Scalable Online State Construction using Recurrent Networks

Khurram Javed
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
kjaved@ualberta.ca

Haseeb Shah
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
hshah1@ualberta.ca

Rich Sutton
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
rsutton@ualberta.ca

Martha White
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
whitem@ualberta.ca

Abstract

State construction from sensory observations is an important component of a reinforcement learning agent. One solution for state construction is to use recurrent neural networks. Two popular gradient-based methods for recurrent learning are back-propagation through time (BPTT), and real-time recurrent learning (RTRL). BPTT looks at the complete sequence of observations before computing gradients and is unsuitable for online updates. RTRL can do online updates but scales poorly to large networks. In this paper, we propose two constraints that make RTRL scalable. We show that by either decomposing the network into independent modules or learning a recurrent network incrementally, we can make RTRL scale linearly with the number of parameters. Unlike prior scalable gradient estimation algorithms, such as UORO and Truncated-BPTT, our algorithms do not add bias or noise to the gradient estimate. Instead, they trade off the functional capacity of the recurrent network to achieve scalable learning. We demonstrate the effectiveness of our approach on a prediction learning benchmark inspired by animal learning.

Keywords: agent-state construction, recurrent learning, online learning, scalable recurrent learning

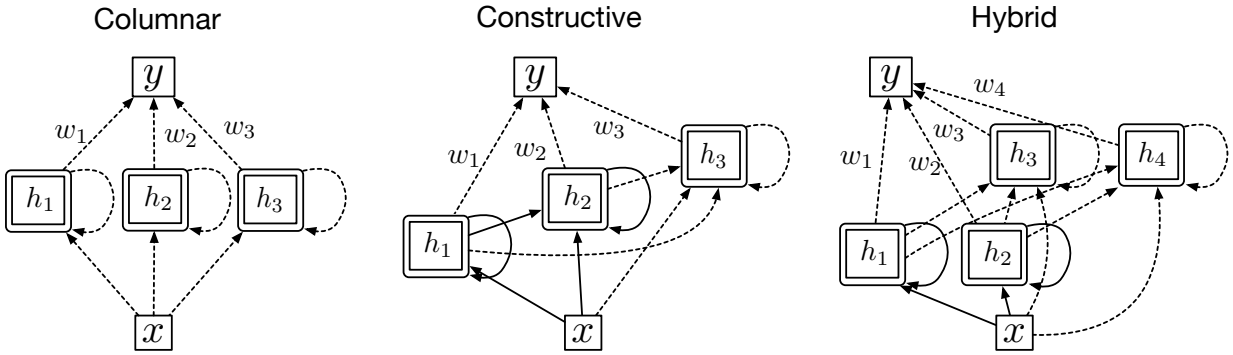

Figure 1: Three structures of recurrent neural networks for which gradients can be estimated in a scalable way without bias or noise. Dotted lines represent parameters that are updated at every step, whereas solid lines are weights that are fixed to prevent bias in the gradient estimate of remaining parameters. Recurrent networks with a columnar structure can be trained end-to-end using gradients without any truncation, only requiring $O(n)$ operations and memory per step. However, columnar networks do not have hierarchical recurrent features—recurrent features made out of other recurrent features. Constructive networks have hierarchical recurrent features, however must be trained incrementally to prevent bias in the gradient estimate. Incremental learning is achieved by initializing all w_i to zero, and learning h_1, h_2, and h_3 in order. Finally, columnar and constructive networks can be combined to get hybrid networks. The pairs (h_1, h_2) and (h_3, h_4) do not depend on each other, and can learn in parallel. However, (h_3, h_4) must be learned after the pair (h_1, h_2) has been learned and fixed.

1 Introduction

Learning by interacting with the world is a powerful framework for building systems that can autonomously achieve goals in complex worlds. A key ingredient for building autonomous systems is agent-state construction—learning a compact representation of the history of interactions that helps in predicting and controlling the future. One solution for state construction is to use differentiable recurrent neural networks (RNNs) learned to minimize prediction errors.

State construction using RNNs requires structural credit assignment—identifying how to change network parameters to improve predictions. In RNNs, a parameter can influence a prediction made in the future. Effective credit assignment requires tracking the influence of the parameter on future predictions. Two popular algorithms for gradient-based structural credit assignment in recurrent neural networks are Back-propagation through time (BPTT) (Werbos, 1988; Robinson and Fallside, 1987) and real-time recurrent learning (RTRL) (Williams and Zipser 1989).

