A Fixed-Point Formulation for Recurrent Neural Networks

Somjit Nath, Taher Jafferjee and Martha White
Department of Computing Science
University of Alberta
Edmonton, Canada
{somjit, jafferje, whitem}@ualberta.ca

Abstract

Recurrent neural networks (RNNs), along with their many variants, provide a powerful tool for predicting sequential data with temporal dependencies. Two problems concerning RNNs, however, are the ability to capture long-term dependencies and, their long training times. There have been a variety of strategies to improve training in RNNs, particularly by approximating an algorithm called Real-Time Recurrent Learning. These strategies, however, can still be computationally expensive and focus computation on computing gradients back-in-time. In this work, we show that state in the RNN can be framed as a fixed-point problem. Using this formulation, we provide an asynchronous fixed-point iteration update that significantly improves run-times and stability of learning the hidden state.

1 Introduction

Neural Networks are powerful models that can give reasonably good results on a variety of machine learning tasks. However, they have certain limitations since they work under the assumption of independence in training and test samples. This is not the case for temporal data, in which predictions at time t may be reliant on data from previous time-steps. To address this shortcoming, RNNs [3, 4] were introduced to learn a hidden state update which summarizes past interaction. On each time-step, the previous state is inputted into the neural network, in addition to the input observation for the current time-step.

RNNs are typically trained either using Backpropagation-through-time (BPTT) [15] or approximations to an algorithm called Real-Time Recurrent Learning (RTRL) [16, 11]. The update for BPTT is a variant of standard backpropagation, computing gradients all the way back in time; this is problematic because the computational cost scales linearly with the number of time-steps. A much more common alternative is truncated BPTT (T-BPTT) [17]) which only computes the gradient back up to some maximum number of steps. Exact gradients can also be computed online by RTRL; however it requires high computational complexity and therefore is not used in practice.

Recently, there have been some efforts towards approximating gradients for back-propagation, both for feedforward NNs and RNNs. Synthetic gradients and BP(λ) [5] use an idea similar to returns from reinforcement learning: they approximate gradients by bootstrapping off estimated gradients in later layers [5, 2]. There are also several methods estimating RTRL [18]—which is itself an estimate of the true gradient back-in-time—including NoBackTrack [9] and Unbiased Online Recurrent Optimization (UORO) [13] which use an unbiased rank-1 approximation to the full matrix gradient. There are also several methods that solve a fixed point problem for Recursive Backpropagation, but for a restricted setting where convergence to a stationary state-vector is desired [1, 12, 8]. Finally, there are some methods that use selective memory back-in-time to compute gradients for the most pertinent samples,

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

using skip connections [6]. All of these methods, however, attempt to approximate the gradient back-in-time, for the current observation and state.

We reformulate state estimation for RNNs as a fixed-point problem, enabling different optimization strategies that do not need to compute gradients back in time. We develop an asynchronous updating mechanism with experience replay, that takes advantage of the fixed-point formulation. This avoids the need to compute full gradients, and avoids sweeping backwards from the current point. Instead, we can keep a longer buffer, while only updating randomly from this buffer with one-step updates. The one-step updates reduce the time complexity of this algorithm in comparison to RTRL and other competitors, and at the same time allow for capturing long-term dependencies, as the updates are from experience stored in the buffer. Further, it should avoid issues with exploding or vanishing gradients [10], as each gradient is only for one step. We demonstrate that the algorithm is effective on several problems with long-term dependencies, and improves over T-BPTT, particularly in terms of the number of gradients that need to be computed per-step.

2 Formulation as a fixed point problem

A simple RNN [3] consists of a hidden-state which is dependent on its value at the previous time-step multiplied by the recurrent weights V and the observation at the current time-step, multiplied by the input weights U. We denote $\mathbf{W}$ as concatenated $[V, U]$. Indeed, a defining feature of these networks is their utilization of memory in the form of recurrent hidden state.

