Fixed Point Propagation: A New Way To Train Recurrent Neural Networks Using Auxiliary Variables

by

Somjit Nath

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Recurrent neural networks (RNNs), along with their many variants, provide a powerful tool for online prediction in partially observable problems. Two issues concerning RNNs, however, are the ability to capture long-term dependencies and 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 learning the hidden state in RNNs 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 state update.

Preface

This thesis is an original work by Somjit Nath. No part of this thesis has been previously published. It will be submitted to International Conference on Learning Representations, (ICLR), 2020.

To my parents

The measure of intelligence is the ability to change.

– Albert Einstein.

Acknowledgements

First of all, I would like to thank my supervisor Prof. Martha White for her incredible support and guidance. Her feedback on the project was constant and those inputs always kept me motivated on this project. Secondly, I would like to thank my examining committee for taking time to read my work and help me out with valuable inputs and constructive criticism. Lastly, I would like to thank all my lab-mates from RLAI for helping me out with small problems and fostering a great environment for research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There has been great advancements in the field of Machine Learning for solving computational problems. Given a batch of data in advance i.e. off-line, several machine learning algorithms have been developed to make predictions from them. We also have algorithms that tackle online learning i.e. learning from streaming data. Here the data is given at each and every step and we have to make predictions continually from the data. This is called online learning a problem we choose to tackle. However, most online prediction problems are partially observable: the most recent observation is insufficient to make accurate predictions. Augmenting inputs with history can significantly improve accuracy, but can require a long history when there are long-term dependencies back-in-time. So we need a better method to solve such partially observable online prediction problems.

Neural Networks are computational models developed by drawing inspiration from how neurons process data in human brain. They are composed of layers of connected nodes called artificial neurons. This interconnected layers of nodes comprises of the Artificial Neural Network model which can perform various tasks like data representation, feature learning, etc. One key aspect of normal Neural Networks is that the inputs at every time-step must be independent of each other, i.e. it cannot process sequential inputs and hence is not useful. So they cannot be used for our problem, however, Recurrent Neural Networks (RNNs) [10, 17] learn a state which *summarizes* a history of data. Specifically, RNNs contain recurrent connections to their hidden layers which allow for past information to propagate through time. This state need not correspond to a true underlying state; rather, it is a subjective, constructed state to facilitate prediction. This enables RNNs to remember and process previous information and hence can be easily used for sequential data prediction. RNNs have been widely used in a variety of applications. in speech recognition [6, 14, 15, 27], image captioning [23, 24, 41], speech synthesis [26] and reinforcement learning [9, 16].

In this work, we formulate a new algorithm for training Recurrent Neural Networks which improves upon the computational time as well as captures long-term dependencies better. The major list of contributions are:

- Formulating training RNNs as a fixed-point problem.
- Developing Fixed Point Propagation (FPP) Algorithm for training RNNs.
- Comparing FPP with the most popular RNN training algorithm (BPTT).
- Discussing the advantages and disadvantages of FPP with supporting experiments.

The thesis is divided into 5 chapters. Chapter 2 includes the necessary background on RNN and a survey of the recent advances made in RNN training. Chapter 3 formalizes the problem setting and describes the new algorithm in detail. The algorithm is evaluated in Chapter 4 by testing it on several environments. Chapter 5 has concluding remarks and directions for future work.

Chapter 2

Background

2.1 Problem Setting

We consider a partially observable online setting, where an immediate observation is not sufficient for prediction. More formally, assume there is a sequence of n observations, $\mathbf{o}_1, \ldots, \mathbf{o}_n$, which provide only partial information about an unknown underlying sequence of states. After obtaining an observation $\mathbf{o}_i$, the agent makes a prediction $\hat{\mathbf{y}}_i$ and sees the actual outcome $\mathbf{y}_i$. The goal is to minimize this prediction error. Given only $\mathbf{o}_i$, however, the agent is unlikely to make accurate predictions about $\mathbf{y}_i$, because $\mathbf{o}_i$ is not a sufficient statistic to predict $\mathbf{y}_i$: $p(\mathbf{y}|\mathbf{o}_i, \mathbf{o}_{i-1}, \mathbf{o}_{i-2}, \ldots) \neq p(\mathbf{y}|\mathbf{o}_i)$. The agent could have obtained lower prediction error by using a history of observations. The length of such a history, however, may need to be prohibitively long, even when this history could have been summarized compactly.

An alternative is to construct state using a Recurrent Neural Network (RNN), by learning a state-update function. Given a current (constructed) state $\mathbf{s}_{t-1} \in \mathbb{R}^k$, and a new observation $\mathbf{o}_t \in \mathbb{R}^d$, the parameterized state-update function $f_{\mathbf{W}} : \mathbb{R}^k \times \mathbb{R}^d \to \mathbb{R}^k$, with parameters $\mathbf{W}$, produces the next (constructed) state $\mathbf{s}_t = f_{\mathbf{W}}(\mathbf{s}_{t-1}, \mathbf{o}_t)$. For example, $f_{\mathbf{W}}$ could be a linear weighting of $\mathbf{s}_{t-1}$ and $\mathbf{o}_t$, with a ReLu activation:

$$f_{\mathbf{W}}(\mathbf{s}_{t-1}, \mathbf{o}_t) = \max([\mathbf{s}_{t-1}, \mathbf{o}_t]\mathbf{W}, \mathbf{0})$$

There are also more complex state-update functions, like the gating updates in Long Short Term Memory (LSTM) [16].

The prediction error is the objective for learning these parameters $\mathbf{W}$ for the state-update. For the current state $\mathbf{s}_t$, a prediction is made by parameterized function $g_{\boldsymbol{\beta}} : \mathbb{R}^k \rightarrow \mathbb{R}^m$ for learned parameters $\boldsymbol{\beta}$. For example, the prediction could be a linear weighting of the state, $g_{\boldsymbol{\beta}}(\mathbf{s}) = \boldsymbol{\beta}^\top \mathbf{s}$. We denote the prediction error as $\ell_{\boldsymbol{\beta}} : \mathbb{R}^k \times \mathbb{R}^m \rightarrow \mathbb{R}$ for a given $\boldsymbol{\beta}$. For example, this loss could be

$$\ell_{\boldsymbol{\beta}}(\mathbf{s}_t; \mathbf{y}_t) = \|g_{\boldsymbol{\beta}}(\mathbf{s}_t) - \mathbf{y}_t\|_2^2.$$

The goal in RNNs is to minimize, for some start state $\mathbf{s}_0$,

$$\min_{\boldsymbol{\beta},\mathbf{W}} \sum_{i=1}^{n} \ell_{\boldsymbol{\beta}}(f_{\mathbf{W}}(...f_{\mathbf{W}}(\underbrace{f_{\mathbf{W}}(\mathbf{s}_0, \mathbf{o}_1)}_{\mathbf{s}_1}, \mathbf{o}_2), ..., \mathbf{o}_i); \mathbf{y}_i). \tag{2.1}$$

2.2 Vanilla Recurrent Neural Network

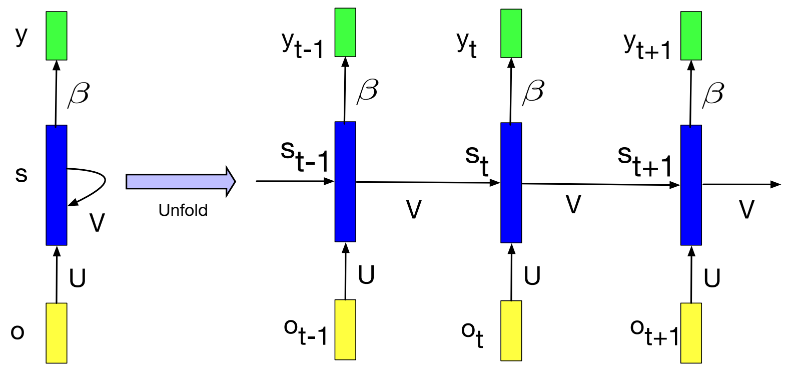


Figure 2.1: Unfolding of a Vanilla RNN

Vanilla RNNs are the simplest form of RNNs, with one hidden layer. One defining feature of recurrent neural networks is their utilization of memory, also known as the hidden state. Each hidden state is of the form:

$$\mathbf{s}_t = a_1(b + \mathbf{s}_{t-1}\mathbf{V} + \mathbf{o}_t\mathbf{U}) = f_{\mathbf{W}}(\mathbf{s}_{t-1}, \mathbf{o}_t) \tag{2.2}$$

where $\mathbf{o}_t$ is the input at time t, $\mathbf{s}_t$ is the value of the hidden state at time t, b is the bias, a_1 is the activation function, $\mathbf{V}$ are the weights of the recurrent edges and $\mathbf{U}$ are the weights of the input connections and $\mathbf{W} = [\mathbf{U}, \mathbf{V}]$ The dynamics of the network can also be visualized by unfolding it, in which the state of the network at time t can be treated as the t^{th} hidden layer. The network then becomes like a deep network, where back-propagation can be applied. This method is called Backpropagation through time (BPTT) [42]. When training VRNNs, given some state $\mathbf{s}_t$, we compute output $\mathbf{y}_t$ as

