Evaluating Attribution Methods using White-Box LSTMs

Yiding Hao
Yale University
New Haven, CT, USA
yiding.hao@yale.edu

Abstract

Interpretability methods for neural networks are difficult to evaluate because we do not understand the black-box models typically used to test them. This paper proposes a framework in which interpretability methods are evaluated using manually constructed networks, which we call *white-box networks*, whose behavior is understood *a priori*. We evaluate five methods for producing attribution heatmaps by applying them to white-box LSTM classifiers for tasks based on formal languages. Although our white-box classifiers solve their tasks perfectly and transparently, we find that all five attribution methods fail to produce the expected model explanations.

1 Introduction

Attribution methods are a family of interpretability techniques for individual neural network predictions that attempt to measure the importance of input features for determining the model's output. Given an input, an attribution method produces a vector of *attribution* or *relevance scores*, which is typically visualized as a heatmap that highlights portions of the input that contribute to model behavior. In the context of NLP, attribution scores are usually computed at the token level, so that each score represents the importance of a token within an input sequence. These heatmaps can be used to identify keywords upon which networks base their decisions (Li et al., 2016; Sundararajan et al., 2017; Arras et al., 2017a,b; Murdoch et al., 2018, *inter alia*).

One of the main challenges facing the evaluation of attribution methods is that it is difficult to assess the quality of a heatmap when the network in question is not understood in the first place. If a word is deemed relevant by an attribution method, we do not know whether the model actually considers that word relevant, or whether the attribution method has erroneously estimated its importance. Indeed, previous studies have argued that attribution methods are sensitive to features unrelated to model behavior in some cases (e.g., Kindermans et al., 2019), and altogether insensitive to model behavior in others (Adebayo et al., 2018).

To tease the evaluation of attribution methods apart from the interpretation of models, this paper proposes an evaluation framework for attribution methods in NLP that uses only models that are fully understood *a priori*. Instead of testing attribution methods on black-box models obtained through training, we construct *white-box* models for testing by directly setting network parameters by hand. Our focus is on white-box LSTMs that implement intuitive strategies for solving simple classification tasks based on formal languages with deterministic solutions. We apply our framework to five attribution methods: *occlusion* (Zeiler and Fergus, 2014), *saliency* (Simonyan et al., 2014; Li et al., 2016), *gradient* $\times$ *input*, (G $\times$ I, Shrikumar et al., 2017), *integrated gradients* (IG, Sundararajan et al., 2017), and *layer-wise relevance propagation* (LRP, Bach et al., 2015). In doing so, we make the following contributions.

- We construct four white-box LSTMs that can be used to test attribution methods. We provide a complete description of our model weights in Appendix A.[1] Beyond the five methods considered here, our white-box networks can be used to test any attribution method compatible with LSTMs.

- Empirically, we show that all five attribution methods produce erroneous heatmaps for our white-box networks, despite the models' transparent behavior. As a preview of our re-

[1] We also provide code for our models at https://github.com/yidinghao/whitebox-lstm.

Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, pages 300–313
Online, November 20, 2020.

Task: Determine whether the input contains one of the following subsequences: ab, bc, cd, or dc.
Output: *True*, since the input aacb contains two (non-contiguous) instances of ab.

Occlusion	Saliency	G × I	IG	LRP
a a c **b**	a **a** c **b**	a a c **b**	**a** a c **b**	a a c **b**
a **a** c **b**	a a c **b**	a a c **b**	a **a** c **b**	a a c **b**

Table 1: Sample heatmaps for two white-box networks: a "counter-based" network (top) and an "FSA-based" network (bottom). The features relevant to the output are the two as and the b.

sults, Table 1 shows sample heatmaps computed for two models designed to identify the non-contiguous subsequence ab in the input aacb. Even though both models' outputs are determined by the presence of the two as and the b, all four methods either incorrectly highlight the c or fail to highlight at least one of the as in at least one case.

- We identify two general ways in which four of the five methods do not behave as intended. Firstly, while saliency, G × I and IG are theoretically invariant to differences in model implementation (Sundararajan et al., 2017), in practice we find that these methods can still produce qualitatively different heatmaps for nearly identical models. Secondly, we find that LRP is susceptible to numerical issues, which cause heatmaps to be zeroed out when values are rounded to zero.

2 Related Work

Several approaches have been taken in the literature for understanding how to evaluate attribution methods. On a theoretical level, *axiomatic* approaches propose formal desiderata that attribution methods should satisfy, such as implementation invariance (Sundararajan et al., 2017), input translation invariance (Kindermans et al., 2019), continuity with respect to inputs (Montavon et al., 2018; Ghorbani et al., 2019), or the existence of relationships between attribution scores and logit or softmax scores (Sundararajan et al., 2017; Ancona et al., 2018; Montavon, 2019). The degree to which attribution methods fulfill these criteria can be determined either mathematically or empirically.

Other approaches, which are more experimental in nature, attempt to directly assess the relationship between attribution scores and model behavior. A common test, due to Bach et al. (2015) and Samek et al. (2017) and applied to sequence modeling by Arras et al. (2017a), involves ablating or perturbing parts of the input, from those with the highest attribution scores to those with the lowest, and counting the number of features that need to be ablated in order to change the model's prediction. Another test, proposed by Adebayo et al. (2018), tracks how heatmaps change as layers of a network are incrementally randomized.

A third kind of approach evaluates the extent to which heatmaps identify salient input features. For example, Zhang et al. (2018) propose the *pointing game task*, in which the highest-relevance pixel for an image classifier input must belong to the object described by the target output class. Within this framework, Kim et al. (2018), Poerner et al. (2018), Arras et al. (2019), and Yang and Kim (2019) construct datasets in which input features exhibit experimentally controlled notions of importance, yielding "ground truth" attributions against which heatmaps can be evaluated.

Our paper incorporates elements of the ground-truth approaches, since it is straightforward to determine which input features are important for our formal language tasks. We enhance these approaches by using white-box models that are guaranteed to be sensitive to those features.

3 Formal Language Tasks

Formal languages are often used to evaluate the expressive power of RNNs. Here, we focus on formal languages that have been recently used to probe LSTMs' ability to capture three kinds of dependencies: *counting*, *long-distance*, and *hierarchical* dependencies. We define a classification task based on each of these formal languages.

3.1 Counting Dependencies

Counter languages (Fischer, 1966; Fischer et al., 1968) are languages recognized by automata equipped with counters. Weiss et al. (2018) demonstrate using an acceptance task for the languages $\mathtt{a}^n\mathtt{b}^n$ and $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ that LSTMs naturally learn to use cell state units as counters. Merrill's (2019) asymptotic analysis shows that LSTM acceptors accept only counter languages when their weights are fully saturated. Thus, counter languages may be viewed as a characterization of the expressive power of LSTMs.

We define the *counting task* based on a simple

example of a counting language.

Task 1 (Counting Task)**.** Given a string in $x \in \{\texttt{a}, \texttt{b}\}^*$, determine whether or not x has strictly more as than bs.

Example 2. The counting task classifies aaab as *True*, ab as *False*, and bbbba as *False*.

A counter automaton can solve the counting task by incrementing its counter whenever an a is encountered and decrementing it whenever a b is encountered. It outputs *True* if and only if its counter is at least 1. We expect attribution scores for all input symbols to have roughly the same magnitude, but that scores assigned to a will have the opposite sign to those assigned to b.

3.2 Long-Distance Dependencies

Strictly piecewise (SP, Heinz, 2007) languages were used by Avcu et al. (2017) and Mahalunkar and Kelleher (2018, 2019a,b) to test the propensity of LSTMs to learn long-distance dependencies, compared to Elman's (1990) simple recurrent networks. SP languages are regular languages whose membership is defined by the presence or absence of certain *subsequences*, which may or may not be contiguous. For example, ad is a subsequence of abcde, since both letters of ad occur in abcde, in the same order. Based on these ideas, we define the *SP task* as follows.