In this section, we formulate RNN training as a fixed-point algorithm for the hidden state for RNNs. The key idea we present is to learn the state function $\mathbf{s} : \mathcal{H} \to \mathbb{R}^d$ where $\mathcal{H}$ is the set of unknown states (encoded as histories), formulated as a fixed point problem. For all $h \in \mathcal{H}$, we want to find the solution to the following fixed point problem

$$\begin{aligned} f(\mathbf{s}(h), \mathbf{o}(h')) &= \mathbf{s}(h') \quad \text{for all } h' \text{ such that } P(h, h') > 0 \\ g(\mathbf{s}(h)) &= y(h) \end{aligned} \tag{1}$$

where $P : \mathcal{H} \times \mathcal{H} \to [0, 1]$ is the transition dynamics; $y(h)$ is the expected target for a state h; and f, g are (learned) functions producing next state from current state and observations, $\mathbf{o}$ and targets from the state, respectively. We can also consider a slightly relaxed first condition in Eq. 1

$$\sum_{h' \in \mathcal{H}} P(h, h') f(\mathbf{s}(h), \mathbf{o}(h')) = \sum_{h' \in \mathcal{H}} P(h, h') \mathbf{s}(h').$$

For fixed f, g, this is a standard fixed point iteration problem to obtain $\mathbf{s}(h)$ for all $h \in \mathcal{H}$. But f and g are not fixed functions: f depend on the weights $\mathbf{W}$ of the RNN, while g depends on the output weights of the RNN, $\boldsymbol{\beta}$. More generally, when also learning f, g, this is an iterative fixed point, where both are being optimized jointly.

We first consider the setting where we can maintain a table of values: one state vector for each state $h \in \mathcal{H}$. When finding $\mathbf{s}$, we will assume fixed f, g; however, we do know we will be learning these incrementally as well and so consider parameterized functions. For RNNs, we will find such state vectors implicitly using $\mathbf{s}(h') = f_{\mathbf{W}}(\mathbf{s}(h), \mathbf{o}(h'))$, for some parametrized function $f_{\mathbf{W}}$ where we learn the weights $\mathbf{W}$. Then, each fixed point iteration step involves updating $\mathbf{W}$ towards the fixed point formula, $\mathbf{s}(h') = f_{\mathbf{W}}(\mathbf{s}(h), \mathbf{o}(h'))$. Given a state $\mathbf{s}(h')$ and $\mathbf{s}(h)$, we adjust the weights to make $f_{\mathbf{W}}(\mathbf{s}(h), \mathbf{o}(h'))$ closer to $\mathbf{s}(h')$ and $\mathbf{s}(h)$ such that $g_\beta(\mathbf{s}(h))$ is closer to $y(h)$. Simultaneously, we also consistently update g_β.

In practice, of course, we cannot practically store a state vector $\mathbf{s}(h)$ for each h. Instead, we will find approximations to such states, and store only a subset of their values by using a buffer which acts like experience replay. Imagine keeping the recent N items in a buffer. On each step, you add the transition $\mathbf{s}_{t-1}, \mathbf{o}_t, \mathbf{o}_{t+1}, \mathbf{s}_{t+1}, y_t$ to the buffer. At each training step, we can sample a transition (say time step, T, where $T > t - N$) from the buffer. The state $\mathbf{s}_T$ can be created using $f_{\mathbf{W}}(\mathbf{s}_{T-1}, \mathbf{o}_T)$, and the weights will be adjusted to make $\mathbf{s}_T$ more useful for predicting $\mathbf{s}_{T+1}$ and y_T. The $\mathbf{W}$ can be updated using a gradient descent step, with a fixed $\boldsymbol{\beta}$, to reduce Mean Squared Error (MSE) between $f_{\mathbf{W}}(f_{\mathbf{W}}(\mathbf{s}_{T-1}, \mathbf{o}_T), \mathbf{o}_{T+1})$, which we would call as **State Loss** and $\mathbf{s}_{T+1}$ and between $f_{\mathbf{W}}(\mathbf{s}_{T-1}, \mathbf{o}_T)\boldsymbol{\beta}$ and y_T, which can be a cross-entropy or MSE loss, which we will refer to as **Target Loss**. Then, $\boldsymbol{\beta}$ can be updated with gradient descent to also improve prediction accuracy. Once $\mathbf{s}_T$ has been updated by this procedure—by updating $\mathbf{W}$—then that value could be update in the transition before it, to give a more recent $\mathbf{s}_{T+1}$ for the previous transition.

