DeepBase: Deep Inspection of Neural Networks

Technical Report

Thibault Sellam
Columbia University
tsellam@cs.columbia.edu

Kevin Lin
Columbia University
kl2806@columbia.edu

Ian Huang
Columbia University
iyh2110@columbia.edu

Yiru Chen
Columbia University
yc3515@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 well across a range of tasks such as language translation and object recognition, it remains unclear what high-level logic, if any, they follow. Understanding this logic may lead to more transparency, 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, but those analyses require considerable manual effort. Our insight is that many of those studies follow a common analysis pattern, which we term Deep Neural Inspection. There is opportunity to provide a declarative abstraction to easily express, execute, and optimize them.

This paper describes DeepBase, a system to inspect neural network behaviors through a unified interface. We model logic with user-provided *hypothesis functions* that annotate the data with high-level labels (e.g., part-of-speech tags, image captions). DeepBase lets users quickly identify individual or groups of units that have strong statistical dependencies with desired hypotheses. We discuss how DeepBase can express existing analyses, propose a set of simple and effective optimizations to speed up a standard Python implementation by up to 72×, and reproduce recent studies from the NLP literature.

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 with, and deploy new models [7, 14, 35].

However, it is still unclear how and why neural networks are so effective [18, 41]. Does a model learn to decompose its task into understandable sub-tasks? Does it memorize training examples [66], and can it generalize to new situations? Grasping the internal representation of neural networks is currently a major challenge for the machine learning community. While the field is still in its infancy, many hope that increasing our understanding of trained models will enable more rapid experimentation and development of NN models, help identify harmful biases, and explain predictions [18]—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 for positive product reviews, then it potentially recognizes positive sentiment. Numerous papers have applied these ideas by manually inspecting visualizations of behaviors [21, 31, 50] or writing analysis-specific scripts [6, 56], and in domains such as detecting syntax and sentiment in language [31, 50], parts and whole objects from images [21, 37], image textures [6], and chimes and tunes in audio [5].

This class of analysis is ubiquitous in the deep learning literature [3, 6, 8, 28, 32, 43, 46], it is particularly well represented in neural net interpretability workshops [62], yet each analysis is implemented in an ad-hoc, one-off basis. In contrast, we find that they belong in a common class of analysis that we term ***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. These DNI analyses share a common set of operations, yet each analysis currently requires considerable engineering (hundreds or thousands of lines of code) to implement, and often runs inefficiently. *We believe there is tremendous opportunity to provide a declarative abstraction to easily express, execute, and optimize DNI analysis.*

Our main insight is that DNI analyses primarily use statistical measures to quantify the affinity between hidden unit behaviors and hypotheses, and simply differ in the specific

NN models, hypotheses, or types of hidden unit behaviors that are studied. Section 2.1 illustrates how existing DNI analyses fit this pattern [3, 31, 67]. To this end, we designed DeepBase, a system to perform large-scale Deep Neural Inspection through a declarative interface. Given groups of hidden units, hypotheses, and statistical affinity measures, DeepBase quickly computes the affinity score between each *(hidden units group, hypothesis)* pair. The aim is for DeepBase to accelerate the development and usage of this class of neural network analysis in the ML community.

Designing a fast DNI system is challenging because the cost is cubic with respect to the size of the dataset, the number of hidden units, and the number of hypotheses. Even trivial examples are computationally expensive. Consider analyzing a character-level recurrent neural net (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 of extracting these activations, storing them, and matching them with hundreds of hypotheses can be incredibly slow.

To address this challenge, DeepBase uses pragmatic optimizations. First, users provide hypothesis logic as functions evaluated over input data, which may be computationally expensive. DeepBase can cache the output of hypothesis functions and reuse their results when re-running the same DNI analysis on new models. Second, users can specify convergence thresholds so that DeepBase can terminate quickly while returning accurate but approximate scores. DeepBase natively provides popular measures such as correlation and linear prediction models. Third, DeepBase reads the dataset, extracts unit behaviors, and evaluates the user-defined hypothesis logic in an online fashion, and can terminate the moment the affinity scores have converged. Finally, DeepBase leverages GPUs—commonplace in deep learning—to offload and parallelize the costs of extracting unit behaviors and computing affinity metrics based on e.g., logistic regression.

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

- The design and implementation of an end-to-end DNI system called DeepBase, along with simple pragmatic optimizations, including caching, early stopping via convergence criteria, streaming execution, and GPU execution.
- A walk-through of how to generate hypothesis functions from existing ML libraries.
- Extensive performance experiments based on a SQL autocompletion RNN model. We show that with all optimizations including caching, DeepBase outperforms a standard Python baseline by up to 72X, and an in-RDBMS implementation using MADLib [24] by 100 – 419×, depending on the specific affinity measure.
- Experimental results using DeepBase to analyze a state-of-the-art Neural Machine Translation model architecture [33] (English to German). We compare DeepBase to existing scripts [8] and validate the results of recent NLP research [2, 56].

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 DeepBase also Convolutional Neural Networks (CNNs). See Appendix E for a results comparison with a recent system called NetDissect [67], and our prior work for more CNN and Reinforcement Learning examples [12].

2 BACKGROUND AND USE CASES

This section introduces current examples of DNI analyses and how they are implemented. These uses cases serve as the motivation for the system described in the rest of the paper. Appendix A provides a quick primer on Neural Networks (NNs) and explains the terminology. The important concept is that a NN is composed of hidden units, and when a NN is evaluated over an input record (e.g., a sequence of characters that form a sentence, or a matrix of pixels that form an image), each hidden unit performs an action that emits a behavior value (e.g., an activation, or the derivative of an activation) for each element of the record (e.g., character or pixel). We refer the reader to Appendix A for details.

2.1 Motivating Example

We use a recurrent model that performs SQL query autocompletion as a motivating example. Given a SQL string, the model can read a window of 100 characters (padded if necessary) and predict the character that follows. Technically, the neural net comprises three layers: one input layer that reads one-hot-encoded characters, one recurrent (LSTM) layer with 500 hidden units, and one fully connected layer for the final output. The analysis focuses on the recurrent layer.