Task 3 (SP Task)**.** Given $x \in \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}^*$, determine whether or not x contains at least one of the following subsequences: ab, bc, cd, dc.

Example 4. In the SP task, aab is classified as *True*, since it contains the subsequence ab. Similarly, acb is classified as *True*, since it contains ab non-contiguously. The string aaa is classified as *False*.

The choice of SP languages as a test for long-distance dependencies is motivated by the fact that symbols in a non-contiguous subsequence may occur arbitrarily far from one another. The SP task yields a variant of the pointing game task in the sense that the input string may or may not contain an "object" (one of the four subsequences) that the network must identify. Therefore, we expect an input symbol to receive a nonzero attribution score if and only if it comprises a subsequence.

3.3 Hierarchical Dependencies

The *Dyck language* is the language D generated by the following context-free grammar, where ε is the empty string.

$$S \rightarrow SS \mid (S) \mid [S] \mid \varepsilon$$

D contains all balanced strings of parentheses and square brackets. Since D is often viewed as a canonical example of a context-free language (Chomsky and Schützenberger, 1959), several recent studies, including Sennhauser and Berwick (2018), Bernardy (2018), Skachkova et al. (2018), and Yu et al. (2019), have used D to evaluate whether LSTMs can learn hierarchical dependencies implemented by pushdown automata. Here, we consider the *bracket prediction task* proposed by Sennhauser and Berwick (2018).

Task 5 (Bracket Prediction Task)**.** Given a prefix p of some string in D, identify the next valid closing bracket for p.

Example 6. The string [([ ] requires a prediction of), since the (is the last unclosed bracket. Similarly, (( ) [requires a prediction of]. Strings with no unclosed brackets, such as [( )], require a prediction of *None*.

In heatmaps for the bracket prediction task, we expect the last unclosed bracket to receive the highest-magnitude relevance score.

4 White-Box Networks

We use two approaches to construct white-box networks for our tasks. In the *counter-based* approach, the cell state contains a set of counters, which are incremented or decremented throughout the computation. The network's final output is based on the values of the counters. In the *automaton-based* approach, we use the LSTM to simulate an automaton, with the cell state containing a representation of the automaton's state. We use a counter-based network to solve the counter task and an automaton-based network to solve the bracket prediction task. We use both kinds of networks to solve the SP task. All networks perfectly solve the tasks they were designed for. This section describes our white-box networks at a high level; a detailed description is given in Appendix A.

In the rest of this paper, we identify the alphabet symbols a, b, c, and d with the one-hot vectors for indices 1, 2, 3, and 4, respectively. The vectors $\boldsymbol{f}^{(t)}$, $\boldsymbol{i}^{(t)}$, and $\boldsymbol{o}^{(t)}$ represent the forget, input, and output gates, respectively. $\boldsymbol{g}^{(t)}$ is the value added to the cell state at each time step, and σ represents

the sigmoid function. We assume that the hidden state $\boldsymbol{h}^{(t)}$ and cell state $\boldsymbol{c}^{(t)}$ are updated as follows.

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \boldsymbol{g}^{(t)}$$
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh\left(\boldsymbol{c}^{(t)}\right)$$

4.1 Counter-Based Networks

In the counter-based approach, each position of the cell state contains the value of a counter. To adjust the counter in position j by some value $v \in (-1, 1)$, we set $g_j^{(t)} = v$, and we saturate the gates by setting them to $\sigma(m) \approx 1$, where $m \gg 0$ is a large constant. For example, our network for the counting task uses a single hidden unit, with the gates always saturated and with $g^{(t)}$ given by

$$g^{(t)} = \tanh\left(u \begin{bmatrix} 1 & -1 \end{bmatrix} \boldsymbol{x}^{(t)}\right),$$

where $u > 0$ is a hyperparameter that scales the counter by a factor of $v = \tanh(u)$.[2] When $\boldsymbol{x}^{(t)} =$ a, we have $g^{(t)} = v$, so the counter is incremented by v. When $\boldsymbol{x}^{(t)} =$ b, we compute $g^{(t)} = -v$, so the counter is decremented by v.

For the SP task, we use seven counters. The first four counters record how many occurrences of each symbol have been observed at time step t. The next three counters record the number of bs, cs, and ds that form one of the four distinguished subsequences with an earlier symbol. For example, after seeing the input aaabbc, the counter-based network for the SP task satisfies

$$\boldsymbol{c}^{(6)} = v \begin{bmatrix} 3 & 2 & 1 & 0 & 2 & 1 & 0 \end{bmatrix}^\top.$$

The first four counters represent the fact that the input has 3 as, 2 bs, 1 c, and no ds. Counter #5 is $2v$ because the two bs form a subsequence with the as, and counter #6 is v because the c forms a subsequence with the bs.

The logit scores of our counter-based networks are computed by a linear decoder using the $\tanh$ of the counter values. For the counting task, the score of the *True* class is $h^{(t)}$, while the score of the *False* class is fixed to $\tanh(v)/2$. This means that the network outputs *True* if and only if the final counter value is at least v. For the SP task, the score of the *True* class is $h_5^{(t)} + h_6^{(t)} + h_7^{(t)}$, while the score of the *False* class is again $\tanh(v)/2$.

[2] We use $u = 0.5$ for the counting task, $u = 0.7$ for the SP task, and $m = 50$ for both tasks.

4.2 Automata-Based Networks

We consider two types of automata-based networks: one that implements a finite-state automaton (FSA) for the SP task, and one that implements a pushdown automaton (PDA) for the bracket prediction task.

Our FSA construction is similar to Korsky and Berwick's (2019) FSA construction for simple recurrent networks. Consider a deterministic FSA $\mathcal{A}$ with states Q and alphabet Σ. To simulate $\mathcal{A}$ using an LSTM, we use $|Q| \cdot |\Sigma|$ hidden units, with the following interpretation. Suppose that $\mathcal{A}$ transitions to state q after reading input $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$. The hidden state $\boldsymbol{h}^{(t)}$ is a one-hot representation of the pair $\left\langle q, \boldsymbol{x}^{(t)} \right\rangle$, which encodes both the current state of $\mathcal{A}$ and the most recent input symbol. Since the FSA undergoes a state transition with each input symbol, the forget gate always clears $\boldsymbol{c}^{(t)}$, so that information written to the cell state does not persist beyond a single time step. The output layer simply detects whether or not the FSA is in an accepting state. Details are provided in Appendix A.3.

Next, we describe how to implement a PDA for the bracket prediction task. We use a stack containing all unclosed brackets observed in the input string, and make predictions based on the top item of the stack. We represent a bounded stack of size k using $2k + 1$ hidden units. The first $k - 1$ positions contain all stack items except the top item, with (represented by the value 1, [represented by -1, and empty positions represented by 0. The kth position contains the top item of the stack. The next k positions contain the height of the stack in unary notation, and the last position contains a bit indicating whether or not the stack is empty. For example, after reading the input ([(() with a stack of size 4, the stack contents ([(are represented by

$$\boldsymbol{c}^{(5)} = \begin{bmatrix} 1 & -1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}^\top.$$

The 1 in position 4 indicates that the top item of the stack is (, and the 1, -1, and 0 in positions 1–3 indicate that the remainder of the stack is ([. The three 1s in positions 5–8 indicate that the stack height is 3, and the 0 in position 9 indicates that the stack is not empty.