BPTT and RTRL are not suitable for online state construction for large problems. BPTT requires storing all past activations for estimating the gradient. Additionally, it requires computation proportional to the length of the sequence seen so far. RTRL can estimate the gradient on the go, and does not require more computation as the length of the sequence grows. However, RTRL scales poorly with an increase in the number of parameters of the RNN. Both BPTT and RTRL can be approximated, to make them more suitable for online learning.

A promising direction to scale gradient-based credit-assignment to large networks is to approximate the gradient. Elman (1990) proposed to ignore the influence of parameters on future predictions completely for training RNNs. This resulted in a scalable but biased algorithm. Williams and Peng (1990) proposed a more general algorithm called Truncated BPTT (T-BPTT). T-BPTT tracks the influence of all parameters on predictions made up to k steps in the future. T-BPTT limits the BPTT computation to last k steps, and works well for a range of problems (Mikolov *et al.*, 2009, 2010; Sutskever, 2013 and Kapturowski *et al.*, 2018). Its main limitation is that the resultant gradient is blind to long-range dependencies. Mujika *et al.* (2018) showed that on a simple copy task, T-BPTT failed to learn dependencies beyond the truncation window. Tallec *et al.* (2017) demonstrated T-BPTT can even diverge when a parameter has a negative long-term effect on a target and a positive short-term effect. A diagonal approximation to RTRL was used by Hochreiter and Schmidhuber (1997) in the original LSTM paper that scales linearly with the number of parameters. The same approximation is also a special case of the SnAp-k algorithm proposed by Menick *et al.* (2021) when $k = 1$. Diagonal-RTRL is not blind to long-term dependencies, but introduces significant bias in the gradient estimate for dense recurrent networks.

Existing approximations to gradient-based learning approximate the estimate of the gradient, but keep the function class of the network the same. These approximations either introduce bias, which can result in poor performance or even divergence, or noise in the gradient estimate, making learning extremely slow. In this work, we propose a different strategy: instead of introducing bias or noise in the gradient estimate, we limit the function class of the RNNs to allow scalable, unbiased and noise-free gradient estimation.

1.1 Recurrent learning

A recurrent neural network consist of a hidden state vector represented by $\mathbf{h}_t \in \mathbb{R}^d$. The agent uses a weight vector $\mathbf{w}_t \in \mathbb{R}^d$ to make a prediction y_t at time t as:

$$y_t = \sum_{k=0}^{d} h_{t,k} w_{t,k} \tag{1}$$

where

$$\mathbf{h}_t = f\left(\mathbf{h}_{t-1}, \mathbf{x}_t, \theta_t\right). \tag{2}$$

$h_{t,k}$ and $w_{t,k}$ are the kth element of $\mathbf{h}_t$ and $\mathbf{w}_t$, respectively. θ_t are the parameters of the RNN at time t and f is the dynamics function of the recurrent network. Given this notation, we can write the gradient of a prediction y_t w.r.t the parameters $\theta_{1:t}$ as

$$\frac{\partial y_t}{\partial \theta_{1:t}} = \frac{\partial y_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \theta_{1:t}} \tag{3}$$

We can expand the second term using the recursive relation:

$$\frac{\partial \mathbf{h}_t}{\partial \theta_{1:t}} = \frac{\partial \mathbf{h}_t}{\partial \theta_t} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \theta_{1:t-1}} \tag{4}$$

to get

$$\frac{\partial y_t}{\partial \theta_{1:t}} = \frac{\partial y_t}{\partial \mathbf{h}_t} \left(\frac{\partial \mathbf{h}_t}{\partial \theta_t} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \theta_{1:t-1}} \right) \tag{5}$$

We can compute the gradient in equation 5 using BPTT or RTRL. BPTT stores the network activations from prior steps and uses the expansion in equation 4 repeatedly to compute the gradient. RTRL, on the other hand, maintains the jacobian $\frac{\partial \mathbf{h}_k}{\partial \theta_{1:k}}$ using equation 4 at every step. To get the gradient w.r.t the prediction, it uses the pre-computed jacobian in equation 3. As a results, gradients are readily available at every step. However, computing the jacobian using equation 4 requires $O(|\mathbf{h}_t|^2|\theta_t|)$ operations and $O(|\mathbf{h}_t|\theta_t|)$ memory, and scales poorly to large networks.

