Searching for Meaning in RNNs using Deep Neural Inspection

Kevin Lin, Eugene Wu
Department of Computer Science
Columbia University
{kl2806,ew2493}@columbia.edu

Abstract

Recent variants of Recurrent Neural Networks (RNNs)—in particular, Long Short-Term Memory (LSTM) networks—have established RNNs as a deep learning staple in modeling sequential data in a variety of machine learning tasks. However, RNNs are still often used as a black box with limited understanding of the hidden representation that they learn. Existing approaches such as visualization are limited by the manual effort to examine the visualizations and require considerable expertise, while neural attention models change, rather than interpret, the model. We propose Deep Neural Inspection to *search* for neurons based on existing interpretable models, features, or programs.

1 Introduction

Recurrent neural networks (RNNs) are widely-used to model sequential data across a variety of tasks including machine translation (Sutskever et al. [2014]), speech recognition (Graves et al. [2013]) and language modeling (Mikolov et al. [2010]). Despite their effectiveness, RNNs still maintain a reputation for being black boxes] as they can use up to millions of parameters and learn complex dependencies between inputs and outputs that vary with time. The process of interpreting different parts of the RNN continues to be a laborious, primarily manual process.

Visualization is a poweful tool to understand characteristics of a RNN model. Karpathy et al. [2015] identified RNN neurons that track higher-level characteristics such as quotations or a limited form of counting by rendering individual neuron hidden states and interpreting the visualizations. Similarly, visualizing model state (Wattenberg et al. [2016]) and neuron outputs (Yosinski et al. [2015]), particularly in CNN models, helps show developers what the "neuron wants to see". Interactive tools that visualize activations and state help developers interactively find individual neurons (Strobelt et al. [2016], Carter et al. [2016]). Ultimately, these approaches are bottlenecked by an expert that must manually interpret visualizations or hundreds or thousands of neurons, and automated approaches are desirable.

We propose to automate the process of identifying interpretable neurons in RNN models. Our primary observation is that many RNN models are used in long-established domains where many interpretable models, rules, and features already exist. For instance, given a parse tree of an input string, it is natural to ask if any neurons are learning components of the parse tree (e.g., addition expression, or nouns). Identifying such neurons can shed light on the structure that the model is learning.

In the rest of this paper, we describe Deep Neural Inspection as a way to *query* a RNN using domain-specific knowledge. Section 3 describes how we model existing knowledge as labeling functions (called *features*) and quantify the query relevance of a given neuron.

It is often not clear how to retrofit existing models and how to represent them as labels; Section 4 describes how to generate features in language model settings from parse trees as well as context-free grammars. The technique is very simple to use, and we show how Deep Neural Inspection can automatically find neurons with interpretable roles in Long Short-Term Memory networks (LSTMs) trained on languages of varying complexity.

2 Background

This section reviews major current approaches that tackle model interpretation—visualization, surrogate models, attention and model modification—and places Deep Neural Inspection into context.

Recurrent Neural Networks: While Recurrent Neural Networks (RNNs) have historically been known to be natural models for sequential tasks, they have also proven to be difficult to train due to the exploding or vanishing gradient problem (Bengio et al. [1994]). In recent years, a number of architectural variants have been introduced that aim to address this problem. One particularly popular variant is the Long-Short Term Memory network (LSTM) (Hochreiter and Schmidhuber [1997]). Since then, a large number of LSTM variants have been proposed, among which some such as Gated Recurrent Units, Recurrent Highway Networks and Hierarchical Multiscale LSTM's have recently achieved impressive results on a number of problems (Greff et al. [2016], Cho et al. [2014], Zilly et al. [2016], Chung et al. [2016]).

Despite these successes, interpretability of the features recurrent networks extract is relatively unexplored and how the internal state is propagated across time in regards to these features remains largely unknown. Hermans and Schrauwen [2013] explored the temporal effects of input perturbations as well as long-term interactions through an analysis of parenthesis closing. Recently, it has been shown that a single layer LSTM architecture was able to learn a unit that correlates well with sentiment despite not being trained to handle this task (Radford et al. [2017]). In this work, we choose to focus on classic LSTM architectures and explore the features learned by comparing the states of the network against features derived from the abstract syntactic structure of the text.