$$\mathbf{y}_t = a_2(\boldsymbol{\beta}\mathbf{s}_t) = g_{\boldsymbol{\beta}}(\mathbf{s}_t) \tag{2.3}$$

where, a_2 is the output activation function. Furthermore, given some prediction $\hat{\mathbf{y}}_t$, we compute the prediction error ℓ_t at time t as:

$$\ell_t = Error(\mathbf{y}_t, \hat{\mathbf{y}}_t)$$

The partial derivatives

$$\frac{\partial \ell_t}{\partial U}, \frac{\partial \ell_t}{\partial V}, \frac{\partial \ell_t}{\partial \beta}$$

are then computed for the RNNs three parameters, $\mathbf{U}, \mathbf{V}, \boldsymbol{\beta}$. These computations, though, involve unrolling the RNN and computing these gradients for time $t-1, ..., 0$. This results in very slow training because the computational cost scales linearly with the number of time-steps. A much more common alternative is truncated BPTT (T-BPTT) [44]) which only computes the gradient up to some maximum number of steps. So, for T-step truncation, the gradients are only computed for time $t-1, ..., t-T$. Depending on the dataset being processed, this can substantially reduce training time. This approximation, though, is not robust to long-term dependencies [38]. Intuitively, it can only capture dependencies as far back as the truncation parameter being used. This can significantly reduce the utility of RNNs, as their usefulness lies in the capacity to capture temporal dependencies in sequential data.

Another alternative to T-BPTT is an algorithm called Real-Time Recurrent Learning (RTRL) [32, 43, 45] which computes the exact error gradients by taking advantage of the recurrent structure of the network. This online algorithm, however, has high computational complexity per step and therefore is not used in practice.

2.3 Issues with training RNNs

There are known stability and computations issues with training RNNs online [31, 38].

Computational Issue: Computing gradients for the objective in Eq. 2.1, however, can be prohibitively expensive. A large literature on optimizing RNNs focuses on approximating this gradient, either through approximations to RTRL or using improvements to BPTT. RTRL [45] uses a recursive gradient form, which can take advantage of gradients computed up until the last observation to compute the gradient for the next observation. This estimate is only exact in the offline case. Further, in either online or offline, RTRL costs $O(k^4)$ computation per observation. In BPTT, gradients are computed back-in-time, by unrolling the network. In the online setting, it is infeasible to compute gradients all the way back to the beginning of time. Instead, this procedure is truncated to T steps back-in-time. T-BPTT is suitable for the online setting, and costs $O(Tk^2)$ at each time step, i.e., for each observation.

Arguably the most widely-used strategy is T-BPTT, because of its simplicity. Unfortunately, though, T-BPTT has been shown to fail in settings where dependencies back-in-time are further than T [38], as we affirm in our experiments. Yet, the need for simple algorithms remains.

Exploding and Vanishing Gradient problem: Another problem while training RNNs is the exploding and vanishing gradient problem. This arises due to the recurrent structure of the network. The gradients are being propagated back through time can explode meaning the long term components of the gradients can increase exponentially or they can vanish, meaning they can decay exponentially to 0. Exploding gradient occurs when the spectral radius of the recurrent weight matrix is greater than 1 and if it is less than 1, we will have the vanishing gradient problem. This especially prevents RNNs form learning long-term dependencies. To combat this problem, we can take two approaches- (a) modifying the architecture of the RNN (like LSTMs [16]) or

(b) change the algorithm of training (like using gradient clipping or regularization [31]).

2.4 Approaches for Solving RNNs

Table 2.1: RNN Algorithms over the years

Year	**Author**	**Algorithm**
1990	Elman [10]	RNNs introduced
1990	Williams [45]	Real Time Recurrent Learning (RTRL)
1994	Bengio [3]	Exploding Gradient problem
1997	Schuster [37]	Bidrectional RNN
1997	Hochreiter [16]	Long Short Term Memory (LSTM)
1999	Gers [11]	Introduction of forget gates into LSTM structure
2005	Graves [13]	Bidirectional LSTM improves performance further
2013	Pascanu [31]	Avoiding exploding gradients by gradient clipping
2014	Koutnik [20]	Clockwork RNN architecture
2015	Olliver [30]	NoBackTrack Algorithm
2016	Neil [29]	Phased LSTM
2017	Tallec [38]	Unbiased Online Recurrent Optimization
2017	Jaderberg [18]	Decoupled Neural Interfaces using Synthetic Gradients
2017	Campos [4]	Skip RNN
2017	Chang [7]	Dilated RNN
2017	Ke [19]	Sparse Attentive Backtracking
2018	Mujika [28]	Approximating RTRL with Random Kronecker Factors
2018	Liao [22]	Reviving Recurrent BackPropagation
2019	Roth [34]	Kernel RNN Learning

2.4.1 Algorithmic Advances

Recently, there have been some efforts towards approximating gradients for back-propagation, both for feed-forward NNs and RNNs. Synthetic gradients and BP(λ) [18] use an idea similar to returns from reinforcement learning: they approximate gradients by bootstrapping off estimated gradients in later layers [8, 18]. BP(λ) uses an idea similar to returns, but uses a recursive form for the gradients rather than the state. The problem is not solved as a fixed point problem, though they do bootstrap-off gradient estimates. They do not solve for gradients, conditioned on inputs, and rather use one sample estimate of the "return" for their targets. Solving the problem as a fixed point problem, where the value function is the gradient conditioned on input, would be prohibitively expensive.

There are several methods estimating RTRL—which is itself an estimate of the true gradient back-in-time—including NoBackTrack [30] which gives an unbiased gradient estimate by using a rank-one approximation of the full matrix gradient. Building up on that Unbiased Online Recurrent Optimization (UORO) [38] employs same approximation but is much easier to use for complex architectures. RTRL gradients can also be approximated using Kronecker factors [28] which reduces the variance in gradient estimation and thus improves learning. Kernel RNN Learning (KERNL) [34] also approximates the gradient by using a rank-reduced product of a sensitivity weight and an eligibility trace which can be learned. Finally, there are some methods that use selective memory back-in-time to compute gradients for the most pertinent samples, using skip connections [19]. All of these methods, however, attempt to approximate the gradient back-in-time, for the current observation and state, and so suffers to some extent from the same issues as BPTT and RTRL.

Liao et. al. [22] enhances upon recurrent back-propagation (RBP) [2, 33] which is an old algorithm that has several strict assumptions on the state and hence is not generally applied. This paper [22] provides introduces some new RBP variants like Neumann RBP (using Neumann series) and Conjugate RBP (using conjugate gradients).

2.4.2 Architectural Advances

Note that in addition to a variety of optimization strategies, different architectures have also been proposed to facilitate learning long-term dependencies with RNNs. The most commonly used are LSTMs [16], which use gates to remember and forget parts of the state. Another strategy was to use bi-directional networks both for RNNs [37] and LSTMs [13], where we calculate the gradients both forward and backward in time. Clockwork RNNs [20] introduce a hidden state that is partitioned into separate modules each having its own clock-period when they are processed. Phased LSTMs [29] add a separate time-gate to the LSTM structure controlled by a parameterized oscillation which impose sparse updating thus requiring much lesser compute time and accelerating training. Skip RNNs [4] use skip connections which skips the state updates and thus enables faster convergence and lesser computation. Dilated RNNs [7] uses dilated RNN skip connections which not only improve the training efficiency but also help in solving gradient problems and thus enables the model to capture long term dependencies better. In this work, we focus on a general purpose RNN algorithm, that could be combined with each of these architectures for further improvements.

Chapter 3

Fixed Point Formulation

In this section, we investigate an alternative optimization strategy that does not attempt to approximate the gradient back-in-time, by reformulating state estimation for RNNs as a fixed-point problem.

We develop an asynchronous updating mechanism on a sliding window buffer, that takes advantage of the fixed-point formulation. The algorithm explicitly optimizes state vectors, as auxiliary variables, with many efficient one-step—or short term multi-step updates—across the buffer. Instead of focusing computation to get a more accurate gradient estimates for this time-step, our algorithm, called Fixed Point Propagation (FPP), can more effectively use computation to update several states. Further, it is a sound strategy to use short, truncated gradient updates, because FPP is a standard gradient descent optimization with auxiliary variables. We demonstrate that the algorithm is effective on several problems with long-term dependencies, and improves over T-BPTT, particularly in terms of stability and computation.

Here, we formulate RNN training as a fixed-point problem. The key idea is to explicitly learn the state, as an auxiliary variable. We first formalize the problem in the ideal case, with access to the unknown underlying states. We then approximate this formulation for an online setting.

