DeepBase: Deep Inspection of Neural Networks

Thibault Sellam
Columbia University
tsellam@cs.columbia.edu

Kevin Lin
Columbia University
kl2806@columbia.edu

Ian Huang
Columbia University
iyh2110@columbia.edu

Michelle Yang
UC Berkeley
michelleyang@berkeley.edu

Carl Vondrick
Columbia University
cv2428@columbia.edu

Eugene Wu
Columbia University
ewu@cs.columbia.edu

ABSTRACT

Although deep learning models perform remarkably across a range of tasks such as language translation, parsing, and object recognition, it remains unclear whether, and to what extent, these models follow human-understandable logic or procedures when making predictions. Understanding this can lead to more interpretable models, better model design, and faster experimentation. Recent machine learning research has leveraged statistical methods to identify hidden units that behave (e.g., activate) similarly to human understandable logic such as detecting language features, however each analysis requires considerable manual effort. Our insight is that, from a query processing perspective, this high level logic is a *query* evaluated over a *database of neural network hidden unit behaviors*.

This paper describes DeepBase, a system to inspect neural network behaviors through a query-based interface. We model high-level logic as *hypothesis functions* that transform an input dataset into time series signals. DeepBase lets users quickly identify individual or groups of units that have strong statistical dependencies with desired hypotheses. In fact, we show how many existing analyses are expressible as a single DeepBase query. We use DeepBase to analyze recurrent neural network models, and propose a set of simple and effective optimizations to speed up existing analysis approaches by up to 413×. We also group and analyze different portions of a real-world neural translation model and show that learns syntactic structure, which is consistent with prior NLP studies, but only requires 3 DeepBase queries.

1 INTRODUCTION

Neural networks (NNs) are revolutionizing a wide range of machine intelligence tasks with impressive performance, such as language understanding [23], image recognition [21], and program synthesis [17]. This progress is partly driven by the proliferation of deep learning libraries and programming frameworks that drastically reduce the effort to construct, experiment on, and deploy new models [8, 14, 35].

However, it is still unclear how and why neural networks are so effective [1, 18]. Does a model learn to decompose its task into understandable subtasks? Does it memorize training examples [65], can it generalize to new situations? Understanding the internal representation of a trained model learns will enable more rapid experimentation and development of NN models, help identify harmful biases, and explain predictions—all critical in real-world deployments.

A prevailing paradigm is to study how individual or groups of hidden units (neurons) behave when the model is evaluated over test data. One approach is to identify if the behavior of a hidden unit mimics a high level functionality—if a unit only activates between opening and closing quotation marks, then it potentially has learned how to recognize quotations. Numerous papers have applied this ideas by manually inspecting visualizations of behaviors [21, 31, 51] or using analysis-specific scripts [7, 56], and in domains such as detecting syntax and sentiment in language [31, 51], parts and whole objects from images [21, 37], image textures [7], and chimes and tunes in audio [6].

This class of analysis is ubiquituous in the deep learning literature [4, 7, 9, 28, 32, 42, 45] and we term it *Deep Neural Inspection* (*DNI*). Given user-provided hypothesis logic (e.g., "detects nouns", "detects keywords"), DNI seeks to quantify the extent that the behavior of hidden units (e.g., the magnitude or the derivative of their output) is similar to the hypothesis logic when running the model over a test set. Although different instances of DNI may share a common set of operations, each analysis requires considerable engineering resources to implement from scratch, and can execute inefficiently. *We believe there is tremendous opportunity to provide a declarative abstraction to easily express, execute, and optimize DNI analyses as queries.*

Our main insight is that although DNI analyses may use different NN models, hypotheses, or types of hidden unit behaviors, they all use statistical measures to quantify the affinity between hidden unit behaviors and hypotheses. These measures can then be used to identify groups of hidden units to debug, compare models based on the logic that their hidden units mimic, and better understand how NNs achieve high empirical performance. To this end, we present DeepBase, a system to perform large-scale *Deep Neural Inspection* through a declarative query interface. Given groups of hidden units, hypotheses, and statistical measures of learn-ability, DeepBase quickly computes the affinity score between each (hidden units group, hypothesis) pair. DeepBase is embedded into SQL using a new `INSPECT` clause so that users can easily filter, group, and inspect hidden units within a single statement. Section 2.2 describes many previously manual analyses [4, 31, 66] that can be easily expressed as DNI queries.

Designing a fast DNI system is challenging because the cost is a cubic function dependent on the size of the dataset, the number of hidden units, and the number of hypotheses. Even trivial examples are computationally expensive. Consider analyzing simple character-level recurrent neural nets (RNNs) with 128 hidden units over a corpus of 6.2M characters [31]. Assuming each activation is stored as a 4-byte float, each RNN model requires 3.1GB to store its activations. While this fits in memory, the process to extract and

materialize these activations, and matching them with hundreds of hypotheses can be incredibly slow.

To address these challenges, DeepBase uses pragmatic optimizations based on existing DNI use cases. First, the users provide hypothesis logic in the form of functions evaluated over input data, which are potentially expensive to compute. When DeepBase is re-run on new models but the hypothesis logic remains unchanged, we cache the output of the hypothesis functions to avoid incurring redundant costs. Second, users can specify convergence thresholds so that DeepBase can terminate quickly while returning accurate but approximate scores. Since many DNI analyses use correlation measures and linear prediction models, DeepBase natively supports those measures. Third, DeepBase reads the dataset, extracts unit behaviors, and evaluates the user-defined hypothesis logic in an online fashion, to avoid unnecessary costs when the affinity scores have converged. Finally, DeepBase leverages GPUs—commonplace in deep learning—to offload the cost of unit behavior extraction and compute expensive affinity metrics such as logistic regression in parallel.

Our primary contribution is to formalize Deep Neural Inspection and develop a declarative query language to specify DNI analyses. We also contribute:

- The design and implementation of an end-to-end DNI system called DeepBase, along with several simple but effective optimizations, including caching, early stopping via convergence criteria, streaming execution, and GPU execution.
- A walk-through of how hypothesis functions can be easily generated from existing machine learning libraries (Section 4.2, Appendix B).
- A verification procedure based on input perturbation to provide more confidence that a unit group of high affinity scores are indeed affected by the hypothesis function (Section 4.4).
- We compare the runtime to inspect a SQL auto-completion RNN model. We show that with all optimizations including hypothesis caching, DeepBase outperforms an implementation based on MADLib [24] by 100 – 413×, depending on the specific affinity measure.
- Experimental results using DeepBase to analyze a wide variety of RNN models to show the flexibility and effectiveness of the optimizations, along with the analysis of a state-of-the-art Neural Machine Translation model architecture [33] (English to German) that finds results consistent with NLP research [56] (Section 6).

This paper focuses the discussion, and application of DeepBase, on Recurrent Neural Networks (RNNs), a widely used class of NNs used for language modeling, program synthesis, image recognition, and more. We do this to simplify the exposition while focusing on an important class of NNs, however the system design and techniques are applicable to other classes such as Convolutional Neural Networks (CNNs).

2 BACKGROUND AND USE CASES

This section introduces the basic types of neural network models and neural network terminology used in this paper. Further, we describe representative use cases of DNI that are currently implemented and used in an manual manner. These uses cases serve as the motivation for the system described in the rest of the paper.

2.1 Neural Network Primer

We provide a brief review of neural networks. For an extensive overview, we refer readers to Goodfellow et al. [22].

Neural networks are mappings that transform an input vector s to an output vector y. The mapping is parameterized by a vector of weights w, which is learned from data. To capture nonlinear relationships between s and y, we can stack linear transformations followed by nonlinear functions:

$$y = w_{n+1}^T h_n \tag{1}$$

$$h_n = \sigma\left(w_n^T h_{n-1}\right) \tag{2}$$

$$h_0 = s \tag{3}$$

where h_n is a vector of *hidden units* in the nth layer that represent intermediate states in the neural network, which are also frequently called activations or "neurons". The function σ must be nonlinear for the model to learn nonlinear mappings, and today most neural networks use the rectified linear unit $\sigma(x) = \max(0, x)$. When interpreting neural networks, we are often interested in understanding what the hidden units h_n are learning to detect, which is the focus of this paper.

Recurrent neural networks (RNNs) are popular models for operating on sequences, for example processing text as a sequence of words. Given an input sequence where s_t is the t-th element in the sequence, a recurrent network follows the recurrence relation:

$$y_t = w_2^T h_t \tag{4}$$

$$h_t = \sigma\left(w_1^T [s_t, h_{i-1}]\right) \tag{5}$$

$$h_0 = 0 \tag{6}$$

where the intermediate hidden units h_t are a function of the tth element in the sequence and the previous hidden units from the previous element in the sequence. Importantly, the parameters w are independent of the position in the sequence, allowing the model to process arbitrary length sequences. Modern recurrent networks use a more sophisticated update mechanism to stabilize optimization, for example with LSTMs [26], which we use in our experiments. However, the high-level question of interpretation remains the same: we are interested in analyzing and understanding the intermediate activations h_i in the network.

Recall that h_t is a vector with multiple dimensions. For clarity, we refer to a specific dimension in h_t as u_i, which represents a single hidden unit. We call any value that can be computed for u_i at each time step a *behavior*. For example, we can compute measures such as u_i's gradient with regards to the input (the derivative $\frac{\delta u_i}{\delta x_t}$).