Visualization: Another approach to interpreting neural networks is through interactive visualization. There have been efforts in developing software tools that provide a interactive interface for exploring activation patterns and inspecting models. Tzeng and Ma [2005] developed a visualization system for interpreting feedforward neural networks. Visualizations of the highly successful convolutional neural network architectures (Zeiler and Fergus [2014], Kahng et al. [2017]), as well as RNN models (Karpathy et al. [2015], Strobelt et al. [2016], Rong and Adar [2016]) hav been proposed. Although visualizations are highly effective for a very small set of neurons, *finding* the neurons to visualize continues to be a challenge.

Intepreting Models: Augmenting neural networks with neural attention has allowed models to focus on a subset of the information that they are given for instance in neural translation (Bahdanau et al. [2014]). These approaches extend interpretability by showing the aspects of the data that they are focusing on. However, our approach differs in that we do not change the training of the model and instead focus on interpreting the hidden states after training.

Another approach is to train simple surrogate models to learn features (Ribeiro et al. [2016]) or training data (Krishnan and Wu [2017]) that are related to a given test prediction. These approaches can help provide intuition to complement Deep Neural Inspection.

Linear Probes: The closest work to our approach is the work on linear probes Alain and Bengio [2016], in which the utilization of linear classifiers that use the hidden state of a given intermediate layer was studied. The classifier accuracy is used to visualize the state of the network, either at test time, or at epochs during training. Deep Neural Inspection uses similar approach for measuring the “predictiveness” of a given neuron in the network, however the prediction is of features in the feature library.

3 Interpreting Neuron Function

This section introduces the conceptual framework for interpreting neurons in recurrent neural network models by leveraging existing libraries of intepretable features. The discussion is grounded in the context of RNN-based language models.

3.1 Setup

RNNs: RNNs are a popular architecture for sequential input as they take in data sequentially and their connections create directed cycles that allow information to persist. RNNs take in a sequence of vectors $x_1, \ldots, x_n$ one by one and keeps a hidden state vector h_t at time step t. The hidden states are an internal representation of the features of the input data that contain information about long-distance relations that is useful for a variety of tasks such as machine translation or text classification. At each time step t, the RNN learns a function over x_t and the previous hidden state h_{t-1} to produce the current hidden state h_t. x_i is generated by mapping the input symbol w_i (a character in this work) to a vector using e.g., hot one encoding.

In short, the model predicts the probability of the next data point given the previous points in the input sequence: $p(w_{t+1}|w_1, \ldots, w_t) = \text{softmax}(Wh_t + b)$

Interpretable Features: Neural network models are commonly used as a replacement or in the context of a well-established application domain for which years or decades of manual effort have been put into developing well-understood rules, models, or feature extractors that have been supplanted by deep neural networks. For instance, LSTMs are commonly used for language translations, a domain where numerous parsing models have been developed. Thus, it is natural to ask: "what characteristics of existing parsing models is the LSTM learning?", or "are any neurons learning particular sentence structures?".

To this end, we model interpretable domain knowledge as a *feature library* $F = \{f_1, \cdots, f_m\}$, where feature $f_i(\{w_1, \cdots, w_n\}) = \{l_1, \cdots, l_n\}$ generates a label $l_i \in \mathbb{R}$ for each input point w_i. Binary features generate binary labels to represent concepts such as "is this character part of a noun?" or "is this charecter within a quotation?". Continuous features can express concepts such as "level of parenthesis nesting" or forms of counting; their values are normalized to $[0, 1]$.

The benefit of this general description is that it encompases exsiting prediction models, feature extraction functions, detection rules, as well as imperitive functions written in e.g., Python. Section 4 will describe an automated method of generating features from language parsers as well as parse trees.

3.2 Finding Predictive Neurons

We view Deep Neural Inspection as a search problem: each feature is a query and the task is to identify the neurons that are predictive of the feature. We model query relevance of a neuron by using its ability to predicte the feature's output label. Borrowing ideas from Alain and Bengio [2016], we use a linear classifier/regression model for binary/continuous features to predict the label given the current hidden state as input.