3.1 The Fixed-Point Objective

Consider an idealized setting, where the goal is to learn a state function $\mathbf{s} : \mathcal{H} \to \mathbb{R}^k$ for a discrete set of true states $\mathcal{H}$. For all $h \in \mathcal{H}$, we want to find the solution to the following fixed point problem

$$f_{\mathbf{W}}(\mathbf{s}(h), \mathbf{o}(h')) = \mathbf{s}(h') \quad \forall\ h' \text{ s.t. } \mathbf{P}(h, h') > 0 \tag{3.1}$$

where $\mathbf{P} : \mathcal{H} \times \mathcal{H} \to [0, 1]$ is the transition dynamics; $\mathbf{o} : \mathcal{H} \to \mathbb{R}^k$ is the vector-valued observation function; and $f_{\mathbf{W}}$ is a parameterized function producing next state from the current state and observations. In addition to satisfying this fixed point problem, the learned state should also enable accurate prediction about the given targets: minimize $\ell_{\boldsymbol{\beta}}(\mathbf{s}(h); \mathbf{y}(h))$ for all h, where $\mathbf{y}(h)$ is the expected target for a true state h. This results in the following optimization, with the fixed point problem as a constraint

$$\begin{aligned} &\min_{\boldsymbol{\beta}, \mathbf{W}, \mathbf{s}} \sum_{h, h' \in \mathcal{H}} \mathbf{P}(h, h') \ell_{\boldsymbol{\beta}}(f_{\mathbf{W}}(\mathbf{s}(h); \mathbf{o}(h')), \mathbf{y}(h')) \\ &\text{s.t. } f_{\mathbf{W}}(\mathbf{s}(h), \mathbf{o}(h')) = \mathbf{s}(h') \quad \forall\ h' \text{ s.t. } \mathbf{P}(h, h') > 0 \end{aligned} \tag{3.2}$$

The satisfiability of this will depend on $f_{\mathbf{W}}$ and if $\mathbf{s}(h)$ and $\mathbf{o}(h')$ can uniquely determine $\mathbf{s}(h')$.

In practice, we cannot explicitly learn state as a function of true state. We can, however, approximate this objective on a batch of observed data. Assume n observations have been observed, $\mathbf{o}_1, \ldots, \mathbf{o}_n$, with corresponding targets $\mathbf{y}_1, \ldots, \mathbf{y}_n$. Let the state variables be stacked in a matrix $\mathbf{S} \in \mathbb{R}^{k \times n}$ and observations as a matrix $\mathbf{O} \in \mathbb{R}^{d \times n}$, with $\mathbf{S} = [\mathbf{s}_0, \ldots, \mathbf{s}_n]$ and $\mathbf{O} = [\mathbf{o}_1, \ldots, \mathbf{o}_n]$. The fixed point problem becomes $\mathbf{S} = F_{\mathbf{W}}(\mathbf{S}, \mathbf{O})$ for operator

$$F_{\mathbf{W}}(\mathbf{S}, \mathbf{O}) \overset{\text{def}}{=} [\mathbf{S}_{:,0}, f_{\mathbf{W}}(\mathbf{S}_{:,0}, \mathbf{O}_{:,1}), \ldots, f_{\mathbf{W}}(\mathbf{S}_{:,n-1}, \mathbf{O}_{:,n})]\,. \tag{3.3}$$

The resulting optimization, for this batch, is

$$\min_{\boldsymbol{\beta}, \mathbf{W}, \mathbf{S}} \sum_{i=1}^{n} \ell_{\boldsymbol{\beta}}(f_{\mathbf{W}}(\mathbf{s}_{i-1}, \mathbf{o}_i); \mathbf{y}_i) \qquad \text{s.t. } \mathbf{S} = F_{\mathbf{W}}(\mathbf{S}, \mathbf{O}). \tag{3.4}$$

The solution to this new optimization corresponds to the solution for the original RNN problem in (2.1)—when also optimizing over $\mathbf{s}_0$ in (2.1)—because the fixed-point constraint forces variables $\mathbf{s}_i$ to be representable by $f_{\mathbf{W}}$. Therefore, the reformulation as a fixed point problem has not changed the solution;

rather, it has only made explicit that the goal is to learn these states and facilitates the use of alternative optimization strategies.

Reformulations like the one in (3.4) have been widely considered in optimization, because (3.4) is actually an auxiliary variable reformulation of (2.1). In this case, the auxiliary variables are the states $\mathbf{S}$. Using auxiliary variables is a standard strategy in optimization—under the general term *method of multipliers*—to decouple terms in an optimization and so facilitate decentralized optimization.

Such auxiliary variable methods have even been previously considered for optimizing neural networks. Carreira-Perpiñán & Wang introduced the Method of Auxiliary Coordinates (MAC), which explicitly optimize hidden vectors in the neural network. Taylor *et al.* proposed a similar strategy, but introduced an additional set of auxiliary variables to obtain further decoupling and a particularly efficient algorithm for the batch setting. Scellier & Bengio introduced Equilibrium Propagation for symmetric neural networks, where the state of the network is explicitly optimized to obtain a stationary point in terms of the energy function. Gotmare *et al.* built on these previous ideas to obtain a stochastic gradient descent update for distributed updates to blocks of weights in a neural network. Our proposed optimization can be seen as an instance of the objective considered for MAC [5, Equation 1], though we arrived at it from a different perspective: with the goal to obtain a fixed point formulation in terms of state.

The solution strategies proposed for neural networks with auxiliary variables, however, do not apply to the RNN setting, because all the auxiliary variables are coupled to the beginning of time. The solution strategy in MAC relies on storing all hidden layers—the auxiliary variables—for a sample. For RNNs, this would correspond to storing all states and observations to the beginning of time, which is prohibitive in an online setting. Further, in MAC, each update is for one i.i.d. sample at a time. It is possible that the large literature on such optimizations will be fruitful for obtaining practical algorithms in the future, but to the best of our knowledge, no easy-to-use algorithm yet exists. In the next section, we propose a new stochastic algorithm, called Fixed

Point Propagation, that optimizes this objective online.

Note: Recurrent Backpropagation and related variants [2, 22, 33, 36] also use fixed points for their optimization, but in a different way. These algorithms only address a restricted class of RNNs, that assume a fixed input and converge to a single low-energy state—a fixed point of the dynamics for that given input. These RNNs are actually highly related to Graph NNs [35], because the temporal nature only arises from cyclic connections, rather than from temporal data. Their problem setting is fundamentally different from our online prediction setting, and is usually used for associate memory with Hopfield networks or semi-supervised problems. Recurrent Backpropagation cannot be used for our setting and so we do not further discuss this class of RNN algorithms.

3.2 Fixed Point Propagation

We need an online algorithm to optimize (3.4) and thus (a) we cannot store all the data and (b) each update should be efficient—using the minimal computation of $O(k^2)$. To obtain such an algorithm, we propose two simple innovations that are nonetheless effective for satisfying these criteria. First, we maintain a windowed buffer of states and observations. Though this buffer could still be prone to missing long-term dependencies, it can be significantly longer than the truncation level in BPTT, since the computation per step of our algorithm is independent of the buffer size. We use random sampling to simulate asynchronous fixed point updates for (3.4).

Second, we stochastically sample the objective in such a way as to avoid extra computation, but still satisfy the requirements of stochastic gradient descent. As in MAC-QP [5], we reformulate this constrained objective into an unconstrained objective with a quadratic penalty, with $\lambda > 0$

$$L(\boldsymbol{\beta}, \mathbf{W}, \mathbf{S}) \overset{\text{def}}{=} \sum_{i=1}^{n} \ell_{\boldsymbol{\beta}}(f_{\mathbf{W}}(\mathbf{s}_{i-1}, \mathbf{o}_i); \mathbf{y}_i) + \frac{\lambda}{2} \sum_{i=1}^{n} \|\mathbf{s}_i - f_{\mathbf{W}}(\mathbf{s}_{i-1}, \mathbf{o}_i)\|_2^2 \qquad (3.5)$$

Once in this unconstrained form, we can perform stochastic gradient descent on this objective in terms of $\mathbf{W}$ and $\mathbf{S}$, for an iteratively increasing λ. To use stochastic gradient descent, the objective needs to break up into a sum of losses, $L(\boldsymbol{\beta}, \mathbf{W}, \mathbf{S}) = \sum_{i=1}^{n} L_i(\boldsymbol{\beta}, \mathbf{W}, \mathbf{S})$, where we define

$$L_i(\boldsymbol{\beta}, \mathbf{W}, \mathbf{S}) \overset{\text{def}}{=} \ell_{\boldsymbol{\beta}}(f_{\mathbf{W}}(\mathbf{s}_{i-1}, \mathbf{o}_i); \mathbf{y}_i) + \frac{\lambda}{2}\|\mathbf{s}_i - f_{\mathbf{W}}(\mathbf{s}_{i-1}, \mathbf{o}_i)\|_2^2.$$

For our buffer, we can stochastically sample i and update our variables with ∇L_i. Fortunately, because the auxiliary variables break connections across time, this gradient is zero for most variables, except $\boldsymbol{\beta}, \mathbf{W}, \mathbf{s}_{i-1}$ and $\mathbf{s}_i$. Therefore, each stochastic update can be computed efficiently. Note that in practice, we simply fix $\lambda = 1$ without iteratively increasing it, as this results in good performance.