2.2 Motivating Example

We will use an RNN with a single LSTM layer that performs SQL query auto-completion as a motivating example. We use a toy grammar of trivial SELECT statement rules to simplify the exposition, and scale up to more complex grammars in our Experiments section. Given a SQl string, the model reads a window of 100 characters (padded with a null character if necessary) to predict the next character. For a given window, it reads each character as described in the primer. The model can be iteratively run in a generative mode to output full query strings by using the previous prediction to predict

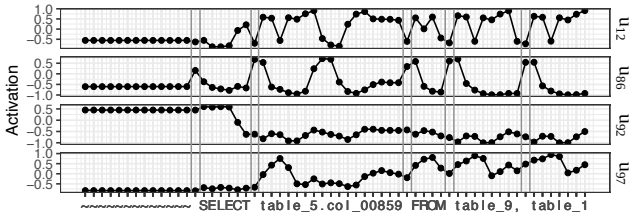


Figure 1: Activations over time for the SQL auto-completion model. What is the model learning?

and emit the next character. The layer contains 128 hidden units and a final fully connected layer makes the prediction. We convert each character into a one-hot vector that indicates the original character, and train the model on a thousand randomly generated queries to predict the next character with the cross-entropy loss function.

The model achieved 80% prediction accuracy on a held-out test set of 1,152 queries, as compared with random guess accuracy of $\frac{1}{32}$. In addition, 75% of its generated queries can be parsed in average. Those statistics indicate that the model can reliably predict the next character. *But what did the model really learn?* One hypothesis is that the model "memorized" all possible queries. Another is that it learns an N-gram model that uses the previous N − 1 characters to predict the next. Or the model learned portions of the SQL grammar. We would also like to check these hypotheses across models with different architectures or training parameters, or for a specific set of units.

2.3 Approaches for Interpretation

The machine learning community has developed a variety of approaches for interpretation, which we discuss below.

Manual Visual Inspection: Manual approaches [31, 60], such as LSTMVis [60], visualize each unit's activations and let users manually check that the units behave as expected. For instance, if a unit only spikes for table names, it suggests that the model behaves akin to that grammar rule and possibly has "learned" it. Unfortunately, visual inspection is difficult, even for simple settings. Figure 1 plots the activations of 4 units on the prefix of a query. We easily observe that units are inactive when reading the padding character "~". However, interpreting the fluctuations is difficult. u_{12} appears to spike down on whitespaces (highlighted), mirrored by u_{86}. u_{97} tends to activate within the words `FROM` and `table`. But those observations are simply guesses on a small string; scaling this manual analysis to all units and all queries is impractical. Ideally, we would express these hypotheses and formally test them at scale.

Saliency Analysis: This approach seeks to identify the input symbols that have the largest "effect" on a single or group of units. For instance, an NLP researcher may want to find words that an LSTM's output is sensitive to [38], or the image pixels that most activates a unit [21]. This analysis may use different behaviors, such as the unit activation or its gradient. Typically, the procedure collects a unit's behaviors, finds the top-k highest value behaviors, and reports the corresponding input symbols. For instance, whitespace and period trigger the five highest activations for u_{86} in Figure 1. This DNI approach has been used to analyze image object detection [55, 57, 67], NLP [38], and sentiment [51].

Statistical Analysis: Many datasets are annotated: text documents are annotated with parse trees or linguistic features, while image pixels are annotated with object information. Such annotations can help analyze groups of units. For instance, Kim et. al [32] use logistic regression to predict annotations of high-level concepts from unit activations. NetDissect [7] finds the image pixels that cause a unit to highly activate (similar to saliency analysis), and computes the Jaccard distance between those pixels and annotated pixels of e.g., a dog. In general, these techniques compute a statistical measure between unit behaviors and annotations of the input data, and have been used to e.g., find semantic neurons [42], find units that learn texture [7], and evaluate transfer learning [4, 9, 45].

2.4 Desiderata of a DNI System

DNI analysis using the existing approaches is powerful and spans domains and applications. Unfortunately, each analysis currently requires custom, adhoc implementation despite following a common analysis goal: **the user wants measure the extent that groups of hidden units in one or more trained models behave in a manner that is similar to, or indicative of, a human-understandable function, when evaluated over the same test dataset**. For instance, we may want to measure the extent that the activation behavior of each LSTM unit in our SQL auto-completion model have high correlation with a function that detects the presence SQL keywords by emitting 1 for keyword characters and 0 otherwise.

DeepBase is a system that provides a declarative abstraction to efficiently express and execute these analyses. DeepBase takes as input a test set, a trained model, a set of Python functions that encode hypothesizes of what the model may be learning (we call them *hypothesis functions*), and a scoring function, e.g., a measure of statistical dependency. From those inputs, DeepBase produces a set of scores that quantify the affinity between the hypotheses and the model's hidden units. Such a system should support:

Arbitrary Hypothesis Logic: Different applications and domains care about different hypotheses. In autocompletion, does the model learn to count characters? In machine translation, do units learn sentiment or language nuances such as relational dependencies? In visual object recognition, pixels correspond to different types of objects—do units detect pixels containing dogs or cats? A system should be flexible about the types of logic that can be used as queries.

Many Models and Units: Modern neural network models can contain tens of thousands of hidden units, and researchers may want to compare across different model architectures, training epochs, parameters, datasets, or groups of units. A DNI system should allow users to easily specify which combination of hidden units and models to inspect.

Different Affinity Measures: Different use cases and users may define affinity differently. They may correlate activations of individual units [31], compute mutual information between a unit's behavior and annotations [42], use a linear model to predict high-level logic from unit activations [4, 32], or use another measure.

A DNI system should be fast for common measures, and support user-defined measures.

Mix and Match Users should be able to easily specify the combination of hypothesis functions, models, hidden units, and datasets that they want to inspect.

Analyze Quickly Developers use inspection functionality to interactively debug and understand the characteristics of their models. Thus, any system should both scale to a large number of models, test data, and queries, while maintaining acceptable query performance.

3 PROBLEM DEFINITION

We now define the deep neural inspection problem, using the SQL auto-completion model in Section 2.2 as the example.

Problem Setup: Let $D = \{d_1, \cdots\}$ be a dataset of records, or sequences, where each $d_i = [s_1, \cdots]$ is a sequence of symbols. In our SQL auto-completion example, each record is a window of 100 symbols, and each symbol is a hot-one encoded character. For other data types, a symbol may be an image pixel, word, or vector depending on the model. We assume all records have length n_s by padding shorter sequences with Null symbols.

A model $M = \{u_1^M, \cdots, u_n^M\}$ contains $|M|$ hidden units. Logically, $M(d)$ is evaluated by reading each input symbol s_i one at a time, which triggers a single behavior $b_i \in \mathbb{R}$ from each hidden unit u[1]. For instance, each line graph in Figure 1 plots a unit's activations when reading each character in the input query. In this paper, we report results based on unit activation, however DeepBase is agnostic to the specific definition of behavior extracted from the model. This flexibility is important because existing papers may use the gradient of the activations instead of their magnitude [67]. Finally, let $u(d) = [b_1, \ldots, b_{n_s}]$ be the *Unit Behavior* when the model is evaluated over all symbols in d, and $U(d) = \{u(d) | u \in U\}$ be the *Group Behavior* for a set of units in the model $U \subseteq M$.

We model high level logic in the form of a *Hypothesis Function*, $h(d) \in \mathbb{R}^{n_s}$, that outputs a *Hypothesis Behavior* when evaluated over d. For instance, a hypothesis that the model has learned to detect the keyword "SELECT" would emit 1 for those characters and 0 otherwise. There is no restriction on the complexity of a hypothesis function and Section 4.2 describes example functions; the only constraint is that the hypothesis behavior is size n_s.

So far, we modeled both hidden unit and hypothesis behavior over input data as time series of the same length. We quantify the affinity between each unit $u \in U$ in a group of units U and a hypothesis h based on a user-defined statistical affinity measure $l(U, h, D) = (\mathbb{R}^{|U|}, \mathbb{R})$ that outputs a scalar affinity score for each unit, along with a score for the group as a whole. Similar to [31], we may study each unit individually, and compute the correlation between their behavior and a grammar rule such as "SELECT" keyword detection. Alternatively, we may group the units in a layer and measure whether we can predict the occurence of the keyword from their behavior, using a classifier such as logistic regression [4]; the model's F1 score is the group affinity, and each unit's score is its model coefficient. Although $l(U, h, D)$ is user-defined, commonly used measures exhibit properties that are amenable to optimization.

[1]In the context of windowing over streaming data, the RNN model internally encodes a dynamically size sliding window over the symbols seen so far.

Basic Problem Definition: Given the above definitions, we are ready to define the basic version of DNI:

Definition 1 (DNI-Basic). *Given dataset* D*, a subset of units* $U \subseteq M$ *of an RNN model* M*, hypothesis* h*, statistical measure* l*, return the set of tuples* (u, s_u, s_U) *where the score* s_u *is defined as* $l(U, h, D) = ([s_u | u \in U], s_U)$.

Note that we specify as input a set of units U rather than the full model M. This is because the statistical measure l() may assign different affinity scores depending on the group units that it analyzes. For instance, if the user inspects units in a single layer using logistic regression, then only the behaviors of those units will be used to fit the linear model and their coefficients will be different than when inspecting all in the model. This highlights the value of embedding DeepBase within a SQL-like language.

