Learning Operations on a Stack with Neural Turing Machines

Anonymous Author(s)
Affiliation
Address
email

Abstract

Multiple extensions of Recurrent Neural Networks (RNNs) have been proposed recently to address the difficulty of storing information over long time periods. In this paper, we experiment with the capacity of Neural Turing Machines (NTMs) to deal with these long-term dependencies on well-balanced strings of parentheses. We show that not only does the NTM emulate a stack with its heads and learn an algorithm to recognize such words, but it is also capable of strongly generalizing to much longer sequences.

1 Introduction

Although neural networks shine at finding meaningful representations of the data, they are still limited in their capacity to plan ahead, reason and store information over long time periods. Keeping track of nested parentheses in a language model, for example, is a particularly challenging problem for RNNs [9]. It requires the network to somehow memorize the number of unmatched open parentheses. In this paper, we analyze the ability of Neural Turing Machines (NTMs) to recognize well-balanced strings of parentheses. We show that even though the NTM architecture does not explicitely operate on a stack, it is able to emulate this data structure with its heads. Such a behaviour was unobserved on other simple algorithmic tasks [4].

After a brief recall of the Neural Turing Machine architecture in Section 3, we show in Section 4 how the NTM is able to learn an algorithm to recognize strings of well-balanced parentheses, called *Dyck words*. We also show how this model is capable to strongly generalize to longer sequences.

2 Related Work

Grammar induction Deep learning models are often trained on large datasets, generally extracted from real-world data at the cost of an expensive labeling step by some expert. In the context of Natural Language Processing, an alternative is to generate data from an artificial language, based on a predefined grammar. Historically, these *formal languages* have been used to evaluate the theoretical foundations of RNNs [14].

Hochreiter and Schmidhuber [7] tested their new Long Short-Term Memory (LSTM) on the *embedded Reber language*, to show how their output gates can be beneficial. This behaviour was later extended to a variety of context-free and context-sensitive languages [3]. However, as opposed to these previous works focused on character-level language modeling, here our task of interest is the *membership problem*. This is a classification problem, where positive examples are generated by a given grammar, and negative examples are randomly generated with the same alphabet.

Differentiable memory To enhance their capacity to retain information, RNNs can be augmented with an explicit and differentiable memory module. Memory Networks and Dynamic Memory

Submitted to 29th Conference on Neural Information Processing Systems (NIPS 2016). Do not distribute.

Networks [16, 11, 17] use a hierarchical attention mechanism on an associative array to solve text QA tasks involving reasoning. Closely related our work, Stack-augmented RNNs [8] are capable of inferring algorithmic patterns on some context-free and context-sensitive languages, including $a^n b^n$, $a^n b^n c^n$, and $a^n b^m c^{n+m}$.

3 Neural Turing Machines

The Neural Turing Machine (NTM) [4] is an instance of memory-augmented neural networks, consisting of a neural network *controller* which interacts with a large (though bounded, unlike a Turing machine) *memory tape*. The NTM uses soft read and write heads to retrieve information from the memory and store information in memory. The dynamics of these heads are governed by one or multiple sets of weights $\mathbf{w}_t^r$ for the read head(s) and $\mathbf{w}_t^w$ for the write head(s). These are controlled by the controller (either a Feed-forward network, or an LSTM), and maintain the overall architecture differentiable. The read head returns a read vector $\mathbf{r}_t$ as a weighted sum over the rows of the memory bank M_t:

$$\mathbf{r}_t = \sum_i \mathbf{w}_t^r(i) M_t(i) \tag{1}$$

Similarly, the write head modifies the memory M_t by first erasing a weighted version of some *erase vector* $\mathbf{e}_t$ from each row in the memory (Equation 2), then adding a weighted version of an *add vector* $\mathbf{a}_t$ (Equation 3). Both vectors $\mathbf{e}_t$ and $\mathbf{a}_t$ are generated by the controller.

$$\tilde{M}_{t+1}(i) \quad \longleftarrow \quad M_t(i) \cdot (1 - \mathbf{w}_t^w(i)\, \mathbf{e}_t) \tag{2}$$

$$M_{t+1}(i) \quad \longleftarrow \quad \tilde{M}_{t+1}(i) + \mathbf{w}_t^w(i)\, \mathbf{a}_t \tag{3}$$

The weights $\mathbf{w}_t^r$ and $\mathbf{w}_t^r$ are produced through a series of differentiable operations, called the addressing mechanisms. These fall into two categories: a *content-based addressing* comparing each memory locations with some key $\mathbf{k}_t$, and a *location-based addressing* responsible for shifting the heads (similar to a Turing machine). Even though recent works [13, 6] tend to drop the location-addressing, we chose to use the original formulation of the NTM and keep both addressing mechanisms.