Finally, we can generalize this procedure to incorporate more than one step of propagation back-in-time, simply by generalizing the fixed-point operator. Consider the more general T-step fixed point problem $\mathbf{S} = F_{T,\mathbf{W}}(\mathbf{S}, \mathbf{O})$ where

$$\begin{aligned} &F_{T,\mathbf{W}}(\mathbf{S}, \mathbf{O}) \overset{\text{def}}{=} \\ &\Big[\mathbf{S}_{:,0}, \mathbf{S}_{:,1}, \ldots, \mathbf{S}_{:,T-1}, \underbrace{f_{\mathbf{W}}(\ldots f_{\mathbf{W}}(f_{\mathbf{W}}(\mathbf{S}_{:,0}, \mathbf{O}_{:,1}), \mathbf{O}_{:,2}), \ldots), \mathbf{O}_{:,T})}_{\hat{\mathbf{s}}_{:,T}}, \\ &\qquad \ldots, \\ &\qquad f_{\mathbf{W}}(\ldots f_{\mathbf{W}}(f_{\mathbf{W}}(\mathbf{S}_{:,n-T-1}, \mathbf{O}_{:n-T}), \mathbf{O}_{:,n-T+1}), \ldots), \mathbf{O}_{:,n})\Big]. \end{aligned}$$

For $T = 1$, we recover the operator provided in (3.3). This generalization mimics the use of T-step methods for learning value functions in reinforcement learning. This generalization could significantly improve the propagation of state values, and provides more flexibility in using the allocated computation per iteration. For example, for a budget of T updates per iteration, we could use T 1-step updates, $T/2$ 2-step updates, all the way up to one T-step update. Intuitively, more shorter length updates should propagate state update more effectively across the buffer, as opposed to allocating all resources to one long gradient update. We test this hypothesis in the experiments.

The final algorithm—Fixed Point Propagation—is summarized in Algorithm 1. Fig. 3.1 represents a pictorial depiction of the same algorithm. The feed-forward path is represented using black arrows whereas the flow of gradients during back-propagation is highlighted using red arrows. The values highlighted by the red-circle indicate values either being added to or sampled from the buffer.

Algorithm 1 Fixed Point Propagation (FPP)

```
Input: truncation $T$ (can start higher and decay to $T = 1$)
Initialize weights $\mathbf{W}$ and $\boldsymbol{\beta}$ randomly
Initialize start state $\mathbf{s}_0 = \mathbf{0}$
Initialize empty buffer $B$
for t = 1, 2, ... do
    If buffer $B$ is full, drop the oldest transition
    Add $(\mathbf{s}_t = f_{\mathbf{W}}(\mathbf{s}_{t-1}, \mathbf{o}_t), \mathbf{o}_t, \mathbf{y}_t)$ to buffer $B$
    Sample length $T$ trajectory from the buffer:
        $\mathbf{s}_{i-T}, \mathbf{o}_{i-T}, \ldots, \mathbf{s}_i, \mathbf{o}_i, \mathbf{y}_i$
    Update $\mathbf{s}_{i-T}, \mathbf{s}_i, \mathbf{W}$ and $\boldsymbol{\beta}$ using (3.7)
end for
```

The loss for general T similarly decomposes into a sum $\sum_{i=T}^{n} L_i(\boldsymbol{\beta}, \mathbf{W}, \mathbf{S})$ for

$$L_i(\boldsymbol{\beta}, \mathbf{W}, \mathbf{S}) = \ell_{\boldsymbol{\beta}}(\hat{\mathbf{s}}_i(\mathbf{s}_{i-T}, \mathbf{W}); \mathbf{y}_i) + \frac{\lambda}{2}\|\mathbf{s}_i - \hat{\mathbf{s}}_i(\mathbf{s}_{i-T}, \mathbf{W})\|_2^2 \tag{3.6}$$
$$\text{where} \quad \hat{\mathbf{s}}_i(\mathbf{s}_{i-T}, \mathbf{W}) \overset{\text{def}}{=} f_{\mathbf{W}}(\ldots f_{\mathbf{W}}(f_{\mathbf{W}}(\mathbf{s}_{i-T-1}, \mathbf{o}_{i-T}), \mathbf{o}_{i-T+1}), \ldots), \mathbf{o}_i).$$

For each stochastic sample i, ∇L_i is only non-zero for $\boldsymbol{\beta}, \mathbf{W}, \mathbf{s}_{i-T}$ and $\mathbf{s}_i$. Though these updates can simply be computed using automatic differentiation on L_i, we provide the following explicit updates, with shorthand $\hat{\mathbf{s}}_i$ for $\hat{\mathbf{s}}_i(\mathbf{s}_{i-T}, \mathbf{W})$

$$\begin{aligned}
\nabla_{\mathbf{s}_{i-T}} L_i &= [\nabla_{\hat{\mathbf{s}}_i} \ell_{\boldsymbol{\beta}}(\hat{\mathbf{s}}_i; \mathbf{y}_i) - \lambda(\mathbf{s}_i - \hat{\mathbf{s}}_i)]^\top \nabla_{\mathbf{s}_{i-T}} \hat{\mathbf{s}}_i \\
\nabla_{\mathbf{s}_i} L_i &= \lambda(\mathbf{s}_i - \hat{\mathbf{s}}_i) \\
\nabla_{\mathbf{W}} L_i &= [\nabla_{\hat{\mathbf{s}}_i} \ell_{\boldsymbol{\beta}}(\hat{\mathbf{s}}_i; \mathbf{y}_i) - \lambda(\mathbf{s}_i - \hat{\mathbf{s}}_i)]^\top \nabla_{\mathbf{W}} \hat{\mathbf{s}}_i \\
\nabla_{\boldsymbol{\beta}} L_i &= \nabla_{\boldsymbol{\beta}} \ell_{\boldsymbol{\beta}}(\hat{\mathbf{s}}_i; \mathbf{y}_i)
\end{aligned} \tag{3.7}$$

As alluded to, the advantage of FPP over T-BPTT is that we are not restricted to focusing all computation to estimate the gradient T-steps back-in-time for one state-observation pair. 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 updates more states per

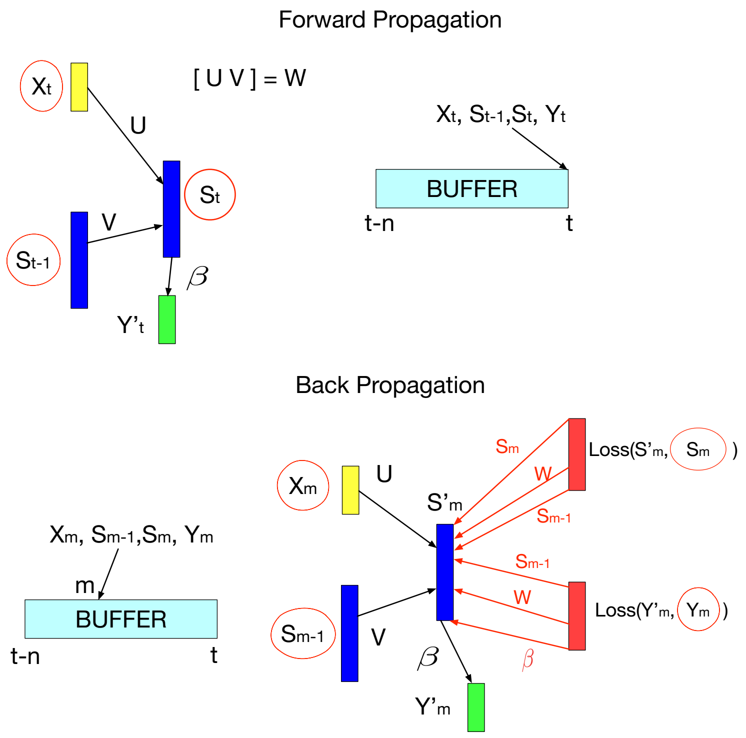


Figure 3.1: Saving and sampling transitions for the Fixed Point Propagation (FPP) algorithm, with $T = 1$.

step, 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 formulation as a stochastic gradient descent on the fixed point objective makes it a sound strategy—as opposed to truncation which is not sound. FPP, therefore, maintains the simplicity of T-BPTT, but provides a more viable direction to obtain sound algorithms for training RNNs.

Chapter 4

Experiments

The goal of the experiments is to investigate (a) the efficacy of FPP for training RNNs and (b) if FPP has the hypothesized advantages over T-BPTT, in terms of computation and robustness. We compare FPP to T-BPTT with the same RNN architecture, as well as only a buffer-based variant of FPP. This variant similarly uses a buffer but does not explicitly learn auxiliary variables; we include it to investigate the importance of the use of auxiliary variables beyond buffer-based updates. We further evaluate the ability of FPP to take advantage of mutiple updates per step. We provide a parameter study for FPP in Section 4.6, to buffer size and truncation.

