DeepBase 2.0 : Scalable Deep Neural Inspection

Yiru Chen
yiru.chen@columbia.edu
Columbia University

Ziao Wang
zw2498@columbia.edu
Columbia University

Yiliang Shi
ys3130@columbia.edu
Columbia University

ABSTRACT

Deep learning is becoming increasingly crucial in both research and industrial fields. However, neural networks are a black box to most, causing models to be difficult to debug and refine. One common approach to opening the black box is by examining the behaviour of portions of the model with regards to hypothesis of possible explanations. This is known as deep neural inspection (DNI). DeepBase is a first attempt at a system that performs general DNI as opposed to implementations of specific techniques. However, as a first prototype, DeepBase operates sequentially, which is not scalable as the number of units, hypothesis and models increase.

This paper describes DeepBase 2.0, an extension of DeepBase 1.0. The main contribution of DeepBase 2.0 is its distributed architecture which supports neural network inspection on a large scale. Additional contributions include an extended SQL interface to the system, allowing users to define a neural network analysis task with a new INSPECT clause to analyze models and hypotheses simultaneously. DeepBase 2.0 is built on Dask. From our experiment, we demonstrate that DeepBase 2.0 achieves 30x faster than DeepBase 1.0. Our system also demonstrates linear scalability.

ACM Reference Format:
Yiru Chen, Ziao Wang, and Yiliang Shi. 2019. DeepBase 2.0 : Scalable Deep Neural Inspection. In *Proceedings of 6113 Advanced Database System (W6113 Advanced Database System).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Previously, we implemented DeepBase 1.0 [9] , a single-threaded deep neural inspection system. In this paper, our work mainly focuses on significantly enhancing scalability and user experience.

Below are some terms appear in this paper:

Behavior. The behaviors of models we want to inspect. In our system, it may refer to the activation of the hidden units, the gradient of hidden units or the hidden unit group output.

Hypothesis. The human-made assumptions about how these hidden units act.

Our DeepBase 2.0 has the following properties:

Domain Specific. Like other model management systems such as ModelDB and MLBase, DeepBase is a domain-specific system. It can be classified as a system for neural network explanation. Ideally, DeepBase 2.0 should support all types of neural networks, which includes feedforward networks, CNNs and RNNs. The commonality between the networks is that regardless of architecture, the output of a single unit is the same. As our system is domain specific, we are able to tightly integrate machine learning specific libraries such as Keras and PyTorch into our implementation.

Scalability. The previous version of DeepBase only allows DNI (short for Deep Neural Inspection)to run on a single thread, which results in exponential runtimes with respect to the number of variables. To create scalability, we need to identify parallelizable points in program execution flow and resource allocation. For the first, SQL syntax enables us to create a well defined logical plan of operators to execute, with clear parallelizable points. We are then able to estimate resource usage and decide on physical operator order. Physical operators execute using the Dask distributed system, which allows us to run computations an arbitrary number of machines. Together with Kubernetes and Helm, which can be used to start an arbitrary number of clusters (machines) and the Google Cloud Storage API for data access, we remove scalability limits for DeepBase.

Programmability. Python programs are very flexible, but requires users to write repeated complex analysis due to data-structure limitations that are separate from logic. Common data storage formats like JSON requires complex parsing to group, filter and aggregate, which can be very inefficient. SQL on the other hand is designed and optimized for the above operations. We thus extend SQL and provide a GUI interface for users to easily update data and perform complex analysis with DeepBase.

Extensibility. Researches are constantly coming up with new variations of DNI. As such, we support user defined functions to generate hypothesis, extract behaviour and compute metrics. These can be passed into the API and called by the system, allowing for flexible extensions to the system, though estimations may be inaccurate.

In the reminder, we outline our main contributions:

- We describe several hypothetical DNI use cases, and the corresponding extended-SQL statement.
- Designed and implemented an end to end distributed system with a user input layer, logical plan, cost estimation, optimization and distributed execution.
- The evaluate scalability, we implemented the Netdissect experiment and test for scale up and scale out.
- Compares different implementations of DeepBase

W6113 Advanced Database System, Mar 22–May 05, 2019, Columbia University, NY

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

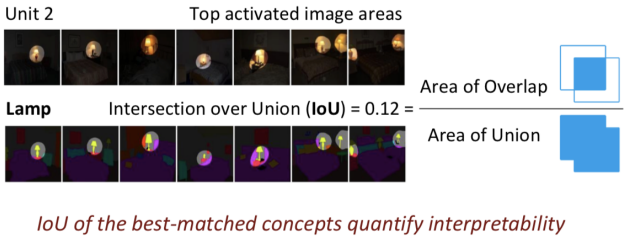

Figure 1: IoU score interpretation [2]

2 BACKGROUND

In this section, we will give overviews of former studies that DeepBase depends on.

2.1 Deep Neural Inspection

Deep learning is efficient for solving problems in many fields like image and natural language processing, yet people still lack knowledge of what actually happens inside models. Figuring out the internals of neural networks is a new challenge in deep learning research. With units and data points in the million, approaches such as visualization can be painful.

Previous works have tried to automate those analyses. For example, [12] measures whether units in a CNN detect a specific object by defining specific pixels and computing the IoU (intersection over Union) score. This connects high-level behaviors with some types of measurement.

Such works could be summarized as Deep Neural Inspection(DNI). DNI is a common class that contains divergent analyses for understanding the behaviors of hidden units under trained neural network models.