Although the above definition is sufficient to express the existing approaches in Section 2.2, it is both inconvenient and inefficient. Developers often train and want to compare many groups of units to understand e.g., what hypotheses the model learns across training epochs. For space considerations, we present the generalized problem definition in Appendix A.

4 DECLARATIVE DEEP NEURAL INSPECTION

We first describe the INSPECT clause and example DNI Queries. We then describe how create hypothesis functions, DeepBase native statistical measures, and a verification procedure to assess the quality of highly scored units. The next section describes the system design and optimizations.

4.1 The INSPECT Clause Syntax

DeepBase models hidden units, hypotheses, and model inputs as relations (or views) in a database. Let `units` be a relation representing hidden units (`uid`) and their models (`mid`), and `hypotheses` represent hypothesis functions (`h`). These relations may contain additional meta-data attributes—e.g., unit layer, training epoch, or the source of the hypothesis function—for filtering and grouping the hidden units.

The INSPECT clause expresses the deep neural inspection operation. We introduce a separate clause rather than use a custom UDF to make it easier to use. DNI is neither a scalar UDF nor an user defined aggregation (UDA). Instead, it outputs a set of records for each input group (UDAs return a single record per group), and the outputs across all groups needs to be flattened into a single relation before further processing. INSPECT performs this flattening automatically.

The syntax specifies unit ids and hypothesis functions to compare, optional affinity measures (e.g., logistic regression), and the dataset of input sequences used to extract unit and hypothesis behaviors. By default, DeepBase measures correlation between individual units and hypotheses:

```
INSPECT <unit>, <hypothesis> [USING metric, ...]
   OVER <sequences>
```

The clause is evaluated prior to the SELECT clause and outputs a temporary relation with schema `(uid, hid, mid, group_score, unit_score)` containing unit, hypothesis, and model ids, and two affinity scores. `group_score` is the affinity between a group of units U and the hypothesis h, whereas `unit_score` specifies the affinity

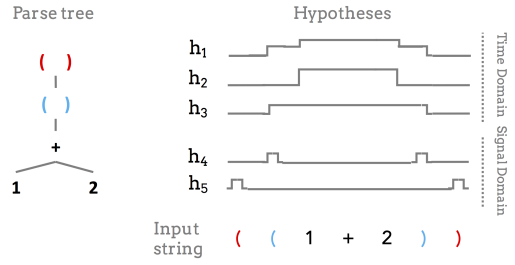


Figure 2: Example parse tree (left) and hypothesis functions. Each h_i is the behavior over each input character symbol.

of each unit u ∈ U; groups are defined using `GROUP BY`. These scores are interpreted depending on the type of statistical measure. For instance, correlation is computed for individual units (each group is a single unit), so the two scores are the same. In contrast, when using logistic regression, the model F1 score represents the affinity of the group, while the coefficients are the individual unit scores. This relation can be renamed but only referenced in later clauses (e.g., `SELECT`, `HAVING`).

Users often inspect models as part of debugging. To provide the flexibility to specify combinations of models and hidden units to analyze, we integrate DNI into a SQL-like language. To continue the motivating example (Section 2.2), the following query groups hidden units by the sql parser model's training epochs, computes the correlation between each unit in layer 0 with a hypothesis that recognizes SQL keywords (e.g., "SELECT", "FROM"), and returns the epoch and id of high scoring units:

```
  SELECT M.epoch, S.uid
INSPECT U.uid AND H.h USING corr OVER D.seq AS S
   FROM models M, units U, hypotheses H, inputs D
  WHERE M.mid = U.mid AND M.mid = 'sqlparser' AND
        U.layer = 0 AND H.name = 'keywords'
GROUP BY M.epoch
  HAVING S.unit_score > 0.8
```

4.2 Hypotheses

Hypotheses are the cornerstone of DNI analyses, as they encode the logic that we search for. Although numerous language-based models, grammars, parsers, annotations, and other information already exist, many do not fit the hypothesis function abstraction. For example, parse trees (Figure 2) are a common representation of an input sequence that characterizes the roles of different subsequences of the input. What is an appropriate way to transform them into hypothesis functions? This section provides examples for generating hypothesis functions from common machine learning libraries; additional examples are found in Appendix B.

Parse Trees: A common use of RNNs is for parsing and language models. In these applications, there are decades of research on language parsing, ranging from context free grammars for programming languages to dependency and constituency parsers for natural language. Figure 2 illustrates an example parse tree for a simple algebraic expression within nested parentheses `((1+2))`. The corresponding parse tree contains leaf nodes that represent characters matching terminals, and intermediate nodes that represent non-terminals.

Given a parse tree, we map each node and node type to a hypothesis function. To illustrate, the red root in Figure 2's parse tree corresponds to the outer `()` characters. It can be encoded as a time-domain representation that activates throughout the characters within the parentheses (h_3), or a signal representation that activates at the beginning and end of the parentheses (h_5). Similarly, h_2 and h_4 are time and signal representations generated by hypothesis functions for the inner blue parentheses. Finally, h_1 is a composite of h_2 and h_3 that accounts of the nesting depth for the parentheses rule. Note that a given parse tree generates a large number of hypothesis functions, thus the cost of parsing is amortized across many hypothesis functions.

This form of encoding can be used for other parse structures such as entity-relationship extraction.

Annotations: Existing machine models are trained from massive corpa of manually annotated data. This ranges from bounding boxes of objects in images to multi-word annotations for information extraction models. Each annotation type is akin to a node type in a parse tree and can be transformed into a hypothesis function that emits 1 when the annotation is present and 0 otherwise.

Similarly, image datasets (e.g., Coco [39], ImageNet [15]) contain annotations in the form of bounding boxes or individual pixel labels. Both can be modeled as hypotheses functions that map a sequence of image pixels to a boolean sequence of whether the pixel is labeled with a specific annotation.

4.3 Natively Supported Measures

DeepBase supports two types of statistical measures.

Independent Measures measure the affinity between a single unit and a hypothesis function and are commonly used in the RNN interpretation literature. Examples in prior work include Pearson's correlation, mutual information [42], difference of means, Jaccard coefficient [66]. In general, DeepBase supports any UDF that takes two behavior time series as input. Independent measures are amenable to parallelization across units, which DeepBase enables by default. We currently implement an incremental variant of Pearson's correlation.

Joint Measures compute the affinity between a group of units U and a hypothesis h, and scores for each unit u ∈ U. For instance, when using logistic regression, we jointly compute one score for the whole group of units (e.g., prediction accuracy), and we assign individual scores based on model's coefficients. The current implementation supports convex prediction models that implement incremental `train` and `predict` methods. If the U's behaviors are predictive of h, then there is evidence that they have learned h [4, 47]. By default, DeepBase uses logistic regression with L1 regularization, and we report the 5-fold F1 cross-validation scores.

4.4 Verification

This paper proposes a declarative system to express existing deep neural inspection analyses. DNI is a data mining procedure that computes a large number of pairwise statistical measures between many groups of units and hypotheses. When using the scores to identify high-scoring units, the decision is susceptible to multiple hypothesis testing issues and can generate false positives. Although a complete treatment to address this problem is beyond the scope of

this paper, DeepBase implements a perturbation-based verification procedure to ensure that the set of high scoring units indeed have higher affinity to the hypothesis function. To do so, we procedure is akin to randomized control trials, where, for a given input record, we perturb it in a way to swap a single symbol's hypothesis behavior, and measure the difference in activations.

Formally, let h() be a hypothesis function that has high affinity to a set of units U. It generates a sequence of behaviors when evaluated over a sequence of symbols:

$$h([s_1, \ldots, s_{k-1}, s_k]) = [b_1, \ldots, b_{k-1}, b_k]$$

After fixing the prefix $s_1, \ldots, s_{k-1}$, we want to change the k^{th} symbol in two ways. We swap it with a *baseline* symbol s_k^b so that b_k^b remains the same, and with a *treatment* symbol s_k^t so that b_k^t changes.

$$h([s_1, \ldots, s_k^b]) = [b_1, \ldots, b_k^b] \text{ s.t. } b_k^b = b_k, s_k^b \neq s_k$$
$$h([s_1, \ldots, s_k^t]) = [b_1, \ldots, b_k^t] \text{ s.t. } b_k^t \neq b_k, s_k^t \neq s_k$$

Let act(s) be U's activation for symbol s, $\Delta_k^b = act(s_k^b) - act(s_k)$ be the change activation for a baseline perturbation, and Δ_k^t be the change for a treatment perturbation. Then the null hypothesis is that Δ_k^t and Δ_k^b, across different prefixes and perturbations, are drawn from the same distribution.

For example, consider the input sentence "He watched Rick and Morty.", where the hypothesis function detects coordinating conjunctions (words such as "and", "or", "but"). We then perturb the input words in two ways. The first is consistent with the hypothesis behavior for the symbol "and", by replacing "and" with another conjunction such as "or". The second is inconsistent with the hypothesis behavior, such as replacing "and" with "chicken". We expect that the change in activation of the high scoring units for the replaced symbol (e.g., "and") is higher when making inconsistent than when making consistent changes. To quantify this, we label the activations by the consistency of the perturbation and then measure the Silhouette Score [53], which scores the difference between the within- and between-cluster distances.