We do not compare with more baselines, because in these datasets BPTT performs well and so provides a gold standard for performance. The use of truncation in BPTT enables us to range from very good performance to poor performance, trading off memory and computation. We can then ask: how does our algorithm perform over this range, compared to truncated BPTT?

4.1 Environments and Datasets

We use the following four tasks for evaluation: CycleWorld [39], Stochastic-World [1], Sequential MNIST [21] (image classification) and Penn-Tree Bank [25] (character prediction).

CycleWorld is a deterministic 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. The main learning indicator of the model is whether it correctly predicts when the next observation should be 1; a classification accuracy of 83%(for 6 Cycleworld) and 90% (for 10 Cycleworld) is obtained by the naive predictor that always predicts 0. We include this task is to control the dependency back-in-time, by varying p. As in previous work, we use a simple RNN with 4 hidden neurons.

StochasticWorld is a stochastic environment producing a binary sequence of observations and targets. The input at time t, X_t has a 50% probability of being 0 and 50% probability of being 1. The output at time t Y_t also has a 50% probability of being 0 and 50% probability of being 1. The probability of Y_t being 1 is increased by increased by 50% (i.e., to 100%) if X_{t-5} is 1, and decreased by 25% (i.e., to 25%) if X_{t-12} is 1. If both X_{t-5} and X_{t-12} is 1, then the probability that Y_t is 1 is 75%. We include this task is to test the robustness of FPP to stochasticity. For this problem, a cross entropy loss of 0.66 indicates that neither of the two dependencies have been learnt. When the RNN learns one dependency (t-5) the cross entropy loss is about 0.51. When the model learns both the dependencies the cross-entropy loss is about 0.45.

$P(Y_t \mid X_{t-5} = 0, X_{t-12}=0)$	50%
$P(Y_t \mid X_{t-5} = 1, X_{t-12}=0)$	100%
$P(Y_t \mid X_{t-5} = 0, X_{t-12}=1)$	25%
$P(Y_t \mid X_{t-5} = 1, X_{t-12}=1)$	75%

For this task, we use a simple RNN with 32 hidden neurons.

Sequential MNIST is a pixel-by-pixel MNIST classification dataset, where the input is each pixel of the dataset, in a sequential manner, and the target is to predict the label of the image. At each and every time-step, a row of pixels of the image (28x28) were given as the input, and each time step had the correct target as the output. We include this tasks to test long-term dependencies. For this task, we used an RNN with 512 hidden units. Each of the algorithms were run for 300 iterations and the results are an average over 5 runs over the entire training data.

Penn Tree Bank Dataset is a character prediction dataset. We include this dataset because language modeling is one of the most common applications of RNNs. We used a vocabulary size of 10000. The Target Loss function used here is the seq2seq loss used in Tensorflow, which is a weighted cross-entropy loss for a sequence of logits. For this task, we used an LSTM with 200 hidden-neurons.

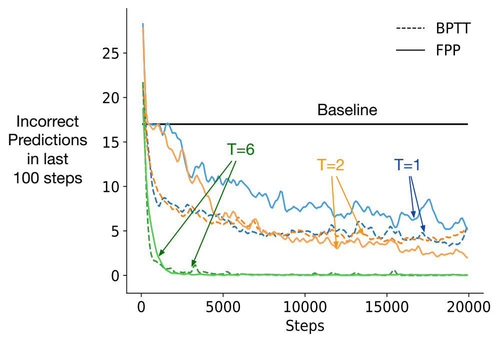
4.2 Comparison to T-BPTT

We first compare FPP to T-BPTT with varying truncation levels, in Cycle-World and StochasticWorld. For all the algorithms, we used a constant buffer size of 1000 and the trajectory length T, with RMSProp optimizer for the weight updates for both BPTT and FPP. From Figure 4.1 (a) and (b), it is evident that FPP learns almost as fast as full-BPTT (6-BPTT for (a) and 10-BPTT for (b)). Figure 4.1 (b) also portrays that for values of T in BPTT less than 10, T-BPTT performs poorly, whereas even for T=5 for FPP, the network does almost as well as 10-BPTT. Both of these methods have similar time-complexity of $O(Tk^2)$—where T is the trajectory length in the case of FPP and the truncation parameter in the case of BPTT. From Figure 4.1 (c), FPP seems to be much more robust to the trajectory length (T). So even T=5 does better than 10-BPTT and having T=10 makes it learn both the dependencies, which is not possible with 10-BPTT which learns only the first dependency.

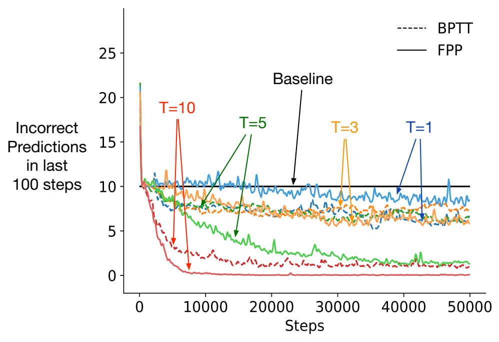
For the Penn-Tree Bank Dataset and Sequential MNIST, we include final performance in Table 4.1 and Table 4.2. FPP performs significantly better than T-BPTT in Sequential MNIST, with accuracies almost 10% higher. The effect is less pronouced in Penn-Tree, but FPP still consistently performs better.

Table 4.1: FPP vs T-BPTT on Sequential MNIST

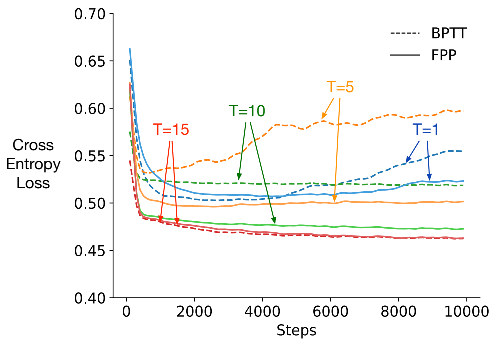
	Accuracy		
Algorithm	T = 5	T = 10	T = 20
BPTT	66.169	59.670	68.874
FPP	66.435	73.820	74.067



(a) 6-CycleWorld



(b)10-CycleWorld



(c) StochasticWorld

Figure 4.1: Learning curves for T-FPP and T-BPTT with different T in three domains. The curves for FPP are solid lines and T-BPTT are dotted lines. The different colors corresponds to different T. Both FPP and T-BPTT perform poorly for T = 1, but FPP begins to perform well for smaller T than T-BPTT.

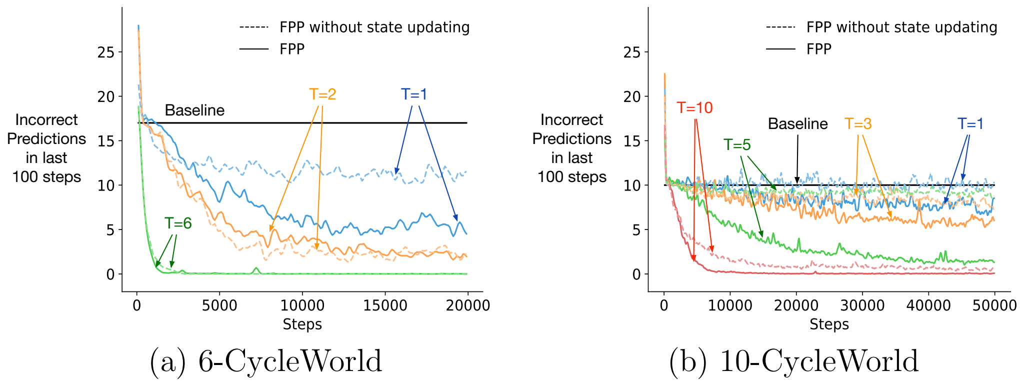
Table 4.2: FPP vs BPTT on Penn-Tree Bank Dataset

	Cross Entropy		
Algorithm	T = 1	T = 5	T = 10
BPTT	6.453	6.595	6.592
FPP	6.476	6.402	6.359

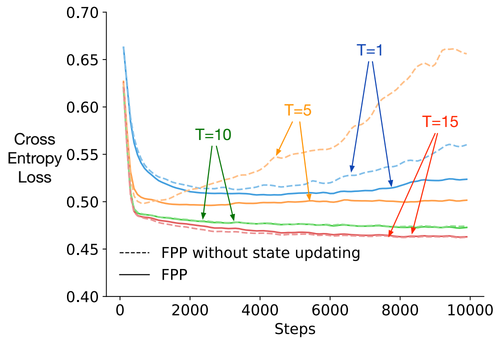
4.3 Importance of Auxiliary Variables

To investigate the benefit of the fixed point formulation and auxiliary variables, we compare FPP with the BPTT-like version of our algorithm. BPTT can also take advantage of a buffer, simply by executing T-length BPTT updates from random states backwards in the buffer. This corresponds to using only the target loss component of the FPP objective, without the explicit use of auxiliary variables.