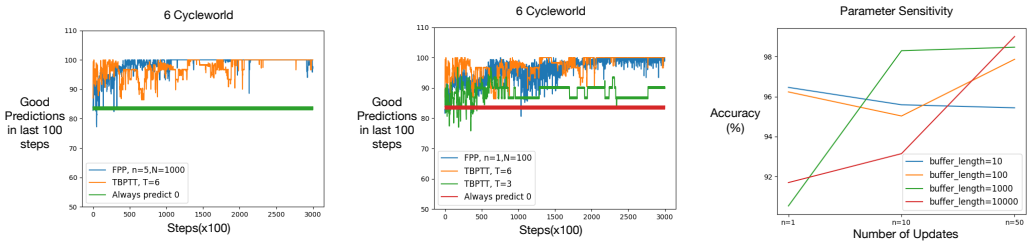
In this way, useful state values in later states can be propagated backwards, to make previous states adjust to predict those states. These updates can be done n times, by sampling n transitions from the buffer. To improve the convergence and stability of learning, we can also update the states of a few randomly chosen transitions stored in the buffer. The entire process is presented in Algorithm 1 in the Appendix.

The advantage of this strategy over truncated BPTT, is that we do not have to compute the entire gradient over T time-steps. Rather, instead of sweeping all the way back, we spread value by fixed point updates on random transitions in the buffer. This has three advantages. First, it avoids an expensive gradient computation for each state, allowing more states to be updated, including updates towards their targets. Second, this actually ensures that targets for older transitions are constantly being reinforced, and spends gradient computation resources towards this goal, rather than spending all computation on computing a more exact gradient for the recent time step. This distributes updates better across time, and should likely also result in a more stable state. Third, the interpretation as a fixed point iteration makes it a sound strategy—as opposed to truncation—where in the realizable setting, $\mathbf{W}$ and $\boldsymbol{\beta}$ should eventually converge to the optimal values to satisfy (1).

3 Experimental Setup and Results

We report the performance of our Fixed Point Propagation (FPP) algorithm on 2 different tasks. Our main goals are to (a) provide a first investigation into if FPP can learn solutions for RNNs and (b) the sensitivity of FPP to its parameters, particularly the buffer length and number of updates n. We include the same model trained with full-BPTT and truncated BPTT. Here, we additionally investigate if performing one-step updates asynchronously—as in FPP—can obtain similar or better performance than T-BPTT, with the same time complexity.

We test the algorithms in two domains with long-term dependencies: Cycleworld [14] and Sequential MNIST [7]. p-Cycleworld is a cycle of p time-steps, where the observation is 1 in time-step p and zero otherwise. The goal is to predict the observation bit. Sequential MNIST is pixel-by-pixel MNIST classification, where pixels are processed consecutively and the target is to predict the label of the image. A simple RNN is used for Cycleworld and an LSTM for Sequential MNIST. Further details of the experiment are provided in the Appendix.