DNI has a high conceptual similarity to query processing in databases. Operators deals with large amounts of data, materialization of intermediate results is expansive and concurrency is highly desirable. As such, we would like to take advantage of these synergies to implement database and distributed systems techniques in a new domain to enable much faster queries on neural network models. More specifically, we would like to implement the concept of operator cost estimation. This would enable us to create better distributed query plans and know ahead of time how long a likely plan would take to execute.

2.2 Netdissection

NetDissect [2] is an instance of DNI. At the core of the analysis, the technique involves computing the IoU score between thresholded activations of a neural network unit over a given dataset, against a binary mask of objects, where the object is a hypothesis of the purpose of a unit (Figure 1). With this score, it is then possible to perform analysis such as finding the number of units that supports a certain hypothesis, finding hypothesis with strong unit activations and so on so forth.

2.3 DeepBase 1.0 Overview

Running a single DNI analysis could be complex as well as long-term because scientists need to write huge amount of code in order to connect high-level behaviors with the measurement of units and layers. And due to the size of models, it requires optimal data management.

DeepBase 1.0 is a DNI system specifies the output as DNI analysis. It allows users to define customized hypothesis and metrics in order to meet the divergent demands for inspection.

There has already been researches based on DeepBase. By writing only several lines of Python code, users could be able to answer series of questions like the time of learning happened during training, and whether hidden units have some types of behaviors.

However, Python code becomes unfriendly when users want to perform more complex analyses, because users have to know the workflow of different APIs, thus the process becomes painful again. For example, below is the usage for DeepBase 1.0:

```
models = [model_1, model_2, ...]
data = [dataset_1, dataset_2, ...]
hypothesis = [hyp_1, hyp_2, ...]
units = [(1, 1), (2, '*')]
scores = [metrics]
inspect(models, data, hypothesis, units, scores)
```

User has to specify every detail for the inspection, which would require user knows the model very well. In DeepBase 2.0, however, user could write a simple query like this:

```
SELECT U.uid, S.score
INSPECT U.uid USING metrics ON D.dataset AS S
FROM Units U, Hypothesis H, Models M
WHERE (U.lid = 1 AND U.uid = 1) OR U.lid = 2
GROUP BY U.lid, M.mid
```

By using this SQL-like query, user doesn't have to specify the layers or hypothesis unless he wants.

2.4 Google Storage

Normally, dataset and models are more than one gigabyte, and it is too expensive to store them on local disk. In DeepBase, we use Google Storage to store them because of its following properties:

- Cheapness: Even though data access has a high frequency, the cost remains low.
- Low latency: It takes less than 100 ms to grab the first byte of data.
- Ease of use: Google Storage provides Python API to call, which is easy to be integrated into our system.
- Durability: The storage is persistent and doesn't require the running of server.

2.5 Dask

Dask is a library for distributed computing in Python. It provides interfaces for big data collections like numpy, pandas as well as scheduling for dynamic tasks. In DeepBase we mainly use Dask for scheduling.

Dask provides user with default scheduler and customized distributed scheduler. In DeepBase we choose customized distributed scheduler because it handles data locality with more sophistication, and so can be more efficient than the multiprocessing scheduler on workloads that require multiple processes.

2.6 Kubernetes

Kubernetes is a container orchestration system for creating and managing large clusters of machines. The abstraction for each machine is a pod, which can contain multiple containers, while services manages sets of pods. This is convenient for scaling and testing Dask as Dask provides out of the box methods to setup schedulers and workers on Kubernetes clusters. All our experiments can run by simply submitting Dask computations to the scheduler service.

3 USE CASES

Eventually, DeepBase could be able to meet divergent demands of neural network explanation for different models. In this section, we will provide several use cases that DeepBase could work out, and give a SQL-like declarative language code to each one of them.

3.1 Saliency Map Visualization

In computer vision, saliency map is used to transform an real life image to a more meaningful image to analyze. The problem[10] is that given an image, a class, we would like to extract each hidden unit's saliency map and directly return the matrix to the visualization system.

To achieve this functionality, users can write the SQL-like query as follows:

```
SELECT S.content
INSPECT U.uid USING saliency() ON D.data AS S
FROM models M, units U, inputs D
WHERE M.mid = U.mid
```

By running this query, DeepBase would extract the saliency map of hidden unit group and return to users.

In FROM clause, "models", "units", and "inputs" are defined as *system tables*, which are populated by DeepBase based on user's upload. The schema of each system table and the output of inspection will be discussed in following sections.

Beside SQL clauses "SELECT", "FROM" and "WHERE", an additional "INSPECT" clause is used to generate the records from the inspection of neural network. In this clause, the user uses an UDF extraction called "saliency" to inspect the model, and therefore could inspect each unit in "units" table.

3.2 Cross Model Comparison

Another common problem in similarity is to calculate the similarity across models with different initialization and even different architectures[7]. DeepBase could take advantages of previous works such as CCA[4] and compute the average correlation coefficient of each model.

```
SELECT M1.mid, M2.mid, AVG(S.group_score)
INSPECT U1.uid, U2.uid USING CCA()
                       ON D.datasets.data AS S
FROM models M1, models M2,
                units U1, units U2
WHERE M1.mid < M2.mid AND U1.mid = M1.mid
                      AND U2.mid = M1.mid
GROUP BY M1.mid, M2.mid
```

In this "INSPECT" clause, CCA represents a metric for the evaluation of models. Again, we will give the definition in details in following sections.

