftarrow sampleAct(i)$
  for all $g(i) \in g$ **do**
    Initialize the list $dPar(i)$
    for all $p \in P(i)$ **do** ▷ $dPar(i, p)$: the distance from each partition p for neuron $g(i)$ to s
      if $s \in p$ **then** $dPar(i, p) \leftarrow 0$
      else $dPar(i, p) \leftarrow |$lowBnd$(i, p) -$ act$(i, s)|$
  for all $g(i) \in g$ **do** ▷ $ord(i)$: the order by which the partitions for neuron $g(i)$ are accessed
    $ord(i) \leftarrow$ ARGSORT($dPar(i)$)
  for all $g(i) \in g$ **do** ▷ Initialization of some variables
    $F_i \leftarrow 1, G_i \leftarrow 1$
    $minBoundary_i \leftarrow \infty, maxBoundary_i \leftarrow -\infty$
  $c \leftarrow 0, top \leftarrow \emptyset$ ▷ Starting with $c = 0$; *top*: current top-k result set
  $inputRun \leftarrow \{s\}$ ▷ *inputRun*: set of inputs that have been run for DNN inference
  while True do
    for all $g(i) \in g$ **do**
      if $ord(i, c)$ does not exist **then return** *top* ▷ Return if all partitions have been seen
      $toRun_i \leftarrow$ neuParIdx$(i, ord(i, c))$
    if *exitFlag* **then break**
    $toRunUnion \leftarrow \bigcup_{g(i) \in g} toRun_i \setminus inputRun$
    $toRunAct \leftarrow$ MODELINFERENCE($model, layer, toRunUnion$) ▷ Run DNN inference in batches
    for all $x \in toRunUnion$ **do**
      Initialize the list $diff$
      for all $g(i) \in g$ **do**
        act$(i, x) \leftarrow toRunAct_{i,x}$
        $diff_i \leftarrow |$act$(i, x) -$ act$(i, s)|$
      $dist(s, x, g) \leftarrow$ DIST($diff$) ▷ Compute the distance between x and s
      if $|top| < k$ **or** $dist(s, x, g) <$ GETMAXDIST(top) **then**
        UPDATE($top, x, dist(s, x, g)$) ▷ Update *top* if x is one of the k-most similar seen
    for all $g(i) \in g$ **do**
      for all $x \in toRun_i$ **do**
        $minBoundary_i \leftarrow$ MIN($minBoundary_i$, act(i, x))
        $maxBoundary_i \leftarrow$ MAX($maxBoundary_i$, act(i, x))
      if $ord(i, c + 1)$ does not exist **then** $F_i \leftarrow \infty$
      if $ord(i, c) == 0$ **then** $G_i \leftarrow \infty$
      $minDist_i \leftarrow$ MIN$(F_i \cdot |minBoundary_i -$ act$(i, s)|, G_i \cdot |maxBoundary_i -$ act$(i, s)|)$
    $t \leftarrow$ DIST($minDist$) ▷ Calculate the threshold t
    if $|top| == k$ **and** GETMAXDIST(top) $\leq t$ **then break** ▷ Termination condition
    $inputRun \leftarrow inputRun \cup toRunUnion$
    $c \leftarrow c + 1$
  return *top*

indexes for the layer and persists them to disk. With this approach, the preprocessing overhead for each layer is incurred once the first time that layer is queried, and only if that layer is queried.

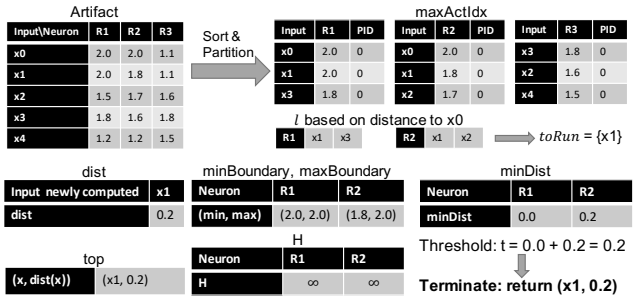


Figure 4: An example of constructing `maxActIdx` ($ratio$=0.6) and query execution for `topk(`$x0, \{R1, R2, R3\}, 1, l1$`-distance)` ($batchSize$=1). Despite $x0$ only being in `maxActIdx` for $R1$ and $R2$, DeepEverest leverages `maxActIdx` to answer the query after only running DNN inference on $x0$ and $x1$.

In Section 5, we show that DeepEverest with this incremental approach significantly outperforms other methods. While DeepEverest must do extra preprocessing to compute and store its indexes compared with caching the activation values directly, it accelerates significantly more queries because it is able to store the indexes for significantly more layers given a storage budget.

4.6 Optimizations

In this section, we present several important optimizations that further improve the performance of our approach. The first optimization, described in Section 4.6.1, accelerates two common types of top-k queries. The second optimization, described in Section 4.6.2, automatically tunes DeepEverest's parameters. Finally, the third optimization, described in Section 4.6.3, accelerates sequences of related queries, as may occur during data exploration.

4.6.1 Maximum Activation Index. For a given sample input and a layer in a DNN, the *maximally activated neurons* are those neurons in the layer for which the activation values for the sample input are the highest. DNN interpretation often involves examining such maximally activated neurons [6, 50, 59, 62] because they respond to the input the most and have the greatest impact on the DNN output. A common set of *top-k most-similar* queries ask to find the top-k similar inputs to a sample based on a neuron group consisting of these maximally activated neurons.

To accelerate these *top-k most-similar* queries that target maximally activated neurons as well as *top-k highest queries*, we introduce a straightforward yet effective optimization. The key idea is for DeepEverest to store, for each neuron, a small fraction of the highest activation values together with the corresponding `inputIDs`. We call this data structure the Maximum Activation Index, and denote it with `maxActIdx(neuronID)`. This small fraction of the highest activations automatically becomes each neuron's 0-th partition. We denote the fraction of inputs with the highest activations stored in this index with *ratio*, which is a configurable parameter discussed in Section 4.6.2.