When $\boldsymbol{x}^{(t)}$ is (or [, it is copied to $c_k^{(t)}$, and $c_k^{(t)}$ is copied to the highest empty position in $\boldsymbol{c}_{:k-1}^{(t)}$, pushing the opening bracket to the stack. The empty stack bit is then set to 0, marking the stack

Name	Formula
Saliency	$R^{(c)}_{t,i}(\boldsymbol{X}) = \frac{\partial \hat{y}_c}{\partial x_i^{(t)}}\Big\vert_{x_i^{(t)}=X_{t,i}}$
G × I	$R^{(c)}_{t,i}(\boldsymbol{X}) = X_{t,i}\frac{\partial \hat{y}_c}{\partial x_i^{(t)}}\Big\vert_{x_i^{(t)}=X_{t,i}}$
IG	$R^{(c)}_{t,i}(\boldsymbol{X}) = X_{t,i}\int_0^1 \frac{\partial \hat{y}_c}{\partial x_i^{(t)}}\Big\vert_{x_i^{(t)}=\alpha X_{t,i}} d\alpha$

Table 2: Definitions of the gradient-based methods.

as non-empty. When the current input symbol is a closing bracket, the highest item of positions 1 through $k-1$ is deleted and copied to position k, popping the top item from the stack. Because the PDA network is quite complex, we focus here on describing how the top stack item in position k is determined, and leave other details for Appendix A.4. Let $\alpha^{(t)}$ be 1 if $\boldsymbol{x}^{(t)} =$ (, -1 if $\boldsymbol{x}^{(t)} =$ [, and 0 otherwise. At each time step, $g_k^{(t)} = \tanh\left(m \cdot u^{(t)}\right)$, where $m \gg 0$ and

$$u^{(t)} = 2^k \alpha^{(t)} + \sum_{j=1}^{k-1} 2^{j-1} h_j^{(t-1)}. \qquad (1)$$

Observe that $m \cdot u^{(t)} \gg 0$ when $\alpha^{(t)} = 1$, and $m \cdot u^{(t)} \ll 0$ when $\alpha^{(t)} = -1$. Thus, $g_k^{(t)}$ contains the stack encoding of the current input symbol if it is an opening bracket. If the current input symbol is a closing bracket, then $\alpha^{(t)} = 0$, so the sign of $u^{(t)}$ is determined by the highest item of $\boldsymbol{h}_{:k-1}^{(t-1)}$.

5 Attribution Methods

Let $\boldsymbol{X}$ be a matrix of input vectors, such that the input at time t is the row vector $\boldsymbol{X}_{t,:} = \left(\boldsymbol{x}^{(t)}\right)^\top$. Given $\boldsymbol{X}$, an LSTM classifier produces a vector $\hat{\boldsymbol{y}}$ of logit scores. Based on $\boldsymbol{X}$, $\hat{\boldsymbol{y}}$, and possibly a *baseline input* $\overline{\boldsymbol{X}}$, an attribution method assigns an attribution score $R^{(c)}_{t,i}(\boldsymbol{X})$ to input feature $X_{t,i}$ for each output class c. These feature-level scores are then aggregated to produce token-level scores:

$$R^{(c)}_t(\boldsymbol{X}) = \sum_i R^{(c)}_{t,i}(\boldsymbol{X}).$$

Broadly speaking, our five attribution methods are grouped into three types: one *perturbation-based*, three *gradient-based*, and one *decomposition-based*. The following subsections describe how each method computes $R^{(c)}_{t,i}(\boldsymbol{X})$.

5.1 Perturbation- and Gradient-Based Methods

Perturbation-based methods are premised on the idea that if $X_{t,i}$ is an important input feature, then changing the value of $X_{t,i}$ would cause $\hat{\boldsymbol{y}}$ to change. The one perturbation method we consider is occlusion. In this method, $R^{(c)}_{t,i}(\boldsymbol{X})$ is the change in $\hat{y}_c$ observed when $\boldsymbol{X}_{t,:}$ is replaced by $\mathbf{0}$.

Gradient-based methods rely on the same intuition as perturbation-based methods, but use automatic differentiation to simulate infinitesimal perturbations. The definitions of our three gradient-based methods are given in Table 2. The most basic of these is saliency, which simply measures relevance by the derivative of the logit score with respect to each input feature. G × I attempts to improve upon saliency by using the first-order terms in a Taylor-series approximation of the model instead of the gradients on their own. IG is designed to address the issue of small gradients found in saturated units by integrating G × I along the line connecting $\boldsymbol{X}$ to a baseline input $\overline{\boldsymbol{X}}$, here taken to be the zero matrix.

5.2 Decomposition-Based Methods

Decomposition-based methods are methods that satisfy the relation

$$\hat{y}_c = R^{(c)}_{\text{bias}} + \sum_{t,i} R^{(c)}_{t,i}(\boldsymbol{X}), \qquad (2)$$

where $R^{(c)}_{\text{bias}}$ is a relevance score assigned to the bias units of the network. The interpretation of equation (2) is that the logit score $\hat{y}_c$ is "distributed" among the input features and the bias units, so that the relevance scores form a "decomposition" of $\hat{y}_c$.

The one decomposition-based method we consider is LRP, which computes scores using a backpropagation algorithm that distributes scores layer by layer. The scores of the output layer are initialized to

$$r_i^{(c,\text{output})} = \begin{cases} \hat{y}_i, & i = c \\ 0, & \text{otherwise.} \end{cases}$$

For each layer l with activation $\boldsymbol{z}^{(l)}$, activation function $f^{(l)}$, and output $\boldsymbol{a}^{(l)} = f^{(l)}\left(\boldsymbol{z}^{(l)}\right)$, the relevance $\boldsymbol{r}^{(c,l)}$ of $\boldsymbol{a}^{(l)}$ is determined by the following *propagation rule*:

$$r_i^{(c,l)} = \sum_{l'}\sum_j r_j^{(c,l')} \frac{W_{j,i}^{(l' \leftarrow l)} a_i^{(l)}}{z_j^{(l')} + \text{sign}\left(z_j^{(l')}\right)\varepsilon},$$

where l' ranges over all layers to which l has a forward connection via $\boldsymbol{W}^{(l' \leftarrow l)}$ and $\varepsilon > 0$ is a stabilizing constant.[3] For the LSTM gate interactions, we follow Arras et al. (2017b) in treating multiplicative connections of the form $\boldsymbol{a}^{(l_1)} \odot \boldsymbol{a}^{(l_2)}$ as activation functions of the form $\boldsymbol{a}^{(l_1)} \odot f^{(l_2)}(\cdot)$, where $\boldsymbol{a}^{(l_1)}$ is $\boldsymbol{f}^{(t)}$, $\boldsymbol{i}^{(t)}$, or $\boldsymbol{o}^{(t)}$. The final attribution scores are given by the values propagated to the input layer:

$$R_{t,i}^{(c)}(\boldsymbol{X}) = r_i^{(c,\text{input}_t)}.$$

6 Qualitative Evaluation

To evaluate attribution methods under our framework, we begin with a qualitative description of the heatmaps that are computed for our white-box networks, based on the illustrative sample of heatmaps appearing in Table 3.

6.1 Counting Task

Occlusion, G × I, and IG are well-behaved for the counting task. As expected, these methods assign `a` a positive value and `b` a negative value when the output class for attribution is $c = \textit{True}$. When the number of `a`s is different from the number of `b`s, occlusion assigns a lower-magnitude score to the symbol with fewer instances. When $c = \textit{False}$, all relevance scores are 0. This is because $\hat{y}_{\textit{False}}$ is fixed to a constant value supplied by a bias term, so input features cannot affect its value.

Saliency and LRP both fail to produce nonzero scores, at least in some cases. Saliency scores satisfy $R_{t,1}^{(\textit{True})}(\boldsymbol{X}) = -R_{t,2}^{(\textit{True})}(\boldsymbol{X})$, resulting in token-level scores of 0 for all inputs. Heatmaps #3 and #4 show that LRP assigns scores of 0 to prefixes containing equal numbers of `a`s and `b`s. We will see in Subsection 7.1 that this phenomenon appears to be related to the fact that the LSTM gates are saturated.