From Figure 4.2, it is evident that for smaller values of T, FPP without state updating stills suffers from the same problems as T-BPTT. FPP is particularly robust in the StochasticWorld, which is more reflective of typical partially observable domains than the deterministic cycles in CycleWorld. The poor performance of BPTT and FPP without state updating for smaller T is mainly because of the inherent stochasticity in this problem. Also, from Figure 4.2(b), particularly for T=5, we can see that FPP with state updating still performs a lot worse compared to FPP. The experiment highlights that the addition of the auxiliary variable **S**, which provide a sound update, seems to have a significant impact.



(a) 6-CycleWorld (b) 10-CycleWorld



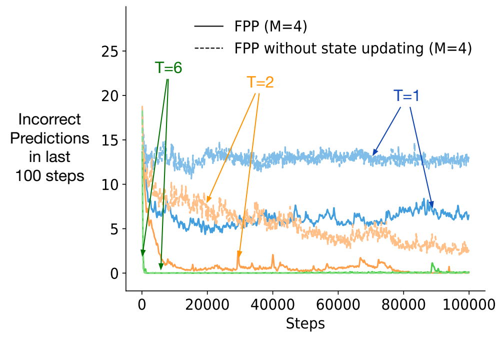
(c) StochasticWorld

Figure 4.2: Learning curves for FPP(without state updating) are dotted and solid for FPP. This result indicates the utility of having the auxiliary variable, to improve stability and performance.

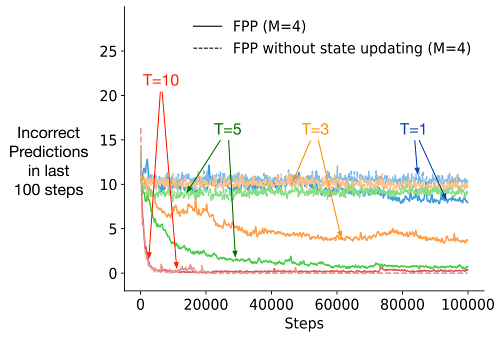
4.4 Multiple T-step Updates

One of the advantages of FPP is its ability to perform multiple updates, which enables better propagation of state values with more updates to the buffer and also should be sound and strictly improve learning. Thus, we can perform M T-step updates in parallel and then update the weights for each of the processes in a sequential manner. This is not obviously possible with BPTT, since it only calculates gradients with respect to the current state-observation pair. As in the previous experiment, FPP and FPP without state updating can both do many T-step updates for the given buffer, with the only difference being that FPP uses auxiliary variables. We compare performance for both the algorithms with M=4 updates per step.

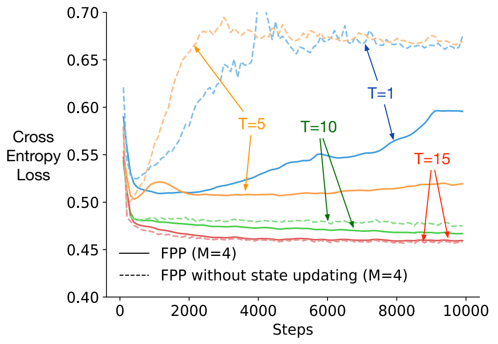
From Fig. 4.3, we see that both version of FPP, which do four updates per step, improve upon BPTT. FPP can better take advantage of multiple updates per step, improving performance particularly on 6-CycleWorld for T=2 over its own performance with one update and over FPP without state updating (M=4). Comparison with FPP (M=1 update per step) and BPTT are given in Fig. 4.4 and Fig. 4.5 respectively.



(a) 6-CycleWorld

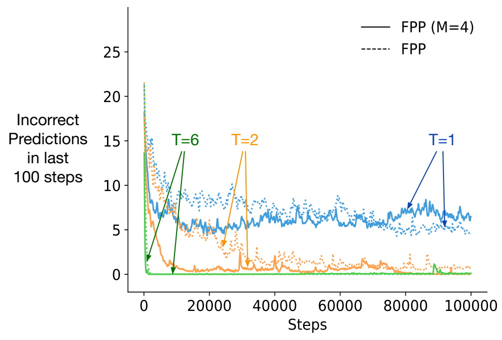


(b) 10-CycleWorld

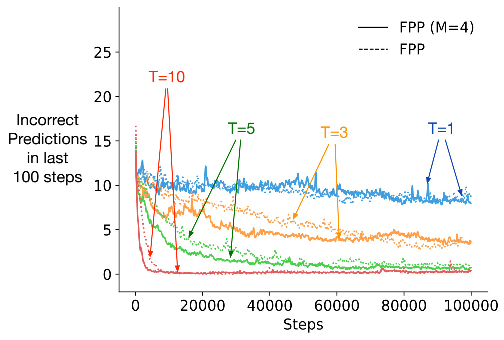


(c) StochasticWorld

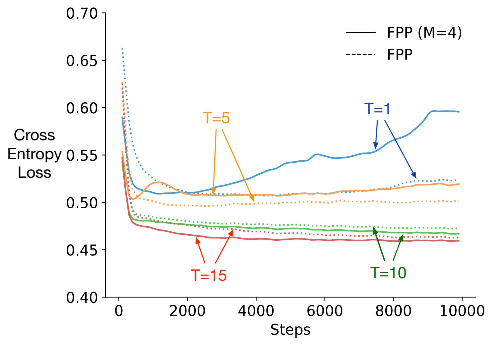
Figure 4.3: M=4 updates per step in FPP and FPP without state updating. Multiple updates across the buffer improves performance for FPP more compared to FPP without state updating suggesting the advantage of using auxiliary variables.



(a) 6-CycleWorld

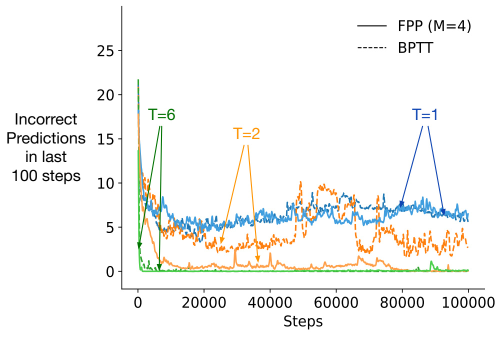


(b) 10-CycleWorld

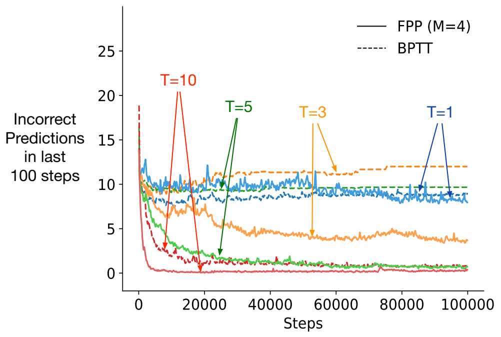


(c) StochasticWorld

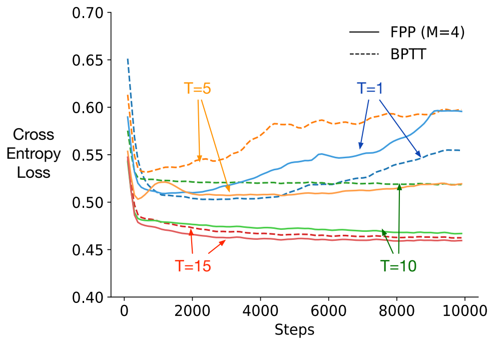
Figure 4.4: Learning curves for 4 updates per step in solid vs 1 update per step FPP in dotted. This graph is a comparison of how much it can help in learning faster.



(a) 6-CycleWorld



(b) 10-CycleWorld



(c) StochasticWorld

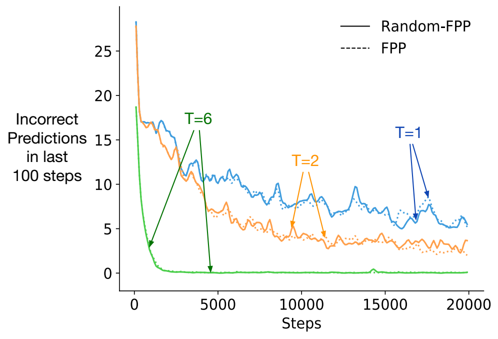
Figure 4.5: Learning curves for 4 updates per step FPP in solid and BPTT in dotted. BPTT cannot take advantage of multiple updates per step but FPP can, and hence the performance is much better.