DeepEverest utilizes `maxActIdx` during query processing to further reduce the number of inputs on which inference is performed, if possible. DeepEverest now has more detailed knowledge of which inputs are most similar to the target input, rather than just the high-level knowledge that the inputs are in the same partition. We observe empirically that the activation values of the *maximally activated neurons* for an input are often likely to be in the top activations stored in `maxActIdx`, and thus `maxActIdx` is effective in improving the query time. DeepEverest modifies query processing described in Section 4.4 of partition 0 to incorporate this information as follows: DeepEverest first finds the neurons for which the sample input is in the Maximum Activation Index (and therefore in the 0-th partition). For these neurons, DeepEverest sorts the other inputs in the 0-th partition by their distance to s. Rather than performing DNN inference on all inputs in each partition 0 for each neuron, DeepEverest builds a global *toRun* set by adding the most similar inputs from all of these neurons until the batch size is reached. Step 4(c) in Section 4.4 is modified to compute $minDist_i$ as $\min\left\{|minBoundary_i - act_{i,s}|, H_i \cdot |maxBoundary_i - act_{i,s}|\right\}$, where H_i is an indicator function that indicates whether the algorithm has seen the input with the highest activation in `maxActIdx(i)`. Specifically, H_i=∞ when highest activation of neuron i has been seen; H_i=1 otherwise. The neurons i for which s is not in `maxActIdx(i)` contribute 0 to the threshold calculation. Figure 4 illustrates an example of how `maxActIdx` is constructed and how the query execution proceeds using `maxActIdx`.

4.6.2 Configuration Selection. Given a storage budget, DeepEverest must decide how to allocate it between the Neural Partition Index and the Maximum Activation Index. A larger *ratio* enables more queries to benefit from the Maximum Activation Index, while more partitions enables DeepEverest to avoid running DNN inference on more inputs that do not contribute to the query results.

DeepEverest uses a heuristic algorithm to select good *nPartitions* and *ratio* given a storage budget. We first pick a value for *nPartitions*, which is key to good performance on queries that target any kind of neuron group (see Section 5.1). Then, we set the value of *ratio* using the remaining storage budget to further accelerate the two types of queries mentioned in Section 4.6.1.

Intuitively when partitions are smaller, fewer unnecessary inputs are reprocessed to compute their activation values because DeepEverest processes inputs partition-by-partition. However, if the partitions are too small, DeepEverest will not leverage the full GPU parallelization because the partition size will be smaller than the optimal batch size. Furthermore, *nPartitions* should be a power of two so that all the binary bits of a `partitionID` are fully utilized.

Given a storage budget, *budget* (in bytes), and a batch size, *batchSize*, DeepEverest sets the value of *nPartitions* to be the maximum power of two that satisfies $nPartitions \leq nInputs/batchSize$ and $cost(nPartitions) < budget$. $cost(nPartitions)$ is the bytes consumed by storing the Neural Partition Index with *nPartitions* and can be calculated as $nNeurons \cdot nInputs \cdot \log_2(nPartitions)/8$. *batchSize* is set to the value that achieves the highest throughput for the DNN. *batchSize* can also be set by the user to indicate the preferred level of parallelization of DNN inference.

Given the remaining storage budget, $budget - cost(nPartitions)$, DeepEverest then sets *ratio* to be the maximum value that satisfies $cost(ratio) \leq budget - cost(nPartitions)$. $cost(ratio)$ is the bytes consumed by storing the Maximum Activation Index

and can be calculated as $ratio \cdot nInputs \cdot nNeurons \cdot 4 \cdot 2$, where $nInputs \cdot nNeurons \cdot 4 \cdot 2$ is the bytes of storing all (`activation`, `inputID`) pairs since `activation` and `inputID` are 4 bytes each. In cases where there is no remaining storage budget after selecting $nPartitions$, DeepEverest sets $ratio$ to 0.

4.6.3 Inter-Query Acceleration. Inter-query acceleration is an optimization technique to accelerate sequences of related queries, as may occur during DNN interpretation. As an example, imagine that a user finds a misclassified image. The user may want to first see the maximally activated neurons in a layer for the image and then find images with similar maximally activated neurons. The user may then decide to change how many neurons they are looking at, e.g., go from the top-3 neurons to the top-4 neurons. These exploratory queries can be related in different ways. In addition to queries that overlap in neurons, the user-specified sample input in a query could also be one of the top-k results for recent queries. Such queries present an opportunity for further optimization as activation values can be reused for related queries.

For inter-query acceleration, DeepEverest leverages an in-memory cache that contains recently used activation values to reduce the number of inputs that it must run DNN inference on. Note that this in-memory cache is different from the disk caches described in Section 4.1. During query execution, DeepEverest inserts the activation values of each input processed by the Neural Threshold Algorithm into the cache. Instead of caching only the activation values for the neuron group being queried, it caches the activations for *all* neurons in the queried layer. This enables DeepEverest to utilize the cache for future related queries that target a different set of neurons in the same layer. DeepEverest utilizes a most recently used (MRU) replacement policy for the in-memory cache. This is because DeepEverest processes partitions in order from most similar to the target input to least similar, and we seek to prioritize keeping the activations from the most similar partitions in the cache. We show in Section 5.6 that given a small in-memory cache budget, DeepEverest with inter-query acceleration achieves up to 5.02× faster query times than DeepEverest without it.

5 EVALUATION

We implement DeepEverest in Python, using C++ to construct the Neural Partition Index and the Maximum Activation Index. We evaluate DeepEverest against the baselines described in Section 4.1.

5.1 Evaluation Setup

Datasets and models. We evaluate DeepEverest on two well-known datasets and models. The first, called *CIFAR10-VGG16*, uses as inputs 10,000 images with resolution 32×32×3 from the test set of CIFAR10 [27], and uses as a DNN a VGG16 model [31, 48] trained on the training set of CIFAR10. The second, called *ImageNet-ResNet50*, uses as inputs 10,000 images with average resolution 469×387×3 from the validation set of ImageNet [46], and uses as a DNN a ResNet50 model [18] trained on the training set of ImageNet. These two sets of models and datasets complement each other in terms of model and input size and DNN inference cost. In all experiments, we pre-load the entire input dataset into memory because its size is orders of magnitude smaller than the size of activations or indexes. The loading times for the activations and indexes do not include the loading time for the input dataset. We set $batchSize$ for each model as the value that achieves the highest inference throughput (128 for *CIFAR10-VGG16*; 64 for *ImageNet-ResNet50*).