The model achieve 80% prediction accuracy on a held-out test set of 1,152 queries, as compared with random guess accuracy of $\frac{1}{32}$. The 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, e.g., it learned that column references tend to follow the `SELECT`

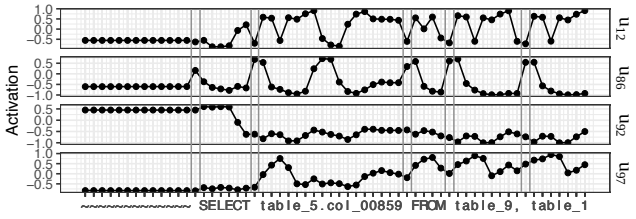


Figure 1: Activations over time for the SQL autocompletion model. What is the model learning?

keyword and that table names usually come after `FROM` but not before `LIMIT`. Ideally, we would also like to check these hypotheses across models with different architectures or training parameters, or for a specific set of hidden units.

2.2 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 can turn out to be challenging, 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, whitespaces and periods trigger the five highest activations for u_{86} in Figure 1. This DNI approach has been used to analyze image object detection [55, 57, 68], in NLP models [38] and sentiment analysis [50].

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.

In our SQL example, we could parse the query and annotate each token with the name of its parent rules (e.g., `where_clause` or `variable_name`). If we find a strong correlation between the activations of a hidden unit and the occurrence of a particular rule while running the model (e.g., "hidden unit 99 has a high value for every token inside `WHERE` predicates"), then we have some evidence that the hidden unit acts as a detector for this rule [31]. We could take the analysis further and test *groups* of hidden units: if we build a classifier on top of their activations and find that it can predict the occurrence of grammar rules with high accuracy, then we have evidence that those neurons behave *collectively* as a detector [3, 8].

Statistical analysis of hidden unit activations is a widespread practice in the machine learning literature. For instance, Kim et. al [32] use logistic regression to predict annotations of high-level concepts from unit activations. NetDissect [6] 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 [43], compare models [51] or more generally evaluate to what extent neural nets learn high-level concepts such as textures or part-of-speech tags [3, 6, 8, 46].

Inspection in Practice: Although the model interpretation literature is extremely active, the software ecosystem of tools to support Deep Neural Inspection is very limited. Authors have focused on reproducibility in the narrow sense, rather than usability, and it is likely that a ML engineer will have to implement her own version of a given approach[1].

We searched online for software used in publications that perform deep neural inspection [3, 6, 8, 28, 32, 43, 46, 50, 56, 65, 68]. Of these, six papers provided code repositories and four of them target computer vision models. In all cases, the scripts (in various languages) are tailored to only reproduce the experiment described the corresponding papers—that is, they are custom implemented for one type of model, one type of analysis, and for one type of dataset. All scripts have different APIs, and several rely on outdated/unsupported versions of deep learning frameworks (LuaTorch, PyTorch, Caffe, or Tensorflow). Popular approaches such as [65] also have "unofficial" implementations that exhibit similar issues.

Figure 2 summarizes the lines of code in each repository after manually removing non-essential code (e.g., non-analysis visualization or imported libraries). Every analysis is at least

[1]These remarks don't apply to the NN visualization community, which publishes and maintains several important software packages [47, 60].

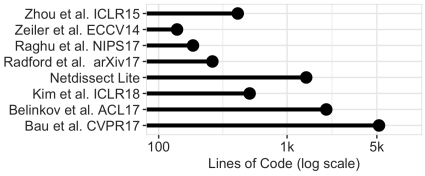


Figure 2: Lines of code (approx) from available code for papers that perform DNI.

several hundred lines of code, and in some cases thousands of lines. Although this is an imperfect measure, it provides a sense of the complexity of current DNI methods.

2.3 Desiderata of a DNI System

DNI analysis using the existing approaches is powerful and spans many domains of application. Unfortunately, each analysis currently requires custom, ad-hoc implementations despite following a common analysis goal: **the user wants to 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 to what extent the activations of each hidden unit in our SQL auto-completion model correlates with the output of 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 hypotheses 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 auto-completion, 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 [43], use a linear model to predict high-level logic from unit activations [3, 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 2.1 as the example.

Problem Setup: Let a dataset D be a $n_d \times n_s$ matrix of symbols where d_i is the i^{th} row (or record) of size $1 \times n_s$ (Table 1 presents our notations). In our SQL auto-completion example, each record is a 100-symbol vector where each symbol is a one-hot encoded character[2]. For other data types, a symbol may be an image pixel, a word, or a vector depending on the model. Records are null-padded to ensure that all records are the same size.

A model M is a vector of n_M hidden units, where u_i^M is the i^{th} unit. Logically, M(d) is evaluated on a record d by reading each input symbol one at a time; each symbol s_i triggers a single behavior $b_i \in \mathbb{R}$ from each hidden unit u[3]. Thus the *Unit Behavior* $u(d) \in \mathbb{R}^{n_s}$ is the vector of behaviors for unit u when the models evaluated over all symbols in d. Let $U(d) \in \mathbb{R}^{|U| \times n_s}$ be the *Group Behavior* for a subset of units U in a model. An example of U may be the units in the first layer, or simply all units in a model.

Each line graph in Figure 1 plots behavior as a unit's activation when reading each character in the input query. This paper reports results based on unit activations, however DeepBase is agnostic to the specific definition of behavior extracted from the model. This flexibility is important because some papers use the gradient of the activations instead of their magnitude [68].

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. In practice, those functions are either written by the user or provided in a library. There is no

[2]Each character is represented by a sparse binary vector, where a 1 at position *i* indicates that the character is set to the *i*th value from the alphabet.
[3]In the context of windowing over streaming data, the RNN model internally encodes a dynamically size sliding window over the symbols seen so far.

Notation	Description
D	A $n_d \times n_s$ matrix of symbols.
d_i	The i^{th} row in D. Called a record.
M	A model M is a vector of hidden units.
u_i^M	i^{th} hidden unit in M.
U	A group of units in a model.
$u(d) \in \mathbb{R}^{n_s}$	Unit u returns vector of behaviors.
$h(d) \in \mathbb{R}^{n_s}$	Hypothesis function h returns vector of behaviors.
n_d, n_s, n_M	Number of records, symbols/record, units in M

Table 1: Summary of notations used in the paper.

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 that it matches the size of a unit behavior.