Our verification technique is based on analyzing the effects of input perturbations on unit activations, however there are a number of other possible verification techniques. For instance, by perturbing the model using ablation [31, 42] (removing the high scoring units and retraining the model) and measuring its effects on the model's output. We leave an exploration of these extensions to future work.

5 SYSTEM DESIGN

This section first describes a baseline system design using an existing ML-in-DB system called MADLib, and its challenge. We then describe the DeepBase system, a set of effective optimizations, and implementation details.

5.1 MADLib Approach and Challenges

Our initial implementation used existing ML-in-database systems [20, 36], specifically the MADLib [24] Postgres extensions, to compute the affinity scores. These systems can express and execute convex optimization problems (such as model training) as user-defined aggregates. For example, the following query trains an SVM model using records in `data(X, Y)` and outputs the model parameters in the `modelname` table. Note that the relation name is passed as a parameter, and the UDA internally scans and manipulates the relation.

```
SELECT SVMTrain('modelname', 'data', 'X', 'Y');
```

One way to use MADLib is to first extract unit and hypothesis behaviors from the input dataset, and materialize them as relations `unitsb` and `hyposb`, respectively. Their schemas `(id, unitid/hypoid, symbolid, behavior)` contain the behavior value for each unit (or hypothesis) and input symbol . A Python driver then rewrites the `INSPECT` clause into one or more large SQL aggregation queries to compute the affinity scores. For example, the correlation between each unit and hypothesis can be expressed as:

```
SELECT U.uid, H.h, corr(U.val, H.val)
  FROM unitsb U, hyposb H GROUP BY U.uid, H.h
```

The first challenge is behavior representation. Deep learning frameworks [2, 13, 49] return behaviors in a dense format. Reshaping the matrices into a sparse format is expensive, and this representation is inefficient because it needs to store a hypothesis or unit identifier for each symbol. To avoid this cost, we store the matrices in a dense representation where each unit (`U.uid_i`) or hypothesis (`H.h_j`) is an attribute. We compute the metrics as follows:

```
SELECT corr(U.uid1, H.h_1),...corr(U.uidn, H.h_m)
  FROM unitsb_dense U JOIN hyposb_dense H ON
       U.symbolid = H.symbolid
```

Unfortunately, there can easily be > 100k pairs of units/hypotheses to evaluate, while existing databases typically limit the number of expressions in a clause to e.g., 1,600 in PostgreSQL by default. We could batch the scores (i.e., the subexpressions `corr(U.uid, H.h_m)`) into smaller groups and run one `SELECT` statement for each batch, but this would force PosgesSQL to perform hundreds of passes over the behavior relations (one full scan for each query). The problem is even more acute with MADLib's complex user-defined functions, such as `SVMTrain`, which incurs a full scan of the behavior tables and a full execution of the UDF for every hypothesis (see Section 6.3). This leads to our second challenge: how to efficiently evaluate hundreds, potentially thousands of units/hypotheses pairs without incurring duplicate work?

The third challenge is that extracting the behavior matrices can be expensive [62]. Unit behaviors require running and logging model behaviors for each record, while hypothesis behaviors require running potentially expensive UDFs. For instance, our experiments use NLTK [10] for text parsing, which is slow and ultimately accounts for a substantial portion of execution costs. Furthermore, users often only want to identify high affinity scores, thus the majority of costs may compute low scores that will eventually be filtered out. Thus, it is important to reduce: the number of records that must be read, the number of unit behaviors to extract and materialize, the number of hypotheses that must be evaluated, and affinity score computation that are filtered out.

5.2 DeepBase Architecture Overview

The architecture of DeepBase follows a standard query plan, with the exception of the `INSPECT` operator. Thus, Figure 3 focuses on a naive design of `INSPECT`, which will be optimized below. The operator first materializes all behaviors from the dataset D. The *Unit Behavior Extractor* takes one or more unit groups as input, and

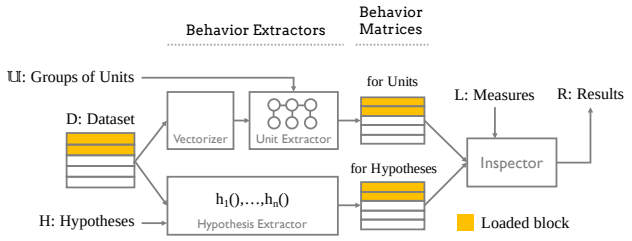


Figure 3: Architecture of the `INSPECT` operator.

generates behaviors for each unit in each group—the assumption is that each group is a subset of units from a single RNN model. Similarly, the *Hypothesis Behavior Extractor* takes a set of hypothesis functions as input and applies them to generate hypothesis behaviors. We concatenate the sequences together, so the extractors output matrices of dimensionality $|\mathbb{U}| \times |D| \cdot n_s$ (for Units) and $|H| \times |D| \cdot n_s$ (for Hypotheses). Since the number and length of records ($|D| \cdot n_s$) can dwarf the number of units and hypotheses, these matrices are "skinny and tall". The `Inspector` takes as input these two matrices, along with a set of statistical measures, and computes affinity scores for each triplet of unit, hypothesis, and measure.

DeepBase is implemented in Python, and DNI analysis of Keras, Tensorflow, and PyTorch models. It extracts unit activations using a GPU, which accelerates activation extraction as compared to a single CPU core. Hypotheses are executed using a single CPU core. DeepBase trains logistic regression models, and more generally all affinity measures based on linear models, as a Keras neural network model on a GPU. Finally, DeepBase can cache the hypothesis behavior matrix in cases where the model repeatedly changes. Our implementation uses simple LRU to pin the matrix in memory, and integrating caching systems such as Mistique [62] for unit and hypothesis behaviors is a direction for future work.

5.3 Optimizations

Below we outline the main optimizations.

5.3.1 Shared Computation via Model Merging. Although affinity score measures are typically implemented as Python user defined aggregates, DeepBase also supports implementations as a Keras computation graph. For instance, the default Logistic Regression measure is implemented as a Keras model. This enables a shared computation optimization we call model merging. The naive approach trains a separate model for every hypothesis, which can be extremely expensive. Instead, DeepBase *merges* the computation graphs of all $|H|$ hypotheses into a single large composite model. The composite model has one output for each hypothesis rather than $|H|$ models with one output each. This lets DeepBase make better use of Keras' GPU parallel execution. It also amortizes the per-tuple overheads across the hypotheses—such as shuffling and scanning the behavior relations, and data transfer to the GPU.

DeepBase produces one composite model for each affinity measure, by duplicating the intermediate and final layers for each hypothesis and force them to share the same input layer. If the model doesn't have a hidden layer (as in logistic regression), DeepBase can further merge all output layers into a single layer with one or more units (if the categorical output is hot-one encoded) per hypothesis; DeepBase then generates a custom cost function that aggregates the costs for all the hypotheses.

5.3.2 Early Stopping. Much of machine learning theory assumes that datasets used to train machine learning models are samples from the "true distribution" that the model is attempting to approximate [19]. DeepBase assumes that the dataset D is a further subsample. Thus, the affinity scores are actually empirical estimates based on sample D.

A natural optimization is to allow the user to directly specify stopping criteria to describe when the scores have sufficiently converged. To do so, a statistical measure l() can expose an incremental computation API:

```
l.process_block(U, h, <rec>)→(<scores>, err)
```

The API takes a set of records as input and returns both the group and unit scores, as well as an error of the group score. Users can thus specify a minimum threshold for `err`. If this API is supported, then DeepBase can terminate computation for the pair of units and hypothesis function early. Otherwise, DeepBase ignores the threshold and computes the measure over all of D.

We expose an API rather than make formal error guarantees because such guarantees may not available for all statistical measures. For example, tight error bounds are not well understood for training non-convex models (e.g., neural nets), and so in practice machine learning practitioners check if the performance of their model converges with empirical methods (i.e., comparing the last score to the overage over a training window [50]). There exists however formal error bounds for statistical measures such as correlation [19]. By default DeepBase implements this API for pairwise correlation and logistic regression models. To estimate error of the correlation score, we use Normal-based confidence intervals from the statistical literature (i.e., Fisher transformation [19]). For logistic regression, we follow established model training procedures and report the difference between the model's current validation score and the average scores over the last N batches, with N set up by default to cover 2,048 tuples.

Early stopping is implemented by iteratively loading and processing blocks of pre-materialized unit and hypothesis behavior matrices in blocks of n_b records. It loads for all units in each group U, and for as many hypotheses as will fit into memory, and checks the error for every statistical measure after each block.

Note that the moment the score for a given hypothesis and unit group has converged, then there is no need to continue reading additional blocks for that hypothesis. Thus, there is a natural trade-off between processing very small blocks of rows, which incurs a the overhead of checking convergence more frequently, and large blocks of rows, which may process more behaviors than are needed to converge to ε. Empirically, we find that setting $n_b = 512$ works well because most measures converge within a few thousand records. This optimization ensures that the query latency is bound by the complexity of the statistical measure, rather than the size of the test dataset.

5.3.3 Streaming Behavior Extraction. A consequence of employing approximation is that DeepBase does not need to read all of the materialized matrices. Our third optimization is to materialize the

behavior and hypothesis matrices in an online fashion, so that the amount of test data that is read is bound by how quickly the confidence of the statistical measures converge. To do so, we read input records in blocks of n_b and extract unit and hypothesis behaviors from them in parallel. An additional benefit of this approach is that affinity scores can be computed and updated progressively, similar to online aggregation queries, so that the user can stop DeepBase after any block.