2 Proposed methods

The key idea behind our methods is to structure a recurrent learning system in a way to make RTRL scale linearly with the number of parameters. We propose three approaches: (1) Columnar networks, (2) Constructive networks, and (3) Hybrid networks.

2.1 Columnar Networks

The first approach, called Columnar Networks, organizes the recurrent network such each scalar recurrent feature is independent of other recurrent features. Let $h_{t,k}$ be the kth index of the state vector $\mathbf{h}_t$. Then, in columnar networks,

$$h_{t,k} = f_k(h_{t-1,k}, \mathbf{x}_t, \theta_{t,k}). \tag{6}$$

Each f_k outputs a scalar recurrent feature, and is called a column. $\theta_{t,k}$ are the sets of parameters of the column i and j, and are disjoint for $i \neq j$. A columnar network consists of d columns. The output of all columns are concatenated to get d-dimensional $\mathbf{h}_t$. In this work, we implement each column as a single LSTM cell with a hidden size of one.

Because recurrent features in a columnar network are independent of each other, we can apply RTRL to each of them individually. The computation cost of RTRL is $O(|\mathbf{h}_t|^2|\theta_i|)$. For a single column, it reduces to $O(|\theta_i|)$, since $|h_i| = 1$. The cost for all the columns is

$$O(|\theta_{t,1}|) + \cdots O(\theta_{t,n}|) = O(|\theta_t|). \tag{7}$$

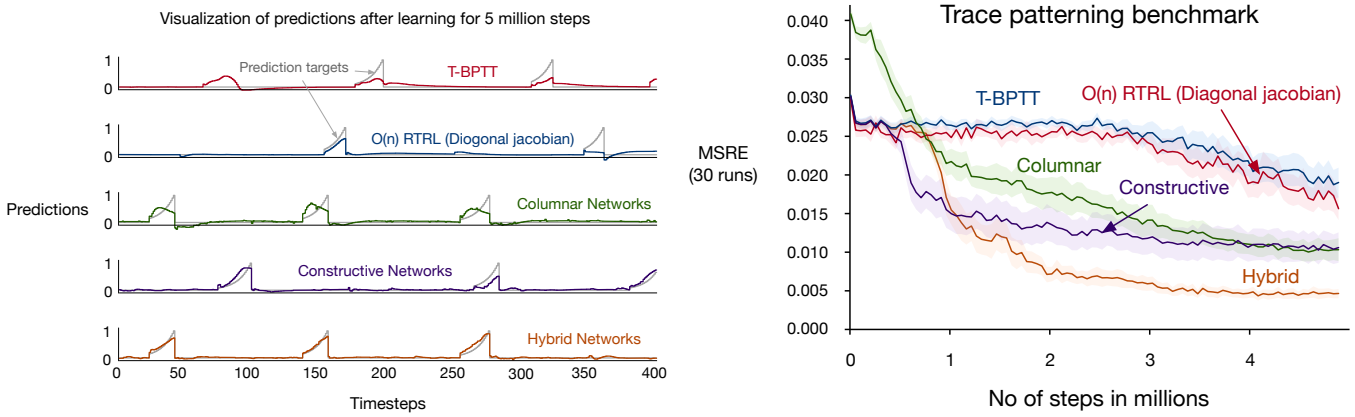

Figure 2: Left: Comparison of the predictions produced by the different methods for the last 400 timesteps of training. Right: Learning curves for the proposed and other online baseline methods on the trace patterning benchmark (lower is better). All the runs are averaged over 30 seeds, and the error regions are the 95% confidence intervals. T-BPTT uses a truncation length k=27—sufficient to capture the longest dependency in the data stream. Dense LSTM networks trained with both T-BPTT and Diagonal-RTRL perform poorly. Columnar and constructive networks perform well; hybrid networks, that combine columnar and constructive, perform the best, showing that both parallel learning, and hierarchical learning is important.

2.2 Constructive and Hybrid Networks

In constructive and hybrid networks, the recurrent network is a directed acyclic graph (DAG) learned incrementally. We initialize the vector $\mathbf{w}_1$ to be zero. Let d be a function from $\mathbb{Z} \to \mathbb{Z}$ that takes as input the index of a recurrent feature, and returns the length of the longest path from the input $\mathbf{x}$ to the recurrent feature in the DAG. Learning in a constructive or hybrid network happens in phases. First, we learn all recurrent features, and their corresponding w_i for which $d(a_j) = 1$. Once those features have been learned, we freeze the θ_j for these features. The corresponding w_j are not frozen and continue to be updated. We then learn features for which $d(a_j) = 2$. Note that these features may take as input the earlier frozen recurrent features. We repeat this process until all the features have been learned. Once again, all w_j continue to be updated continually.

