"I Like the Way You Think!" Inspecting the Internal Logic of Recurrent Neural Networks

Thibault Sellam
tsellam@cs.columbia.edu
Columbia University

Kevin Lin
kl2806@columbia.edu
Columbia University

Ian Yiran Huang
iyh2110@columbia.edu
Columbia University

Carl Vondrick
cvondrick@gmail.com
Google Research

Eugene Wu
ewu@cs.columbia.edu
Columbia University

1 INTRODUCTION

Recurrent neural networks (RNNs) are revolutionizing many domains. Their expressivity and the wide availability of training data enables them to tackle a wide range of problems, including language understanding [8], image generation [9], and program synthesis [5], and the proliferation of libraries and programming frameworks is significantly reducing the effort to construct and deploy them [2, 4, 12].

How do we ensure that learned models behave reliably and as intended? In traditional, non-learned software systems, we ensure reliability through software engineering principles [22] such as abstractions, modularity, testing, and logical requirements. Modern engineers do not write software systems as a single code block, nor do they deploy them without understanding the system logic and extensively testing the system.

Our goal is to develop principles and tools to hold recurrent network networks to the same reliability standards. The usual approach to evaluating and analyzing trained models treats them as black boxes and uses end-to-end metrics such as perplexity or classification accuracy [7] as a proxy for reliability in production [6]. This method often views understanding and debugging the model's *internal logic* as secondary. Consequently, the rich ecosystem of software engineering methodology and tools is not as mature for RNNs.

Understanding the internals of trained models is critical because production systems cannot rely exclusively on end-to-end metrics when the test data is unexpected or does not match the training distribution. For example, adversarial attacks [13, 21] have shown that models with excellent test score may not generalize well and may be fooled by even small perturbations over the input data. Moreover, end-to-end metrics are vulnerable to the "Clever Hans effect" [19]: it is possible to achieve high end-to-end metrics by disregarding the problem that we intend to solve and instead relying on spurious correlations that may not generalize to production settings [3].

Explaining these failures is challenging in end-to-end approaches. In classic software systems, developers can step through a program and use assertions to understand logical errors. However, since the model's logic is embedded as learned weights, these procedures are not avaliable. Although tools can help explain the model's sensitivity to the input [11, 20] or identify salient features using a simpler surrogate model [14, 16], they do not explain the model's internal decision making process.

To illustrate the problem, let us analyze a simple two-layer LSTM model[1] that learnt to evaluate Boolean expressions under left-to-right precedence padded with `X` (for instance, the input sequence `XXX1|0&1&1` yields `true`). The model has 99.9% accuracy on test data, but we wish to understand *how* the model evaluates the input expressions. Does it memorize the training data? Or learn logic? Or a mixture of the two?

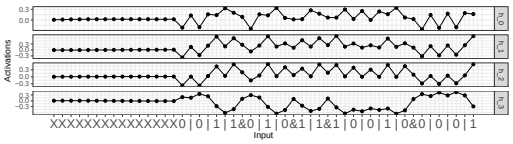


Figure 1: Activations over time of the LSTM layer for one sequence.

Pioneers in the machine learning community have developed effective methods for visualizing the state of individual or groups of hidden units, for example in convolutional networks [1, 15, 17, 23, 24] and recurrent networks [10, 18]. To understand our toy LSTM, we visualize the activations of the recurrent layer (h) as stacked times series (Figure 1) [18]. The LSTM appears to "sleep" while reading the padding character `X`. Discerning more complex patterns, however, is difficult. For instance, the network behaves differently for two instances of the same substring `|0|0|`, and it is unclear why. Visualization relies on the developer to recognize the salient logic embedded in the visualized patterns. This induces a heavy cognitive burden, and is difficult to scale to even modestly sized models or more complex tasks.

There is a discrepancy in the abstractions that existing tools provide and what developers desire. Existing tools operate at the level of hidden state activations and weight matrices, whereas developers reason about logic and functionality. In the above toy example, it would be easier to ask whether the RNN has learned the Boolean `OR` logic than to formulate this question in terms of unit activations and weights. We propose Neural Inspection (NI) as a mechanism to help bridge this gap. The idea is to let developers ask if the model is learning higher level logic using logical constructs (e.g., in Python, automata), and to automatically check and verify which parts of

[1] The first layer contains 4 hidden states, the second is fully connected with sigmoid activation. The model was trained on 5,000 randomly generated sequences and tested on 5,000 held sequences in 5 epochs. We minimized the binary cross-entropy with L2 activity regularization on the LSTM layer, using Keras/Tensorflow's Adam.

the model are following the logic. In our example, do the hidden units learn to process the Boolean OR?

We believe NI is an important primitive towards bringing software engineering principles to model analysis. Understanding whether a given piece of logic is learned in a distributed or localized fashion is the first step towards systematically modularizing the model architecture, generating assertions for the model's internal neuron behavior, and inducing the logic that the model has learned.

2 OVERVIEW

Luigi is our first NI system, and operates in a hypothesis-driven fashion. Developers provide *hypothesis functions* (e.g., Python functions) that describe candidate logic, and a trained neural network to analyze. Luigi will output sets of hidden units ranked by the extent that each set replicates the functions' logic. The hypothesis functions may be simple (e.g., sequential counters, n-gram detectors) or very complex (e.g., POS taggers, parsers). They may be provided as a system library, by the user (e.g., regex filters) or inferred from the data (described below). We find and rank the sets empirically: we execute the model and each hypothesis functions f on a test set of input sequences, we record the model's activations and compare them with f's output. We use the statistical dependency between a subset of the activations and f to score candidate pairings. Next, we will describe two types of pairings: single-unit pairing and multi-unit pairing.