Figure 3 illustrates streaming execution using the orange blocks. The input D contains 5 blocks of records, and only two blocks of unit and hypothesis behaviors have been extracted so far. When all affinity scores have converged, then DeepBase can stop. Although it is possible to furher optimize by terminating hypothesis extraction for hypotheses that have converged, we find that the gains are negligible. This is because 1) training the composite model from model merging costs the same for one hypothesis as it does for all, and 2) some hypothesis extractors, such as creating a parse tree for NLP, incurs a single cost amortized across all parsing-based hypotheses.

6 EXPERIMENTS

Our first evaluation challenge is to assess whether the hidden units that DeepBase scores highly are indeed correct. To this end, we first present an accuracy benchmark to study DNI under conditions when we "force" parts of a model to learn a hypothesis function. Following this analysis, we present scalability experiments using a SQL auto-completion RNN model to show how DeepBase (and its optimizations) scales as we vary the number of hidden units, hypotheses, and records. Finally, we run DeepBase on a real world English-to-German translation model from OpenNMT [33]. Additional details and benchmarks are in Appendix C.

6.1 Setup

We ran DeepBase on three types of RNN models, and ran verification on the high scoring units. The aim of the first two types of models is to predict the next symbol (character) for strings generated from formal grammars; the third model is a sequence-to-sequence natural language translation model.

Datasets: We used three language datasets for our experiments: a simple parentheses grammar for the accuracy benchmark, a SQL grammar for the scalability benchmark, and a publicly available English-to-German translation dataset for the real-world experiment. The grammar-based datasets are generated by sampling from a probabilistic context-free grammar to generate syntactically valid sentences/queries.

The parentheses dataset (used in [59]) consists of strings such as `0(1()1(2()))0()` where a digit representing the current nesting level precedes each balanced parenthesis (up to 4 levels). Its grammar consists of the same production rule ($r_i \rightarrow \varepsilon|0(r_{i+1})r_i$) for $i < 4$ nesting levels, along with a terminating rule ($r_4 \rightarrow \varepsilon|4r_4$) at the 4^{th} level.

To generate synthetic SQL queries, we sample from a Probabilistic Context Free Grammar (PCFG) of SQL. We choose subsets of the grammar (between 95 to 171 production rules) to vary the language complexity, as well as the number of hypothesis functions. The task is to take a window of 30 characters and predict the last character.

We trained the OpenNMT model on the WMT 2015 dataset[2].

Models: The parentheses and SQL tasks use custom models: a one-hot encoded input layer, a LSTM layer, and a fully connected layer with soft-max loss for final predictions (details below). The OpenNMT model [33] is publicly available, it uses an encoder-decoder architecture, where both the encoder and decoder contain two LSTM layers of 500 units, with an additional attention module for the decoder.

Hypotheses: We follow the procedure in Section 4.2 to transform parse trees for each experiment into a set of hypothesis functions. By default, we use the time-domain representations for each node type (e.g., production rule, verb, punctuation). In our experiments, we do not run the parser until one of the hypothesis functions is evaluated; at that point the other hypothesis functions based on the parser do not need to re-parse the input text. For the SQL experiment, to increase the number of hypotheses, we also generate hypothesis functions using the signal representation. We use NLTK's chart parser [10] to sample and parse the SQL grammar, and CoreNLP's [40] constituency and dependency parsers for the OpenNMT experiment.

6.2 Accuracy Benchmark

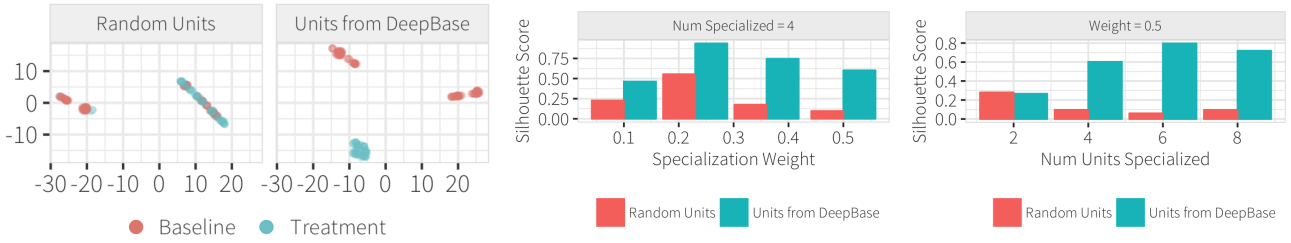
DeepBase is designed to quickly compute affinity scores in order to identify hidden units that are likely to learn and mimic user-understandable hypotheses. The goal in this subsection is to understand the conditions when high affinity scores are indicative that a group of units have learned a given hypothesis.

To establish ground truth, we specially train a 16-unit RNN model by "specializing" a subset of units $S \subseteq M$ to learn a specific hypothesis h. To do so, we introduce an auxiliary loss function g_h that forces the output of the neurons in S to to be close to the output of h. If g_h is the auxiliary loss function and g_T is the loss function for M based on the next character prediction task, the model's loss function is a weighed average of $g_M = w \times g_h + (1 - w) \times g_T$. This setup allows us to vary the number of specialized units $|S|$, and the specialization weight w for how much the specialized units focus on learning the hypothesis. Their defaults are $|S| = 4$, $w = 0.5$.

The challenge in this benchmark is that not all units in S may be needed to learn a given hypothesis. For instance, if the hypothesis is to detect the current input, then one unit may be sufficient. In contrast, all units may be needed to learn a higher level hypothesis. We run DeepBase using logistic regression with L1 regularization, return units with unit_score above 15, and use the perturbation-based verification method in Section 4.4 to assess the quality of the high scoring units.

Figure 4a shows an example clustering of the change in activations between the baseline and treatment perturbations (colors). The hypothesis function used to inspect the model recognizes parentheses symbols, thus the baseline perturbations swap '(' for ')' or vice versa. The treatment swaps '(' for a number. Units selected by DeepBase show clear clusters that distinguish baseline and treatment perturbations, while the change in activation for a set random units (the same number of units) overlap considerably (blue and red are indistinguishable). Figure 4b summarizes the cluster separation using the Silhouette score [53] and shows higher separation than

[2] http://statmt.org/wmt15



(a) T-SNE clustering of change in activation during verification. Each point is one unit's activation. (Weight=0.5, Num Specialized=4)

(b) Silhouette scores varying the number of specialized units. (Weight=0.5)

(c) Silhouette scores varying weight. (Num Specialized=4)

Figure 4: Verification results for parentheses detection hypothesis function.

random units across all weights. Similar results are shown when varying the number of specialized units in Figure 4c.

We also ran the above for two other hypothesis functions: predicting the current nesting level, and predicting that the current nesting level is 4. The former hypothesis is nearly identical to the model task, and we indeed find that none of the units selected by DeepBase distinguish themselves from random during verification. The latter hypothesis is ambiguous: the specialized units may simply recognize the input character 4, or learn the nesting level. After running verification by swapping 4 with other numbers (baseline) or open parentheses (treatment), we find that the change in activations were indistinguishable. Thus suggests that the specialized units learned to recognize the input 4 rather than the logical nesting level.

Takeaways: *Although ground truth does not exist for deep neural inspection analyses, unit specialization provides a (weak) form of ground-truth. In general, DNI analyses (using DeepBase or another system) is a form of data mining, and may misinterpret the behavior of hidden units (e.g., if the hypothesis is very similar to the model task or ambiguous). DeepBase's perturbation-based verification method helps us identify these false positives.*

6.3 Scalability Benchmarks

We now report scalability results on the SQL grammar benchmark. To do so, we vary the number of records in the inspection dataset, hidden units in the model and rules in the grammar used to generate the data and the features. The default setup contains 29,696 records[3], 512 hidden units and 142 grammar rules. Each record has $n_s = 30$ symbols, so there are 890,880 behaviors for each unit and hypothesis. We build two hypotheses per non-terminal in the grammar–the first one returns "1" for each symbol for which the rule is active, the second only triggers for the first and last symbol, which yields 190 hypotheses.

Compared Systems: We start with the naive implementation, then demonstrate the benefits of model merging (`Naive`) and cumulatively add the optimizations described in Section 5: early stopping (`EarlyStop`), online extraction (DeepBase). We compare against the implementation using MADLib [24] (`MADLib`), which fully materializes the behavior matrices, and then computes affinity scores using PostgreSQL native (for correlation) and MADLib (for logistic regression) function implementations.

Setup: We run each experiment three times and report the average runtime. All our experiments are based on 6 Google Cloud virtual machines with 8 virtual CPUs each, where a virtual CPU is implemented a hyper-thread on a 2.3 GHz Intel Xeon E5 CPU. The main memory is 32GB and the OS is Linux Ubuntu 16.04. The virtual machines include nVidia Tesla K80 GPUs with 12 GB GDDR5 memory. All the models are based on Keras with Tensorflow 1.8. MADLib runs on top of PostgreSQL 9.6.9, with the shared buffer size, effective cache size and number of workers tuned following to the manual's guidelines. Hypothesis extraction is performed by creating a parse tree using NLTK, a text analysis Python library, and transforming the tree into hypotheses.