The linear model is trained by executing the trained RNN and a candidate feature f_i over a test corpus $W^t = \{w_1^t, \cdots, w_n^t\}$. For the j-th neuron in a layer, we log its hidden state $h_{t,j}$ for time $t = 1, \ldots, n$. Each pair $(h_{t,j}, l_t)$ is an input and output example used to train an SVM with linear kernels for discrete features and OLS for continuous features, and used default settings using sci-kit learn's SVM package.

The model's 20-fold cross-validation accuracy provides a query relevance score $s_{i,j}$ for a feature f_i and neuron h_j. We then return the top k predictive neurons for a feature above a minimum prediction value.

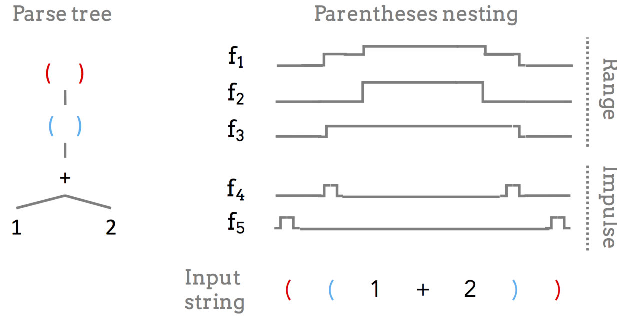


Figure 1: Example parse tree (left) and parentheses nesting feature functions. Each row corresponds to character labels for the corresponding input character. `range` is a time domain representation that activates for all characters within the matching paretheses, while `impulse` only activates at the starting and ending parentheses.

4 Generating Interpretable Parsing Features

Although numerous language-based models, grammars, parsers, and other information already exist, many do not fit the feature library abstraction described above. For example, parse trees (Figure 1) are a common representation of an input sequence that characterizes the roles of different subsequences of the input. What is the appropriate way to transform them into parts of a feature library? For example, parsers that rely on formal context free grammars are written as imperative code—how can the decisions during the parsing process be translated into a feature library?

This section describes how we map the parse tree into two types of feature representations that we call `range` and `impulse`. We also sketch an approach to generate features from imperative parsing code. These are meant to serve as illustrations of how features can be easily generated.

Parse Tree Featurization: Context-free Grammars (CFGs) are defined by a set of production rules that describe all possible strings in the language Hopcroft et al. [2006]. A rule is composed of terminals—strings that appear in the language—and non-terminals—variables used during parsing. For instance, the following rules defines simple arithmetic expressions containing single digit numbers, parentheses and the symbols $+-*/$. The non-terminal e represents an expression:

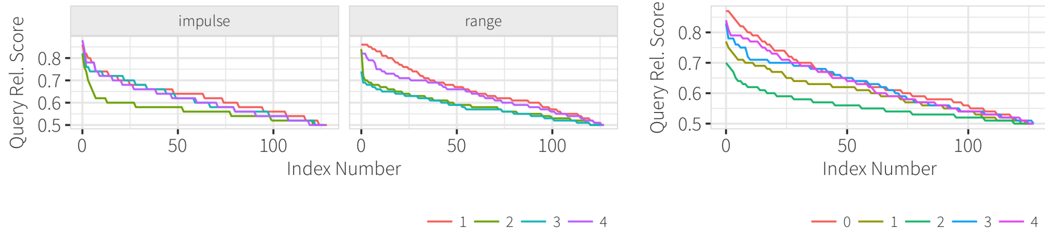
$$e \rightarrow \backslash\mathtt{d} \qquad e \rightarrow e - e \qquad e \rightarrow e + e$$
$$e \rightarrow (e) \qquad e \rightarrow e * e \qquad e \rightarrow e/e$$

Figure 1 illustrates an example parse tree for the input text `((1+2))`, which contains a single add expression within two parenthesis nestings. 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 to several `range` and `impulse` features. To illustrate, the red root in Figure 1's parse tree corresponds to the outer () characters. It can be represented as two types of features: a time-domain representation (`range`) that activates throughout the characters within the parentheses (f_3), or an `impulse` representation that activates at the beginning and end of the parentheses (f_5). Similarly, f_2 and f_4 represent the output of feature functions for the inner blue parentheses. Finally, f_1 is a composite of f_2 and f_3 that accounts of the nesting depth for the parentheses rule.