4.5 Trading off the Length and Number of Updates

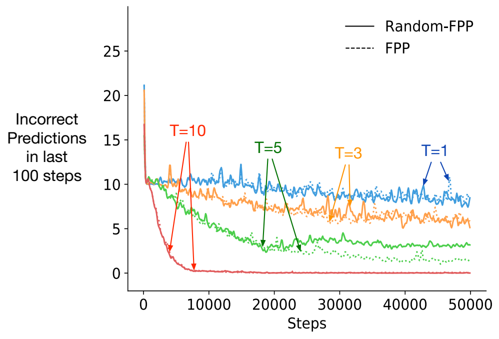
One modification to the FPP algorithm could be to decay the T-step updates, and begin using more updates with smaller T. We can initially start with a high probability of choosing T-step updates and decay this probability over time, and increase the probability of doing T 1-step updates. The decay to T 1-step updates can significantly improve speed, because these 1-step updates can be done in parallel. We find that such a decay performs just as well as continuing to use a full T-step update (see Fig. 4.7), indicating FPP provides more flexibility to take advantage of parallelism while still maintaining high accuracy.

For Decay-FPP, we decay the value of T by 2 and increase the number of updates by 2, after the completion of 20% of training steps. We continue this until Decay-FPP is performing T 1-step updates. For Random-FPP. we decay the probability of choosing T-step updates by 20% after the completion of 20% of training steps.

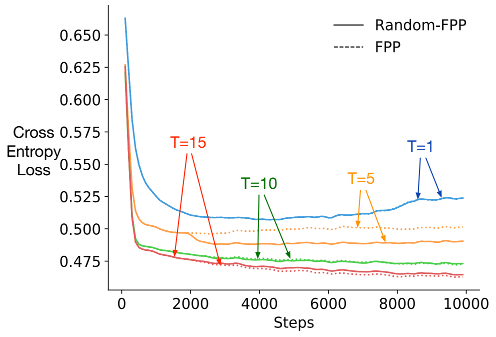
From Figures 4.6 and 4.7, this method seems to do almost as well as FPP and even better in some cases. Though in certain cases, it performed worse depending on T. It would be beneficial to start with longer T-step updates, but then could begin shortening updates later in learning once a reasonable initial set of states had been obtained. As T decreases, it becomes easier to perform more and more updates and also facilitates multiple updates across the buffer. These domains, though are quite simple. Further investigation is needed to understand the utility of spreading updates of different lengths, in early and later learning.



(a) 6 CycleWorld

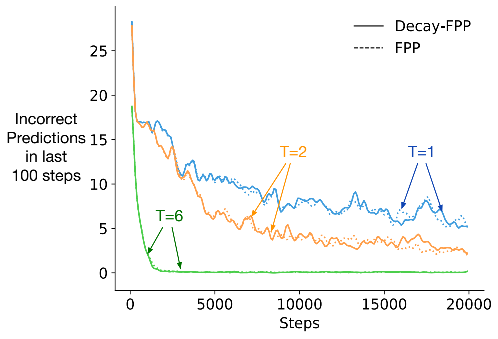


(b) 10 CycleWorld

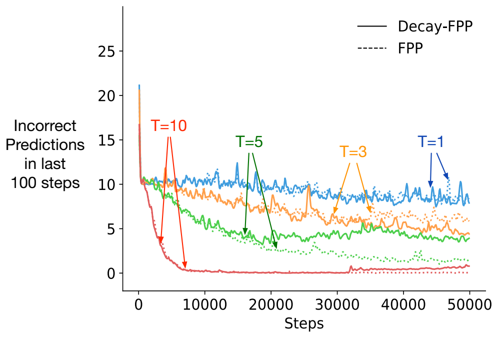


(c) StochasticWorld

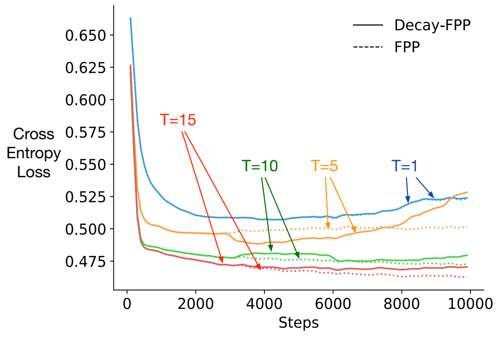
Figure 4.6: Learning curves of FPP in dotted and Random-FPP in solid. Reducing T with increasing steps does not affect performance much and even improves performance in some cases.



(a) 6 CycleWorld



(b) 10 CycleWorld

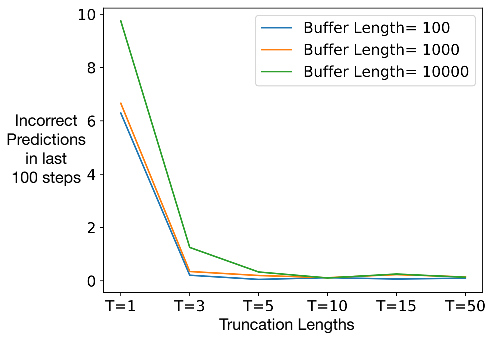


(c) StochasticWorld

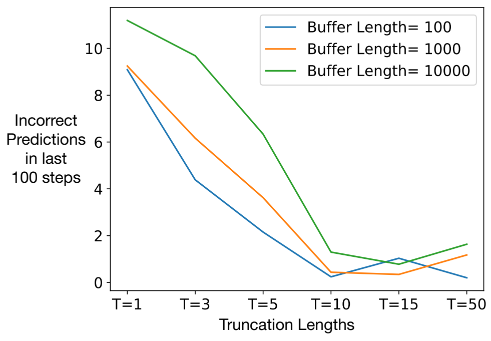
Figure 4.7: Learning curves of FPP in dotted and Decay-FPP in solid. Another strategy for decaying T also yields similar results, which means we can perform multiple updates per step towards the later stages of learning.

4.6 Parameter Study

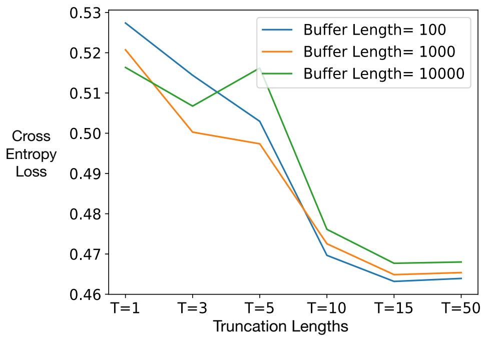
We investigate the sensitivity of FPP to its two parameters: the length of the trajectory T, and the buffer size N. Overall, the inaccuracies/losses on y-axis of Figure 4.8 show that FPP is quite robust to buffer size and number of updates. As expected, for very small T, performance degrades, but otherwise the move from T= 10 to T= 50 does not result in a large difference. The algorithm was quite invariant to buffer size, starting from a reasonable size of 100. For too large a buffer with a small number of updates, performance did degrade somewhat. Overall, though, across this wide range of settings, FPP performed consistently well.



(a) 6-CycleWorld



(b) 10-CycleWorld



(c) StochasticWorld

Figure 4.8: Sensitivity to buffer length and trajectory length in FPP, for buffer sizes 100, 1000 and 10000 and truncations of 1,3,5,10,15 and 50.

Chapter 5

Conclusion and Future Work

The main objective of this thesis is to formulate RNN training as a fixed point problem for the constructed 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. Further, there are clear steps for proving convergence of the approach, and even improving optimization over the auxiliary variables, such as by taking advantage of the natural parallelism of updates. Overall, this work provides evidence that FPP could be a promising direction for robustly training RNNs, without the need to compute or approximate long gradients back-in-time.

For future work, we are looking at comparing FPP with newer algorithms as mentioned in Chapter 2 as well as including a few more data-sets. Also, we are curious on implementing it on some many-to-one prediction problems, where there is a target at the end unlike online problems where you have a target at each time-step. We are also looking at trying to have a theoretical proof on the convergence of our algorithm.