(a) (b) (c)

Figure 1: (a) & (b) FPP vs BPTT on 6-Cycleworld, (c) Parameter Sensitivity plot for 6-Cycleworld

In Cycleworld, we explore both the effect of the number of updates on FPP and T-BPTT, as well as sensitivity of FPP to its buffer size, N, and number of updates, n. The main learning indicator of the model is whether it correctly predicts when the next observation should be 1; a classification accuracy of 83% is obtained by the naive predictor that always predicts 0. From Fig. 1 (a), it is evident that FPP learns as fast as 6-BPTT (which can be considered as a full-BPTT since there are no dependencies beyond 6 time-steps). Fig. 1 (b) also portrays that even though we do just one-step updates, the network still learns to predict the correct values more often than not. Both of these methods have similar time-complexity of $O(kd^2)$—though FPP is less computationally intensive by a constant factor—where d is n in the case of FPP and the truncation parameter in the case of BPTT. Once T drops below 6, T-BPTT suffers considerably, whereas FPP remains more robust, learning even with n = 1.

Another important parameter of this algorithm is the buffer size. As can be seen in Fig. 1 (c), there is no clear indication as to whether higher or lower values of buffer sizes are best. Small values of buffer size often lead to poor performance because of insufficient information stored in the state values of

elements of the buffer, whereas higher values of the buffer size contain transitions that are too old and hence decrease accuracy of learning. This suggests that a promising next step is to include more updates to the states $\mathbf{s}_{t+1}$ in the buffer, to update the states in the fixed point iteration more frequently than the parameters to f and g. Overall, however, the accuracies on y-axis of Fig. 1 (c) show that even for a wide range of buffer sizes and number of updates performed, FPP performs reasonably well.

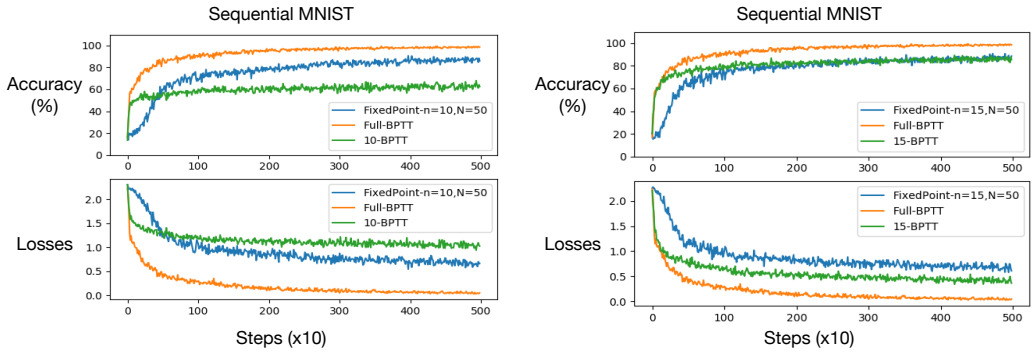


Figure 2: FPP vs BPTT on Sequential MNIST

Next, we examine the properties of FPP on a more complex RNN task: Sequential MNIST. Full BPTT is included as a baseline, to indicate near optimal performance on this task. We then consider how well FPP and T-BPTT can perform relative to this baseline. We include a results with $n = T = 10$ and $n = T = 15$, so that FPP and T-BPTT have comparable computation per step. Fig. 2 shows that FPP significantly outperforms T-BPTT for $n = 10$, and is better able to capture long term dependencies with fewer gradient computations. Instead, by focusing the computation on computing a 10-step gradient, 10-BPTT obtains notably worse—about 60% final accuracy as opposed to about 80%. Increasing n to 15 does not provide a significant gain, though learning is slightly faster. For T = 15, though, T-BPTT can match performance of FPP. This suggests that $n = 10$ is sufficient for FPP, and potentially to obtain further improvements a larger buffer or modifications to the updating strategy should be investigated.

4 Conclusion and Future Work

The main objective of this paper is to formulate RNN training as a fixed point problem for the hidden state, and investigate the properties of this optimization approach as an alternative for RNNs. In particular, the goal is to investigate methods that can better distribute computation, and improve state updating without having to compute expensive—and potentially unstable—gradients back-in-time for each state. We found that our algorithm, called FPP, was indeed more robust to the number of updates, than BPTT was to its truncation level. These initial experiments suggest that FPP could be a promising direction, and that it warrants further investigation.