Query generation. To generate queries, we consider 3 types of layers: *early*, *mid*, and *late*. For *CIFAR10-VGG16*, these correspond to layers activation_2, activation_7, and activation_13, respectively. For *ImageNet-ResNet50*, we use layers activation_2, activation_25, and activation_48. Given an input and a layer, we consider the following types of neuron groups: (a) *Top*: the maximally activated neurons for the given input in a layer; and (b) *RandomHigh*: neurons randomly picked from the top half of non-zero neurons for the given input in a layer. We further consider *small*, *medium*, and *large* neuron groups consisting of 1, 3, and 10 neurons, respectively. Finally, based on the neuron groups, we use the following query types: (a) *FiringMax*: Find the top-k inputs that maximally activate a given neuron group (*top-k highest* query); (b) *SimilarTop*: Find the top-k most similar inputs based on a given input's activations of a *Top* neuron group; and (c) *SimilarHigh*: Find the top-k most similar inputs based on a given input's activations of a *RandomHigh* neuron group. We randomly select inputs from each dataset to generate *SimilarTop* and *SimilarHigh* queries. *SimilarTop* and *SimilarHigh* both belong to *top-k most-similar* queries mentioned in Section 1.

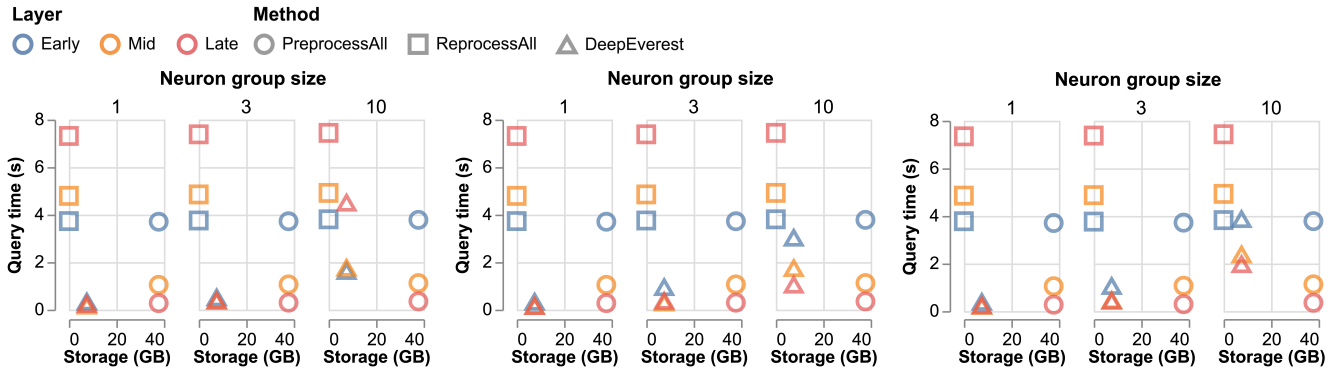
In all experiments, we set k=20, which is a reasonable number of results for a user to inspect after a query. With a smaller k, we expect DeepEverest to achieve larger speedups because it will process fewer inputs and therefore return the results faster, while the query times of baselines will remain similar since they still need to recompute or load all the activations and maintain the query results. With a larger k, the overall speedups could degrade, but DeepEverest can incrementally return the top-k query results, as discussed in Section 6. Therefore, the perceived query time is still significantly improved.

We use $l2$-distance as the distance function. Unless otherwise stated, all numbers reported (e.g., query times) are median values of five queries on random inputs for each query configuration (e.g., query type: *FiringMax*, neuron group size: 3, layer: *late*).

Machine configuration. All experiments are run on an AWS EC2 p2.xlarge instance, which has an Intel Xeon E5-2686 v4 CPU running at 2.3 GHz, with 61 GB of RAM, and an NVIDIA K80 GPU with 12GB of GPU memory. P2 instances are designed for general-purpose GPU applications and ideally suited for machine learning [1], which is neither too cheap nor too expensive. GPU is enabled when running DNNs. AWS EBS gp3 volumes are used for disk storage.

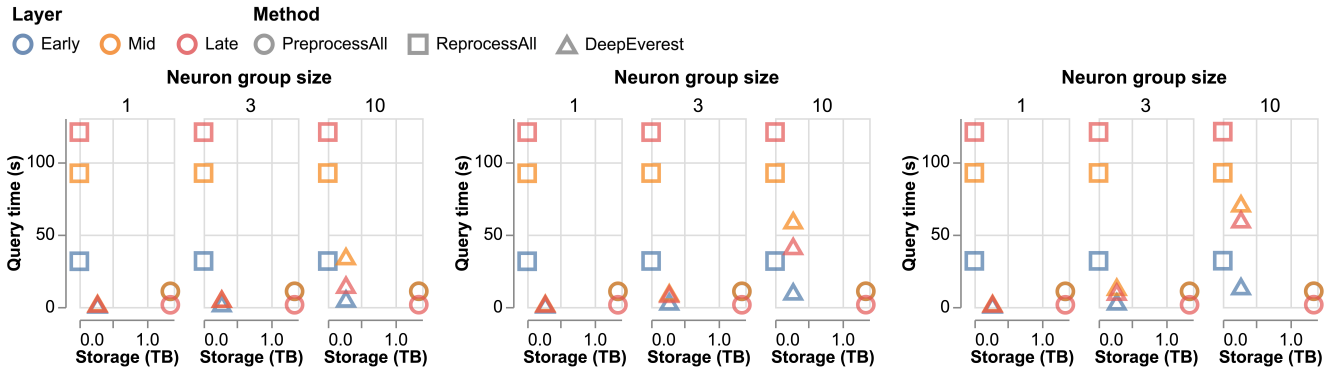
5.2 Fundamental Space-Time Tradeoff

We first evaluate the fundamental trade-off that DeepEverest achieves in terms of storage space and query execution time for individual queries. In this experiment, we first precompute and store the indexes for all layers before executing the benchmark queries. For these experiments, the only optimization DeepEverest uses is the Maximum Activation Index described in Section 4.6.1. DeepEverest has a storage budget of 20% of *PreprocessAll*, and selects $nPartitions$ and $ratio$ using the algorithm described in Section 4.6.2 (for *CIFAR10-VGG16*, $nPartitions = 64, ratio = 0.0047$; for *ImageNet-ResNet50*, $nPartitions = 64, ratio = 0.0076$). We compare DeepEverest against materializing all activation values



(a) *FiringMax* (b) *SimilarTop* (c) *SimilarHigh*

Figure 5: End-to-end query times (per query) and storage sizes on *CIFAR10-VGG16*. *nPartitions* and *ratio* of DeepEverest are selected by our heurstic algorithm given a storage budget of 20% of full materialization.



(a) *FiringMax* (b) *SimilarTop* (c) *SimilarHigh*

Figure 6: End-to-end query times (per query) and storage sizes on *ImageNet-ResNet50*. *nPartitions* and *ratio* of DeepEverest are selected by our heurstic algorithm given a storage budget of 20% of full materialization.

to disk (i.e., *PreprocessAll*) or computing them at query time (i.e., *ReprocessAll*). Figure 5 and Figure 6 show the results.