References

1. `https://r2rt.com/recurrent-neural-networks-in-tensorflow-i.html`. 17

2. Almeida, L. B. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. *Proceedings, 1st First International Conference on Neural Networks* (1987). 8, 13

3. Bengio, Y., Simard, P. & Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* **5,** 157–166. ISSN: 1045-9227 (Mar. 1994). 7

4. Campos, V., Jou, B., Giró i Nieto, X., Torres, J. & Chang, S. Skip RNN: Learning to Skip State Updates in Recurrent Neural Networks. *CoRR* **abs/1708.06834.** arXiv: `1708.06834`. `http://arxiv.org/abs/1708.06834` (2017). 7, 9

5. Carreira-Perpiñán, M. Á. & Wang, W. *Distributed optimization of deeply nested systems.* in *International Conference on Artificial Intelligence and Statistics* (2014). 12, 13

6. Chan, W., Jaitly, N., Le, Q. & Vinyals, O. *Listen, attend and spell: A neural network for large vocabulary conversational speech recognition* in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (Mar. 2016), 4960–4964. doi:`10.1109/ICASSP.2016.7472621`. 2

7. Chang, S. *et al.* Dilated Recurrent Neural Networks. *CoRR* **abs/1710.02224.** arXiv: `1710.02224`. `http://arxiv.org/abs/1710.02224` (2017). 7, 9

8. Czarnecki, W. M., Jaderberg, M., Osindero, S., Vinyals, O. & Kavukcuoglu, K. Understanding Synthetic Gradients and Decoupled Neural Interfaces. *arXiv:1411.4000v2.* arXiv: `1703.00522v1` (2017). 7

9. Düll, S., Udluft, S. & Sterzing, V. *Solving Partially Observable Reinforcement Learning Problems with Recurrent Neural Networks* in *Neural Networks: Tricks of the Trade* (2012). 2

10. Elman, J. L. Finding Structure in Time. *Cognitive Science* **14,** 179–211. ISSN: 1551-6709 (1990). 1, 7

11. Gers, F. A., Schmidhuber, J. & Cummins, F. A. Learning to Forget: Continual Prediction with LSTM. *Neural Computation* **12,** 2451–2471 (2000). 7

12. Gotmare, A., Thomas, V., Brea, J. & Jaggi, M. Decoupling Backpropagation using Constrained Optimization Methods (2018). 12

13. Graves, A. & Schmidhuber, J. *Framewise phoneme classification with bidirectional LSTM networks* in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* **4** (July 2005), 2047–2052 vol. 4. doi:`10.1109/IJCNN.2005.1556215`. 7, 8

14. Graves, A., Mohamed, A. & Hinton, G. E. Speech Recognition with Deep Recurrent Neural Networks. *CoRR* **abs/1303.5778.** arXiv: `1303.5778`. `http://arxiv.org/abs/1303.5778` (2013). 2

15. Hinton, G. *et al.* Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* **29,** 82–97. ISSN: 1053-5888 (Nov. 2012). 2

16. Hochreiter, S. & Schmidhuber, J. Long Short-Term Memory. *Neural Comput.* **9,** 1735–1780. ISSN: 0899-7667 (Nov. 1997). 2, 3, 6–8

17. Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences* **79,** 2554–2558. ISSN: 0027-8424 (1982). 1

18. Jaderberg, M. *et al. Decoupled Neural Interfaces using Synthetic Gradients.* in *International Conference on Machine Learning* (2017). 7

19. Ke, N. R. *et al.* Sparse Attentive Backtracking: Long-Range Credit Assignment in Recurrent Networks. *arXiv:1509.01240v2.* arXiv: `1711.02326` (2017). 7, 8

20. Koutnık, J., Greff, K., Gomez, F. J. & Schmidhuber, J. A Clockwork RNN. *CoRR* **abs/1402.3511.** arXiv: `1402.3511`. `http://arxiv.org/abs/1402.3511` (2014). 7, 9

21. LeCun, Y. & Cortes, C. MNIST handwritten digit database. `http://yann.lecun.com/exdb/mnist/` (2010). 17

22. Liao, R. *et al. Reviving and Improving Recurrent Back-Propagation.* in *International Conference on Machine Learning* (2018). 7, 8, 13

23. Lu, J., Xiong, C., Parikh, D. & Socher, R. Knowing When to Look: Adaptive Attention via A Visual Sentinel for Image Captioning. *CoRR* **abs/1612.01887.** arXiv: `1612.01887`. `http://arxiv.org/abs/1612.01887` (2016). 2

24. Mao, J., Xu, W., Yang, Y., Wang, J. & Yuille, A. L. Explain Images with Multimodal Recurrent Neural Networks. *CoRR* **abs/1410.1090.** arXiv: `1410.1090`. `http://arxiv.org/abs/1410.1090` (2014). 2

25. Marcus, M. P., Marcinkiewicz, M. A. & Santorini, B. Building a Large Annotated Corpus of English: The Penn Treebank. *Comput. Linguist.* **19,** 313–330. ISSN: 0891-2017 (June 1993). 17

26. Mehri, S. *et al.* SampleRNN: An Unconditional End-to-End Neural Audio Generation Model. *CoRR* **abs/1612.07837.** arXiv: `1612.07837`. `http://arxiv.org/abs/1612.07837` (2016). 2

27. Miao, Y., Gowayyed, M. & Metze, F. EESEN: End-to-End Speech Recognition using Deep RNN Models and WFST-based Decoding. *CoRR* **abs/1507.08240.** `http://dblp.uni-trier.de/db/journals/corr/corr1507.html#MiaoGM15` (2015). 2

28. Mujika, A., Meier, F. & Steger, A. Approximating Real-Time Recurrent Learning with Random Kronecker Factors. *CoRR* **abs/1805.10842.** arXiv: `1805.10842`. `http://arxiv.org/abs/1805.10842` (2018). 7, 8

29. Neil, D., Pfeiffer, M. & Liu, S. Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences. *CoRR* **abs/1610.09513.** arXiv: `1610.09513`. `http://arxiv.org/abs/1610.09513` (2016). 7, 9

30. Ollivier, Y. & Charpiat, G. Training recurrent networks online without backtracking. *arXiv* (2015). 7, 8

31. Pascanu, R., Mikolov, T. & Bengio, Y. *On the Difficulty of Training Recurrent Neural Networks* in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (Atlanta, GA, USA, 2013). `http://dl.acm.org/citation.cfm?id=3042817.3043083`. 6, 7

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

33. Pineda, F. J. Generalization of back-propagation to recurrent neural networks. *Physical review letters* (1987). 8, 13

34. Roth, C. Z., Kanitscheider, I. & Fiete, I. R. *Kernel RNN Learning (KeRNL)* in *ICLR 2019* (2019). 7, 8

35. Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M. & Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks* **20,** 61–80 (2008). 13

36. Scellier, B. & Bengio, Y. Equilibrium Propagation: Bridging the Gap between Energy-Based Models and Backpropagation. *Frontiers in Computational Neuroscience* (2017). 12, 13

37. Schuster, M. & Paliwal, K. Bidirectional Recurrent Neural Networks. *Trans. Sig. Proc.* **45,** 2673–2681. ISSN: 1053-587X (Nov. 1997). 7, 8

38. Tallec, C. & Ollivier, Y. Unbiased Online Recurrent Optimization. *arXiv:1411.4000v2 [cs.LG].* arXiv: `1702.05043v3` (2017). 5–8

39. Tanner, B. & Sutton, R. S. *TD(lambda) Networks: Temporal-Difference Networks with Eligibility Traces* in (2005). 17

40. Taylor, G., Burmeister Ryan, Z. X., Singh, B., Patel, A. & Goldstein, T. Training Neural Networks Without Gradients - A Scalable ADMM Approach. *ICML* (2016). 12

41. Vinyals, O., Toshev, A., Bengio, S. & Erhan, D. Show and Tell: A Neural Image Caption Generator. *CoRR* **abs/1411.4555.** arXiv: `1411.4555`. `http://arxiv.org/abs/1411.4555` (2014). 2

42. Werbos, P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* **78,** 1550–1560. ISSN: 0018-9219 (Oct. 1990). 5

43. Williams, R. J. & Zipser, D. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation* **1,** 270–280. ISSN: 0899-7667 (June 1989). 5

44. Williams, R. J. & Peng, J. An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories. *Neural Computation* **2,** 490–501 (1990). 5

45. Williams, R. J. & Zipser, D. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation* (1989). 5–7

Appendix A

Experimental Details

The experimental details of each dataset are provided below.

A.1 6-CycleWorld

Network Type = simple RNN

Hidden Units = 4

Total time-steps = 100000

Batch size = 1

Optimizer = RMSprop

Learning rate = Swept over a range 0.0001-0.1 and the learning rate chosen for each algorithm had the best online prediction

Number of runs = 30

A.2 10-CycleWorld

Network Type = simple RNN

Hidden Units = 4

Total time-steps = 100000

Batch size = 1

Optimizer = RMSprop

Learning rate = Swept over a range 0.0001-0.1 and the learning rate chosen for each algorithm had the best online prediction

Number of runs = 30

A.3 Stochastic World

Network Type = simple RNN

Hidden Units = 32

Total time-steps = 1000000

Batch size = 100

Optimizer = RMSProp

Learning rate = Swept over a range 0.0001-0.1 and the learning rate chosen for each algorithm had the best online prediction

Number of runs = 30

A.4 Sequential MNIST

Network Type = simple RNN

Hidden Units = 512

Image size = 784 pixels

Input dimension = 28 pixels

Iterations = 300

Batch size = 100

Optimizer = RMSProp

Learning rate = Swept over a range 0.0001-0.1 and the learning rate chosen for each algorithm had the best online prediction

Number of runs = 5

A.5 PTB

Network Type = LSTM

Hidden Units = 200

Batch size = 20

Vocabulary Size = 10000

Optimizer = RMSProp

Learning rate = Swept over a range 0.00001-0.1 and the learning rate chosen for each algorithm had the best online prediction

Number of runs = 5