To illustrate, a hypothesis that the model has learned to detect the keyword "SELECT" could be a Python function that emits 1 for those characters and 0 otherwise. Thus for the query `SELECT 1 FROM a`, the hypothesis would be `111111000000000`. The hypothesis behavior need not be binary, and can encode integers or floating point values as well. For instance, a hypothesis that the model counts the number of characters in an input string may return a number between 0 an 100. Further examples are given in Section 4.2.

We quantify the affinity between a group of units U and a hypothesis h with a user-defined statistical affinity measure $l(U, h, D) = (\mathbb{R}^{|U|}, \mathbb{R})$. The first element contains a scalar affinity score for each unit in the group, and the second element is a score for the group as a whole. Either element may be empty. For instance, we may replicate [31] by computing the correlation each unit's activation and a grammar rule such as "SELECT" keyword detection. Alternatively, we may follow [8] and use a linear classifier to predict the occurrence of the keyword from the behavior of all units in the first layer; 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, DeepBase provide 8 common measures (see 4.3) and leverage their approximation properties to optimize the analysis runtime (Section 5).

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.

General Problem Definition: Although the above definition is sufficient to express the existing approaches in Section 2.1, it is inefficient. In practice, developers often train and compare many groups of units, e.g., to understand what hypotheses the model learns across training epochs. We present a more general definition that is amenable to optimizations across models, hypotheses, and measures.

Let $\mathbb{U}$ be a set of unit groups defined by the user. The user may also provide a large corpus of hypotheses H, to understand which hypotheses are learned by the model. The user may also want to evaluate multiple statistical measures L to have different perspectives. With those notations, we define our problem as follows:

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*

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

4 DEEPBASE API AND OVERVIEW

This section describes our Python API, how to create hypothesis functions, DeepBase's native affinity measures, and a verification procedure to assess the quality of highly scored units. The next section describes the system design and optimization.

4.1 Python API

DeepBase is implemented in Python and exposes a Python API. We will use the API to perform two analyses using the SQL-autocompletion example: 1) compute the correlation between every unit's activations and binary hypotheses that indicate the occurrence of grammar rules (as described in Section 2.2), and 2) report the F1 accuracy of a logistic regression classifier that predicts the binary hypothesis behaviors from all hidden unit activations [3, 8]:

```
import deepbase
model   = load_model('sql_char_model.h5')
dataset = load_data('sql_queries.tok')
scores  = [CorrelationScore('pearson'),
           LogRegressionScore(regul='L1',score='F1')]
hypotheses = gram_hyp_functions('sql_query.grammar')
deepbase.inspect([model], dataset, scores, hypotheses)
```

This code loads the `deepbase` module, NN model, and test dataset. It specifies that we wish to compute per-unit correlation scores as well as logistic regression F1 accuracy

with L1 regularization. `hypotheses` is a list of binary hypothesis functions that each returns the presence of a specific grammar rule. Finally, we call `deepbase.inspect()`, which returns a Pandas data frame (i.e., table) that contains an affinity value for each model, score, hypothesis, and hidden unit:

```
model_id, score_id, hyp_id, h_unit_id, val
```

The variable `scores` points to a list of `DBScores` objects. Currently, DeepBase's standard library includes 8 scores (see 4.3) and 2 naive baselines (random class, majority class). The list `hypotheses` contains arbitrary Python functions, which output formats are checked during execution (we defined the specifications in 3).

In practice, the users will often post-process the table returned by `inspect`. For instance, they may wish to return only the top scores (e.g., to find the "sentiment neuron" in [50]), combine the results with other statistics (e.g., to reproduce Figure 2 in [8]), or group the scores by layer and count the number of hidden units with a high score (Figure 5 in [6]). This observation, combined with the fact that the intermediate and final outputs can be very large (multiple GBs for even simple cases) calls for tight integration with a DBMS. A full treatment is outside the scope of this paper, however Appendix B describes how SQL can be extended to support DNI using a new `INSPECT` clause. In addition, Section 5.1.1 describes a baseline built upon a database engine rather than the Python and Tensorflow scripts used in existing papers.

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 3) 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 that we used in our experiments. We note that the purpose of DeepBase is to simplify the use and inspection of hypothesis functions—developing appropriate hypothesis functions to answer NN analysis questions continues to be an open area of research.

Parse Trees: A common use of RNNs is language analysis and modeling. For 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 3 illustrates an example parse tree for a simple algebraic expression within

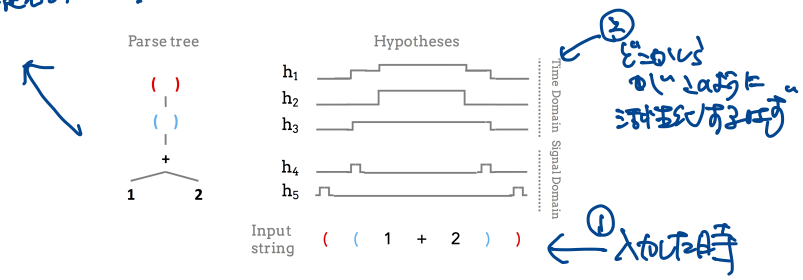


Figure 3: Example parse tree (left) and hypothesis functions. Each h_i is the behavior over each input character symbol.

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 3'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 corpus 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.

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.

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 [43], difference of means, Jaccard coefficient [67], all available in DeepBase by default. In general, DeepBase supports any UDF that takes two behavior vectors as input. Independent measures are amenable to parallelization across units, which DeepBase enables by default.

Joint Measures: compute the affinity between a group of units U and a hypothesis h, and scores for each unit $u \in U$. 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. By default, we the use the logistic regression with L1 regularization trained with SGD (we use the optimizer Adam and Keras' default hyper-parameters), and we report the F1 on 5-fold cross-validation. DeepBase also supports arbitrary Keras and ScikitLearn models, as well as a multivariate implementation of mutual information.

4.4 Verification

We note that DNI is fundamentally a data mining procedure that computes a large number of pairwise statistical measures between many groups of units and hypotheses. When looking for high-scoring units, the decision is susceptible to multiple hypothesis testing issues and can lead to false positives. Most current DNI analysis either do not perform verification (e.g., are best effort), or use one of a variety of methods. One method is to ablate the model [31, 43] ("remove" the high scoring units) and measure its effects on the model's output. 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, the 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, $\Delta_k^b = \text{act}(s_k^b) - \text{act}(s_k)$ 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, 43] (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

DeepBase is implemented in Python and Keras, however it is also possible to embed DNI analysis into an ML-in-DB system such as MADLib [24] through judicious use of UDFs and driver code. This section describes two baseline designs—MADLib-based design and the naive DeepBase design—and their drawbacks. It then introduces pragmatic optimizations to accelerate DeepBase.