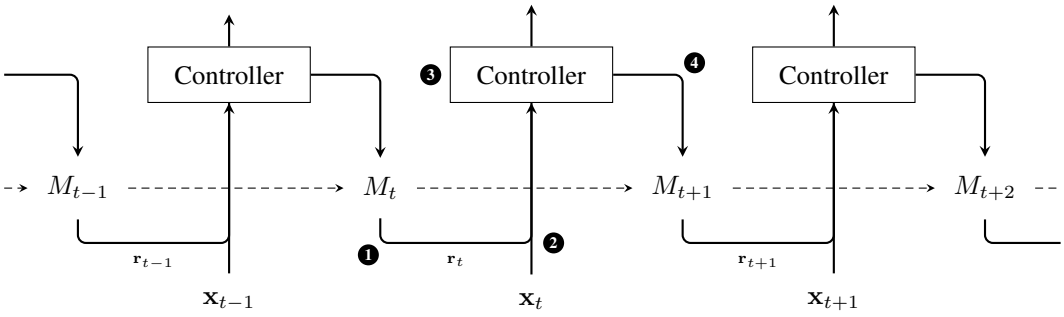

Figure 1: *A Neural Turing Machine, unrolled in time* – (1) The read head first returns a read vector $\mathbf{r}_t$ which is (2) concatenated with the input $\mathbf{x}_t$. Both vectors are sent to (3) the controller (either a Feed-forward network, or an LSTM) which is responsible for the computation of the internal state of the NTM, as well as the read and write heads. (4) This write head is then used to makes changes to the memory M_{t+1}.

4 Experiments

4.1 Dyck words

A *Dyck word* is a balanced string of opening and closing parentheses. Besides the important role they play in parsing tasks, they have multiple connections with other combinatorial objects [15, 2]. In

particular, one convenient and visual way representation of a Dyck word is a path on the integer line (see Figure 2).

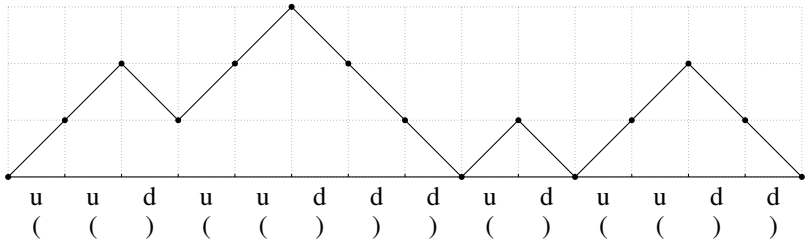

Figure 2: *Example of a Dyck word* – This is an example of a well-balanced string of parentheses (bottom), along with its representation in $\mathcal{A}^*$ (middle) and graphical representation as a path (top).

To avoid ambiguities, we will consider strings of parentheses as words $w \in \{u, d\}^* = \mathcal{A}^*$, where each character u corresponds to an opening parenthesis and d to a closing parenthesis. The subset of $\mathcal{A}^*$ containing the Dyck words of length $< 2n$ is called the *Dyck language* and is denoted $\mathcal{D}_{<2n}$.

4.2 Experimental setup

We are interested here in the membership problem over the Dyck language. We trained a NTM for a classification task, where positive examples are uniformly sampled [2] from the Dyck language $\mathcal{D}_{<12}$, and negative examples are non-Dyck words $w \in \mathcal{A}^*$ of length < 12 with the same number of characters u and d. We use the same experimental setup as described in [4], with a 100-hidden units feed-forward controller, 1 read head, 1 write head, and a memory bank containing 128 memory locations, each of dimension 20. We used a ReLU nonlinearity for the key $\mathbf{k}_t$ and add vector $\mathbf{a}_t$ and a hard sigmoid for the erase vector $\mathbf{e}_t$. We trained the model using the Adam optimizer [10] with a learning rate of 0.001 and batch size 16.

4.3 Stack emulation

The Dyck language is a context-free language that can be recognized by a pushdown automaton [1]. Here, we are interested in the nature of the algorithm the NTM is able to infer only from examples on this task. More specifically, we want to know if, and how, the NTM uses its memory to act as a stack, without specifying the push and pop operations explicitely [8, 5]. In Figure 3, we show the behaviour of the read and write heads on a Dyck word and a non-Dyck word, along with the probability returned by the model of each prefix to be a Dyck word.

We observe that the model is actually emulating a stack with its read head. Each time the NTM reads an opening parenthesis u, the read head is moved upward and conversely when reading a closing parenthesis d. This behaviour is different from what was previously reported on other algorithmic tasks [4], where the content of the memory played a central role. Here, the NTM barely writes anything in memory, but uses its read head for computation purposes, following closely the graphical representation of the words (on the right).

In the case of non-Dyck words, the read head is used up until the first time the model reads a closing parenthesis with no matching opening parenthesis (illustrated by the red line in the graphical representation of the word). Beyond this point, the NTM correctly predicts the word is no longer a Dyck word, and stops using its read head by placing equal mass over all the memory locations.

4.4 Strong generalization

When testing a model, it is often assumed that the training and test data are sampled from the same (unknown) distribution. However, here we are not only interested in the capacity of the NTM to recognize Dyck words of similar length, but also its capacity to learn an algorithm and generalize to longer sequences. This is called *strong generalization* [12].