We extract behaviors in blocks of 512 records, and set the Keras batch size to 512 records. All the models are trained for up to 50 epochs with Keras early stopping. Their average classification accuracy is 49.7%, and 53-69% of randomly generated queries can be parsed (based on the grammar complexity). The approximation defaults use $\varepsilon = 0.025$ and confidence to 95% for correlation, and set convergence for logistic regression to 0.01.

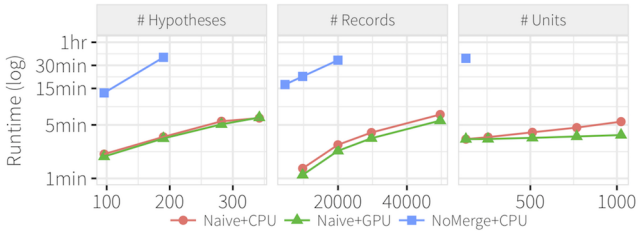


Figure 5: Runtime of `Naive` without model merging on CPU (`NoMerge`), with model merging on CPU (`Naive+CPU`) and on GPU (`Naive+GPU`) using logistic regression measure.

Model Merging and Hardware: We first evaluate different versions of `Naive` to highlight the benefits of model merging and the GPU. We use the logistic regression measure using `Naive` without

[3]Recall that each record in DeepBase is a window of symbols of length n_s as defined by a sliding window of size n_s and stride 5.

model merging on the CPU (NoMerge), with model merging on a CPU (Naive+CPU) and GPU (Naive+GPU). The lack of model merging forces multiple scans of the behaviors in order to train models serially (one full scan per hypothesis). Executing on a CPU vs GPU has minor effects for small models containing few units, but the gap increases with larger models. In fact, performance is nearly insensitive to model size due to massive GPU parallelism. Overall, model merging and using the GPU improve upon NoMerge by 6.9× on average, and by up to 14×. For the rest of the experiments, we use Naive to refer to Naive+GPU.

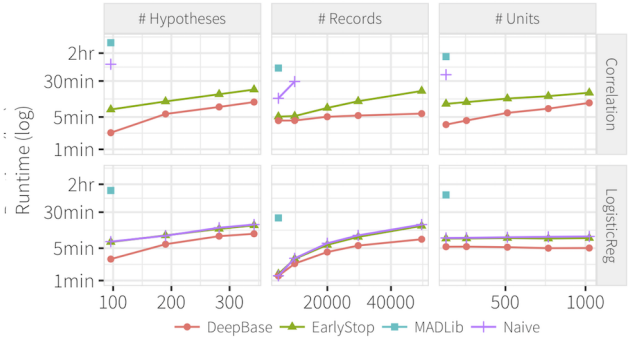


Figure 6: Total runtime using correlation and logistic regression measures, varying # hypotheses, records, and units.

Runtime Comparisons: Figure 6 summarizes the system runtime for each optimization as we vary the number of hypotheses, records, and units for the correlation and logistic regression measures. We limited each run to 2 hours. We find that the runtimes for the two measures exhibit considerably different trends due to the number of measures that must be computed, the cost of inspection, and GPU parallelization. Overall, DeepBase outperforms MADLib by ≈ 25× on average, and up to 30×, for the two measures.

Correlation (top row) is expensive because it must be computed for every unit-hypothesis pair (up to 194,560 pairs in the experiments). MADLib incurs a large number of passes over the behavior relations (up to 121), PostgreSQL/Naive incur considerable table scan and aggregation costs due the lack of early stopping, and EarlyStop highlights the cost to fully extract and materialize the behavior matrices. DeepBase avoids unnecessary extraction costs once all the scores have converged. As expected, the techniques are all sensitive to the number of units and hypotheses, while DeepBase converges to 5min as the dataset grows.

Logistic regression (bottom row) is dominated by the cost to fit logistic regression models for each pair of hypothesis and unit group. The model merging optimization using the GPU enables DeepBase and EarlyStop to run independently of the number of hidden units, and increases linearly as the number of records and hypotheses increase (below, we will see this is due to hypothesis extraction overhead). Overall, DeepBase improves upon Naive by 1.6× on average, and by up to 2.4×; it outperforms MADLib by 21× on average, and by up to 30×.

Runtime Breakdown: Figure 7 shows the cost breakdown by system component: the hypothesis and unit extractors, and the inspector. The EarlyStop column shows that inspector cost is much higher for correlation, while extraction behave nearly identically. The DeepBase column shows that runtime savings are primarily due to lower extraction costs thanks to online extraction.

Cached Hypothesis Extraction: We found that hypothesis extraction due to a slow parsing library can dominate the runtime. However, during model development or retraining, the developer typically has a fixed library of interesting hypothesis functions and wants to continuously inspect how the model behavior is changing. Figure 8 examines this case: the left column incurs all runtime costs, while the right column shows when hypothesis behavior has been cached. We see that it improves correlation somewhat, but its cost is dominated by inspection; whereas for logistic regression, DeepBase improves to run at constant time of ≈ 20s. Caching improves correlation by 1.9× on average, and logistic regression by 12.4× on average and up to 19.5×. Overall, DeepBase outperforms MADLib by up to 413×.

Sensitivity to Error Threshold: Figure 9 examines the sensitivity to varying the error threshold (x-axis) for correlation and logistic regression, using the default experiment parameters. The top row shows correlation: EarlyStop only reduces the inspector costs as the threshold is relaxed, while DeepBase reduces the extraction costs considerably because it only extracts behaviors when necessary. The bottom row shows logistic regression, which exhibits similar trends, though it is far less sensitive to the error threshold because the optimization converges slowly.

Takeaways: *Independent measures such as correlation are computed on a per-unit basis, and the cost is dominated by inspection costs. In contrast, join measures such as logistic regression are computed for each unit-group are dominated by extraction costs. Early stopping improves indpendent measure performance, while online extraction enables DeepBase to run nearly independently of dataset size. When hypothesis behaviors are cached, DeepBase outperforms* MADLib *by up to* 367×.

6.4 Neural Machine Translation

We use DeepBase to inspect an English-German translation model open-sourced by OpenNMT [33]. Each English word is mapped to a 500-dimension vector and fed into the encoder RNN, which produces activations for each time step. At each step, the decoder uses all the activations of the encoder, combined with the decoder's current activation, to predict the next German word. We select

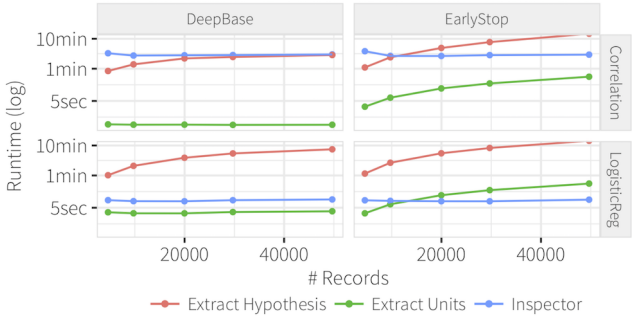


Figure 7: Runtime breakdown of extraction and inspector costs for correlation and logistic regression.

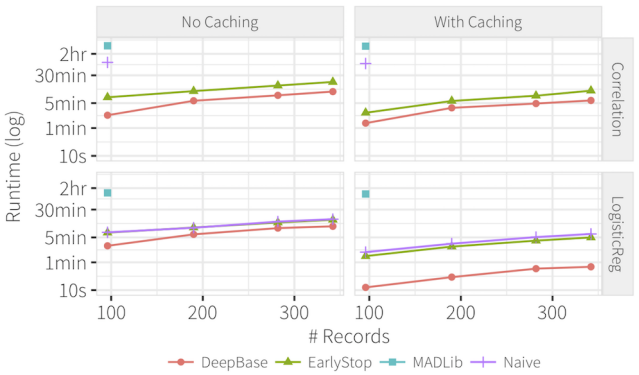


Figure 8: Runtime comparing the effects of cached hypothesis behavior.

and inspect the encoder units using a library of natural language hypotheses generated from CoreNLP. We use one DeepBase query each to inspect individual units, all encoder units, and layer-by-layer; the latter two analyses are the same as in [56].

We create 56 hypothesis functions. 36 are for different part-of-speech tags (DT, CC, CD, etc.) in the Penn Treebank, and 7 for phrase-level structures (NP, VP, PP, etc.). Dependency parsing generates labeled pointers between "head words" and "tail words" that modify the head words. We generate one hypothesis for all head words, and 10 for the tail or the head of five common dependency labels (nmod, nsubj, compound, etc.). Another hypothesis indicates named entities and finally, a hypothesis that acts as a counter for the sentence length.

Individual Units: We first use correlation to study individual units. Prior work found individual interpretable units in character-level language models [31], and we find similar units at the word-level. They learn low-level features (e.g., periods, commas, etc) along with one unit that tracks the sentence length. Going beyond past studies, we find that high affinity units are only present in the trained model and not in an untrained model of the same architecture (Figure 10a).

Encoder Level: We then use logistic regression with L2 regularization to study all 1000 units in a trained and untrained model (Figure 10b). We first confirm recent work showing that model architecture can act as a strong prior [3]. Similarly, the untrained model has high affinity with some low level language features (e.g, periods), but low affinity for almost all high-level features. On the other hand, the trained model has far higher affinity to various POS tags (e.g., CD, RB, VBD, etc.) and phrase structure (e.g., VP, NP) than the untrained model.