5.1 Baseline Designs

5.1.1 DB-oriented Design. Using a database can help manage the massive unit and hypothesis behavior matrices that can easily exceed the main memory [64]. Also, as discussed in Section 4.1, it can be easier for users to post-process DNI results with relational operators (filtering, grouping, joining). We now describe our DB-oriented implementation that uses the MADLib [24] PostgreSQL extensions to perform DNI.

ML-in-database systems [20, 24, 36] such as MADLib express and execute convex optimization problems (e.g., model training) as user-defined aggregates. The following query trains a SVM model over records in `data(X, Y)` and inserts the resulting model parameters in the `modelname` table.

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

Note that the relation names are parameters, and the UDA internally scans and manipulates the relations. An external process still needs to extract unit and hypothesis behaviors from the test dataset and materialize them as the relations `unitsb` and `hyposb`, respectively. Their schemas `(id, unitid/hypoid, symbolid, behavior)` contain the behavior value for each unit (or hypothesis) and input symbol. This can be quite expensive. After loading, a Python driver then submits 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 [1, 13, 48] 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 can 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 6.2). This leads to our second challenge: how to efficiently evaluate hundreds, potentially thousands of units/hypotheses pairs without incurring duplicate work?

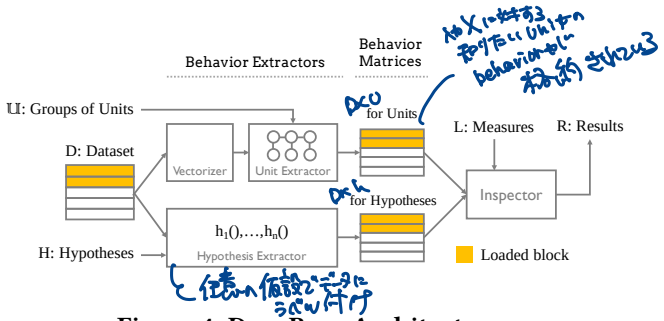


Figure 4: DeepBase Architecture.

The third challenge is that extracting the behavior matrices can be expensive [63]. 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 [9] 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.

Our experiments find that this baseline is far slower than all versions of DeepBase, and point to the bottlenecks to address in order to support deep neural inspection within a database system.

5.1.2 Naive DeepBase Design. Figure 4 presents the naive DeepBase architecture. Its major drawback is the need to excessively materialize intermediate matrices. The design will be optimized in the next subsection.

DeepBase first materializes all behaviors from the dataset D. The *Unit Behavior Extractor* takes one or more unit groups as input, and generates behaviors for each unit in each group—the assumption is that each group is a subset of units from a single model. Similarly, the *Hypothesis Behavior Extractor* takes a set of hypothesis functions as input and runs them to generate hypothesis behaviors. We concatenate the sequences together, so the extractors emit matrices of dimensionality $|D| \cdot n_s \times |\mathbb{U}|$ (for Units) and $|D| \cdot n_s \times |H|$ (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 and desired statistical measures, and computes affinity scores for each unit, hypothesis, and measure triplet.

DeepBase natively supports activation extraction for Keras models, but can be extended with custom Unit Extractors for other frameworks such as PyTorch, or to simply read behaviors from pre-extracted files. For instance, our experiments in Section 6.3 use on a custom PyTorch extractor for the OpenNMT model. Any object that inherits our class `Extractor` and that exposes the following method may be used in DeepBase:

```
extract(model, records, hid_units) → behaviors
```

The input `records` is a list of records, and `hid_units` is a list of integers that uniquely identify hidden units. The output `behaviors` is a NumPy array with one row per symbol and one column per hidden unit. The user may pass additional arguments (e.g., batch size) in the constructor of the class. Note that this is the minimal API, a few additional methods must be written to support the optimizations presented in following sections.

DeepBase 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 [63] for unit and hypothesis behaviors is a direction for future work.

5.2 Optimizations

Below we outline the main optimizations.

5.2.1 Shared Computation via Model Merging. Although affinity score measures are typically implemented as Python User Defined Aggregates, DeepBase also supports Keras computation graphs. 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. This optimization is exact, it does not impact the final scores.

DeepBase produces one composite model for each affinity measure. For a given measure's Keras model, it duplicates the intermediate and final layers for each hypothesis and enforces them to share the same input layers. Thus they share the input layer, but maintain separate outputs. 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 global loss function that averages the losses for each hypothesis. This optimization does not degrade the results: since there is no dependency between the models and their parameters, minimizing the sum of the losses is equivalent to minimizing each loss separately. We do however lose the ability to early-stop the training for the individual hypotheses, as we cannot freeze individual hidden units in Keras.

Model merging is applicable when the *scoring function* provided by the user is based on Keras (e.g., the logistic regression score in our experiments). This is orthogonal to the framework of the model to inspect—the optimization could very well support custom extractors for other frameworks, e.g., PyTorch.