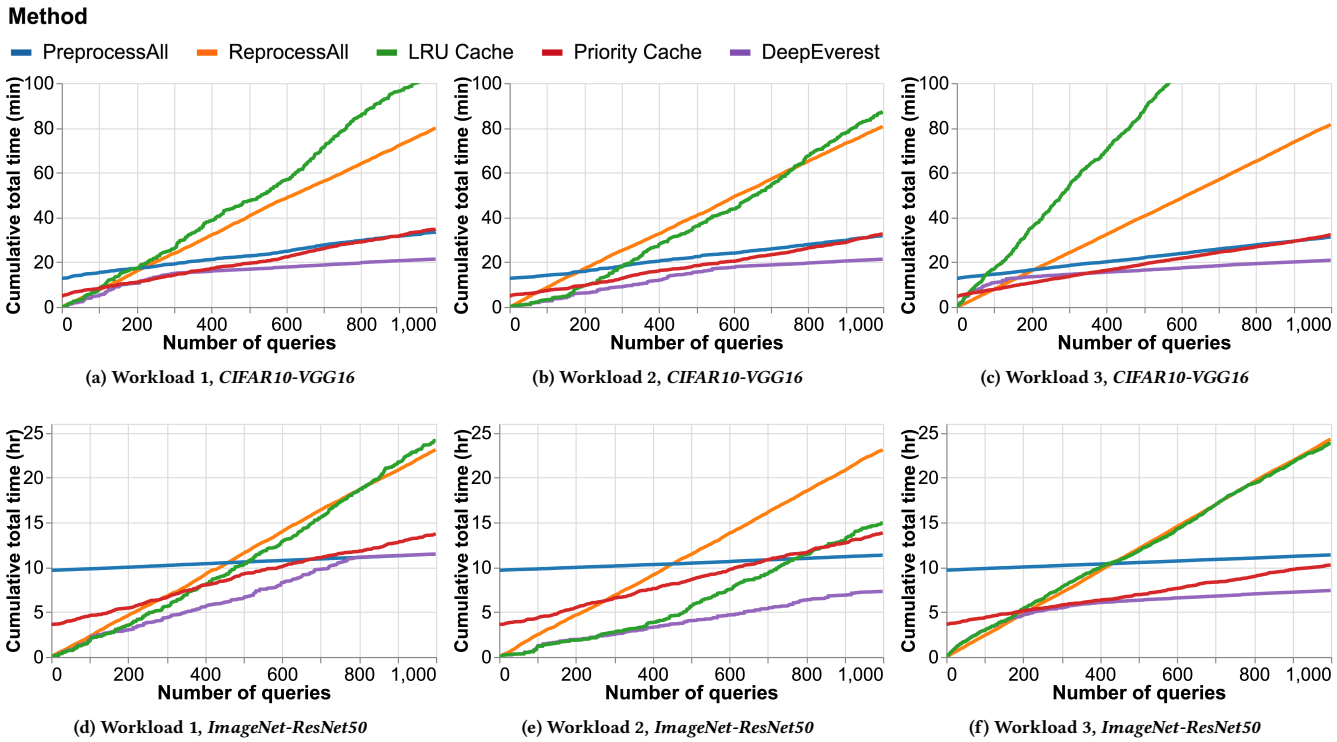
As the figures show, *PreprocessAll* has the highest storage cost (37.8 GB for *CIFAR10-VGG16*, and 1.35 TB for *ImageNet-ResNet50*) since it stores all activations for every input. However, scanning the precomputed activation values generally leads to the fastest query times. The query times of *PreprocessAll* are slower for the early layer of *CIFAR10-VGG16* because it has a large number of neurons, and thus it takes longer to load all the activations. *ReprocessAll* has the lowest storage cost since it does not precompute or store anything ahead of time. Its query times are slow because of the DNN inference on the entire dataset at query time.

DeepEverest achieves the best of both worlds: low storage overhead and fast query times. For *CIFAR10-VGG16*, compared with *ReprocessAll* DeepEverest is 1.65× to 30.9× faster for *FiringMax*, 1.26× to 50.6× faster for *SimilarTop*, and up to 51.7× faster for *SimilarHigh*. For *ImageNet-ResNet50*, compared with *ReprocessAll*, DeepEverest is 2.66× to 62.7× faster for *FiringMax*, 1.55× to 61.3× faster for *SimilarTop*, and 1.30× to 61.1× faster for *SimilarHigh*.

Compared to *PreprocessAll* on both *CIFAR10-VGG16* and *ImageNet-ResNet50*, DeepEverest achieves comparable and sometimes even faster query times for queries that target small and medium-size neuron groups despite using only 20% of *PreprocessAll*'s storage overhead. For queries that target large neuron groups, DeepEverest's query times are slower. We observe this phenomenon again in Figure 10 (discussed in Section 5.4). Due to the curse of dimensionality, there is little difference in the distances between different pairs of inputs. As a result, DeepEverest is not able to reduce the number of inputs run by the DNN at query time as it does for small and medium neuron groups. Table 2 shows the number of inputs run by the DNN at query time to compute the activation values for *SimilarHigh* queries. We find that the number of inputs run by the DNN at query time for queries on larger neuron groups is higher than that of queries on smaller neuron groups.

5.3 Multi-Query Workloads

In this section, we evaluate DeepEverest on multi-query workloads using incremental indexing (see Section 4.5) to avoid start-up overheads. We compare DeepEverest against other on-disk caching techniques. We construct various query workloads to represent possible DNN interpretation patterns. All workloads consist of 1,000 *SimilarHigh* queries that target neuron groups of medium size, which are the most general query type. The



(a) Workload 1, *CIFAR10-VGG16* (b) Workload 2, *CIFAR10-VGG16* (c) Workload 3, *CIFAR10-VGG16*

(d) Workload 1, *ImageNet-ResNet50* (e) Workload 2, *ImageNet-ResNet50* (f) Workload 3, *ImageNet-ResNet50*

Figure 7: Cumulative total time for various multi-query workloads.