One important step is to analyze if this fixed point iteration will converge. This requires that the iterates for Eq. 1 should be bounded, that is the function f must be a contraction, as per Banach's Fixed Point Theorem. The second term, $g(\mathbf{s}(h)) = y(h)$, should enable us to demonstrate contraction properties for certain g, because it is the end of the recursion. However, the functions f, g are general and such an analysis will require some investiation.

Additionally, we would like to improve on the algorithm choices in FPP, which were intentionally initially simple for this first investigation. One important addition is to better consider what is stored in the buffer, to reduce memory complexity and ensure important transitions are remembered. This could include prioritizing particularly important transitions. Additionally, we plan to investigate improvements on updating f and g, and the states in the buffer. As suggested in the text, it is possible that the states should be updated more frequently—at a faster timescale—enabling them to become more stable before the functions defining the fixed point iteration are changed. Overall, there are many avenues to better understand this different approach to optimizing RNNs.

References

[1] L B Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. *Proceedings, 1st First International Conference on Neural Networks*, 1987.

[2] Wojciech Marian Czarnecki, Max Jaderberg, Simon Osindero, Oriol Vinyals, and Koray Kavukcuoglu. Understanding Synthetic Gradients and Decoupled Neural Interfaces. *arXiv:1411.4000v2 [cs.LG]*, 2017.

[3] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. ISSN 1551-6709. doi: 10.1207/s15516709cog1402_1. URL `http://dx.doi.org/10.1207/s15516709cog1402_1`.

[4] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982. ISSN 0027-8424. URL `http://www.pnas.org/content/79/8/2554`.

[5] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled Neural Interfaces using Synthetic Gradients. In *International Conference on Machine Learning*, 2017.

[6] Nan Rosemary Ke, Anirudh Goyal, Olexa Bilaniuk, Jonathan Binas, Laurent Charlin, Chris Pal, and Yoshua Bengio. Sparse Attentive Backtracking: Long-Range Credit Assignment in Recurrent Networks. *arXiv:1509.01240v2*, 2017.

[7] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL `http://yann.lecun.com/exdb/mnist/`.

[8] Renjie Liao, Yuwen Xiong, Ethan Fetaya, Lisa Zhang, KiJung Yoon, Xaq Pitkow, Raquel Urtasun, and Richard S Zemel. Reviving and Improving Recurrent Back-Propagation. In *International Conference on Machine Learning*, 2018.

[9] Yann Ollivier and Guillaume Charpiat. Training recurrent networks online without backtracking. *arXiv*, 2015.

[10] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1310–III–1318. JMLR.org, 2013. URL `http://dl.acm.org/citation.cfm?id=3042817.3043083`.

[11] B. A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: a survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228, Sept 1995. ISSN 1045-9227. doi: 10.1109/72.410363.

[12] Fernando J Pineda. Generalization of back-propagation to recurrent neural networks. *Physical review letters*, 1987.

[13] Corentin Tallec and Yann Ollivier. Unbiased Online Recurrent Optimization. *arXiv:1411.4000v2 [cs.LG]*, 2017.

[14] Brian Tanner and Richard S. Sutton. Td(lambda) networks: Temporal-difference networks with eligibility traces. 2005.

[15] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990. ISSN 0018-9219. doi: 10.1109/5.58337.

[16] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, June 1989. ISSN 0899-7667. doi: 10.1162/neco.1989.1.2.270.

[17] Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2:490–501, 1990.

[18] Ronald J Williams and David Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1989.

APPENDIX

A Algorithm

The entire algorithm for the Fixed Point Iteration is presented below.