General Featurization: We note that most iteration procedures over the input symbols can be featurized to understand if neurons are learning characteristics of the procedure. As an example, a shift-reduce parser (Aho et al. [1986]) is a loop that, based on the next input character, decides whether to apply a production rule or read the next character:

```
initialize stack
```



(a) Lines correspond to nesting level.

(b) Lines correspond to the digit value.

Figure 2: Sorted neuron query relevance scores for parentheses (left) and digit value (right). While most neurons are marginally prediction, a few are highly predictive.

```
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.

5 Experiments

Our experiments seek to understand 1) whether the predicted neurons indeed affect the model's end-to-end predictive capacity in relation to its predicted feature function, and 2) how Deep Neural Inspection can help inspect and better understand the learning characteristics of RNN-based language models over languages of varying complexity. For all experiments, we train a 1-layer LSTM with 128 hidden units, with tanh activation, using Keras (citechollet2015keras). We use three language datasets based on context-free grammars of varying complexity and show that the neurons identified by Deep Neural Inspection have a quantitative and qualitative effect on the models.

5.1 Parentheses

We use toy language of balanced parentheses; a digit preceeds each parentheses nesting (up to 4), thus the model should maintain the current nesting level to successfully predict this digit value (Strobelt et al. [2016]). The production rules are shown below, and the dataset is generated from the grammar:

$e_0 \rightarrow 0e_0|(e_1)$ $\quad e_1 \rightarrow e_1 1|1e_1|(e_2)$ $\quad e_2 \rightarrow e_2 2|2e_2|(e_3)$

$e_3 \rightarrow e_3 3|3e_3|(e_4)$ $\quad e_4 \rightarrow 4e_4|\epsilon$

Figure 2 shows the query relevance scores of each neuron for the parentheses production rules (using both `range` and `impulse` representations), and the digit value of the nesting level (0 means no nesting). Many neurons are highly predictive for the first and deepest levels of nesting, however far fewer neurons appear to predict the intermediate levels. The top neurons for different nesting and representations are not correlated with each other, suggesting that the features are learned by *different* neurons.

5.2 Arithmetic

We now use the slightly more complex arithmetic grammar in Section 4, with input data generated from the grammar. Figure 3 shows query relevance for each production rule. The `range` representation has higher relevance scores than `impulse` representation, and some neurons appear to learn digits and matching parentheses very well (≈ 90% SVM test

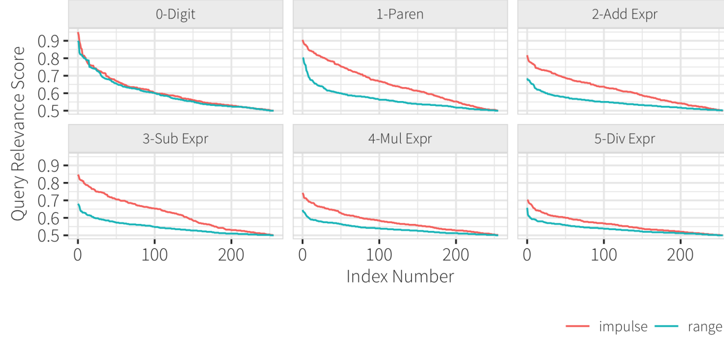


Figure 3: Sorted list of neuron prediction accuracies for each production rule. Different neurons appear to learn both `range` and `impulse` representations; digits and parentheses appear to be learned better than the $+-*/$ expressions.