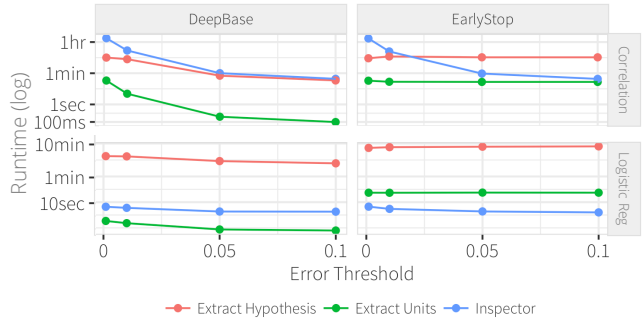


Figure 9: Runtime when varying error threshold for early stopping. Note different Y-axis scales.

Unit groups: We now inspect each layer separately, and use Logistic Regression with L1 to identify unit groups with non-zero coefficients. Previous work [9, 56] showed that both encoder layers learn POS features, but layer 0 is slightly more predictive and more distributed (spread over more units). Similarly, we find that layer 0 yields higher F1 scores and selects more units for most hypotheses. Going beyond prior work, we find that the unit group size varies widely depending on the language feature. In layer 1 for example, 372 units are found to detect verbs, 62 units to detect coordinating conjunctions (e.g. 'and', 'or', 'but'), while only 9 units to detect punctuation such as ".".

Verification: Finally, we report verification results for three high affinity POS hypotheses (determiners, coordinating conjunctions, possessive pronouns) in the unit group study. For each hypothesis, we perturbed words in sentences from the WMT 2015 dataset where the hypothesis is non-zero. Figure 10c verifies that the change in activations for the units found by DeepBase are well separated than for randomly selected units. The verification result goes beyond, and validates, those results reported in [56].

Takeaways: *DeepBase expresses the types of analyses in recent NLP studies to understand neural activations in machine translation [9, 56] and reports results consistent with [56] using only* 3 *DeepBase queries. Our library of natural language hypothesis functions automates model inspection for syntactic features that NLP researchers are commonly interested in, and is easy to extend. Integrating with SQL lets users easily inspect and perform different analyses by comparing different models and units, such as at the individual, group, and layer levels.*

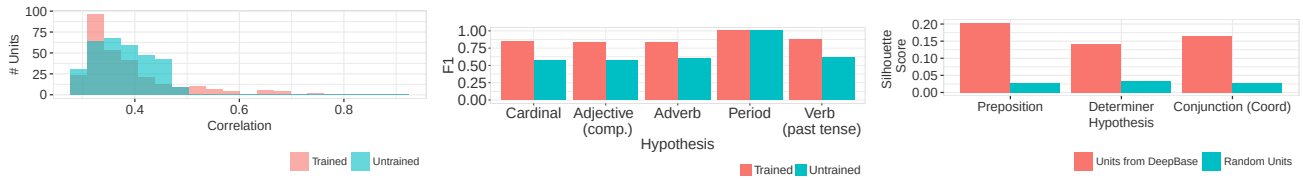
7 RELATED WORK

Interpreting Deep Learning Models: Neural network understanding is an active field of research, and there are many proposed approaches towards model interpretation.

In our Background section 2, we reported three methods: visualization of the hidden unit activations [25, 31, 60], saliency analysis [21, 38, 55, 57, 61, 67] and statistical neural inspection [4, 7, 28, 32, 42, 45]. These methods represent the common approaches in the neural net understanding literature, and motivate the design of DeepBase.

Other approaches including using generative methods [43, 44] that generate synthetic inputs by inverting the transformation induced by the hidden layers of a neural net (the most compelling example reveal e.g., textures, body parts or objects). However, most of the literature focuses on computer vision applications, and the process relies heavily on human visual inspection.

Another form of inspection is occlusion analysis, by which machine learning engineers selectively replace patches of an image by a black area and observe which hidden units are affected as a result [64]. Currently, most studies that fall under this category are ad-hoc and target image analysis. Our verification method 4.4 is an attempt to generalize and automate this process by defining input



(a) Histogram of correlations for all encoder units in OpenNMT. High correlations are only found in the trained model.

(b) L2 Logistic Regression F1 measure for different hypotheses. Both models learn low-level hypotheses (period); only the trained model learns higher level concepts.

(c) Silhouette scores of three high-affinity hypotheses when performing verification.

Figure 10: Deep neural inspection on OpenNMT translation model. Results compared against an untrained OpenNMT model.

perturbations (of which occlusion is one type of input perturbation) with respect to the desired hypothesis function.

Because the field is still in its infancy [18, 27], the majority of existing implementations are specialized research prototypes and there is a need for general software systems in the same way TensorFlow and Keras simplify model construction and training. A notable example is Lucid [46], which bundles feature inversion, saliency analysis, with visualization into a larger grammar. Lucid has similar goals as DeepBase, however its focus is on images and still relies on manual analysis.

Visual Neural Network Tools: Our work is related to neural net visual debugging tools. Numerous visualization tools have been developed to inspect the architecture of deep models [29, 58], do step debugging to check the validity of the computations [12], visualize the convergence of gradient descent during training [30] and drill into test sets to understand where models make errors [30].

Machine Learning Interpretation: A related field of research seeks to augment machine learning predictions with explanations, to help debugging or augment software produces based on classifier. A common, classifier-oblivious approach is surrogate models, which approximates a complex model by a simpler one (e.g., classification tree or logistic regression). They train a simple, often linear, model over examples in the neighborhood of a test datapoint so that users can interpret the rationale for a specific model decision [5, 52]. Other approaches modify the machine learning model so that predictions are inherently interpretable. For example, PALM [34] uses a decision tree that fits a neural network to data points in each leaf node, so that the decision path constitutes the explanation for a misprediction. In contrast, DeepBase seeks to identify general behaviors with respect to a test dataset by inspecting individual and groups of unit behaviors.

Databases and Models: A number of database projects have proposed integrating machine learning models, training, and prediction into the database [11, 16, 20, 24, 35]. Recent projects such as ModelDB [63] and Modelhub [41] propose to manage historically trained models, and can be used by DeepBase to select models and hidden units to inspect. Similarly, systems such as Mistique [62] can be used in conjunction with DeepBase to manage the process of extracting and caching unit activations. Further, fast timeseries analysis algorithms such as KShape [48] can be used in lieu of pearsons correlation to quickly compute per-unit affinity scores.

8 CONCLUSION AND DISCUSSION

Programming frameworks for deep learning have enabled machine learning to impact a large set of applications, and better NN analysis tools will contribute to a better understanding of how and why neural networks work. Towards this goal, this paper describes DeepBase, a system to quickly inspect neural network behavior through a declarative SQL-like interface. We find that DeepBase can express a large fraction of existing deep neural inspection analyses, but improves the analysis runtime by up to 86× and automates common processes. When applying DeepBase to real-world translation models, we find results consistent with prior NLP-specific research.

We intend to extend DeepBase with more statistical measures (e.g., mutual information), deeper integration with GPUs and fast database engines, and apply DeepBase to a broader range of applications (involving, among others, Convolutional Neural Networks). Looking further, we envision Deep Neural Inspection as a core primitive of a larger neural network verification and inspection framework [54]. DeepBase allows users, or automated processes to query neural network models using high level hypotheses. We imagine curating *libraries of hypotheses* based on decades of existing models, features, and annotations across application domains. In addition, these tools may be used to identify biases in neural networks, decompose NNs into smaller components, enforce activation behavior for unit groups, and ultimately open up NN black boxes.

REFERENCES

[1] How many computers to identify a cat? 16,000. https://www.nytimes.com/2012/06/26/technology/in-a-big-network-of-computers-evidence-of-machine-learning.html.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[3] J. Adebayo, J. Gilmer, I. Goodfellow, and B. Kim. Local explanation methods for deep neural networks lack sensitivity to parameter values. *ICLR*, 2018.

[4] G. Alain and Y. Bengio. Understanding intermediate layers using linear classifier probes. *arXiv preprint arXiv:1610.01644*, 2016.

[5] D. Alvarez-Melis and T. S. Jaakkola. A causal framework for explaining the predictions of black-box sequence-to-sequence models. *arXiv preprint arXiv:1707.01943*, 2017.

[6] Y. Aytar, C. Vondrick, and A. Torralba. Soundnet: Learning sound representations from unlabeled video. In *Advances in Neural Information Processing Systems*, pages 892–900, 2016.

[7] D. Bau, B. Zhou, A. Khosla, A. Oliva, and A. Torralba. Network dissection: Quantifying interpretability of deep visual representations. *arXiv preprint arXiv:1704.05796*, 2017.