6.2 SP Task

We obtain radically different heatmaps for the two SP task networks, despite the fact that they produce the same classifications for all inputs.

For the counter-based network, all methods except for saliency assign positive scores for $c = \textit{True}$ to symbols constituting one of the four subsequences, and scores of zero elsewhere. The saliency heatmaps do not adhere to this pattern, and instead generally assign higher scores to tokens occurring near the end of the input. Heatmaps #7–10 show that LRP fails to assign positive scores to the first symbol of each subsequence, while the other methods generally do not.[4] The LRP behavior reflects the fact that the initial `a` does not increment the subsequence counters, which determine the final logit score. In contrast, the behavior of occlusion, G × I, and IG is explained by the fact that removing either the `a` or the `b` destroys the subsequence. Note that the `a`s in heatmap #9 receive scores of 0 from occlusion and G × I, since removing only one of the two `a`s does not destroy the subsequence.

For the FSA-based network, saliency, G × I, and LRP assign only the last symbol a nonzero score when the relevance output class c matches the network's predicted class. IG appears to produce erratic heatmaps, exhibiting no immediately obvious pattern. Although occlusion appears to be erratic at first glance, its behavior can be explained by the fact that changing $\mathbf{x}^{(t)}$ to $\mathbf{0}$ causes $\mathbf{h}^{(t)}$ to be $\mathbf{0}$, which the LSTM interprets as the initial state of the FSA; thus, $R_t^{(c)}(\mathbf{X}) \neq 0$ precisely when $\mathbf{X}_{t+1:,:}$ is classified differently from $\mathbf{X}$. In all cases, the heatmaps for the FSA-based network diverge significantly from the expected heatmaps.

6.3 Bracket Prediction Task

The heatmaps for the PDA-based network also differ strikingly from those of the other networks, in that the gradient-based methods never assign nonzero scores. This is because equation (1) causes $\boldsymbol{g}^{(t)}$ to be highly saturated, resulting in zero gradients. In the case of LRP, the matching bracket is highlighted when $c \neq \textit{None}$. When the matching bracket is not the last symbol of the input, the other unclosed brackets are also highlighted, with progressively smaller magnitudes, and with brackets of the opposite type from c receiving negative scores. This pattern reflects the mechanism of (1), in which progressively larger powers of 2 are used to determine the content copied to $c_k^{(t)}$. When the relevance output class is $c = \textit{None}$, LRP assigns opening brackets a negative score, revealing the fact that those input symbols set the bit $c_{2k+1}^{(t)}$ to indicate that the stack is not empty. Although occlusion sometimes highlights the matching bracket, it does not appear to be consistent in doing so. For example, it fails to highlight the matching bracket

[3] We use $\varepsilon = 0.001$.

[4] Although it is difficult to see, IG assigns a small positive score to the `b`s in heatmaps #7 and #8.

Network	#	c	Target	Occlusion	Saliency	G × I	IG	LRP
Counting	1	True	True	a a a b b	a a a b b	a a a b b	a a a b b	a a a b b
	2	True	False	b b b a a	b b b a a	b b b a a	b b b a a	b b b a a
	3	True	False	a a a b b b	a a a b b b	a a a b b b	a a a b b b	a a a b b b
	4	True	False	a a b b b	a a b b b	a a b b b	a a b b b	a a b b b
	5	False	True	a a a b b	a a a b b	a a a b b	a a a b b	a a a b b
	6	False	False	a a b b b	a a b b b	a a b b b	a a b b b	a a b b b
SP (Counter)	7	True	True	a c b	a c b	a c b	a c b	a c b
	8	True	True	a c b b	a c b b	a c b b	a c b b	a c b b
	9	True	True	a a c b	a a c b	a a c b	a a c b	a a c b
	10	True	True	a b c a b	a b c a b	a b c a b	a b c a b	a b c a b
	11	True	False	a a c c	a a c c	a a c c	a a c c	a a c c
	12	False	True	a c b	a c b	a c b	a c b	a c b
	13	False	False	a a c c	a a c c	a a c c	a a c c	a a c c
SP (FSA)	14	True	True	a c b	a c b	a c b	a c b	a c b
	15	True	True	a c b b	a c b b	a c b b	a c b b	a c b b
	16	True	True	a a c b	a a c b	a a c b	a a c b	a a c b
	17	True	True	a b c a b	a b c a b	a b c a b	a b c a b	a b c a b
	18	True	False	a a c c	a a c c	a a c c	a a c c	a a c c
	19	False	True	a c b	a c b	a c b	a c b	a c b
	20	False	False	a a c c	a a c c	a a c c	a a c c	a a c c
Bracket (PDA)	21	]	]	([[([	([[([	([[([	([[([	([[([
	22	)	)	([[([ ]	([[([ ]	([[([ ]	([[([ ]	([[([ ]
	23	None	None	([[ ]])	([[ ]])	([[ ]])	([[ ]])	([[ ]])
	24	]	]	[([ ] [( )	[([ ] [( )	[([ ] [( )	[([ ] [( )	[([ ] [( )
	25	)	]	[([ ] [( )	[([ ] [( )	[([ ] [( )	[([ ] [( )	[([ ] [( )

Table 3: Selected heatmaps based on $R_t^{(c)}(\boldsymbol{X})$. **Red** represents positive values and **blue** represents negative values. Heatmaps with all values within the range of $\pm 1 \times 10^{-5}$ are shown as all 0s.

u	v	$\hat{y}_{True}$	Saliency	G × I	IG
0.6	0.537	0.151	a c c b	a c c b	a c c b
0.7	0.604	0.533	a c c b	a c c b	a c c b
0.8	0.664	0.581	a c c b	a c c b	a c c b
1	0.762	0.642	a c c b	a c c b	a c c b
4	0.999	0.761	a c c b	a c c b	a c c b
8	1.000	0.762	a c c b	a c c b	a c c b
16	1.000	0.762	a c c b	a c c b	a c c b
64	1.000	0.762	a c c b	a c c b	a c c b

Table 4: Gradient-based heatmaps of $R_t^{(True)}(\texttt{accb})$ for the counter-based SP network, with $0.6 \leq u \leq 64$.

m	$\sigma(m)$	$c^{(t)}$	Accuracy	% Blank
4	0.982	-8.74×10^{-3}	90.1	0.2
5	0.993	-3.48×10^{-3}	96.1	2.2
6	0.998	-1.32×10^{-3}	99.8	6.5
7	0.999	-4.91×10^{-4}	100.0	22.0
8	1.000	-1.81×10^{-4}	100.0	42.1
9	1.000	-6.68×10^{-5}	100.0	69.9
10	1.000	-2.46×10^{-5}	100.0	92.3
11	1.000	-9.05×10^{-6}	100.0	98.7
12	1.000	-3.33×10^{-6}	100.0	99.8

Table 5: The results of the LRP saturation test, including the value of m, the average value of $c^{(t)}$ when the counter reaches 0, the network's testing accuracy, and the percentage of examples with blank heatmaps for prefixes with equal numbers of as and bs.

in heatmap #21, and highlights one other bracket in heatmaps #23–24.

7 Detailed Evaluations

We now turn to focused investigations of particular phenomena that attribution methods exhibit when applied to white-box networks. Subsection 7.1 begins by discussing the effect of network saturation on the gradient-based methods and LRP. In Subsection 7.2 we apply Bach et al.'s (2015) ablation test to our attribution methods for the SP task.

7.1 Saturation

As mentioned in the previous section, network saturation causes gradients to be approximately 0 when using sigmoid or tanh activation functions. To test how attribution methods are affected by saturation, Table 4 shows heatmaps for the input accb generated by gradient-based methods for different instantiations of the counter-based SP network with varying degrees of saturation. Recall from Section 4 that counter values for this network are expressed in multiples of the scaling factor v. We control the saturation of the network via the parameter $u = \tanh^{-1}(v)$. For all three gradient-based methods, scores for a decrease and scores for b increase as u increases. Additionally, saliency scores for the first c decrease when u increases. When $u = 8$, v is almost completely saturated, causing G × I to produce all-zero heatmaps.