5.2.2 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 sub-sample. 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, recs)→(scores, err)
```

The API takes an iterator over records `recs` as input and returns both the group and unit scores in `scores`, as well as an error of the group score `err`. Users can thus specify a maximum 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 [49]). 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. Records on disk are assume to have been shuffled record-wise. 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. We shuffle the blocks symbol-wise in-memory before running inspection. The SGD based approaches shuffle the behaviors further during training.

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.

5.2.3 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 4 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 further 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 derived from the parse tree.

6 EXPERIMENTS

Our experiments study the scalability of DeepBase, as well as its ability to generate DNI scores that are comparable to prior DNI analyses. To this end, we first present scalability experiments using a SQL auto-completion RNN model to show how the baselines, DeepBase, and its optimizations scale as we vary the number of hidden units, hypotheses, and records. We then use DeepBase to analyze a real world English-to-German translation model from OpenNMT [33], and report results from reproducing DNI analysis from Belinkov et al [8].

We provide additional experiments in the Appendix. In Appendix C, we present a set of experiments to evaluate the accuracy of DeepBase's scores. In Appendix D, we complete our SQL auto-completion scalability benchmark by commenting the results provided by the system. In Appendix E, we extend our analysis to convolutional neural nets and compare DeepBase to NetDissect, an existing system to analyze computer vision models.

6.1 Setup Overview

We ran DeepBase on two types of RNN models: the first predicts the next symbol (character) for SQL query strings generated from a subset of the SQL grammar, while the second is a sequence-to-sequence English to German translation model called OpenNMT.

Datasets: We used two language datasets: a collection of SQL queries for the scalability benchmark, and a publicly available English-to-German translation dataset for the real-world experiment[4]. 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 character that follows.

Models: The SQL use-case is based on 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: For the SQL experiment, we follow the procedure in Section 4.2 to transform parse trees 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;

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

at that point the other hypothesis functions based on the parser do not need to re-parse the input text. To increase the number of hypotheses, we also generate hypothesis functions using the signal representation. We use NLTK's chart parser [9] to sample and parse the SQL grammar. For the translation experiments, we use Part-of-Speech tagger of CoreNLP[40], which we can directly use as a hypothesis.

6.2 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[5], 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 symbol is consumed by the rule or a descendant rule). The second only returns "1" for the first and last symbol and returns "0" otherwise. This yields 190 hypotheses.

Systems: We start with the Python baseline implementation `PyBase`, then cumulatively add the optimizations described in Section 5: model merging (`+MM`), early stopping (`+MM+ES`), and online extraction (DeepBase). In addition, we measure the benefits of the GPU by comparing the model-merging baseline with a GPU (`+MM (GPU)`) and without (`+MM (CPU)`). We compare against the `MADLib` implementation, which fully materializes the behavior matrices, and then computes affinity scores using PostgreSQL native (for correlation) and MADLib (for logistic regression) functions.

We run each configuration 3 times report and the average. For each experiment, we run the smallest-scale baseline to completion, and then enforce a 30-minute timeout for larger-scale settings.

Setup: All our experiments are based on 6 Google Cloud virtual machines with 32GB RAM running Ubuntu 16.04, and 8 virtual CPUs each, where each virtual CPU is a hyper-thread on a 2.3 GHz Intel Xeon E5 CPU. Each VM includes a nVidia Tesla K80 GPUs with 12 GB GDDR5 memory. All models are based on Keras with Tensorflow 1.8. MADLib uses 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 [9] and transforming the tree into many 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 95% confidence for correlation, and error threshold of 0.01 for logistic regression (cf. Section 5.2.2).

[5]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.

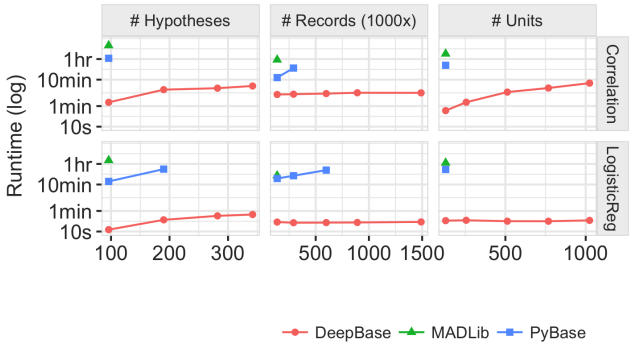


Figure 5: Runtime of MADLib and Python baselines as compared to DeepBase with all optimizations for logistic regression measure.

Comparing Baselines: Figure 5 compares the MADLib and Python baseline systems for both affinity measures (rows) as we vary the number of hypotheses, records, and hidden units in the model (columns). We also include DeepBase with all optimizations for reference.

Correlation (top row) is generally 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). Both `MADLib` and `PyBase` incur considerable full table scan and aggregation costs. Logistic regression (bottom row) is dominated by the cost to fit logistic regression models for each pair of hypothesis and unit group.

Overall, we find that `PyBase` performs faster than `MADLib` on the smallest experimental settings. We believe this is largely because of the overheads of using PostgreSQL extensions and the fact that the in-memory Python implementation of logistic regression is quite fast. DeepBase's optimizations avoids unnecessary extraction costs once all the scores have converged. DeepBase improves upon `PyBase` by 72× on average, and by up to 96×; it outperforms `MADLib` by 200× on average, and by up to 419×.

Optimization Benefits for Correlation Measure: Figure 6 reports runtimes for three variants of DeepBase for correlation. Correlation is a cheap measure to compute and is executed on the CPU. Since we use a CPU, model merging (which is an GPU-oriented optimization) is disabled. Thus we compare `PyBase`, with early stopping, and with lazy extraction.

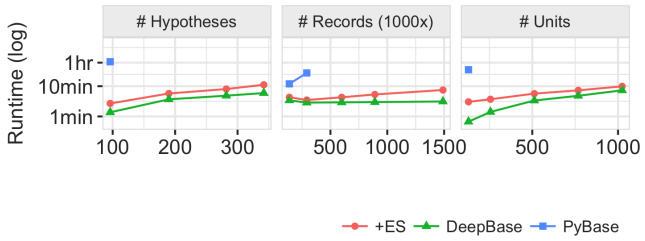


Figure 6: Runtime of DeepBase with different optimizations enabled for correlation measure.

We find that the primary performance gains are due to the early stopping optimization, while lazily extracting behaviors provides considerable, but smaller benefit. We see that lazy extraction provides a benefit as the number of records increases (middle plot), and similarly, the benefit of lazy extraction reduces as the number of hidden units increases (right plot) because the bottleneck becomes the large number of pair-wise correlation computations.

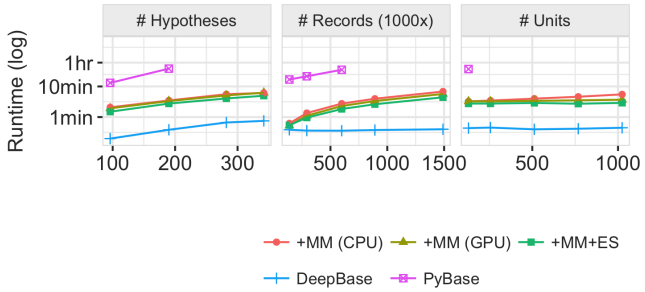


Figure 7: Runtime of DeepBase with different optimizations enabled for logistic regression measure.