3.3 Netdissect

DeepBase is also able to conduct cutting-edge experiments. Below is the query for running Netdissect on DeepBase.

```
SELECT S.uid, S.hid, S.unit_score
INSPECT U.uid, H.hid USING IoUIndex(error=0.001)
                     ON broden AS S
FROM models M, hypothesis H, units U
WHERE M.mid = 1 AND U.lid = 47
                AND U.uid = M.mid
GROUP BY U.uid, H.hid
```

In the query above, we inspect units in model 1's 47th layer. Notice that "hypothesis" is another system table that contains hypothesis functions, which are normally UDFs (User Defined Functions). We will give the exact definition in following sections.

3.4 Post Analysis

DeepBase supports post analysis as well. Users could use the result of inspection as a temporary table and then do more things with it. The semantics would be same as creating a view in SQL.

Take the previous Netdissect query as an example, selected units might have different scores in the result. To query the unit that is most likely to detect the concept, the user could write query as following:

```
CREATE VIEW T(uid, hid, unit_score) AS
SELECT S.uid, S.hid, S.unit_score
INSPECT U.uid, H.hid USING IoUIndex(error=0.001)
                     ON broden AS S
FROM models M, hypothesis H, units U
WHERE M.mid = 1 AND U.lid = 47
                AND U.uid = M.mid
GROUP BY U.uid, H.hid

SELECT T.mid, T.hid, T.uid, T.unit_score
FROM T, (
    SELECT temp.mid, temp.hid, MAX(temp.unit_score)
                                    as max_unit_score
    FROM T as temp
    GROUP BY temp.mid, temp.hid)
WHERE T.mid = temp.mid AND T.hid = temp.hid
                       AND T.unit_score = max_unit_score
```

By taking advantages of view that inspects the scores for each unit, user could then get the unit with the maximum score, which means that unit is most likely to detect the concept of the hypothesis.

4 PROGRAMMING INTERFACE

The commonality amongst the many neural network interpretation papers is the extraction and comparison of models against the 'interpretation'. We thus define a set of system tables on which users can query and update based on their interpretation need.

4.1 Types

In addition to standard types (INT, FLOATS, STRING), we would ideally support two extra types of objects, DIMENSION, and ARRAY. DIMENSION objects tuples of arbitrary length, and supports the equal operator. A value of -1 in the tuple indicates variable dimension. Equal returns true if all non variable dimensions are equal. The Array object contains objects of arbitrary dimension, is the default representation of data. Our system unfortunately do not support type checking currently, which we leave for future work.

4.2 Data Model

Our System defines a set of system tables below. Each contains a a number of default columns. Users can add columns to the schemas whose values can be used in filter operations. We operate on the assumption that all data exists in the system. The user is responsible for population the systems table with the initial location of data.

Models Table

- mid (INT): Primary key
- model_type (STRING): One of <KERAS>, <TENSORFLOW>, <PYTORCH>
- path (String): Location of state file

The models table is populated by the user using INSERT into MODELS(Mid, Framework, Model_definition). Upon insertion, our system will populate the Layers and Units table.

Units Table

- uid (INT): Unique identifier all units in all models
- lid (INT FOREIGN KEY REFERENCES Layers): Information of which layer the unit belongs to
- mid (INT FOREIGN KEY REFERENCES Models): Information of which model the unit belongs to

The Units table is populated when processing the layers table by the system.

DataSets Table

- name (STRING): Primary Key
- Data_Type (STRING)
- size (INT):

Like the Models table, users are required to update the system table with their dataset. All columns must be provided. Users may add additional columns to the table at any time.

Hypothesis Table

- Hid (INT): Primary key
- Hname (STRING): Hypothesis name
- dependencies (STRING): List of libraries that must be imported
- function (STRING): access path to function.
- input_type (STRING): Defines the type of input
- return_type (STRING): Defines the type of returned value
- input_dimension (DIMENSION): Dimension of input object
- output_dimension (STRING): A function that outputs a Dimension object.

The hypothesis table is populated when users registers user defined function to the system. Apart from the Hid which is assigned when a hypothesis function is registered, all other columns in the table are properties that describes the function and must be filled out by the user when registering the function.

4.3 Extractor

Extractors are functions that take as input data and a list of uids and returns an object of type array. By default the system extracts the activations of each unit. User defined Extractors must define properties: Name, Output_Dimension, where Output_Dimension is a function that returns a Dimension object after receiving a Dimension object as input.

4.4 Metric

Metric is a type of interface which allows users to define custom extractors. From user's perspective, each metric is a function with two default parameters error and confidence.

4.5 Error/Confidence Bar

A common method in big data processing is AQP (approximate query processing). To achieve a faster execution of DNI, users could also specify an error/confidence rate in their query like in BlinkDB[1]. Different from BlinkDB, the bar in our semantics is not a clause. Instead, users could simply pass the error/confidence rate as a parameter of metric. That is to change *corr()* to *corr(error=0.001, confidence=0.9)* in the above query.

4.6 Inspect API

The inspect table is defined against system tables with the following semantics:

We add the INSPECT clause to the to the SQL syntax. INSPECT accepts arguments in the following format:

```
INSPECT [<U.ID>],  [<H.ID>]
ON <Metric>, [<Metric>], [<Metric>]
USING <DATASET.IDt>
[FROM tables]
[WHERE  condition]
[GROUP BY <U.ID>, <H.ID>]
```

Inspect returns a Table view containing all expressions in the Group By clause, in addition to the output of Metric. As the outputs of Metrics is a table, the inspect clause automatically flattens the values. Generally, the schema is as following:

- uid (INT): Each units' id in Units table
- hid (INT): Hypothesis id
- group_score (REAL): Result of inspection, if applicable
- unit_score (REAL): Result of inspection, if applicable

4.7 User Interface

Our demo frontend runs on a cloud server. By accessing the ip address, user could upload models, datasets as well as hypothesis. After that, user can run a query. Currently, we allow users to run inspection over all the models/datasets in our storage.