On the other hand, IG is still able to produce nonzero heatmaps even at $u = 64$. Thus, IG is much more resistant to the effects of saturation than G $\times$ I.

According to Sundararajan et al. (2017), gradient-based methods satisfy the axiom of *implementation invariance*: they produce the same heatmaps for any two networks that compute the same function. This formal property is seemingly at odds with the diverse array of heatmaps appearing in Table 4, which are produced for networks that all yield identical classifiers. In particular, the networks with $u = 8$, 16, and 64 yield qualitatively different heatmaps, despite the fact that the three networks are distinguished only by differences in v of less than 0.001. Because the three functions are technically not equal, implementation invariance is not violated in theory; but the fact that IG produces different heatmaps for three nearly identical networks shows that the intuition described by implementation invariance is not borne out in practice.

Besides the gradient-based methods, LRP is also susceptible to problems arising from saturation. Recall from heatmaps #3 and #4 of Table 3 that for the counting task network, LRP assigns scores of 0 to prefixes with equal numbers of as and bs. We hypothesize that this phenomenon is related to the fact $c^{(t)} = 0$ after reading such prefixes, since the counter has been incremented and decremented in equal amounts. Accordingly, we test whether this phenomenon can be mitigated by desaturating the gates so that $c^{(t)}$ does not exactly reach 0. Recall that the white-box LSTM gates approximate $1 \approx \sigma(m)$ using a constant $m \gg 0$. We construct networks with varying values of m and compute LRP scores on a randomly generated testing set of 1000 strings, each of which contains at least one prefix with equal numbers of as and bs. In Table 5 we report the percentage of examples for which such prefixes receive LRP scores of 0, along with the network's accuracy on this testing set and the average value of $c^{(t)}$ when the counter reaches 0. Indeed, the percentage of prefixes receiving scores of 0 increases as the approximation $c^{(t)} \approx 0$ becomes more exact.

7.2 Ablation Test

So far, we have primarily compared attribution methods via visual inspection of individual examples. To compare the five methods quantitatively, we apply the ablation test of Bach et al. (2015) to our two white-box networks for the SP task.[5] Given an input string classified as *True*, we iteratively remove the symbol with the highest relevance score, recomputing heatmaps at each iteration, until the string no longer contains any of the four subsequences. We apply the ablation test to 100 randomly generated input strings, and report the average percentage of each string that is ablated in Table 6. A peculiar property of the SP task is that removing a symbol preserves the validity of input strings. This means that, unlike in NLP settings, our ablation test does not suffer from the issue that ablation produces invalid inputs.

Method	SP (Counter)	SP (FSA)
Occlusion	$61.8_{\pm 12.2}$	$\mathbf{52.6}_{\pm 11.7}$
Saliency	$97.8_{\pm 1.1}$	$96.0_{\pm 2.5}$
G $\times$ I	$65.7_{\pm 14.4}$	$96.0_{\pm 2.5}$
IG	$\mathbf{47.5}_{\pm 7.6}$	$94.9_{\pm 2.9}$
LRP	$64.3_{\pm 12.7}$	$96.0_{\pm 2.5}$
Random	$96.1_{\pm 2.5}$	
Optimal	$\mathbf{42.7}_{\pm 3.8}$	

Table 6: Mean and standard deviation results of the ablation test, normalized by string length and expressed as a percentage. "Optimal" is the best possible score.

Saliency, G $\times$ I, and LRP perform close to the random baseline on the FSA network; this is unsurprising, since these methods only assign nonzero scores to the last input symbol. While Table 3 shows some variation in the IG heatmaps, IG also performs close to the random baseline. Only occlusion performs considerably better, since it is able to identify symbols whose ablation would destroy subsequences.

On the counter-based SP network, IG performs remarkably close to the optimal benchmark, which represents the best possible performance on this task. Occlusion, G $\times$ I, and LRP achieve a similar level of performance to one another, while saliency performs worse than the random baseline.

8 Conclusion

Of all the heatmaps considered in this paper, only those computed by G $\times$ I and IG for the counting task fully matched our expectations. In other cases, all attribution methods fail to identify at least some of the input features that should be considered relevant, or assign relevance to input features that do

[5] We do not consider the counting task because its heatmaps are already easy to understand, and we do not consider the PDA network because the gradient-based methods fail to produce nonzero heatmaps for that network.

not affect the model's behavior. Among the five methods, saliency achieves the worst performance: it never assigns nonzero scores for the counting and bracket prediction tasks, and it does not identify the relevant symbols for either of the two SP networks. Saliency also achieves the worst performance on the ablation test for both the counter-based and the FSA-based SP networks. Among the four white-box networks, the two automata-based networks proved to be much more challenging for the attribution methods than the counter-based networks. While the LRP heatmaps for the PDA network correctly identify the matching bracket when available, no other method produces reasonable heatmaps for the PDA network, and all five methods fail to interpret the FSA network.

Taken together, our results suggest that attribution heatmaps should be viewed with skepticism. This paper has identified cases in which heatmaps fail to highlight relevant features, as well as cases in which heatmaps incorrectly highlight irrelevant features. Although most of the methods perform better for the counter-based networks than the automaton-based networks, in practical settings we do not know what kinds of computations are implemented by a trained network, making it impossible to determine whether the network under analysis is compatible with the attribution method being used.

In future work, we encourage the use of our four white-box models as qualitative benchmarks for evaluating interpretability methods. For example, the style of evaluation we have developed can be replicated for attribution methods not covered in this paper, including DeepLIFT (Shrikumar et al., 2017) and contextual decomposition (Murdoch et al., 2018). We believe that insights gleaned from white-box analysis can help researchers choose between different attribution methods and identify areas of improvement in current techniques.

Acknowledgments

I would like to thank Dana Angluin and Robert Frank for their advice and mentorship on this project. I would also like to thank Yoav Goldberg, John Lafferty, Tal Linzen, R. Thomas McCoy, Aaron Mueller, Karl Mulligan, Shauli Ravfogel, Jason Shaw, and the reviewers for their helpful feedback and discussion.

References

Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. 2018. Sanity Checks for Saliency Maps. In *Advances in Neural Information Processing Systems 31*, volume 31, pages 9505–9515, Montreal, Canada. Curran Associates, Inc.

Marco Ancona, Enea Ceolini, Cengiz Öztireli, and Markus Gross. 2018. Towards better understanding of gradient-based attribution methods for Deep Neural Networks. In *ICLR 2018 Conference Track*, Vancouver, Canada. OpenReview.

Leila Arras, Franziska Horn, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. 2017a. "What is relevant in a text document?": An interpretable machine learning approach. *PLOS ONE*, 12(8):e0181142.

Leila Arras, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. 2017b. Explaining Recurrent Neural Network Predictions in Sentiment Analysis. In *Proceedings of the 8th Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis*, pages 159–168, Copenhagen, Denmark. Association for Computational Linguistics.

Leila Arras, Ahmed Osman, Klaus-Robert Müller, and Wojciech Samek. 2019. Evaluating Recurrent Neural Network Explanations. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 113–126, Florence, Italy. Association for Computational Linguistics.

Enes Avcu, Chihiro Shibata, and Jeffrey Heinz. 2017. Subregular Complexity and Deep Learning. In *Proceedings of the Conference on Logic and Machine Learning in Natural Language (LaML 2017), Gothenburg, 12–13 June 2017*, volume 1 of *CLASP Papers in Computational Linguistics*, pages 20–33, Gothenburg, Sweden. Centre for Linguistic Theory and Studies in Probability (CLASP), University of Gothenburg.

Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. 2015. On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. *PLOS ONE*, 10(7):e0130140.

Jean-Philippe Bernardy. 2018. Can Recurrent Neural Networks Learn Nested Recursion? *Linguistic Issues in Language Technology*, 16(1):1–20.

N. Chomsky and M. P. Schützenberger. 1959. The Algebraic Theory of Context-Free Languages. In P. Braffort and D. Hirschberg, editors, *Studies in Logic and the Foundations of Mathematics*, volume 26 of *Computer Programming and Formal Systems*, pages 118–161. North-Holland Publishing Company, Amsterdam, Netherlands.

Jeffrey L. Elman. 1990. Finding Structure in Time. *Cognitive Science*, 14(2):179–211.

Patrick C. Fischer. 1966. Turing Machines with Restricted Memory Access. *Information and Control*, 9(4):364–379.

Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. 1968. Counter Machines and Counter Languages. *Mathematical systems theory*, 2(3):265–283.

Amirata Ghorbani, Abubakar Abid, and James Zou. 2019. Interpretation of Neural Networks Is Fragile. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):3681–3688.

Jeffrey Nicholas Heinz. 2007. *Inductive Learning of Phonotactic Patterns*. PhD Dissertation, University of California, Los Angeles, Los Angeles, CA, USA.

Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, and Rory Sayres. 2018. Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV). In *International Conference on Machine Learning, 10–15 July 2018, Stockholmsmässan, Stockholm Sweden*, volume 80 of *Proceedings of Machine Learning Research*, pages 2668–2677, Stockholm, Sweden. PMLR.

Pieter-Jan Kindermans, Sara Hooker, Julius Adebayo, Maximilian Alber, Kristof T. Schütt, Sven Dähne, Dumitru Erhan, and Been Kim. 2019. The (Un)reliability of Saliency Methods. In Wojciech Samek, Grégoire Montavon, Andrea Vedaldi, Lars Kai Hansen, and Klaus-Robert Müller, editors, *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, number 11700 in Lecture Notes in Computer Science, pages 267–280. Springer International Publishing, Cham, Switzerland.

Samuel A. Korsky and Robert C. Berwick. 2019. On the Computational Power of RNNs. *Computing Research Repository*, arXiv:1906.06349.

Jiwei Li, Xinlei Chen, Eduard Hovy, and Dan Jurafsky. 2016. Visualizing and Understanding Neural Models in NLP. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 681–691, San Diego, CA, USA. Association for Computational Linguistics.

Abhijit Mahalunkar and John Kelleher. 2019a. Multi-Element Long Distance Dependencies: Using SP_k Languages to Explore the Characteristics of Long-Distance Dependencies. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pages 34–43, Florence, Italy. Association for Computational Linguistics.

Abhijit Mahalunkar and John D. Kelleher. 2018. Using Regular Languages to Explore the Representational Capacity of Recurrent Neural Architectures. In *Artificial Neural Networks and Machine Learning – ICANN 2018*, volume 11141 of *Lecture Notes in Computer Science*, pages 189–198, Rhodes, Greece. Springer International Publishing.

Abhijit Mahalunkar and John D. Kelleher. 2019b. Understanding Recurrent Neural Architectures by Analyzing and Synthesizing Long Distance Dependencies in Benchmark Sequential Datasets. *Computing Research Repository*, arXiv:1810.02966v3.

William Merrill. 2019. Sequential Neural Networks as Automata. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pages 1–13, Florence, Italy. Association for Computational Linguistics.

Grégoire Montavon. 2019. Gradient-Based Vs. Propagation-Based Explanations: An Axiomatic Comparison. In Wojciech Samek, Grégoire Montavon, Andrea Vedaldi, Lars Kai Hansen, and Klaus-Robert Müller, editors, *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, number 11700 in Lecture Notes in Computer Science, pages 253–265. Springer International Publishing, Cham, Switzerland.

Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. 2018. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1–15.

W. James Murdoch, Peter J. Liu, and Bin Yu. 2018. Beyond Word Importance: Contextual Decomposition to Extract Interactions from LSTMs. In *ICLR 2018 Conference Track*, Vancouver, Canada. OpenReview.

Nina Poerner, Hinrich Schütze, and Benjamin Roth. 2018. Evaluating neural network explanation methods using hybrid documents and morphosyntactic agreement. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, volume 1: Long Papers, pages 340–350, Melbourne, Australia. Association for Computational Linguistics.

Wojciech Samek, Alexander Binder, Grégoire Montavon, Sebastian Lapuschkin, and Klaus-Robert Müller. 2017. Evaluating the Visualization of What a Deep Neural Network Has Learned. *IEEE Transactions on Neural Networks and Learning Systems*, 28(11):2660–2673.

Luzi Sennhauser and Robert Berwick. 2018. Evaluating the Ability of LSTMs to Learn Context-Free Grammars. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 115–124, Brussels, Belgium. Association for Computational Linguistics.

Avanti Shrikumar, Peyton Greenside, Anna Shcherbina, and Anshul Kundaje. 2017. Not Just a Black Box: Learning Important Features Through Propagating

Activation Differences. *Computing Research Repository*, arXiv:1605.01713.

Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2014. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In *ICLR 2014 Workshop Proceedings*, Banff, Canada. arXiv.

Natalia Skachkova, Thomas Trost, and Dietrich Klakow. 2018. Closing Brackets with Recurrent Neural Networks. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 232–239, Brussels, Belgium. Association for Computational Linguistics.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic Attribution for Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3319–3328, Sydney, Australia. PMLR.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. On the Practical Computational Power of Finite Precision RNNs for Language Recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, volume 2: Short Papers, pages 740–745, Melbourne, Australia. Association for Computational Linguistics.

Mengjiao Yang and Been Kim. 2019. Benchmarking Attribution Methods with Relative Feature Importance. *Computing Research Repository*, arXiv:1907.09701.

Xiang Yu, Ngoc Thang Vu, and Jonas Kuhn. 2019. Learning the Dyck Language with Attention-based Seq2Seq Models. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 138–146, Florence, Italy. Association for Computational Linguistics.

Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *Computer Vision – ECCV 2014*, volume 8689 of *Lecture Notes in Computer Science*, pages 818–833, Zurich, Switzerland. Springer International Publishing.

Jianming Zhang, Sarah Adel Bargal, Zhe Lin, Jonathan Brandt, Xiaohui Shen, and Stan Sclaroff. 2018. Top-Down Neural Attention by Excitation Backprop. *International Journal of Computer Vision*, 126(10):1084–1102.

A Detailed Descriptions of White-Box Networks

This appendix provides detailed descriptions of our four white-box networks.

A.1 Counting Task Network

As described in Subsection 4.1, the network for the counting task simply sets $g^{(t)}$ to $v = \tanh(u)$ when $\boldsymbol{x}^{(t)} = \texttt{a}$ and $-v$ when $\boldsymbol{x}^{(t)} = \texttt{b}$. All gates are fixed to 1. The output layer uses $h^{(t)} = \tanh\left(c^{(t)}\right)$ as the score for the *True* class and $v/2$ as the score for the *False* class.

$$
\begin{aligned}
g^{(t)} &= \tanh\left(u \left[\begin{array}{cc} 1 & -1 \end{array}\right] \boldsymbol{x}^{(t)}\right) \\
f^{(t)} &= \sigma(m) \\
i^{(t)} &= \sigma(m) \\
o^{(t)} &= \sigma(m) \\
\hat{\boldsymbol{y}}^{(t)} &= \left[\begin{array}{c} 1 \\ 0 \end{array}\right] h^{(t)} + \left[\begin{array}{c} 0 \\ v/2 \end{array}\right]
\end{aligned}
$$

A.2 SP Task Network (Counter-Based)