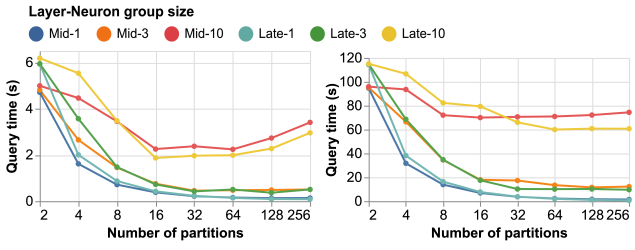
first query of each workload targets a *RandomHigh* neuron group from a randomly selected layer. Each later query has probabilities p_{same} of querying the same layer as the previous query, p_{prev} to query a previously queried layer, and p_{new} to query a layer that has not been queried yet. Workload 1 sets these to p_{same}=0.5, p_{prev}=0.3, p_{new}=0.2. Workload 2 sets these to p_{same}=0.5, p_{prev}=0.4, p_{new}=0.1. Workloads 1 and 2 are intended to simulate the exploration process of users that are likely to initially target layers they are interested in, and gradually explore more layers. Additionally, we construct Workload 3 in which queries are independent of each other; each layer is targeted uniformly at random by each query. This is not a realistic interpretation pattern but is meant to show the worst-case workload for DeepEverest.

We measure the cumulative total time, which includes both preprocessing and query execution, and cumulative storage for each method. DeepEverest is given a storage budget of 20% of full materialization, and $nPartitions$ and $ratio$ are selected by our heuristic algorithm. *LRU Cache* and *Priority Cache* have the same 20% storage budget. The results for cumulative total time are shown in Figure 7. The time to initially compute and store the data on disk is included with the 0-th query for *PreprocessAll* and *Priority Cache*. We report the storage results in the text.

DeepEverest consistently performs the best for Workloads 1 and 2 using less than 20% of the storage of full materialization. We observe that after some number of queries, the cumulative total time of DeepEverest grows more slowly. For example, in the results for *CIFAR10-VGG16* shown in Figure 7a and Figure 7b, it plateaus after around 300 queries for Workload 1 and around 550 queries for Workload 2. This indicates that DeepEverest has built and stored the Neural Partition Index and Maximum Activation Index for all layers in the DNN model. All later queries are much faster because they benefit from these indexes and DeepEverest's Neural Threshold Algorithm. For *ImageNet-ResNet50*, DeepEverest completes building and storing indexes for all layers after around 780 queries for Workload 1 and never completes for Workload 2 as we observe that the storage of DeepEverest is only 11% of full materialization after 1,000 queries. DeepEverest finishes building its indexes after fewer queries in Workload 1 than in Workload 2 since Workload 1 has a higher probability of querying new layers.

While DeepEverest has the fastest query times, its storage also grows more slowly than the baseline approaches (except for *ReprocessAll* which does not have any storage overhead). For both datasets and models, *PreprocessAll* uses full storage after its preprocessing step. Similarly, *Priority Cache* consumes its 20% storage budget after preprocessing. *LRU Cache* consumes its storage budget after around 50 to 200 queries. As discussed above, DeepEverest finishes building the indexes for all layers and consumes its storage budget after around 300 to 500 queries on *CIFAR10-VGG16*, and for *ImageNet-ResNet50* it fills its storage after 780 queries for Workload 1 and never does so for Workload 2.

For Workload 3, which is an unlikely DNN interpretation pattern, the cumulative total time for DeepEverest is slightly worse than the best performing method for the first 200 to 300 queries on both datasets and models because during that time DeepEverest builds indexes for many new layers that have not been queried before. However, DeepEverest performs the best after 400 queries because



(a) *CIFAR10-VGG16* (b) *ImageNet-ResNet50*

Figure 8: Query times of *SimilarHigh* queries when varying *nPartitions*. Note the log scale on the *x*-axis.

Table 2: Number of inputs run by the DNN at query time for *SimilarHigh* queries on *CIFAR10-VGG16*.

Layer-Neuron group size	Number of partitions						
	4	8	16	32	64	128	256
mid-1	3334	1429	667	323	159	79	40
mid-3	5462	2902	1441	736	727	556	409
mid-10	8941	6869	4339	4219	3486	3492	3316
late-1	3334	1429	667	323	159	79	40
late-3	5968	2372	1106	618	682	388	390
late-10	9008	5565	2870	2745	2162	2137	1911

more queries target previously seen layers and benefit from its indexes and query execution.

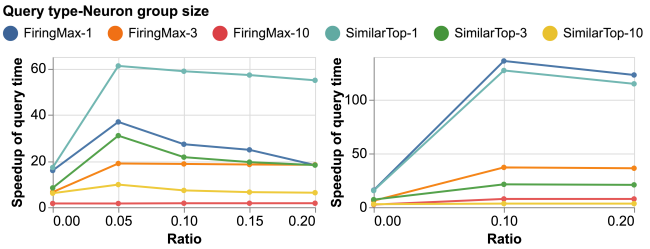
We further observe that users typically pause between queries. DeepEverest can use that time to compute and persist its index to disk, which would yield even better user-perceived query times.

5.4 DeepEverest Tuning

We now study the impact of DeepEverest's tunable parameters and how DeepEverest performs using the configuration selection heuristic described in Section 4.6.2 with different storage budgets.

Impact of Number of Partitions. We first examine how the number of partitions, *nPartitions*, affects the query times. In this experiment, we measure the query times of DeepEverest on *SimilarHigh* queries after building the Neural Partition Index with varying *nPartitions*. The Maximum Activation Index is disabled for this experiment. The results are shown in Figure 8. We also measure the number of inputs on which DeepEverest performs DNN inference during query processing. Table 2 shows the results for *CIFAR10-VGG16*. Similar trends are observed for *ImageNet-ResNet50*.

The query time initially decreases as *nPartitions* increases. This is because when partitions are larger, inputs that do not contribute to the result end up being processed by DeepEverest. Hence, as partitions get smaller, the number of inputs run by the DNN at query time decreases. Then, after *nPartitions* increases past a certain value (64 for *CIFAR10-VGG16* and 128 for *ImageNet-ResNet50*), the query time no longer decreases despite the number of inputs run by the DNN at query time continuing to decrease. Recall that the Neural Threshold Algorithm runs inference on all inputs in a partition as it processes that partition. When *nPartitions* is so large that the partition size is below the optimal *batchSize* for



(a) *CIFAR10-VGG16* (b) *ImageNet-ResNet50*

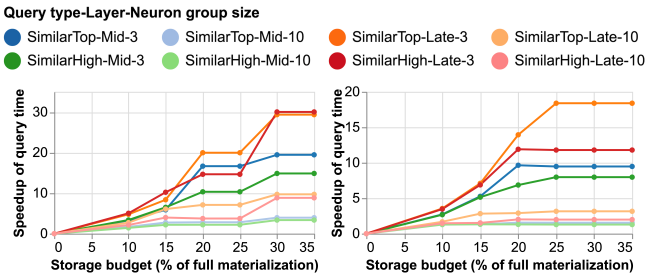
Figure 9: Speedups of query times against *ReprocessAll* when varying *ratio*, evaluated on the late layer for each model with *nPartitions* set to 16.