5 ARCHITECTURE

Figure 3 shows the general architecture of DeepBase, which consists of a master and several worker nodes. A user issues requests using the DeepBase declarative task language to the DeepBase master. At the meantime, the user uploads a set of models and datasets to the system. The system parses the request into a logical plan, which describes the most general workflow to perform the inspection task.

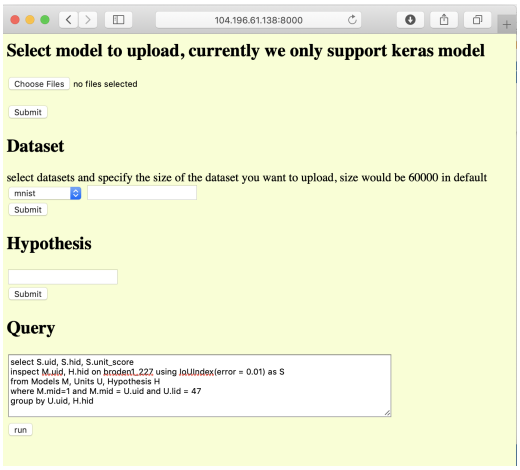

Figure 2: User Interface

The system uses the google file system to store the user-uploaded data and populates the system table with the information of datasets and models.

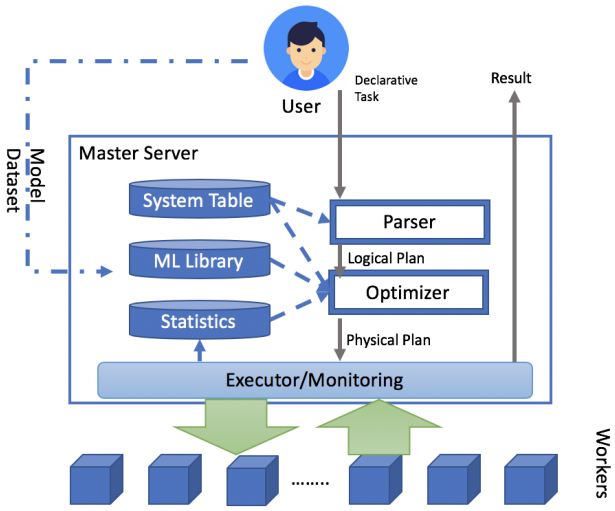

Figure 3: System Architecture

The logical plan search space consists of the combinations of task placements, the beginning time of different tasks and algorithms choice problems, and is too vast to be explored entirely. Part of the logical plan consists of select, from, where operators which are the same as a traditional database management system. Rule-based optimizations are borrowed to optimize this part, like predicate pushdown and statistical estimation. The inspect clause includes the execution of neural network inspection. It calls the functions to extract the hidden unit output and compute the corresponding hypothesis if it is required. Considering the resource distribution, task replicates, data sharing communication cost and data locality, this clause is much harder than traditional database management system. Therefore, an optimizer tries to model it as a integer linear programming problem to find a placement of task that is testable at a reasonable speed. Finally, the physical plan is passed to the executor, which executes the operators on the Dask framework in a parallel manner.

The optimizer takes the physical environment, task runtime estimation, task runtime requirement, tasking dependence as input. For the same logical operator, there would be different kinds of physical operators to choose from. The system will choose the one with the lowest estimated cost while meeting the user's requirement at the same time. From example, there are metric with early stopping, approximated metric to implement the metric function. Through specifying the constraints, acquiring the statistics, system table and modeling the workflow, the optimizer solves the linear programming problem and reaches a final physical plan to place the task with an optimal order on different machines. There are still some further optimization strategies to deploy - bit packing and model merging. We choose rule-based optimization to achieve these(see section 6).

Another critical aspect of DeepBase is its extensibility to novel hypothesis and extraction functions. We envision users constantly add new extract function and hypothesis to the system, with the requirement that developers implement new algorithms with the same format of output schema required in the DeepBase and describe their properties using a particular contract. The contract specifies the output data type of algorithm (e.g., binary), the algorithm's parameters, run-time estimation and possible run-time optimizations (e.g., replicate vs. approximation, see section 6). The easy extensibility of DeepBase will simultaneously make it an attractive platform for ML experts to debug the machine learning systems and allow users to explore the possibility of neural network explanation creatively.

6 PARSER

In DeepBase, the parser takes SQL-like query as input, and outputs a logical plan.

6.1 Parse A Query

The parser is implemented based on moz-sql-parser, an open-source parser designed by Mozilla. Normally, a SQL parser would parse a SQL query into a tree, whereas each node represents an relational algebra operation. For example, *SELECT* would be mapped into a projection. By doing this, it is easy to achieve optimization towards the query itself like predicate push-down. However, moz-sql-parser will parse the query into tokens. That is, for each keyword in SQL, the parser will generate a value dictionary for it, and the system could efficiently access each token's value. Below is an example of parsed tokens for the query in $3.3.

```
{
"select": ["S.uid", "S.hid", "S.unit_score"],
"inspect": ["U.uid", "H.hid",
            {"value": "inspect", "name": "S"}],
"on": "broden",
"using": {
            "corr": {
                        "eq": ["error", 0.001]
```

