Training Recurrent Neural Networks via Forward Propagation Through Time

Anil Kag [1] **Venkatesh Saligrama** [1]

Abstract

Back-propagation through time (BPTT) has been widely used for training Recurrent Neural Networks (RNNs). BPTT updates RNN parameters on an instance by back-propagating the error in time over the entire sequence length, and as a result, leads to poor trainability due to the well-known gradient explosion/decay phenomena. While a number of prior works have proposed to mitigate vanishing/explosion effect through careful RNN architecture design, these RNN variants still train with BPTT. We propose a novel forward-propagation algorithm, FPTT , where at each time, for an instance, we update RNN parameters by optimizing an instantaneous risk function. Our proposed risk is a regularization penalty at time t that evolves dynamically based on previously observed losses, and allows for RNN parameter updates to converge to a stationary solution of the empirical RNN objective. We consider both sequence-to-sequence as well as terminal loss problems. Empirically FPTT outperforms BPTT on a number of well-known benchmark tasks, thus enabling architectures like LSTMs to solve long range dependencies problems.

1. Introduction

Recurrent Neural Networks (RNNs) have been successfully employed in many sequential learning tasks including language modelling, speech recognition, and terminal prediction. An RNN is described by its parameters $W \in \mathcal{W}$ and the transition function $f : \mathcal{W} \times \mathcal{X} \times \mathcal{H} \to \mathcal{H}$ which takes RNN parameters W, current input $x_t \in \mathcal{X}$ and previous hidden state $h_{t-1} \in \mathcal{H} \subseteq \mathbb{R}^D$ to output the next state h_t:

$$h_t = f(W, x_t, h_{t-1}) \tag{1}$$

Given the training dataset $\{x_i, y_i\}_{i=1}^N$ with N examples of $T-$length sequences x_i, y_i, we optimize the following empirical risk function:

$$[W^*, v^*] = \arg\min_{W,v} L(W, v) = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} \ell(y_t^i, \hat{y}_t^i)$$
$$\forall i, t;\ \hat{y}_t^i = v^\top h_t^i;\ h_t^i = f(W, x_t^i, h_{t-1}^i);\quad h_0^i = 0 \tag{2}$$

where, RNN parameters W and classifier $v \in \mathbb{R}^D$ are optimized. This objective is generally optimized by gradient descent. The gradient expression for each example, $i \in [N]$, is a sum of products of partial gradients, and is commonly referred to as the error back-propagated through time (BPTT). This is a direct result of the fact that the hidden state h_t^i, at each time, is recursively updated by the same parameter W. As a consequence, via chain rule, we get:

$$\frac{\partial L}{\partial W} = \sum_{i=1}^{N} \sum_{t=1}^{T} \frac{\partial \ell(y_t^i, \hat{y}_i^t)}{\partial W}$$
$$= \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} \frac{\partial \ell(y_t^i, \hat{y}_i^t)}{\partial \hat{y}_i^t} \frac{\partial \hat{y}_i^t}{\partial h_t^i} \sum_{j=1}^{t} \Big(\prod_{s=j}^{t} \frac{\partial h_s^i}{\partial h_{s-1}^i} \Big) \frac{\partial h_{j-1}^i}{\partial W} \tag{3}$$

We highlight two fundamental aspects of BPTT:

- **Trainability:** Unless the partial terms $\frac{\partial h_s}{\partial h_{s-1}}$ stay close to identity, the product of these terms could explode or vanish. As such, if gradients vanish, earlier times, $t \ll T$ have little contribution to the overall error. If gradients explode, we see an opposite effect, namely, later states may have little contribution.
- **Complexity:** Gradient computation is expensive for large T, since it involves a sum-product of T terms, resulting in $\Omega(T^2)$ scaling for each example, and for N examples, computational cost for processing the dataset once scales as $\Omega(NT^2)$. Note that this calculation assumes naive computation of the sum-product. In practice, with additional memory overhead, an efficient gradient propagation scheme would only result in cost linear in length of the sequence. BPTT has a $\Omega(T)$ memory overhead associated with storing all the intermediate hidden states in the time horizon.

In this work, our focus is on simplifying the RNN training procedure. Once the RNN parameters are learnt, the inference process remains same as before. Our goal is to reduce

[1]Department of Electrical and Computer Engineering, Boston University, USA. Correspondence to: Anil Kag <anilkag@bu.edu>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

computation of BPTT so that each step only involves taking a derivative for a single time.

Challenges. Minimizing Eq. 2 poses two challenges:
(a) *Dynamics.* Equation 1 enforces a temporal constraint on allowable transitions.
(b) *Time-Invariance.* The transition matrices W are fixed, and as a result the dynamics of RNNs is time-invariant.

Let us examine a few potential directions in this context. *Method of Multipliers* allows for eliminating constraints imposed by (a) and (b) by introducing a regularizer based on an augmented Lagrangian. This approach is often adopted in distributed optimization (Boyd et al., 2011). Eliminating (a) could be accomplished with ADMM methods with squared norm penalty, or other specialized functions (Gu et al., 2020). For (b), leveraging the key insight in distributed optimization, we can write the condition (b) as $W_t = W_{t-1}$ for all t, and rewrite it as a penalty. Nevertheless, while computationally block-coordinate descent allows for efficiency, memory expands substantially ($O(TD^2 + NTD)$).