		1	5	10
Digit Range	Top K	8+3)*(6*7/(5+6))*(1*0/(9-9)), (5-9+8*0)*(6*7- 5+2)*(	+3+5+3)/*/////*/*/*(1*7 ///*/*/*/*1/8/85*5/5*9/, /(6/779/90/	888897907)979779977 98070807987)9779994 378800)977
	Rand K	))*(8+0)+(9+7)*(8+8)*(0- 7)/(0+8), (5*9*(3+7)+3+3+3*	9-2), (8-2)/9/9*(6-2), (1/9*(2-2)*(9-2-9)/(2-2- 9-2, (2-2-9-9	/(6+9,,(6,6669999, (6,2669669,,,,,,,(6*6*9* 9,(6+669*99996
+Expr Range	Top K	1+1+1+1+1-)-2+1 /1*1/(1+1+1+1-1+1- 1+1-1+1+1+1+1-1+1	1+5)/(7-1)*(0+6)/(3+1- 7*4-8*5-1*8), (0*5+ 9*2)/(4*2-6/5)-(9-6	)/0*6+8-8-4/4-6/4-7-9- 4+0-0/3, 6/7*6/2*(3/2- 4+2)+9/2/(8-0)*5*6/8
	Rand K	-9)*(7-2-7*9)-0*6*3 *9+1/8*(7+0), (3+7+0- 1)*(2*7+4 +1)+(0+4	3/(3*5+8/4), 1*8+8- 0+0+8+9+2-3/6+3+ 9+9-2+9-4, (2+2-	1*1+4-8+3/7*(7-2)+2 /1*(3+9)+(9+7)*(1+3), 8*1+7+8+8+4+2+2+(8

Figure 4: Generated text when fixing the top k or random k neuron state to 1.

accuracy). In contrast, the binary expressions—particularly $*$ and /—have low scores across all neurons.

Figure 4 shows text generated by setting the hidden state of $k \in \{1, 5, 10\}$ neurons to 1, for the top k neurons found for the digits or + expression features using `range` representation. We see that increasing k for digits directly increases the probability of generating a digit in the output, noticably more than setting random neurons. We see a similar effect for + expressions, however because the query score in Figure 3 decreases rapidly, increasing k beyond 1 does not increase the number of + expressions in the output.

To understand the neurons' end-to-end impact, we measure the RNN model's predictive sensitivity to the top k neurons for a given feature by altering their outputs during test prediction. For a given feature function f_i, we alter the model in two ways: $\texttt{Rm-Neuron}_k$ forces their hidden state to 0; $\texttt{Oracle}_k$ forces their hidden state to have the same pattern as f_i.[1] We report the test accuracy for predicting the value of the nesting digit.

Figure 5 plots the prediction accuracy as we vary k for the digit feature. We find that removing the top 1 neuron does not have a large effect since multiple neurons are predictive of the nesting, whereas removing the top 5 and 10 have significantly more impact on the accuracy with respect to the feature, whereas dropping random neurons exhibited a minor effect. This suggests that the neurons identified using Deep Neural Inspection are indeed

[1] In our experiments, `Oracle` sets the hidden state by linearly extrapolating the feature's label to the range $[-1, 1]$. In practice, we found that setting this value may require tuning, since a neuron may activate when its hidden state is -1, however we do not explore this in the paper.

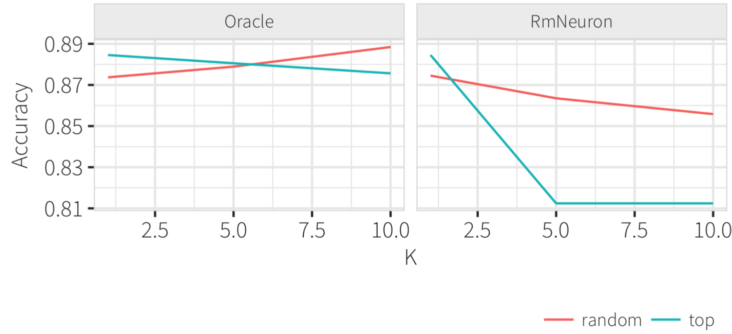


Figure 5: End to end accuracy for predicting the type of the next character (parentheses, digit, operator, et) after dropping k neurons, or setting to the feature's label.

Top 1