Optimization Benefits for Logistic Regression Measure: Figure 7 reports the results when adding optimizations based on early stopping, lazy extraction as well as GPU-based optimizations. We see that model merging (`+MM`) provides a considerable benefit by reducing the number of logistic regression models that need to be trained for each hypothesis. The benefits of using a GPU appear for models with many hidden units. We find that early stopping (`+MM+ES`) does not provide any speedup because materializing the behavior matrices is a large bottleneck; adding lazy extraction (DeepBase) reduces the runtime by 6× on average and by up to 11× as compared to `+MM+ES`.

Runtime Breakdown: Figure 8 shows the cost breakdown by system component: the hypothesis and unit extractors, and the inspector. The `+MM+ES` 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.

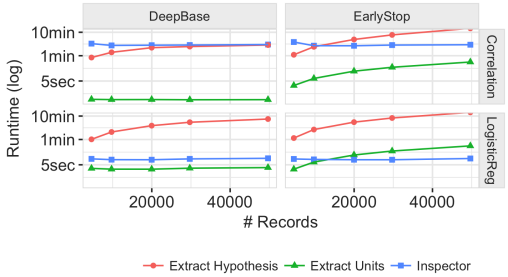


Figure 8: Runtime breakdown of extraction and inspector costs for correlation and logistic regression.

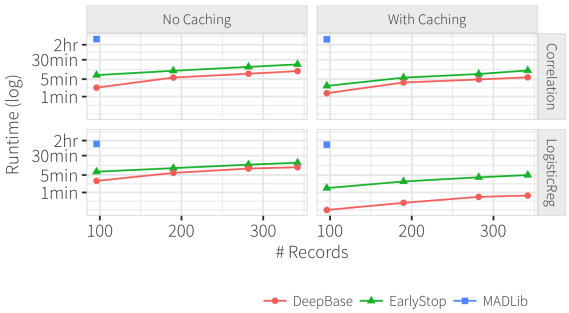


Figure 9: Runtime comparing the effects of cached hypothesis behavior.

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 9 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 converges to ≈ 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×.

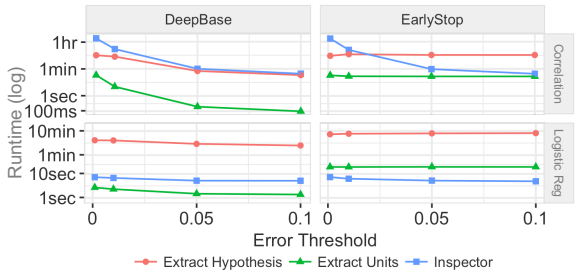


Figure 10: Runtime when varying error threshold for early stopping. Note different Y-axis scales.

Sensitivity to Error Threshold: Figure 10 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: `+MM+ES` 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, joint measures such as logistic regression are computed for each unit-group are dominated by extraction costs. Early stopping improves independent measure performance, while online extraction enables DeepBase to run nearly independently of dataset size. DeepBase outperforms the MADLib and Python baselines by two orders of magnitude and took at most* 10.3 *minutes for the slowest setting.*

6.3 Neural Machine Translation

We reproduce the analysis of existing studies by applying DeepBase on an public English-German translation model. We first replicate the methodology of Belinkov et al. [8] and verify that our results are consistent with those returned by their scripts[6]. We then broaden the analysis and show that we can make observations that are similar in spirit to those presented in related work [2, 56] with only a few queries.

In addition to those results, we present a comparison of DeepBase with NetDissect [6], a recent interpretation method for convolutional Neural Nets in Appendix E.

6.3.1 Comparison with Belinkov et al. Belinkov et al. have shown that sequence-to-sequence neural translation models learn part-of-speech tags as a byproduct of translation. They train a classifier from the encoder's hidden layer activations and observe that they can predict the tags of the input words with high accuracy. This section replicates this analysis.

We ran DeepBase and the baseline scripts on the same datasets, using the same 46 POS tags and the same score function. We used an English-to-German translation corpus available online[7], annotated with the Stanford CoreNLP tagger[8]. We use 4,823 sentences for training, 636 for validation and 544 for testing (each sentence contains 24.2 words in average). Our score is the precision of a multi-class logistic regression model trained on the encoder's hidden unit outputs, as described in the the original paper. We limit to 35 training epochs, with a patience (i.e., number of epochs without improvement before early-stopping) of 5, the scripts' default.

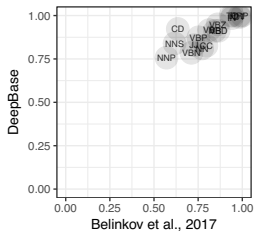


Figure 11: Precision for every tag, computed by Belinkov et al. and by DeepBase. We filtered out the tags that cover less than 1.5% of the data. The sample Pearson correlation is r=0.84.

Experimental Setup Differences: An important difference between the two approaches is that they run in different environments. Belinkov et al. supports `seq2seq-attn` models, a legacy library that runs on top of Torch for Lua. Unfortunately, we found no way to import `seq2seq-attn` models into Python. After consulting the library's authors, we chose to support `seq2seq-attn`'s successor, `OpenNMT`, which runs in PyTorch for Python. We extended DeepBase extraction library to support this model. Because of this incompatibility, we run each system on a different model. For DeepBase, we use a public model from OpenNMT, a sequence-to-sequence model with 2 LSTM layers of 500 hidden units each, available online[9]. For Belinkov et al., we use a custom model trained with `seq2seq-attn` and strived to replicate OpenNMT's setup as closely as possible: we used the same NN architecture and training data, and similar training parameters. We expect the results of the analysis to be strongly correlated, but not identical, because the training and experiments environment is not identical, and the two models are implemented differently.

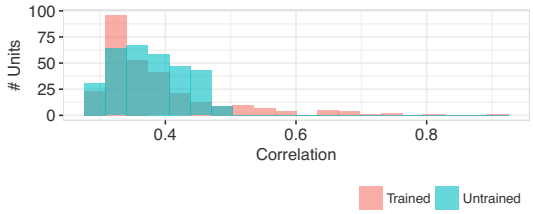
Results: Figure 11 presents the affinity scores for every POS tag computed by the two approaches. The strong correlation (0.84) between the approaches suggest analysis consistency.

Belinkov et al.'s scripts ran in 1 hour and 10 minutes, while DeepBase ran in 55.1 minutes. Aside from the differences in frameworks, we explain the difference as follows. Belinkov et al. modify the NMT model in-place by freezing the weights of the translation model and inserting the POS tag classifier directly in the encoder. Since the dataset is relatively small, they must make many passes over the data before the classifier converges (at least 35), running the full translation model each time. By contrast, DeepBase extracts the activations once (this takes 38.3 minutes) and makes the subsequent passes on the cached version (7.4 minutes), which amortizes

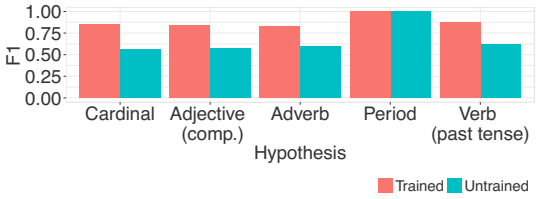
[6] github.com/boknilev/nmt-repr-analysis
[7] drive.google.com/file/d/0B6N7tANPyVeBWE9WazRYaUd2QTg/view
[8] stanfordnlp.github.io/
[9] opennmt.net/

the activation extraction time. Note that none of DeepBase's optimizations apply to this use case: model merging is irrelevant because there is only one hypothesis (the function is not binary, it returns one the 42 distinct POS speeches at each step), and early stopping/lazy materialization does not help because the dataset is small.

Takeaways: *DeepBase can easily express the analysis presented [8], its scores are consistent with the scripts provided by the authors and its runtime is competitive.*



(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.

Figure 12: Deep neural inspection on OpenNMT translation model. Results compared against an untrained OpenNMT model.

6.3.2 Additional Results. We now broaden our analysis, and show that we can replicate and verify the conclusion of recent work [2, 31, 56] with only a few additional queries. For the remainder of those experiments, we add 7 hypotheses for phrase-level structures (NP, VP, PP, etc.) to the POS presented previously.

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 12a).

Encoder Level: We then use logistic regression with L2 regularization to study all 1000 units in a trained and untrained model (Figure 12b). We first confirm recent work showing that model architecture can act as a strong prior [2]. 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.