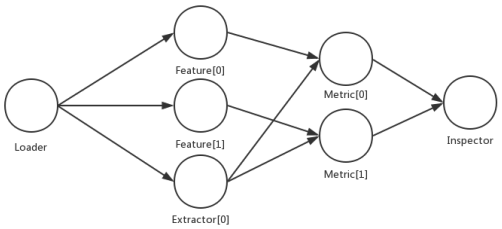

Figure 4: Logical Plan Graph

```
                    }
        },
"from": [{"value": "Models", "name": "M"},
            {"value": "Units", "name": "U"},
            {"value": "Hypothesis", "name": "H"}],
"where": {
            "eq": ["U.mid", "M.mid"],
            "eq": ["U.lid", 47]
        },
"groupby": {
            "value": ["U.uid", "H.hid"]
        },
"having": {
            "gt": ["S.unit_score", "x"]
        },
}
```

We choose this types of parser is because of the observation that all the queries in DeepBase have a lot of commons in query grammar and structure, for example the tables in from clause and the attributes that are grouped by. Therefore, we could manually optimize this part by hardcoding some information, and focus more on the optimization of inspect.

6.2 Generate Logical Plan

Different from traditional database's logical plan, the logical plan in DeepBase is in terms of executing our inspect clause since it is the most significant overhead to optimize rather than reading data from metadata tables.

The logical plan is a DAG whereas each node represents an operation. The source of the graph would always be the loader since the task that system needs to do is firstly read the data and the destination of the graph would always be inspector, which is an ending mark. Figure 2 shows an example for an inspection that uses on two hypothesis, one metric and one model.[5]

7 QUERY OPTIMIZATION

The Deepbase algebra is amenable to several optimizations that improve performance. To demonstrate this, Deepbase comprises four steps of optimization: merge task, choose the appropriate physical operators, estimate the runtime resource requirement and use ILP optimizer to generate the plan, and run a scheduler to map the physical plan to Dask cluster. In practice, we have a greedy heuristics implementation which guarantees fast scheduling.

Figure 5: Optimization Overview

7.1 Rule-based Optimization

There are two types of rule-based optimization.

Physical operator selector The first type of optimization involves selecting the physical implementation for each logical operator is rule-based. The select algorithm is decided on the logical operator type and user information. Some of the logical operators can be computed batch by batch, while some of the operators can only be calculated using the whole dataset. Another rule to choose is stateful or stateless. This situation happens when doing batch computing. For example, you need to keep the latest batch computation result in the memory and then compute the average with the next batch result. Also, users will give computational information of error and confidence. If users are tolerated of low precision, the system will choose the approximation algorithm for the user which will accelerate the workflow. Here are some example of physical operators the system will choose from: *ImageBatchedLoader*, *IoUIncrementalMetric*, *HypothesisBatchedLoader*, *KerasOneshotExtractor*, *KerasPersistentExtractor*, etc.

Task Merge In a logical plan, there are always hundreds of thousands of small tasks. It is not a good idea to distribute small tasks around different machines. Thus, the system will merge the small tasks to be one medium task and circulate it around.

Rule 1: Merge all logical ops with the same data op and model op input into one physical op.

Rule 2: Merge all logical metrics with the same metric type and inputs into one physical metric operator.

To prevent massive operators, Deepbase restricts to merge at most 100 logical operators in the merge step.

7.2 Estimation

In the estimation part, we estimate the memory cost, runtime, transfer time and whether GPU is needed. The value is based on pre-run results and empirical observations. Also, we consider the parameter size of input and output to come up with a precise estimation. For example, one metric operator takes 1 second to compute 5 hypothesis and 2 units. Our estimation for 10 hypotheses and 2 units would be 2 seconds. The estimation runtime is linear with the computation. The choice of GPU usage is decided by the user. Where available, extractors typical default to GPU computations where available.

7.3 Integer Linear Programming

The Inspect clause is much harder than the usual ones. According to the input parameters of the inspect declarative language specified task,**inspect** clause will invoke the activation extraction, hypothesis if required, and metric computing functions. All of the

computation can be processed batch by batch to mitigate the memory limitation problem. We treat the minimal computation unit as a single task. We can have three types of computation tasks: extraction, hypothesis, and metric. We hope to have a solution where the tasks that share the same batch of data should be placed at the same machine to decrease the communication cost if possible. The tasks which share the same models should be placed at one machine if possible. And the task dependencies are satisfied, which means, the optimizer is aware of the order of the tasks. Therefore, we model this distributed scheduler optimization problem as a integer linear programming. [3], [11]

We model each machine j with four parameters CC_j, CM_j, GC_j, GM_j in table 1.For each task i, we model the task with seven parameters cc_i, cm_i, gc_i, gm_i, $time_i$, $deps_i$ in table 2.

CC_j	CPU Cores of machine j
CM_j	CPU Memory of machine j
GC_j	GPU Cores of machine j
GM_j	GPU Memory of machine j

Table 1: Machine Parameters

cc_i	CPU Cores needed by task i
cm_i	CPU Memory needed by task i
gc_i	GPU Cores needed by task i
gm_i	GPU Memory needed by task i
$time_i$	Estimated running time of task i with enough resources.
$size_i$	Output size of task i
$deps_i = \{d_1, d_2, ..., d_k\}$	task i requires the outputs of task $d_1 ... d_k$

Table 2: Task Parameters

7.3.1 Main variables. We define binary variable X_{ijt}. $X_{ijt} = 1$ stands for task i is running on machine j at timestamp t. When all of the tasks are finished, the system is supposed to stop running. Therefore, we define binary variable S_t to represent whether the whole system is working at time t. When $S_t = 1$, it means there are still some tasks working. When $S_t = 0$, it means the whole system stops, the entire task finishes and the results are returned to users. Then we have constraint: $S_t <= S_{t-1}$, which means if machine is working at time t, then it must be working at time $t - 1$; and constraint: $S_t >= X_{ijt}$ for all i and j.