```
 if (irq->dev > 1 > 0) {
          if (tarlet_cpu(pid) || (unsigned long) irq) && i < ns->idle_int) && uelound_node_node) ==
'\n' " kdb_enable_name " interval.trive.cn.....
                    * define things.  if the internal merialilly.  if the rouncenstling that the inter to
storents
```

Top 10

```
if (kernel_lock_irqs_read_lock(&reques_ret_node)) && i < param_bits_opty_lock) {
          if (task_task_inter(task)) {
             printk_type_lock(p, param_lock);
          if (ret)
             return -ein_param_enable();
          warn_on_once(&se->name, num);
          return null;
          }
```

Figure 6: Generated text when fixing the top k neurons for parentheses to 1, on the Linux kernel.

related to the model's ability to learn digits. We find that knowing the feature label (left facet) did not improve this measure measurably.

5.3 Linux Source Code

We now use the C-99 grammar and train using the Linux kernel source code. We use an open-source C parser (Bendersky [2016]) and report results using `range` and `impulse` features for 10 parse tree nodes of varying complexity—simple structures such as statement terminating semicolons, to function declarations, to matching braces for multi-line expressions. For lack of space, we report the output of the generative process when manipulating the top 1 and 10 neurons for the parentheses `range` feature representation. We find that there are noticably more matching parentheses that are generated in the output, and this observation holds for the digit, function declaration, and statement terimantor features.

6 Conclusion and Discussion

In this paper, we described the Deep Neural Inspection technique to query a neural network for neurons that learn existing domain-specific knowledge, and illustrated procedures for turning existing knowledge such as parse trees and parsers into features to query the model.

This work was inspired by MRI analysis in neuroscience, which seeks to identify brain regions that activate based on external stimulus. Deep Neural Inspection is similar, but can evaluate large sets of external stimuli (features) in parallel. For instance, in neuroscience research, Wang et al. [2017] discovered hidden states of a Meta-Reinforcement Learning RNN model (Wang et al. [2016]) that correlate to dopaminergic activity in prefrontal cortex, when the network is trained with experimental observations. Finding and analyzing the hidden states of the model helped explain previously puzzling aspects of dopamine

dynamics. Deep Neural Inspection could be used to systematically discover the correlation between the hiddens states of a RNN model and neural activities from biological recordings.

We plan to extend Deep Neural Inspection by taking hidden state information from previous time steps into account when computing the query relevance score. In addition, we plan to identify clusters of neurons that together learn the same feature. These cluster might be replaced with deterministic features as a form of model simplification, or to generate verifications for monitoring the network during prediction.

Deep Neural Inspection could be applied to the field of Transfer Learning. For example, text features identified while learning a simple language model by using a small dataset could apply as regularization of hidden states while trying to learn a more complicated language model with a large dataset. Regularation for text related task is yet poorly explored. Deep Neural Inspection could provide a set of candidates for "text regularizer", which are learned by network.

References

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 2014.

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-sne effectively. *Distill*, 2016. doi: 10.23915/distill.00002. URL `http://distill.pub/2016/misread-tsne`.

Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

Hendrik Strobelt, Sebastian Gehrmann, Bernd Huber, Hanspeter Pfister, and Alexander M Rush. Visual analysis of hidden state dynamics in recurrent neural networks. *arXiv preprint arXiv:1606.07461*, 2016.

Shan Carter, David Ha, Ian Johnson, and Chris Olah. Experiments in handwriting with a neural network. *Distill*, 2016. doi: 10.23915/distill.00004. URL `http://distill.pub/2016/handwriting`.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2016.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *arXiv preprint arXiv:1607.03474*, 2016.

Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

Michiel Hermans and Benjamin Schrauwen. Training and analysing deep recurrent neural networks. In *Advances in neural information processing systems*, pages 190–198, 2013.

Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*, 2017.

F-Y Tzeng and K-L Ma. Opening the black box-data driven visualization of neural networks. In *Visualization, 2005. VIS 05. IEEE*, pages 383–390. IEEE, 2005.

Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, 2014.

Minsuk Kahng, Pierre Andrews, Aditya Kalro, and Duen Horng Chau. Activis: Visual exploration of industry-scale deep neural network models. *arXiv preprint arXiv:1704.01942*, 2017.

Xin Rong and Eytan Adar. Visual tools for debugging neural language models. In *International Conference on Machine Learning Visualization for Deep Learning Workshop*, 2016.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *KDD*, 2016.

Sanjay Krishnan and Eugene Wu. Palm: Machine learning explanations for iterative debugging. In *HILDA*, 2017.

Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. *arXiv preprint arXiv:1610.01644*, 2016.

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006.

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley Boston, 1986.

Eli Bendersky. pycparser. `https://github.com/eliben/pycparser`, 2016.

Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Joel Z. Leibo, Hubert Soyer, Dharshan Kumaran, and Matt Botvinick. Meta-reinforcement learning: a bridge between prefrontal and dopaminergic function. *Cosyne Abstract*, 2017.

Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Rémi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *CoRR*, 2016.