Unit groups: We now inspect each layer separately, and use Logistic Regression with L1 to identify unit groups with non-zero coefficients. Previous work [8, 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 ".".

Takeaways: *DeepBase expresses and computes results that are consistent with analyses in recent NLP studies that seek to understand neural activations in machine translation [8, 56, 56], with orders of magnitude less engineering effort. 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. Our declarative API lets users easily inspect and perform different analyses by comparing different models at the granularity of individual, groups of, and layers of hidden units.*

7 RELATED WORK

Interpreting Neural Networks: Many approaches were proposed for model interpretation. Section 2 reported three methods: visualization of the hidden unit activations [25, 31, 60], saliency analysis [21, 38, 55, 57, 61, 68] and statistical neural inspection [3, 6, 28, 32, 43, 46]. These methods are common in the neural net understanding literature, and motivate the design of DeepBase. Other approaches 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) [44, 45]. However, most of the literature focuses on computer vision, and the process relies heavily on human inspection. Another form of analysis 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 [65]. 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 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 [47], which bundles feature inversion, saliency analysis, with visualization into a larger grammar. Lucid has similar goals as DeepBase, however it focuses on images and still relies on manual analysis.

Visual Neural Network 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 [11], 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 data point so that users can interpret the rationale for a specific model decision [4, 52]. Other approaches modify the machine learning model so that predictions are inherently interpretable, such as PALM [34]. 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 [10, 16, 20, 24, 35]. Recent projects such as ModelDB [64] and Modelhub [42] 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 [63] can be used in conjunction with DeepBase to manage the process of extracting and caching unit activations.

8 CONCLUSION AND DISCUSSION

Programming frameworks for deep learning have enabled machine learning to impact a large set of applications. Yet efforts to understand and inspect the behaviors of hidden units in NN models are largely manual and one-of, and require considerable expertise and engineering effort. Better NN analysis tools will contribute to a better understanding of how and why neural networks work. Towards this goal, this paper defined *Deep Neural Inspection* to characterize existing inspection analyses, and presented DeepBase, a system to quickly inspect neural network behavior through a declarative API. With a few lines of code, DeepBase can express a large fraction of existing deep neural inspection analyses, but improves the analysis run-times by up to 72× as compared to existing baseline designs. Further, we reproduced results consistent with prior NLP research [8] on real-world translation models [33].

We intend to extend DeepBase with more statistical measures and deeper integration with GPUs, support distributed environments, and apply DeepBase to a broader range of applications (e.g., bias detection, reinforcement learning). 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 decompose NNs into smaller components, enforce activation behavior for unit groups, and ultimately open up NN black boxes.

9 ACKNOWLEDGMENTS

We acknowledge: NVIDIA Corporation for donating a nVidia Titan XP GPU, the Google Faculty Research Award, the Amazon Research Award, and NSF grants 1527765 and 1564049. Thanks to Yonathan Belinkov for his code, guidance, and advice [8], and Yiliang Shi for advice.

REFERENCES

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Y. Belinkov, N. Durrani, F. Dalvi, H. Sajjad, and J. Glass. What do neural machine translation models learn about morphology? *arXiv*, 2017.

[9] S. Bird and E. Loper. Nltk: the natural language toolkit. In *ACL*, 2004.

[10] 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.

[11] 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.

[12] Y. Chen, Y. Shi, B. Chen, T. Sellam, C. Vondrick, and E. Wu. Deep neural inspection using deepbase. In *NIPS LearnSys Workshop*, 2018.

[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] J. Markoff. 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.

[42] 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.

[43] A. S. Morcos, D. G. Barrett, N. C. Rabinowitz, and M. Botvinick. On the importance of single directions for generalization. *arXiv*, 2018.

[44] 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.

[45] 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.
[46] 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.
[47] 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.
[48] A. Paszke, S. Gross, S. Chintala, and G. Chanan. Pytorch, 2017.
[49] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.
[50] A. Radford, R. Jozefowicz, and I. Sutskever. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*, 2017.
[51] M. Raghu, J. Gilmer, J. Yosinski, and J. Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In *NIPS*, 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. *arXiv:1610.02391*, 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] J. H. U. Tal Linzen, T. U. Grzegorz Chrupała, and T. U. Afra Alishahi. Proceedings of the 2018 emnlp workshop blackboxnlp: Analyzing and interpreting neural networks for nlp. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Association for Computational Linguistics, 2018. https://blackboxnlp.github.io/.
[63] 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.
[64] 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.
[65] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.
[66] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.
[67] B. Zhou, D. Bau, A. Oliva, and A. Torralba. Interpreting deep visual representations via network dissection. *arXiv preprint arXiv:1711.05611*, 2017.
[68] 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 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 $\frac{\delta u_i}{\delta s_t}$).

B SQL EXTENSIONS VIA INSPECT

This section describes how DNI can be integrated into a SQL-like language as a new INSPECT clause. We introduce a separate clause because 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 INSPECT operator needs to flatten the groups into a single relation before sending to other relational operators.

NNs as Relations: 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 metadata attributes—e.g., unit layer, training epoch, or the source of the hypothesis function—for filtering and grouping the hidden units.

INSPECT: 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 of each unit $u \in U$; 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).

SQL Integration: Users often inspect models as part of debugging. We now show the full query syntax to express the DNI analysis from the motivating example in Section 2.1. The 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
```