[8] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on*

Knowledge Discovery and Data Mining, pages 1387–1395. ACM, 2017.
[9] Y. Belinkov, N. Durrani, F. Dalvi, H. Sajjad, and J. Glass. What do neural machine translation models learn about morphology? *arXiv*, 2017.
[10] S. Bird and E. Loper. Nltk: the natural language toolkit. In *ACL*, 2004.
[11] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *VLDB*, 2014.
[12] S. Cai, E. Breck, E. Nielsen, M. Salib, and D. Sculley. Tensorflow debugger: Debugging dataflow graphs for machine learning. *Reliable Machine Learning in the Wild*, 2016.
[13] F. Chollet et al. Keras, 2015.
[14] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.
[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.
[16] A. Deshpande and S. Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD*, pages 73–84. ACM, 2006.
[17] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.
[18] F. Doshi-Velez and B. Kim. Towards a rigorous science of interpretable machine learning. *arXiv:1702.08608*, 2017.
[19] B. Efron and T. Hastie. *Computer age statistical inference*, volume 5. Cambridge University Press, 2016.
[20] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, pages 325–336. ACM, 2012.
[21] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.
[22] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
[23] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1764–1772, 2014.
[24] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.
[25] M. Hermans and B. Schrauwen. Training and analysing deep recurrent neural networks. In *Advances in neural information processing systems*, pages 190–198, 2013.
[26] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
[27] F. Hohman, M. Kahng, R. Pienta, and D. H. Chau. Visual analytics in deep learning: An interrogative survey for the next frontiers. *arXiv preprint arXiv:1801.06889*, 2018.
[28] A. Kádár, G. Chrupała, and A. Alishahi. Representation of linguistic form and function in recurrent neural networks. *Computational Linguistics*, 2017.
[29] M. Kahng, P. Y. Andrews, A. Kalro, and D. H. P. Chau. A cti v is: Visual exploration of industry-scale deep neural network models. *IEEE transactions on visualization and computer graphics*, 24(1):88–97, 2018.
[30] M. Kahng, D. Fang, and D. H. P. Chau. Visual exploration of machine learning results using data cube analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 1. ACM, 2016.
[31] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.
[32] B. Kim, J. Gilmer, F. Viegas, U. Erlingsson, and M. Wattenberg. Tcav: Relative concept importance testing with linear concept activation vectors. *arXiv preprint arXiv:1711.11279*, 2017.
[33] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. Opennmt: Open-source toolkit for neural machine translation. In *ACL*, 2017.
[34] S. Krishnan and E. Wu. Palm: Machine learning explanations for iterative debugging. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, page 4. ACM, 2017.
[35] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1717–1722. ACM, 2017.
[36] A. Kumar, F. Niu, and C. Ré. Hazy: making it easier to build and maintain big-data analytics. *Queue*, 2013.
[37] Q. V. Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.
[38] J. Li, X. Chen, E. Hovy, and D. Jurafsky. Visualizing and understanding neural models in nlp. *arXiv preprint arXiv:1506.01066*, 2015.
[39] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, 2014.
[40] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.
[41] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Modelhub: Towards unified data and lifecycle management for deep learning. *arXiv preprint arXiv:1611.06224*, 2016.
[42] A. S. Morcos, D. G. Barrett, N. C. Rabinowitz, and M. Botvinick. On the importance of single directions for generalization. *arXiv*, 2018.
[43] A. Nguyen, A. Dosovitskiy, J. Yosinski, T. Brox, and J. Clune. Synthesizing the preferred inputs for neurons in neural networks via deep generator networks. In *Advances in Neural Information Processing Systems*, pages 3387–3395, 2016.
[44] A. Nguyen, J. Yosinski, Y. Bengio, A. Dosovitskiy, and J. Clune. Plug & play generative networks: Conditional iterative generation of images in latent space. *arXiv preprint arXiv:1612.00005*, 2016.
[45] M. Noroozi and P. Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In *European Conference on Computer Vision*, pages 69–84. Springer, 2016.
[46] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, and A. Mordvintsev. The building blocks of interpretability. *Distill*, 2018. https://distill.pub/2018/building-blocks.
[47] S. J. Pan and Q. Yang. A survey on transfer learning. *TKDE*, 2010.
[48] J. Paparrizos and L. Gravano. k-shape: Efficient and accurate clustering of time series. In *SIGMOD*, 2015.
[49] A. Paszke, S. Gross, S. Chintala, and G. Chanan. Pytorch, 2017.
[50] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.
[51] A. Radford, R. Jozefowicz, and I. Sutskever. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*, 2017.
[52] M. T. Ribeiro, S. Singh, and C. Guestrin. 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*, pages 1135–1144. ACM, 2016.
[53] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *JCAM*, 1987.
[54] T. Sellam, K. Lin, I. Y. Huang, C. Vondrick, and E. Wu. "i like the way you think!" inspecting the internal logic of recurrent neural networks. *SylML*, 2018.
[55] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *See https://arxiv. org/abs/1610.02391 v3*, 7(8), 2016.
[56] X. Shi, I. Padhi, and K. Knight. Does string-based neural mt learn source syntax? In *EMNLP*, 2016.
[57] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.
[58] D. Smilkov, S. Carter, D. Sculley, F. B. Viégas, and M. Wattenberg. Direct-manipulation visualization of deep networks. *arXiv preprint arXiv:1708.03788*, 2017.
[59] H. Strobelt, S. Gehrmann, B. Huber, H. Pfister, and A. M. Rush. Visual analysis of hidden state dynamics in recurrent neural networks. *arXiv preprint arXiv:1606.07461*, 2016.
[60] H. Strobelt, S. Gehrmann, H. Pfister, and A. M. Rush. Lstmvis: A tool for visual analysis of hidden state dynamics in recurrent neural networks. *IEEE transactions on visualization and computer graphics*, 24(1):667–676, 2018.
[61] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. *arXiv preprint arXiv:1703.01365*, 2017.
[62] M. Vartak, S. Madden, J. Trindade, and M. Zoharia. Mistique: A system to store and ery model intermediates for model diagnosis. In *SIGMOD*. ACM, 2018.
[63] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Modeldb: a system for machine learning model management. In *HILDA*. ACM, 2016.
[64] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.
[65] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.
[66] B. Zhou, D. Bau, A. Oliva, and A. Torralba. Interpreting deep visual representations via network dissection. *arXiv preprint arXiv:1711.05611*, 2017.
[67] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Object detectors emerge in deep scene cnns. *arXiv preprint arXiv:1412.6856*, 2014.

A GENERAL DNI PROBLEM DEFINITION

Let $\mathbb{U}$ be a set of unit groups. Similarly, users may curate a large corpus of hypotheses H and want to find hypotheses learned by a model. Users may also want to evaluate different or multiple statistical measures L to have different perspectives of the units. We present a more general definition that is amenable to optimizations across models, hypotheses, and measures:

Definition 2 (DNI-General). *Given dataset* D*, set of unit groups* $\mathbb{U}$*, hypotheses* H*, and measures* L*, return the set of tuples* $(u, h, l, s_{u,h,l}, s_{U,h,l})$ *where*

- $l(U, h, D) = ([s_{u,h,l} | u \in U], s_{U,h,l})$
- $l \in L$, $U \in \mathbb{U}$, $h \in H$

B HYPOTHESES

The following are additional examples of hypothesis functions that can be generated from finite state machines and iterative programs.

Finite State Machines: Regular expressions, simple rules, and pattern detectors are easily expressed as finite state machines that explicitly encode state logic. Since each input symbol triggers a state transition, an FSM can be wrapped into a hypothesis function that emits the current state label after reading the symbol. Similarly, the state labels can be hot-one encoded, so that each state corresponds to a separate hypothesis function that emits 1 when the FSM is in the particular state, and 0 otherwise.

General Iterators: More generally, programs that can be modeled as iterative procedures over the input symbols can be featurized to understand if units are learning characteristics of the procedure. As an example, a shift-reduce parser is a loop that, based on the next input character, decides whether to apply a production rule or read the next character:

```
initialize stack
until done
 if can_reduce using A->B      // reduce
   pop |B| items from stack
   push A
 else                          // shift
   push next char
```

Any of the expressions executed, or the state of any variables, between each `push next char` statement that reads the next character, can be used to generate a label for the corresponding character. For instance, a feature may label each character with the maximum size of the stack, or represent whether a particular rule was reduced after reading a character.

C SQL AUTO-COMPLETE INSPECTION

This section extends the scalability experiments in Section 6.3 with an analysis of the inspection results. We can use DeepBase to study what the model learns through its training process by executing a query akin to the example in Section 4.1. We train the SQL auto-completion model by performing several passes of gradient descent over the training data, called epochs. We repeat the process until the model's performance converges or starts to decrease (after 13 epochs in our case). We capture a snapshot of the model after random initialization (then the accuracy is 1.1%), 1st epoch (41% acc), and 4th epoch (45% acc), and perform neural inspection to understand what the model learned.

Figure 11 shows a few of the highest affinity hypotheses. These hypotheses correspond to fundamental SQL clauses that should be learned in order to generate valid SQL, and the model appears to learn them (rather than arbitrary N-grams) even in the first training epoch. Further, the F1 is higher for detecting the string "ORDER", which we expect is needed to learn the ordering expression `ordering_term`.

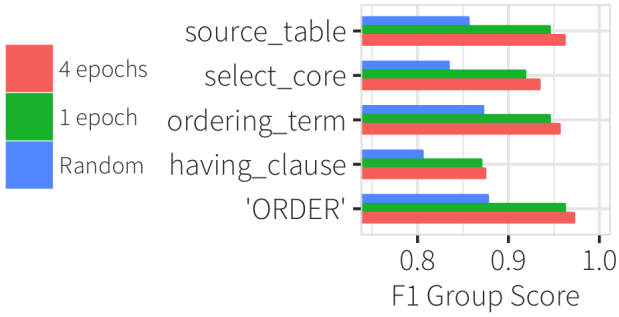


Figure 11: F1 scores for highest affinity hypotheses during training.