DNN inference, the parallelization of the GPU is not fully utilized, which causes some queries to slow down. Therefore, a good value of *nPartitions* creates partitions whose sizes are similar to the optimal *batchSize*, e.g., a good *nPartitions* for *CIFAR10-VGG16* is 64.

Effectiveness of Maximum Activation Index. This set of experiments evaluates the effectiveness of the Maximum Activation Index (`maxActidx`). We measure the speedup of query times compared with *ReprocessAll* when varying *ratio*, which determines the fraction of inputs with activation values materialized in `maxActidx`. Recall that when `maxActidx` is non-empty, it becomes the 0-th partition. For this experiment, we set *nPartitions*=16, which is a setting that performs well (see Figure 8). As discussed in Section 4.6.1, this index is designed to accelerate *FiringMax* and *SimilarTop* queries. We measure the speedups of such queries on different sizes of neuron groups.

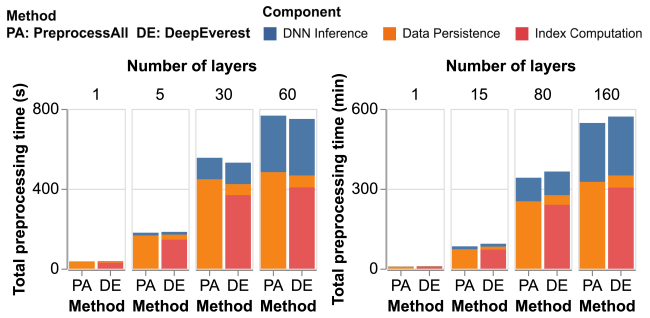
Figure 9 shows the results on *CIFAR10-VGG16* and *ImageNet-ResNet50*. Note that when *ratio*=0, DeepEverest runs without `maxActidx` because it is empty. The speedups of query times are generally much higher when *ratio* is any non-zero value. This is because `maxActidx` enables DeepEverest to return the query results after processing a subset of the inputs from the 0-th partition (i.e., the Maximum Activation Index) for some neurons, rather than processing the entire partition. We also observe that the speedups of query times plateau or drop as *ratio* further increases. This is because loading `maxActidx` from disk takes longer as *ratio* increases. When a small index provides enough information for DeepEverest to find the top-*k* results after processing only some inputs from the 0-th partition, increasing *ratio* degrades the speedups; the additional inputs in the index do not improve the query times, and loading a larger index takes longer. The best value of *ratio* in practice depends on the queries and the distributions of the activations of the neuron group being queried. Empirically, we observe that a small value of *ratio* (e.g., 0.05) is good for *FiringMax* and *SimilarTop* queries on the two datasets and models.

Impact of Storage Budget. In previous sections, we examined the performance of DeepEverest with a storage budget of 20% of full materialization. Here we examine how well DeepEverest performs when the configuration selection algorithm described in Section 4.6.2 has different storage budgets. We measure the speedups of query times compared with *ReprocessAll* for *SimilarTop* and *SimilarHigh* queries that target medium and large neuron



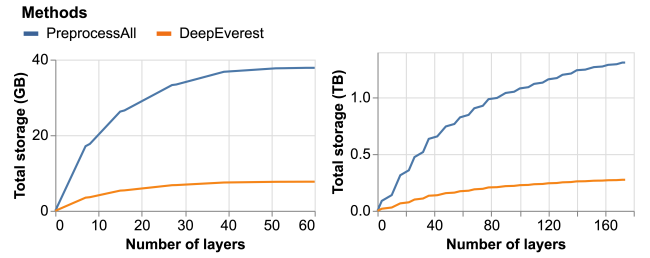
(a) *CIFAR10-VGG16* (b) *ImageNet-ResNet50*

Figure 10: Speedups against *ReprocessAll* by DeepEverest given different storage budget.



(a) *CIFAR10-VGG16* (b) *ImageNet-ResNet50*

Figure 11: Preprocessing times for DeepEverest and *ProprocessAll.*



(a) *CIFAR10-VGG16* (b) *ImageNet-ResNet50*

Figure 12: Preprocessing storage for DeepEverest and *ProprocessAll.*

groups, as shown in Figure 10. We observe that empirically DeepEverest delivers high speedups across different storage budgets, which also suggests that our configuration selection heuristic is robust. With more storage budget, DeepEverest performs better. We also observe that the speedups of queries on medium neuron groups are generally greater than the speedups for queries on large neuron groups due to the curse of dimensionality.

5.5 Preprocessing Costs

In this set of experiments, we evaluate the costs of preprocessing for DeepEverest when given a 20% storage budget of full materialization. We preprocess all the layers for each dataset and model from the first layer to the last layer. Convolutional layers, activation layers, and batch normalization layers are considered separate layers. We measure the cumulative times for each component in preprocessing: DNN inference, data persistence (for *PreprocessAll*, persisting the activations to disk; for DeepEverest, persisting the Neural Partition Index and Maximum Activation Index to disk), and index computation. We force-write the data to disk when measuring the time for data persistence. We also measure the cumulative storage of the two methods as the preprocessing proceeds. Figure 11 and Figure 12 show the preprocessing times and storage of DeepEverest and *PreprocessAll*.

In general, DeepEverest has similar preprocessing times compared with *PreprocessAll*. The time for DeepEverest to build the Neural Partition Index and Maximum Activation Index and persist them to disk is similar to the time for *PreprocessAll* to persist the activations to disk. Comparing the results for the early layers with that of the late layers on both datasets and models, it is obvious that DNN inference takes longer for the late layers than for the early layers. We also observe that data persistence and index computation for the early layers takes longer than the late layers because the sizes of the early layers are usually greater than that of the late layers, as indicated in Figure 12.

Considering these results along with the query performance results shown in Section 5.2, DeepEverest can achieve comparable and sometimes better query times than *PreprocessAll*, with only 20% of its storage overhead and similar preprocessing times.

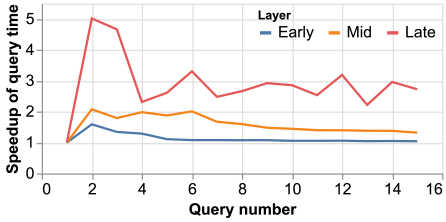


Figure 13: Speedups of query times by DeepEverest with inter-query acceleration against DeepEverest without it. The memory budget for inter-query acceleration is 128 MB.