Our target to minimize is:

$$\sum_t S_t$$

7.3.2 Resource Constraints. Resource Constraints means that at any time, for any machine, the tasks running on that machine should use the existing machine resource. Memory should not exceed total available and the core numbers are limited as well. The formula below defines this constraint: for all j and t:

$$\sum_i X_{ijt} * \{cc_i, cm_i, gc_i, gm_i\} <= \{CC_j, CM_j, GC_j, GM_j\}$$

7.3.3 Continuous Running. A task should run on a machine without pause and resume. That means, we define $Y_{ijt} = |X_{ijt} - X_{ij,t-1}|$ and constraint $\sum_t Y_{ijt} = 2$ for all i and j. Here, we adapt the constraint to Integer Linear Programming like below:

$$Y_{ijt} \in \{0, 1\}$$
$$Y_{ijt} >= X_{ijt} - X_{ij,t-1}$$
$$Y_{ijt} >= X_{ij,t-1} - X_{ijt}$$
$$\sum_t Y_{ijt} = 2 for all i and j$$

7.3.4 Guarantee Execution Time. Our system should guarantee each task have enough time to execute. Once a task is run on one machine, it will execute for $time_i$ without interruption. Specifically, $\sum_t X_{ijt}$ should equal to 0 or $time_i$. Overall, task i is run on at least one machine. So we define binary variable O_{ij} to represent if task i runs on machine j. Then, we derive constraints below:

$$O_{ij} \in \{0, 1\}$$
$$\sum_t X_i jt = time_i * O_{ij}$$
$$\sum_j O_{ij} >= 1$$

7.3.5 Dependencies. To model dependency, e.g. task i depends on task i', the system requires the constraint that end time of i' is less than the begin time of i. So we define binary variables B_{ijt} and E_{ijt} which means the task i on machine j has begun or ended at time t. Therefore, we have constraints:

$$B_{ijt} >= B_{ij,t-1}$$
$$B_{ijt} >= X_{ijt}$$
$$E_{ijt} >= E_{ij,t-1}$$
$$E_{ijt} <= B_{ijt} - X_{ijt}$$

For all i depends on i', we have

$$B_{ijt} <= E_{i'j',t-comm(size_{i'},j',j)} + (1 - Ciji'j')$$

$comm(s, j, j')$ means the communication time needed to transfer data of size s from machine j to j'.

$C_{iji'j'}$ is binary variable representing task i on machine j requires the output of task i' on machine j'. There is a constraint to $C_{iji'j'}$:

for each pair of dependency: i depends on i':

$$\sum_{j'} C_{iji'j'} = 1 for all j$$

7.3.6 Translation to Physical Plan. After solving the integer linear programming, we can easily schedule the task i to j machine at time step t if $X_{ijt} = 1$. Since the actual runtime may slightly vary from the runtime we estimate, we use ordering of the solution to schedule tasks.

7.4 Discussion

The ILP formulation discussed so far is a useful formal tool, and by leveraging powerful commercial solvers (such as Gurobi) we can use it to study the solution space. However, it is not practical for real-world scenarios, and it cannot scale to large problem sizes. This is due to the corresponding explosion of the number of variables and constraints. The practical limit, is hundreds of atom expressions, or tens of complex order. Furthermore, there are prior impossibility results on designing optimal algorithms for commitment on arrival scheduling problems[6]. For these reasons, we focus on greedy heuristics next.

7.5 Greedy Heuristics

The algorithm 1 shows the framework of this greedy heuristic algorithm. This scheduler takes the communication cost and the dependency into consideration. The scheduler will give the task which more tasks depend on high priority. Also the task whose dependencies are on the same machine will be scheduled to the same machine first.

In detail, as shown in the algorithm 1, operator in the ruled based optimized plan has a dependencies list which record all the operators it depends on and a count of all the dependencies. (operator_i, [depend list], count) means depend list operators depend on this operator and there are *count* number of operators depend on operator_i. The operators in the ready task list can run because their dependencies are finished already. The overall algorithm uses a heap in python to maintain the timesstamp. In the end, the algorithm pushes operator's endtime and endtime + net_transfer time into the heap. In this way, every possible timestamp will be checked to schedule possible operator.

The greedy heuristic algorithm will return the physical plan in which every operator has its location(machine and core) and its *starttime* and *endtime*.

Algorithm 1 Framework of Greedy Scheduler.

Input: Rule based optimized plan with estimation information; Cluster configuration
Output: Physical plan with each physical operator containing the information of machine_id, core_id, start_time and estimation

```
Build dependencies index of (operator, [depend list], count);
Build ready Tasks list
Timestamps = [0]; LastT = None;
heapq.heapify(Timestamps)
while Timestamps != [] do
    T = heappop(Timestamps)
    if T == LastT then continue
    end if
    for every running operator do
        if opertor i.endtime < T then
            i operator finished job;
            for all the operators in i.depend_list do
                decrease one depending opertor;
                if no dependency, add to ready list;
            end for
        end if
    end for
    for every core c do
        Best = None
        for operator i in ready list do
            if recource not satisfied then continue
            end if
            if i has dependecy on other machine then
                if i.endtime + net_trans > T then
                    continue
                end if
            end if
            if operator i's depend count > Best's count then
                Best = i
            end if
        end for
        Operator Best will run on core c
        Update the machine vacant resource
        Update the operator Best location info
        Heapq.heappush(timestamps,Best.endtime)
        Heapq.heappush(timestamps,Best.endtime+net_tranfer)
    end for
end while
```

8 EXECUTOR

The physical plan provides the name of the operator, input schema, machine location and priority. One thing we need to ensure is that operators only run if the machine have sufficient resources to do so. In essence, we each operator has 3 parts. First, the operator must be initialized with the parameters passed in. Next, inputs and outputs need to be established. Lastly, we need to manage the sequence that operators run based on its priority while ensuring that the machine does not crash.