Algorithm 1 Fixed-Point Iteration(one iteration)

procedure FORWARD-PROPAGARTION($\mathbf{W}, \boldsymbol{\beta}, \mathbf{s}_{t-1}, \mathbf{o}_t, \mathbf{o}_{t+1}, y_t$)
  State $\mathbf{s}_t = f_{\mathbf{W}}(\mathbf{s}_{t-1}, \mathbf{o}_t)$
  Output $y'_t = g_\beta(\mathbf{s}_t)$
  Next-State $\mathbf{s}_{t+1} = f_{\mathbf{W}}(\mathbf{s}_t, \mathbf{o}_{t+1})$
  Add $\mathbf{s}_{t-1}, \mathbf{o}_t, \mathbf{o}_{t+1}, \mathbf{s}_{t+1}, y_t$ to $buffer$.
end procedure
procedure TRAIN($\mathbf{W}, \boldsymbol{\beta}$)
  Sample $\mathbf{s}_{T-1}, \mathbf{o}_T, \mathbf{o}_{T+1}, \mathbf{s}_{T+1}, y_T \sim buffer$
  $\mathbf{s}_T = f_{\mathbf{W}}(\mathbf{s}_{T-1}, \mathbf{o}_T)$
  $y'_T = g_\beta(\mathbf{s}_T)$
  $\mathbf{s}'_{T+1} = f_{\mathbf{W}}(\mathbf{s}_T, \mathbf{o}_{T+1})$
  State Loss $L_s = MSE(\mathbf{s}'_{T+1}, \mathbf{s}_{T+1})$
  Target Loss $L_t = CE(y'_T, y_T)$
  Total Loss $L = L_s + L_t$
  Calculate $\frac{\partial L}{\partial \mathbf{W}}$ & $\frac{\partial L_t}{\partial \boldsymbol{\beta}}$
  Update $\mathbf{W}$
  Update $\boldsymbol{\beta}$
  $\mathbf{s}_T = f_{\mathbf{W}}(\mathbf{s}_{T-1}, \mathbf{o}_T)$
  $\mathbf{s}'_{T+1} = f_{\mathbf{W}}(\mathbf{s}_T, \mathbf{o}_{T+1})$
  Replace $\mathbf{s}_{T+1}$ by $\mathbf{s}'_{T+1}$ in $buffer$
end procedure

B Losses

Normal RNN training minimizes the cross-entropy loss with respect to the targets. In our case, we take each transition from the buffer, where we want the states and the weights to satisfy Eq. 1. For this we train using a combined **state** and **target** loss as explained earlier. Instead, we can train the parameters just using the **target** loss, as done in normal RNNs. However, including the state loss has the advantage of making the states better at each update and helps to make the learning stable.

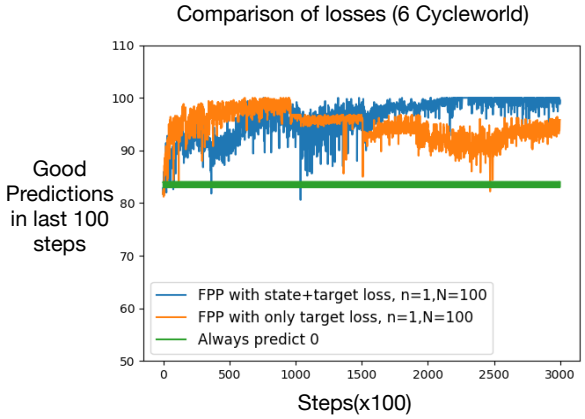


Figure 3: Comparison of losses on 6-Cycleworld

C Experimental Details

The experimental details of each dataset are provided below.

C.1 CycleWorld

Network Type = simple RNN
Hidden Units = 4
Total time-steps = 300000
Optimizer = Adagrad
Learning rate = 0.1
Number of runs = 5

C.2 Sequential MNIST

Network Type = LSTM
Hidden Units = 128
Image size = 784 pixels
Input dimension = 20 pixels
Total time-steps = 5000
Optimizer = Adam
Learning rate = 0.001
Number of runs = 5