5.6 Effectiveness of Inter-Query Acceleration

Finally, we evaluate the effectiveness of inter-query acceleration for sequences of related queries. We randomly select five inputs and construct sequences of related queries for each input on various layers. The first query is a *SimilarHigh* query that targets a neuron group of size 1. For every subsequent query, we add a randomly selected *RandomHigh* neuron to the previous query's neuron group. Thus, the n-th query targets a neuron group of size n. We focus only on adding neurons since it is straightforward for queries that remove neurons from their neuron groups to retrieve activations from the in-memory cache. We measure the speedups for query times of DeepEverest with inter-query acceleration against DeepEverest without it for each query.

Figure 13 shows the median of the speedups for each query on *CIFAR10-VGG16* given a cache budget of 128 MB. We observe that even with this small budget, inter-query acceleration consistently improves DeepEverest's query times across different layers. Not shown in the paper, we also experiment with different budgets. We find that inter-query acceleration consistently speeds up related queries and larger budgets generally lead to larger speedups. In

Figure 13, the speedups for the first query are around 1× since the in-memory cache is initially empty. On later queries when the cache is populated, DeepEverest with inter-query acceleration achieves speedups of 2.22× to 5.02× on the late layer, 1.32× to 2.08× on the mid layer, and 1.05× to 1.59× on the early layer. The speedup for the early layer is smaller because this layer is larger, and hence the in-memory cache can hold fewer inputs' activations of the full layer. We observe larger speedups for the early layer with a larger memory cache budget.

6 DISCUSSION

This section discusses some possible optimizations and extensions for DeepEverest.

Incrementally Returning Query Results. The Neural Threshold Algorithm runs until it has found k inputs whose distances to the sample s are at most the threshold value, t. However, the Neural Threshold Algorithm may be certain that some inputs are part of the top-k set before it has found the complete set. For queries where k>1, after each round of the algorithm, DeepEverest returns inputs in $Y \subseteq U$, where for all $y \in Y$, $dist(s, y, g) \leq t$, and continues running to find the rest of the $k - |Y|$ results. DeepEverest's optimizations enable it to incrementally return the top-k results quickly, and therefore reduces the time required to return the first part of the answer to the user.

Approximation. Modifying DeepEverest to give approximate results is straightforward. Following the definition of the θ-approximation in Fagin's paper [12], a θ-approximation (let $0 < \theta < 1$ be given) to the top-k answers is a collection of k inputs, U, (and their distances to the sample input) such that for each $y \in U$ and each $z \in D \setminus U$, $\theta * dist(s, y, g) \leq dist(s, z, g)$. Let t be the threshold value from eq. (3). DeepEverest can find a θ-approximation to the top-k answers by modifying the termination condition in eq. (4) to be,

$$\max_{(x, dist(s,x,g)) \in top} \{dist(s, x, g)\} \leq t/\theta \tag{5}$$

Early Stopping. DeepEverest can be further modified into an interactive process in which it can show the user the current top-k results with a guarantee about the degree of approximation to the correct top-k results. Based on this guarantee, the user can decide whether they would like to stop the process at any time. Let b be the largest distance to the sample input from the current top-k results, let t be the current threshold value, and let $\theta = t/b$. If the algorithm is stopped early, we have $0 < \theta < 1$ because $b > t$. Therefore, the current top-k results are then a θ-approximation to the correct top-k answers. Hence, the user can be shown the current top-k results and the number θ, with a guarantee that they are being shown a θ-approximation.

7 CONCLUSION

We presented DeepEverest, a system that accelerates top-k queries for DNN interpretation. DeepEverest, with various optimizations, reduces the number of activations computed at query time with low storage overhead, while guaranteeing correct results. With less than 20% of the storage of full materialization, DeepEverest accelerates individual queries by up to 62× and consistently outperforms other methods over various multi-query workloads.

REFERENCES

[1] [n.d.]. Amazon EC2 - P2 Instances. https://aws.amazon.com/ec2/instance-types/p2/. Accessed: 2021-03-30.

[2] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. 2007. Best Position Algorithms for Top-K Queries. In *International Conference on Very Large Data Bases (VLDB)*. ACM, 495–506.

[3] Saleema Amershi, Max Chickering, Steven M Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. ModelTracker: Redesigning Performance Analysis Tools for Machine Learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 337–346.

[4] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. IEEE, 459–468.

[5] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Christian Theobalt, and Gerhard Weikum. 2006. IO-Top-K: Index-Access Optimized Top-K Query Processing. (2006).

[6] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. Network Dissection: Quantifying Interpretability of Deep Visual Representations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6541–6549.

[7] David Bau, Jun-Yan Zhu, Hendrik Strobelt, Agata Lapedriza, Bolei Zhou, and Antonio Torralba. 2020. Understanding the Role of Individual Units in a Deep Neural Network. *Proceedings of the National Academy of Sciences* (2020).

[8] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[9] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. 2006. VisTrails: Visualization Meets Data Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. 745–747.

[10] Rich Caruana, Hooshang Kangarloo, John David Dionisio, Usha Sinha, and David Johnson. 1999. Case-Based Explanation of Non-Case-Based Learning Methods. In *Proceedings of the AMIA Symposium*. American Medical Informatics Association, 212.

[11] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. 253–262.

[12] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal Aggregation Algorithms for Middleware. *J. Comput. System Sci.* 66, 4 (2003), 614–656.

[13] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 580–587.

[14] Abel Gonzalez-Garcia, Davide Modolo, and Vittorio Ferrari. 2018. Do Semantic Parts Emerge in Convolutional Neural Networks? *International Journal of Computer Vision* 126, 5 (2018), 476–494.

[15] Ian J. Goodfellow, Quoc V. Le, Andrew M. Saxe, Honglak Lee, and Andrew Y. Ng. 2009. Measuring Invariances in Deep Networks. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems* (Vancouver, British Columbia, Canada) *(NIPS'09)*. Curran Associates Inc., Red Hook, NY, USA, 646–654.

[16] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Rec.* 14, 2 (June 1984), 47–57. https://doi.org/10.1145/971697.602266

[17] Xixian Han, Jianzhong Li, and Hong Gao. 2015. Efficient Top-K Retrieval on Massive Data. *IEEE Transactions on Knowledge and Data Engineering* 27, 10 (2015), 2687–2699.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[19] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. 2002. Joining Ranked Inputs in Practice. In *Proceedings of the 28th International Conference on Very Large Data Bases*. Elsevier, 950–961.