8.1 Implementation Details

First, we need to decide the type of dataflow system we would like to use. Dask[8] is built to execute compute graphs of functions,executed with either pull or push based dataflow. A key advantage of pushed based is that if a large push based graph is submitted, Dask itself can optimize and assign tasks to clusters. However, we lose the power to specify the exact machine each operator runs on and can only specify that information for the entire graph. As such, we choose to use a pushed based method.

Next, we look at how to ensure order. Dask actors are stateful pushed based instance, while Dask futures are stateless. We create a Dask actor to manage operators on each machine. The actor is keeps track of all existing operators scheduled to run on that machine and the order it runs in.

For communications between operators, we use the publish subscribe model which allows for direct communications between workers on different machines without going though the scheduler, which is typically a bottleneck. The publish subscribe model allows all subscribers to receive a message that a publisher puts into the channel.

In summary, the process is the following: first, create actors on all machines. Next, sort all operators by priority and assign them to the actors. Then we initialize communication channels between

operators with publish subscribe. Finally, for each batch, we start a round of computation on each actor.

9 EXPERIMENT

In our experiments, we implement a version of Netdissect with our system using the basic Image loader, Keras activations extractor, IoU Metrics and Netdissect specific hypothesis extractors. All machines in our experiments are Google Compute Engine instances with 16 cores, 50GB of memory. And some of them have 1 Nvidia T4 GPU. We will specify them in specific experiment. We compare the performance of DeepBase 2.0 against both the naive implementation of DeepBase 1.0, as well as an early stopping version of DeepBase 1.0.

9.1 Performance comparison

In the figure below, we show the basic time necessary to run the experiment specified above on the different variations of DeepBase. In Figure 6, we show the stability of our system over time by examining per batch computation time for DeepBase 2.0 with 3 machines, 150 batches of 32 images 100 hypothesis. Without early stopping, each batch takes around 25 seconds to compute. To achieve an error of less than 0.01 on the IoU score, DeepBase 2.0 takes around 15 iterations for most, though there are stragglers. From Table 3, we can see that DeepBase 2.0 achieves around 30X speedup over DeepBase 1.0. Here, we should mention that although we have three machines with 16 cores each, we only open 8 processes on each machine. Because the google cloud has a limited memory setting, if we open 16 processes, the memory will quickly be used up. There should ideally be a 24X speedup. Deepbase 2.0 achieve higher performance partly because in optimizer, we merge tasks to reuse shared computation, avoiding unnecessary loops and repeated work present in Deepbase 1.0.

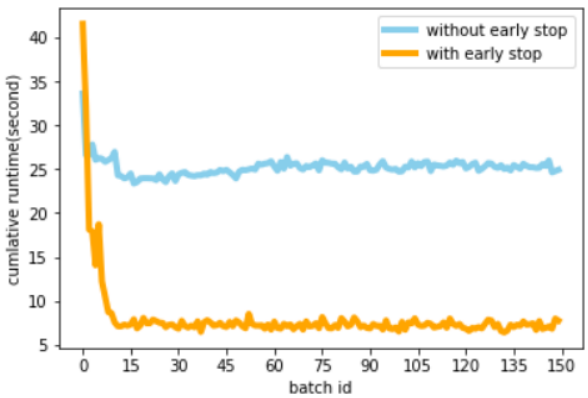

Figure 6: Per-Batch Runtime

9.2 Scalability

In this part, we vary the number of machine and the number of task to show the scalability of our system.

We conduct experiments on 2 machines and 3 machines for the inspection of 500 hidden units and 100 hypothesis using the IOU correlation. The machine setting are described above. These two clusters each contains one GPU machine. We start 8 processes on each machine for our tasks. The 8 shows that on 3 machine, the task has been speeded up by 1.7X. This demonstrates that our system can achieve very high scale out better then linearly scaling.

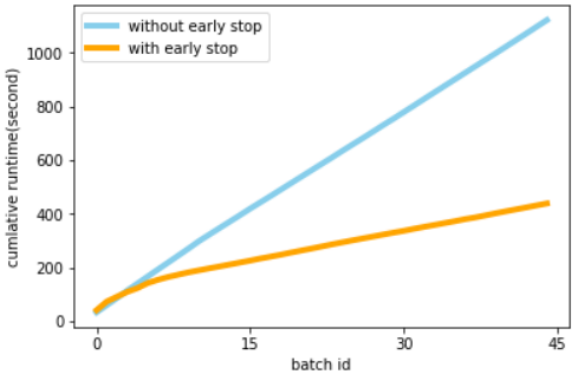

Figure 7: Cumulative Runtime

	Early Stop	Without Early Stop
DeepBase 1.0	> 15 hrs	> 30 hr
DeepBase 2.0	0.5 hr	1 hr