The benefit of the phased approach is that at any given moment, the features that are being learned are independent of each other. As a result, RTRL stays scalable, and only requires $O(|\theta_t|)$ operations per step, similar to Columnar networks.

We call the special case when exactly one feature has $d(i) = j$, for all j the constructive network. The case when there can be multiple features with the same value of $d(.)$ are called hybrid networks, since they both learn incrementally, like constructive network, and learn multiple independent features simultaneously, like columnar networks.

3 Empirical evaluation

We use the trace patterning environment introduced by Rafiee *et al*. (2022) to benchmark our algorithm. It is an online prediction task, that requires the learner to discriminate between patterns—conditional stimuli (CS)—that are followed by the scalar—unconditional stimuli (US)—with a time delay. The goal of the agent is to predict the discounted sum of the US. Correct predictions require the ability to discriminate between patterns that lead to US from those that do not. The time delay between the CS and US requires remembering information from the past. We use an ITI of (80, 120), ISI of (14, 26), and 5 distractors. The US consists of 6 features, 3 of which are active to represent a pattern. Ten randomly chosen patterns lead to the CS. For an explanation for all the terms, please refer to Rafiee *et al*. (2022).

3.1 Baselines

We report the results of our approaches and compare them to T-BPTT, and the diagonal approximation to RTRL. Both T-BPTT and diagonal-RTRL use LSTM networks for learning. The sum of operations done for prediction and learning at each step is the same for all methods. Because T-BPTT is a more expensive algorithm than other methods, the LSTM for T-BPTT has fewer parameters to keep the computation per-step constant.

3.2 Results

We use the TD(λ) algorithm for learning, and report the results in figure 2.2. We tune hyper-parameters for all methods independently over a wide range and plot the results for the best performing hyper-parameters. All three of our approaches outperform truncated-BPTT and diagonal RTRL by a significant margin. The hybrid approach performs the best, showing that both learning hierarchical recurrent features and learning many recurrent features in parallel improve state construction.

4 Conclusion

We propose two algorithms—columnar RNNs and constructive RNNs—and show that they can learn to construct a state for a prediction task. The two algorithms can be combined to give a third algorithm—hybrid RNNs—that combine the strengths of both columnar and constructive. All three of our algorithms outperform LSTM networks trained via truncated-BPTT or the diagonal approximation to RTRL on the trace patterning benchmark under a fixed computation budget.

References

Elman, J. L. (1990). Finding structure in time. Cognitive science.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation.

Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., & Dabney, W. (2018, September). Recurrent experience replay in distributed reinforcement learning. In International conference on learning representations.

Menick, J., Elsen, E., Evci, U., Osindero, S., Simonyan, K., & Graves, A. (2020). A practical sparse approximation for real time recurrent learning. ICLR 2021.

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., & Khudanpur, S. (2010, September). Recurrent neural network based language model. In Interspeech.

Mikolov, T., Kopecky, J., Burget, L., & Glembek, O. (2009, April). Neural network based language models for highly inflective languages. In 2009 IEEE international conference on acoustics, speech and signal processing. IEEE.

Mujika, A., Meier, F., & Steger, A. (2018). Approximating real-time recurrent learning with random kronecker factors. Advances in Neural Information Processing Systems, 31.

Rafiee, B., Abbas, Z., Ghiassian, S., Kumaraswamy, R., Sutton, R., Ludvig, E., & White, A. (2022). From Eye-blinks to State Construction: Diagnostic Benchmarks for Online Representation Learning. Adaptive Behavior.

Robinson, A. J., & Fallside, F. (1987). The utility driven dynamic error propagation network. Cambridge: University of Cambridge Department of Engineering.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. nature.

Sutskever, I. (2013). Training recurrent neural networks (pp. 1-101). Toronto, ON, Canada: University of Toronto.

Tallec, C., & Ollivier, Y. (2017). Unbiased online recurrent optimization. ICLR 2018.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. Neural networks.

Williams, R. J., & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. Neural computation.

Williams, R. J., & Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. Neural computation.