[20] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2010), 117–128.

[21] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* (2019).

[22] Minsuk Kahng, Pierre Y Andrews, Aditya Kalro, and Duen Horng Polo Chau. 2017. ActiVis: Visual Exploration of Industry-Scale Deep Neural Network Models. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2017), 88–97.

[23] Minsuk Kahng, Dezhi Fang, and Duen Horng Chau. 2016. Visual Exploration of Machine Learning Results Using Data Cube Analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 1–6.

[24] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. 2015. Visualizing and Understanding Recurrent Networks. *arXiv preprint arXiv:1506.02078* (2015).

[25] Josua Krause, Adam Perer, and Kenney Ng. 2016. Interacting with Predictions: Visual Inspection of Black-Box Machine Learning Models. In *Proceedings of the*

2016 CHI Conference on Human Factors in Computing Systems. 5686–5697.

[26] Sanjay Krishnan and Eugene Wu. 2017. PALM: Machine Learning Explanations For Iterative Debugging. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*. 1–6.

[27] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[28] Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. 2015. Principles of Explanatory Debugging to Personalize Interactive Machine Learning. In *Proceedings of the 20th International Conference on Intelligent User Interfaces*. 126–137.

[29] Zachary C Lipton. 2018. The Mythos of Model Interpretability. *Queue* 16, 3 (2018), 31–57.

[30] Mengchen Liu, Jiaxin Shi, Zhen Li, Chongxuan Li, Jun Zhu, and Shixia Liu. 2016. Towards Better Analysis of Deep Convolutional Neural Networks. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2016), 91–100.

[31] Shuying Liu and Weihong Deng. 2015. Very Deep Convolutional Neural Network Based Image Classification Using Small Training Sample Size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*. IEEE, 730–734.

[32] Ting Liu, Andrew W Moore, and Alexander Gray. 2006. New Algorithms for Efficient High-Dimensional Nonparametric Classification. *Journal of Machine Learning Research* 7, Jun (2006), 1135–1158.

[33] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. 2006. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience* 18, 10 (2006), 1039–1065.

[34] Minghuang Ma, Haoqi Fan, and Kris M Kitani. 2016. Going Deeper into First-Person Activity Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1894–1903.

[35] Parmita Mehta, Stephen Portillo, Magdalena Balazinska, and Andrew J. Connolly. 2020. Toward Sampling for Deep Learning Model Diagnosis. In *36th IEEE International Conference on Data Engineering, ICDE*. IEEE, 1910–1913.

[36] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. 2017. ModelHub: Deep Learning Lifecycle Management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1393–1394.

[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems* 26 (2013), 3111–3119.

[38] Vinod Nair and Geoffrey E Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *ICML*.

[39] Anh Nguyen, Alexey Dosovitskiy, Jason Yosinski, Thomas Brox, and Jeff Clune. 2016. Synthesizing the Preferred Inputs for Neurons in Neural Networks via Deep Generator Networks. In *Advances in Neural Information Processing Systems*. 3387–3395.

[40] NVIDIA. [n.d.]. NVIDIA Data Center Deep Learning Product Performance. https://developer.nvidia.com/deep-learning-performance-training-inference.

[41] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. 2018. The Building Blocks of Interpretability. *Distill* 3, 3 (2018), e10.

[42] Stephen M Omohundro. 1989. *Five Balltree Construction Algorithms*. International Computer Science Institute Berkeley.

[43] HweeHwa Pang, Xuhua Ding, and Baihua Zheng. 2010. Efficient Processing of Exact Top-K Queries over Disk-Resident Sorted Lists. *The VLDB Journal* 19, 3 (2010), 437–456.

[44] Nicolas Papernot and Patrick McDaniel. 2018. Deep k-Nearest Neighbors: Towards Confident, Interpretable and Robust Deep Learning. *arXiv preprint arXiv:1803.04765* (2018).

[45] 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*. 1135–1144.

[46] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[47] Thibault Sellam, Kevin Lin, Ian Huang, Michelle Yang, Carl Vondrick, and Eugene Wu. 2019. DeepBase: Deep Inspection of Neural Networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1117–1134.

[48] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[49] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 535–546. https://doi.org/10.1109/ICDE.2017.109

[50] Rupesh Kumar Srivastava, Jonathan Masci, Sohrob Kazerounian, Faustino Gomez, and Jürgen Schmidhuber. 2013. Compete to Compute. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) *(NIPS'13)*. Curran Associates Inc., Red Hook, NY, USA, 2310–2318.

[51] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing Properties of Neural Networks. In *International Conference on Learning Representations*. http://arxiv.org/abs/1312.6199

[52] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. 2004. Top-K Query Evaluation with Probabilistic Guarantees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. 648–659.

[53] Manasi Vartak, Joana M F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *Proceedings of the 2018 International Conference on Management of Data*. 1285–1300.

[54] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. MODELDB: A System for Machine Learning Model Management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 1–3.

[55] Eric Wallace, Shi Feng, and Jordan Boyd-Graber. 2018. Interpreting Neural Networks with Nearest Neighbors. *arXiv preprint arXiv:1809.02847* (2018).

[56] Yair Weiss, Antonio Torralba, and Rob Fergus. 2009. Spectral Hashing. In *Advances in Neural Information Processing Systems*, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou (Eds.), Vol. 21. Curran Associates, Inc.

[57] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 446–460. https://doi.org/10.14778/3297753.3297763

[58] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. 2015. Understanding Neural Networks Through Deep Visualization. *arXiv preprint arXiv:1506.06579* (2015).

[59] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I (Lecture Notes in Computer Science)*, David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.), Vol. 8689. Springer, 818–833. https://doi.org/10.1007/978-3-319-10590-1_53

[60] Shile Zhang, Chao Sun, and Zhenying He. 2016. Listmerge: Accelerating Top-K Aggregation Queries over Large Number of Lists. In *International Conference on Database Systems for Advanced Applications*. Springer, 67–81.

[61] Bolei Zhou, David Bau, Aude Oliva, and Antonio Torralba. 2018. Interpreting Deep Visual Representations via Network Dissection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41, 9 (2018), 2131–2145.

[62] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. 2014. Object Detectors Emerge in Deep Scene CNNs. *arXiv preprint arXiv:1412.6856* (2014).

[63] Bolei Zhou, Yiyou Sun, David Bau, and Antonio Torralba. 2018. Revisiting the Importance of Individual Units in CNNs via Ablation. *arXiv preprint arXiv:1806.02891* (2018).