Table 3: Total time to run 150 batches

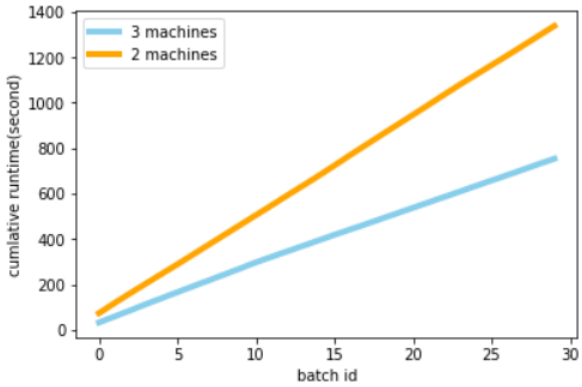

Figure 8: Same task on 2 machines and 3 machines

Then, we vary the number of tasks. We conduct this experiment on 2 machines. One with GPU and the other one only has CPU. We conduct the inspection of 500 hidden units and 100 hypotheses using the IOU correlation. Then, we cut the hypotheses to 50 and conduct the inspection of 500 hidden units and 50 hypotheses using the IOU correlation. The 9 shows we achieve 1.9x faster when we reduce the hypotheses number by 50%.

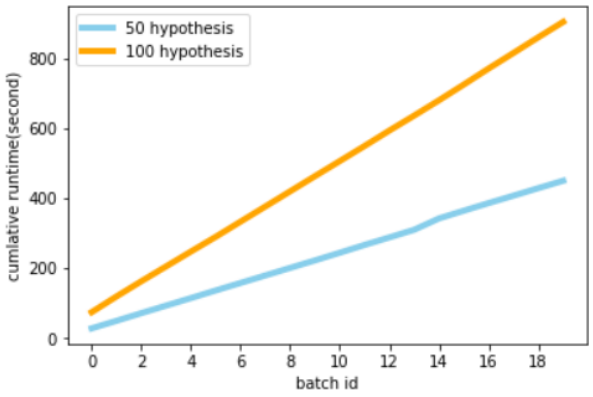

Figure 9: Comparison between 50 and 100 hypotheses

10 DISCUSSION

Our system enables high performance and better scalability when conducting the neural network inspection with greater ease of use compared to DeepBase 1.0.

Dask provides very limited support for controlling the number of tasks that run simultaneously. Together with poor garbage collection, there is a tendency to for memory to spike, which would kill a machine. Dask can also be unstable, and logic to recover should failure occur is difficult since we have to manage what is running ourselves. As a fairly new library, Dask can often have strange type incompatibility and unintuitive behaviour. For instance, ActorFuture are not compatible with other Future objects, making it difficult to connect stateful and stateless operators. Other issues we ran into include instability in the underlying tensorflow libraries and other unexpected behaviour in a distributed setting with python2, resulting in memory pointer corruptions. We have tried a number of different executor architecture, including running pull based dask graphs, submitting stateless futures and creating a new actor for each operator. We went with the publish subscribe to simplify coordination between operators. One limitation though is that the number of pub-sub channels is limited on a machine by the number of open files allowed, which imposes additional constraints operator instance.

Integer Linear Programming is NP complete problem which may be time-consuming.

Another improvement we can make in future work is to generalize hypothesis operators.Our current hypothesis loader is experiment-specific, which makes the system hard to extend. The reason is that the physical operator of hypothesis has to iterate over the hypothesis and it is very likely that user-uploaded hypothesis is not compatible with our system. It would be ideal that an API is provided to user so that the user could just specify some parameters for the hypothesis. This would improve the extensibility of our system.

A related improvement is a general way to check the input and output schema of operators before it runs. Input and output formats can vary greatly, and could contain dictionaries, objects, lists and more. While is does not pose a issue with query computation with a correct query, this is a necessary element we will add in future versions.

11 CONCLUSION

In this paper, we presented DeepBase 2.0, a distributed deep neural inspecting system. It offers a SQL-like query language for efficient usage and run inspection distributively.

We implemented a prototype of DeepBase 2.0 that is able to run an end-to-end query and conducted Netdissect experiment on DeepBase. Our experiment shows that the execution is 30x faster than DeepBase 1.0, which demonstrates the scalability of our system. Our system also demonstrates linear scalability.

REFERENCES

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/2465351.2465355

[2] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. Network Dissection: Quantifying Interpretability of Deep Visual Representations. In *Computer Vision and Pattern Recognition*.

[3] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based scheduling: If you're late don't blame us!. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[4] David R. Hardoon, Sandor R. Szedmak, and John R. Shawe-taylor. 2004. Canonical Correlation Analysis: An Overview with Application to Learning Methods. *Neural Comput.* 16, 12 (Dec. 2004), 2639–2664. https://doi.org/10.1162/0899766042321814

[5] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. 2013. MLbase: A Distributed Machine-learning System.. In *Cidr*, Vol. 1. 2–1.

[6] Brendan Lucier, Ishai Menache, Joseph (Seffi) Naor, and Jonathan Yaniv. 2013. Efficient Online Scheduling for Deadline-sensitive Jobs: Extended Abstract. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*. ACM, New York, NY, USA, 305–314. https://doi.org/10.1145/2486159.2486187

[7] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. 2017. SVCCA: Singular Vector Canonical Correlation Analysis for Deep Learning Dynamics and Interpretability. arXiv:arXiv:1706.05806

[8] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*. Citeseer.

[9] Thibault Sellam, Kevin Lin, Ian Yiran Huang, Yiru Chen, Michelle Yang, Carl Vondrick, and Eugene Wu. 2018. DeepBase: Deep Inspection of Neural Networks. arXiv:arXiv:1808.04486

[10] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. arXiv:arXiv:1312.6034

[11] Eugene Wu, Samuel Madden, and Michael Stonebraker. 2013. Subzero: a fine-grained lineage system for scientific databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 865–876.

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