In Figure 4, we compare the NTM with an LSTM trained on the same task with similar hyperparameters. While the LSTM shows signs of strong generalization on sequences twice as long as what it was

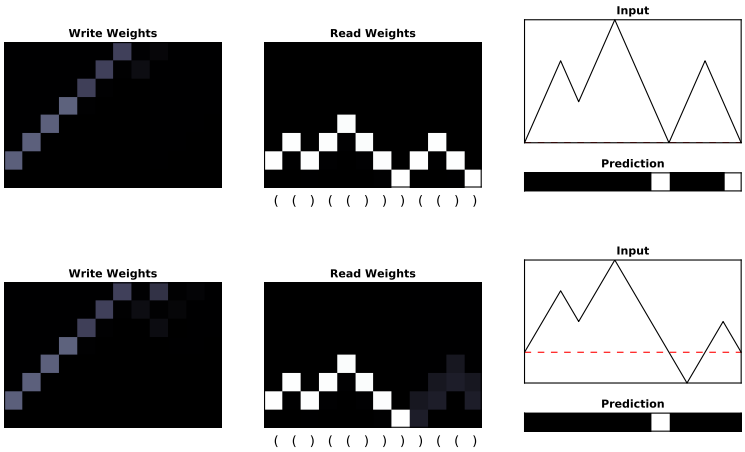

Figure 3: *Read and write heads of the NTM* – Examples of the behaviour of the NTM on a Dyck word (top) and a non-Dyck word (bottom) of length 12. For each example, we show the write (left) and read (center) weights as the NTM reads the input string.

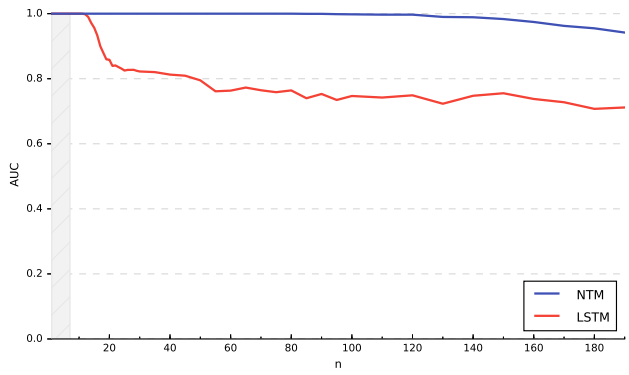

Figure 4: *Generalization on* $\mathcal{D}_{<2n}$ – Strong generalization performance of a NTM (blue) and an LSTM (red) on sequences in $\mathcal{D}_{<2n}$, for different values of n. The gray area represents the training regime ($\mathcal{D}_{<12}$) for both models. The performance is reported as the Area Under the Curve (AUC).

given during training, the AUC starts dropping for much longer sequences. On the other hand, the NTM generalizes perfectly even for much longer sequences (up to 20 times longer than the training regime). Beyond $n \approx 120$, the AUC starts to slightly decrease, most likely due to overflow issues: the stack emulated by the read head is limited by the number of memory locations, here 128.

5 Conclusion

Through an experiment on an artificial language called the Dyck language, we have shown that Neural Turing Machines are not only able to use their memory for storage, but can also use their heads for computational purposes. This allows the NTM to strongly generalize to inputs much longer, effectively learning an algorithm (contrary to only learning patterns in the data). The size of the memory allocated for the NTM being the only constraint. An interesting line of research could then be to run a similar experiment on a model trained under a memory-restricted regime, like a single memory location, and see how the NTM can emulate a stack under this stronger constraint.

References

[1] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. Context-Free Languages and Push-Down Automata. In *Handbook of Formal Languages*, pages 111–174. Springer, 1997.

[2] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 1 edition, 2009.

[3] Felix A Gers and Jürgen Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. *Neural Networks, IEEE Transactions on*, 12(6):1333–1340, 2001.

[4] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *CoRR*, abs/1410.5401, 2014.

[5] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to Transduce with Unbounded Memory. *CoRR*, abs/1506.02516, 2015.

[6] Çaglar Gülçehre, Sarath Chandar, Kyunghyun Cho, and Yoshua Bengio. Dynamic Neural Turing Machine with Soft and Hard Addressing Schemes. *CoRR*, abs/1607.00036, 2016.

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

[8] Armand Joulin and Tomas Mikolov. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. *CoRR*, abs/1503.01007, 2015.

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

[10] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[11] Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. Key-Value Memory Networks for Directly Reading Documents. *CoRR*, abs/1606.03126, 2016.

[12] Scott E. Reed and Nando de Freitas. Neural Programmer-Interpreters. *CoRR*, abs/1511.06279, 2015.

[13] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy P. Lillicrap. One-shot Learning with Memory-Augmented Neural Networks. *CoRR*, abs/1605.06065, 2016.

[14] Hava (Eve) Tova Siegelmann. Foundations of recurrent neural networks, 1993.

[15] Richard P. Stanley. *Enumerative combinatorics. Volume 2*. Cambridge studies in advanced mathematics. Cambridge university press, Cambridge, New York, 1999.

[16] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. Weakly Supervised Memory Networks. *CoRR*, abs/1503.08895, 2015.

[17] Caiming Xiong, Stephen Merity, and Richard Socher. Dynamic Memory Networks for Visual and Textual Question Answering. *CoRR*, abs/1603.01417, 2016.