The seven counters for the SP task are implemented as follows. First, we compute $\boldsymbol{g}^{(t)}$ under the assumption that one of the first four counters is always incremented, and one of the last three counters is always incremented as long as $\boldsymbol{x}^{(t)} \neq \texttt{a}$.

$$
\boldsymbol{g}^{(t)} = \tanh\left(u \left[\begin{array}{c} \boldsymbol{I}_4 \\ \hdashline \begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \end{array}\right] \boldsymbol{x}^{(t)}\right)
$$

Then, we use the input gate to condition the last three counters on the value of the first four counters. For example, if $h_1^{(t-1)} = 0$, then no as have been encountered in the input string before time t. In that case, the input gate for counter #5, which represents subsequences ending with b, is set to $i_5^{(t)} = \sigma(-m) \approx 0$. This is because a b encountered at time t would not form part of a subsequence if no as have been encountered so far, so counter #5 should not be incremented.

$$
\begin{aligned}
\boldsymbol{i}^{(t)} = \sigma\Bigg(2m \left[\begin{array}{cc:c} \multicolumn{2}{c:}{\mathbf{0}} & \mathbf{0} \\ \hdashline \begin{array}{cc} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{array} & \begin{array}{cc} 0 & 0 \\ 1 & 0 \\ 1 & 0 \end{array} & \begin{array}{c} \\ \mathbf{0} \\ \end{array} \end{array}\right] \boldsymbol{h}^{(t-1)} \\
+m \left[\begin{array}{ccccccc} 1 & 1 & 1 & 1 & -1 & -1 & -1 \end{array}\right]^\top\Bigg)
\end{aligned}
$$

All other gates are fixed to $\mathbf{1}$. The output layer sets the score of the *True* class to $h_5^{(t)} + h_6^{(t)} + h_7^{(t)}$ and the score of the *False* class to $v/2$.

$$
\begin{aligned}
\boldsymbol{f}^{(t)} &= \sigma(m\mathbf{1}) \\
\boldsymbol{o}^{(t)} &= \sigma(m\mathbf{1}) \\
\hat{\boldsymbol{y}}^{(t)} &= \left[\begin{array}{c:ccc} \mathbf{0} & 1 & 1 & 1 \\ \mathbf{0} & 0 & 0 & 0 \end{array}\right] \boldsymbol{h}^{(t)} + \left[\begin{array}{c} 0 \\ v/2 \end{array}\right]
\end{aligned}
$$

A.3 FSA Network

Here we describe a general construction of an LSTM simulating an FSA with states Q, accepting states $Q_F \subseteq Q$, alphabet Σ, and transition function $\delta : Q \times \Sigma \to Q$. Recall that $\boldsymbol{h}^{(t)}$ contains a one-hot representation of pairs in $Q \times \Sigma$ encoding the current state of the FSA and the most recent input symbol. The initial state $\boldsymbol{h}^{(0)} = \mathbf{0}$ represents the starting configuration of the FSA.

At a high level, the state transition system works as follows. First, $\boldsymbol{g}^{(t)}$ first marks all the positions corresponding to the current input $\boldsymbol{x}^{(t)}$.[6]

$$g^{(t)}_{\langle q,x\rangle} = \begin{cases} v, & x = \boldsymbol{x}^{(t)} \\ 0, & \text{otherwise} \end{cases}$$

The input gate then filters out any positions that do not represent valid transitions from the previous state q', which is recovered from $\boldsymbol{h}^{(t-1)}$.

$$i^{(t)}_{\langle q,x\rangle} = \begin{cases} 1, & \delta(q', x) = q \\ 0, & \text{otherwise} \end{cases}$$

Now, we describe how this behavior is implemented in our LSTM.

The cell state update is straightforwardly implemented as follows:

$$\boldsymbol{g}^{(t)} = \tanh\left(u\boldsymbol{W}^{(c,x)}\boldsymbol{x}^{(t)}\right),$$

where

$$W^{(c,x)}_{\langle q,x\rangle,j} = \begin{cases} 1, & j \text{ is the index for } x \\ 0, & \text{otherwise.} \end{cases}$$

Observe that the matrix $\boldsymbol{W}^{(c,x)}$ essentially contains a copy of $\boldsymbol{I}_4$ for each state, such that each copy is distributed across the different cell state units designated for that state.

The input gate is more complex. First, the bias term handles the case where the current case is the starting state q_0. This is necessary because the initial configuration of the network is represented by $\boldsymbol{h}^{(0)} = \mathbf{0}$.

$$b^{(i)}_{\langle q,x\rangle} = \begin{cases} m, & \delta(q_0, x) = q \\ -m, & \text{otherwise} \end{cases}$$

The bias vector sets $i^{(t)}_{\langle q,x\rangle}$ to be 1 if the FSA transitions from q_0 to q after reading x, and 0 otherwise. We replicate this behavior for other values of $\boldsymbol{h}^{(t-1)}$ by using the weight matrix $\boldsymbol{W}^{(i,h)}$, taking the bias vector into account:

$$\boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{W}^{(i,h)}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}^{(i)}\right),$$

where

$$W^{(i)}_{\langle q,x\rangle,\langle q',x'\rangle} = \begin{cases} m - b^{(i)}_{\langle q,x\rangle}, & \delta(q', x) = q \\ -m - b^{(i)}_{\langle q,x\rangle}, & \text{otherwise.} \end{cases}$$

The forget gate is fixed to $-\mathbf{1}$, since the state needs to be updated at every time step. The output gate is fixed to $\mathbf{1}$.

$$\boldsymbol{f}^{(t)} = \sigma(-m\mathbf{1})$$
$$\boldsymbol{o}^{(t)} = \sigma(m\mathbf{1})$$

The output layer simply selects hidden units that represent accepting and rejecting states:

$$\hat{\boldsymbol{y}}^{(t)} = \boldsymbol{W}\boldsymbol{h}^{(t)},$$

where

$$W_{c,\langle q,x\rangle} = \begin{cases} 1, & c = \textit{True} \text{ and } q \in Q_F \\ 1, & c = \textit{False} \text{ and } q \notin Q_F \\ 0, & \text{otherwise.} \end{cases}$$

A.4 PDA Network

Finally, we describe how the PDA network for the bracket prediction task is implemented. Of the four networks, this one is the most intricate. Recall from Subsection 4.2 that we implement a bounded stack of size k using $2k + 1$ hidden units, with the following interpretation:

- $\boldsymbol{c}^{(t)}_{:k-1}$ contains the stack, except for the top item
- $c^{(t)}_k$ contains the top item of the stack
- $\boldsymbol{c}^{(t)}_{k+1:2k}$ contains the height of the stack in unary notation
- c_{2k+1} is a bit, which is set to be positive if the stack is empty and nonpositive otherwise.

We represent the brackets `(`, `[`, `)`, and `]` in one-hot encoding with the indices 1, 2, 3, and 4, respectively. The opening brackets `(` and `[` are represented on the stack by 1 and -1, respectively. T

[6] We use $v = \tanh(1) \approx 0.762$.