The user can easily change the layer or types of models to inspect with slight modifications of the query, or further join the output with other analysis queries:

C ACCURACY BENCHMARK

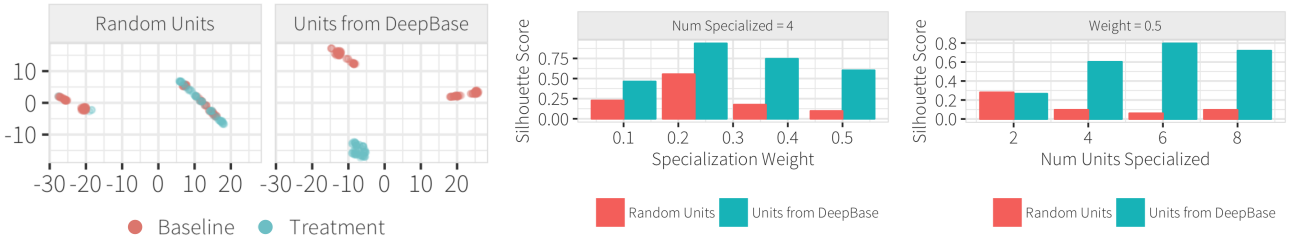
This set of experiments attempts to assess whether the hidden units that DeepBase scores highly are indeed correct. To this end, we first present an accuracy benchmark to study DNI under conditions when we "force" parts of a model to learn a hypothesis function.

Dataset: We generated a dataset by sampling from a Probabilistic Context Free Grammar (PCFG). The dataset (used in [59]) consists of strings such as `0(1(2((44))))` where a digit representing the current nesting level may precede each balanced parenthesis (up to 4 levels). Its grammar consists of the same production rule $r_i \rightarrow i\ r_i \mid (r_{i+1})$ for $i < 4$ nesting levels, along with a terminating rule $r_4 \rightarrow \varepsilon \mid 4\ r_4$ for the 4^{th} level.

Setup: 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.

Results: Figure 13a shows an example clustering of the change in activations between the baseline and treatment perturbations (colors). The hypothesis function used to inspect



(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 13: Verification results for parentheses detection hypothesis function.

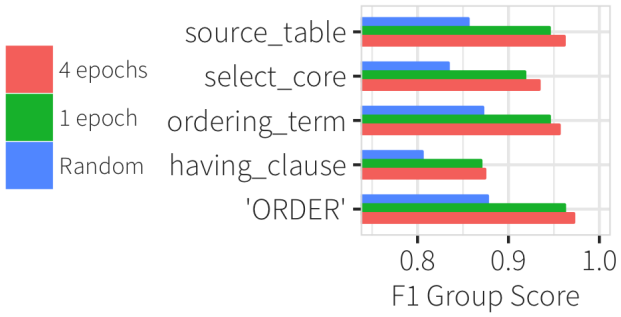


Figure 14: F1 scores for highest affinity hypotheses during training.

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 13b summarizes the cluster separation using the Silhouette score [53] and shows higher separation than random units across all weights. Similar results are shown when varying the number of specialized units in Figure 13c.

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.*

D SQL AUTO-COMPLETE INSPECTION

This section extends the scalability experiments in Section 6.2 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 B. 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 14 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`.

E DEEPBASE FOR CNNS

NetDissect is a recent DNI tool developed specifically for CNN models [6]. It detects groups of hidden units which

act as object and texture detectors (e.g., "the hidden units or layer 2 channel 12 activate specifically for chairs"). To do so, it runs the CNN models on images with pixel-level annotations, and checks which hidden units have activations that correlate with the occurrence of the labels. The affinity measure of the Intersection over Union (i.e., Jaccard similarity), after discretizing the hidden unit activations with quantile-binning.

We replicated this experiment with DeepBase on over 10K images from a corpus of annotated images provided by NetDissect's authors and designed specifically for this purpose, the Broden dataset[10]. We compare our results with those returned by a public version of NetDissect[11]. We used a pretrained VGG 16 model, trained on ImageNet data[12]. Figure 15 presents the result of the analysis. We find that DeepBase's scores are strongly correlated with NetDissect's, which shows that DeepBase's declarative interface can express the analysis and the system can produce consistent results. We explain the difference in scores by the fact that several components of the pipeline are non deterministic (among others, the online quantile approximation algorithm and the image up-sampling algorithm used to align the masks with the activations) and environment differences.

[10]http://netdissect.csail.mit.edu/data/broden1_227.zip
[11]http://netdissect.csail.mit.edu
[12]http://netdissect.csail.mit.edu/dissect/vgg16_imagenet/

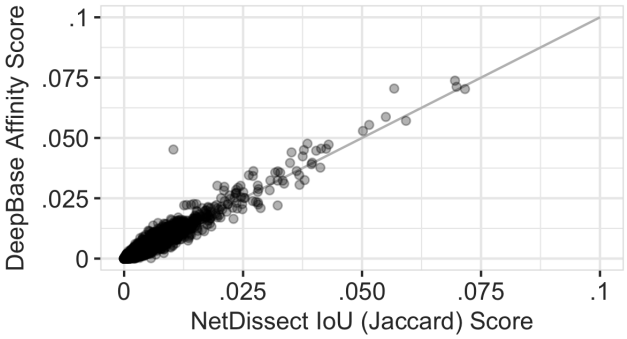


Figure 15: Comparison of NetDissect and DeepBase inspection scores.