Single-unit Pairing: The simplest pairing measures the statistical dependency between a single unit's activations and f's output—using Pearson's correlation, mutual information, or some another measure. Thus, it is possible to identify specialized neurons, and understand whether a representation is local to a few units or distributed across the whole model.

To illustrate this approach, we used Luigi to replicate Karpathy et al. [10] who identified specialized units in character-level language models. We trained a 128 units-wide LSTM to predict the next character given the current one on a subset of Linux' source code[2], ran the model over another subset of the code base, logged the activations, and cross-correlated them with the output of dozens of hypothesis functions that describe possible learnt features. These functions identify various grammatical structures (e.g., brackets), language keywords and parsing behaviors (e.g., nesting level), and were automatically generated. Luigi identified a dozen of specialized units for common grammatical structures, including parentheses, curly brackets, indentation, comments and common keywords such as `struct` and `return`. Figure 2 displays the output of units 53 and 127, which appear to detect comments and parentheses respectively.

Correlation does not guarantee that the unit has learned a given hypothesis function. To this end, Luigi supports *verification* modes. The first is to alter the dataset in such a way that the hypothesis function f under investigation changes, but not the other functions. If the target unit fires across many dataset transformations, it is further evidence that learns the logic embedded in the function. The second is generative—Luigi forces the unit's value, samples from the model, and checks that it triggers the paired function's output (in isolation).

[2]We trained the model with Keras on 6,206,996 characters of C source code and achieved 63.64% accuracy on training data.

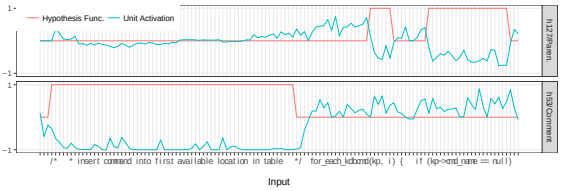
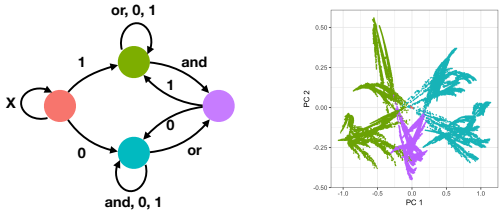


Figure 2: Activations of two LSTM units overlaid with the output of the comments and parenthesis feature functions.



(a) Candidate FSM to evaluate padded Boolean expression.

(b) LSTM activations h proj. on 2 first principal components.

Figure 3: Matching between RNN activations and FSM states.

Multi-unit Pairing: This approach checks whether a group of units replicates f's logic. To do so, Luigi runs both f and the model on test data, and it infers f's output from the model's activations with a classifier (e.g., logistic regression, SVM). High prediction accuracy suggests that the units may be learning the hypothesis function.

To illustrate, consider inspecting the boolean expression example. We manually wrote several finite state machines that could solve the task (e.g., Figure 3a), encoded each as a hypothesis function that outputs the current state. Luigi then checked how well it could infer the result of each function from the LSTM's hidden state h (using logistic regression). The highest ranked pairing provided more than 99% accuracy, which suggests that the model replicates its logic very closely. Figure 3b shows the LSTM activations h colored by their corresponding hypothetical FSM states. The consistent clustering provides more evidence of the pairing accuracy (the sub-clusters correspond to different characters).

The next step is to leverage feature selection and regularization to identify subsets of units that appear to learn a given f, and disregard the irrelevant units.

3 CONCLUSION

This short paper described a model-checking primitive called Luigi that seeks to identify high-level logic learned by units or groups of units. We envision that once found, it can be used to install dynamic assertions to check deviation from the logic, generate richer sets of test cases, modularize components of the model, and incorporate other software engineering principles. Our near-term goal is to scale Luigi to thousands of hypothesis functions and real-world sequential models.

REFERENCES

[1] 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.

[2] 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.

[3] T. Bolukbasi, K.-W. Chang, J. Y. Zou, V. Saligrama, and A. T. Kalai. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In *Advances in Neural Information Processing Systems*, pages 4349–4357, 2016.

[4] 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.

[5] 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.

[6] P. A. Flach. The geometry of roc space: understanding machine learning metrics through roc isometrics. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 194–201, 2003.

[7] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[8] 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.

[9] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[10] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[11] P. W. Koh and P. Liang. Understanding black-box predictions via influence functions. *arXiv preprint arXiv:1703.04730*, 2017.

[12] 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.

[13] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[14] S. M. Lundberg and S.-I. Lee. Consistent feature attribution for tree ensembles. *arXiv preprint arXiv:1706.06060*, 2017.

[15] A. Mahendran and A. Vedaldi. Understanding deep image representations by inverting them. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5188–5196, 2015.

[16] 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.

[17] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[18] 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.

[19] B. L. Sturm. A simple method to determine if a music information retrieval system is a "horse". *IEEE Transactions on Multimedia*, 16(6):1636–1644, 2014.

[20] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. *arXiv preprint arXiv:1703.01365*, 2017.

[21] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[22] H. Van Vliet, H. Van Vliet, and J. Van Vliet. *Software engineering: principles and practice*, volume 3. Wiley New York, 1993.

[23] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[24] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.