We begin by describing $\boldsymbol{g}^{(t)}$. Due to the complexity of the network, we describe the weights and biases individually, which are combined as follows.

$$\boldsymbol{g}^{(t)} = \tanh\left(m\left(\boldsymbol{z}^{(g,t)}\right)\right), \text{ where}$$
$$\boldsymbol{z}^{(g,t)} = \boldsymbol{W}^{(c,x)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(c,h)}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}^{(c)}$$

First, the bias vector sets $c^{(t)}_{2k+1}$ to be 1, indicating that the stack is empty. This ensures that the initial hidden state $\boldsymbol{h}^{(t)} = \mathbf{0}$ is treated as an empty stack.

$$\boldsymbol{b}^{(c)} = \begin{bmatrix} \mathbf{0} \\ \hdashline 2 \end{bmatrix}$$

$\boldsymbol{W}^{(c,x)}$ serves three functions when $\boldsymbol{x}^{(t)}$ is an open bracket, and does nothing when $\boldsymbol{x}^{(t)}$ is a closing bracket. First, it pushes $\boldsymbol{x}^{(t)}$ to the top of the stack, represented by $c^{(t)}_k$. The values $\pm 2^k$ are determined by equation (1) in Subsection 4.2. Second, it sets $\boldsymbol{g}^{(t)}_{k+1:2k}$ to $\mathbf{1}$ in order to increment the unary counter for the height of the stack. Later, we will see that the input gate filters out all positions except for the top of the stack. Finally, $\boldsymbol{W}^{(c,x)}$ sets the empty stack indicator to -1, indicating that the stack is not empty.

$$\boldsymbol{W}^{(c,x)} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \hdashline 2^k & -2^k & 0 & 0 \\ \hdashline \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \hdashline -2 & -2 & 0 & 0 \end{bmatrix}$$

$\boldsymbol{W}^{(c,h)}$ performs two functions. First, it completes equation (1) for $c^{(t)}_k$, setting it to be the second-highest stack item from the previous time step. Second, it copies the top of the stack to the first $k-1$ positions, with the input gate filtering out all but the highest position.

$$\boldsymbol{W}^{(c,h)} = \left[\begin{array}{ccccc:c:c:c} & & \mathbf{0} & & & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \hdashline 2 & 4 & \cdots & & 2^{k-1} & 0 & \mathbf{0} & 0 \\ \hdashline & & \mathbf{0} & & & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \hdashline & & \mathbf{0} & & & 0 & \mathbf{-1} & 0 \end{array}\right]$$

Finally, the -1s serve to decrease the empty stack indicator by an amount proportional to the stack height at time $t-1$. Observe that if $\boldsymbol{x}^{(t)}$ is a closing bracket and $\boldsymbol{h}^{(t-1)}$ represents a stack with only one item, then

$$\boldsymbol{W}^{(c,x)}_{2k+1,:}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(c,h)}_{2k+1,:}\boldsymbol{h}^{(t-1)} + b^{(c)}_{2k+1}$$
$$= -1 + 2 = 1,$$

so the empty stack indicator is set to 1, indicating that the stack is empty. Otherwise,

$$\boldsymbol{W}^{(c,x)}_{2k+1,:}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(c,h)}_{2k+1,:}\boldsymbol{h}^{(t-1)} \leq -2,$$

so the empty stack indicator is nonpositive.

Now, we describe the input gate, given by the following.

$$\boldsymbol{i}^{(t)} = \sigma\left(m\left(\boldsymbol{z}^{(i,t)}\right)\right)$$
$$\boldsymbol{z}^{(i,t)} = \boldsymbol{W}^{(i,x)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(i,h)}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}^{(i)}$$

$\boldsymbol{W}^{(i,x)}$ sets the input gate for the first $k-1$ positions to 0 when $\boldsymbol{x}^{(t)}$ is a closing bracket. In that case, an item needs to be popped from the stack, so nothing can be copied to these hidden units. When $\boldsymbol{x}^{(t)}$ is an opening bracket, $\boldsymbol{W}^{(i,x)}$ sets $i^{(t)}_k = 1$, so that the bracket can be copied to the top of the stack.

$$\boldsymbol{W}^{(i,x)} = 2\begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{-1} \\ \hdashline 1 & 1 & 0 & 0 \\ \hdashline \multicolumn{4}{c}{\mathbf{0}} \end{bmatrix}$$

$\boldsymbol{W}^{(i,h)}$ uses a matrix $\boldsymbol{T}_n \in \mathbb{R}^{n \times n}$, defined below.

$$\boldsymbol{T}_n = \begin{bmatrix} 1 & -1 & 0 & \dots & 0 & 0 \\ 0 & 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -1 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

Suppose $\boldsymbol{v}$ represents the number s in unary notation: v_j is 1 if $j \leq s$ and 0 otherwise. $\boldsymbol{T}_n$ has the special property that $\boldsymbol{T}_n\boldsymbol{v}$ is a one-hot vector for s. Based on this, $\boldsymbol{W}^{(i,h)}$ is defined as follows.

$$\boldsymbol{W}^{(i,h)} = 2\left[\begin{array}{c:c:c} \mathbf{0} & (\boldsymbol{T}_k)_{:k-1,:} & \mathbf{0} \\ \hdashline & \mathbf{0} & \\ \hdashline & (\boldsymbol{T}_k)_{:k-1,:} & \\ \hdashline & \mathbf{0} & \end{array}\right]$$

$\boldsymbol{W}^{(i,h)}_{:k-1,k+1:2k}$ contains $\boldsymbol{T}_k$, with the last row truncated. This portion of the matrix converts $\boldsymbol{h}^{(t-1)}_{k+1:2k}$, which contains a unary encoding of the stack height, to a one-hot vector marking the position of the top of the stack. This ensures that, when pushing to the stack, the top stack item from time $t-1$ is only copied to the appropriate position of $\boldsymbol{h}^{(t)}_{:k-1}$. The other copy of $\boldsymbol{T}_k$, again with the last row omitted, occurs in $\boldsymbol{W}^{(i,h)}_{k+2:2k,k+1:2k}$. This copy of $\boldsymbol{T}_k$ ensures that when the unary counter for the

stack height is incremented, only the appropriate position is updated. Finally, the bias vector ensures that the top stack item and the empty stack indicator are always updated.

$$\boldsymbol{b}^{(i)} = \begin{bmatrix} -\mathbf{1} \\ \hdashline 1 \\ \hdashline -\mathbf{1} \\ \hdashline 1 \end{bmatrix}$$

The forget gate is responsible for deleting portions of memory when stack items are popped.

$$\begin{aligned} \boldsymbol{f}^{(t)} &= \sigma\left(m\left(\boldsymbol{z}^{(f,t)}\right)\right) \\ \boldsymbol{z}^{(f,t)} &= \boldsymbol{W}^{(f,x)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(f,h)}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}^{(f)} \end{aligned}$$

$\boldsymbol{W}^{(f,x)}$ first ensures that no stack items are deleted when an item is pushed to the stack.

$$\boldsymbol{W}^{(f,x)} = 2\begin{bmatrix} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \hdashline 0 & 0 & 0 & 0 \\ \hdashline \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \hdashline 0 & 0 & 0 & 0 \end{bmatrix}$$

Next, $\boldsymbol{W}^{(f,h)}$ marks the second highest stack position and the top of the unary counter for deletion, in case an item needs to be popped.

$$\boldsymbol{W}^{(f,h)} = 2\left[\begin{array}{c|c|c} \mathbf{0} & \begin{array}{c} -\left(\boldsymbol{T}_k\right)_{2:,:} \\ \hdashline \mathbf{0} \\ \hdashline -\boldsymbol{T}_k \\ \hdashline \mathbf{0} \end{array} & \mathbf{0} \end{array}\right]$$

Finally, the bias term ensures that the top stack item and empty stack indicator are always cleared.

$$\boldsymbol{b}^{(i)} = \begin{bmatrix} \mathbf{1} \\ \hdashline -1 \\ \hdashline \mathbf{1} \\ \hdashline -1 \end{bmatrix}$$

To complete the construction, we fix the output gate to $\mathbf{1}$, and have the output layer read the top stack position:

$$\begin{aligned} \boldsymbol{o}^{(t)} &= \sigma(m\mathbf{1}) \\ \hat{\boldsymbol{y}}^{(t)} &= \boldsymbol{W}\boldsymbol{h}^{(t)}, \end{aligned}$$

where

$$W_{c,j} = \begin{cases} 1, & c = \texttt{)} \text{ and } j = k \\ -1, & c = \texttt{]} \text{ and } j = k \\ 1, & c = \textit{None} \text{ and } j = 2k+1 \\ 0, & \text{otherwise.} \end{cases}$$