Online Gradient Method (OGD). We can view $\ell_t(W) = \ell(y_t, v^\top f(W, x_t, h_{t-1})$ as the instantaneous loss incurred in round t by "playing" the parameter W. We can update $W_{t+1} = W_t - \eta \nabla \ell_t(W_t)$, based on the observed loss. While this could work, there is no reason why W_t's converge, and furthermore, it is unclear how to choose a constant parameter, W, based on the sequence of updates. In general, we have observed in experiments that training performance based on time-varying transition matrices does not reflect test-time and does not generalize well.

Follow-the-Regularized-Leader Rule. (McMahan et al., 2013) Rather than optimizing the instantaneous loss as in OGD, we utilize all of the previously seen losses, namely, $L_t(W) = \sum_{j=1}^{t} \ell_j(W)$ and attempt to find a update direction. Nevertheless, this approach suffers from the same issue as BPTT, since for large $t \approx T$, finding a descent direction involves back-propagation through $\approx T$ steps. Furthermore, this method adds a multiplicative factor of T in the run-time in comparison to BPTT.

Our Forward-Propagation Method. We propose a novel forward-propagation-through-time (FPTT) method based on instantaneous dynamic regularization. FPTT at each time takes a gradient step to minimize an instantaneous risk function. The instantaneous risk is the loss at time t plus a dynamically evolving regularizer. This dynamics is controlled by a state-vector, which summarizes past losses. FPTT has the in-built property that the point of convergence of W_t sequence, is also a stationary point of the global empirical risk Equation 2. The resulting method has a light-weight footprint and is computationally efficient. For sequence-to-sequence modelling tasks our learning scheme integrates easily since the losses are instantaneous, i.e., at timestep t, we immediately get feedback for the updated W_t. For terminal prediction problems we present a simple scheme to construct surrogate losses at any timestep using the label for the entire sequence.

We then conduct a number of experiments on benchmark datasets and show that our proposed method is particularly effective on tasks that exhibit long-range dependencies. In summary, our proposed method suggests that vanilla LSTMs are effective tools for inferring long-term dependencies, and exhibit performance matching state-of-the-art competitors–even those with higher capacities and well-designed architectures.

Toy Example. As a sneak preview, we demonstrate effectiveness of FPTT on the Add Task (see Sec. 4 for details) against BPTT on training LSTMs under an identical test/train split. Figure 1 shows that FPTT solves this problem while BPTT fails to find the correct parameters, it stays near the same loss value throughout the training phase. BPTT's poor behavior on this task has been observed in previous works (Kag et al., 2020; Zhang et al., 2018).

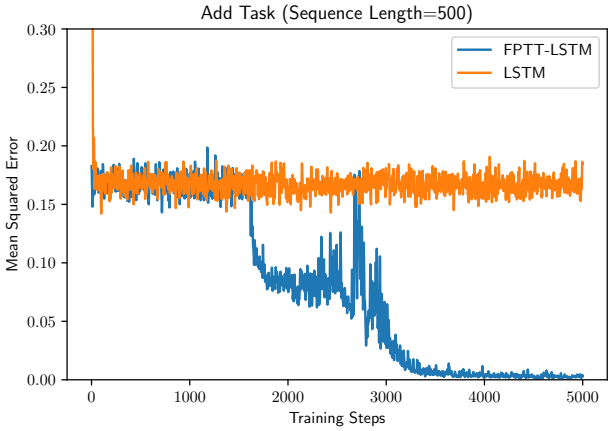


Figure 1. Add Task ($T = 500$): Comparison between standard learning and forward propagation.

Contributions.

- We proposed forward-propagation-through-time (FPTT) as an alternative to conventional BPTT.
- FPTT takes a gradient step of an instantaneous time-dependent risk function at each time. The risk function is the regularized loss, with a dynamic evolving regularization. The dynamic penalty requires minimal memory, thus allowing for rapid gradient computation.
- We construct surrogate losses for terminal prediction tasks, to guide FPTT to learn in intermediate time.
- We perform empirical evaluations to demonstrate the utility and superiority of FPTT over BPTT.
- Our FPTT algorithm can be readily deployed in any deep learning library. We have released our implementation at `https://github.com/anilkagak2/FPTT`

2. Related Work

There is a vast literature on RNNs that span novel architectural designs and algorithmic/methodological improvements. Here, we list only the closely related works.

Learning Algorithms. While a number of RNN training methods have been proposed, BPTT remains the single most dominant method (Rumelhart et al., 1986; Werbos, 1990). BPTT unrolls the recurrent logic for the entire time horizon and computes gradient through this horizon. It has been observed to be computationally expensive (storing the hidden states and computing gradient for entire time horizon) and unless the RNN architecture is carefully designed, this leads to vanishing / exploding gradients (Hochreiter, 1991; Bengio et al., 2013). Truncated BPTT (Williams & Peng, 1990) is the variant of BPTT where gradient flow is truncated after a fixed number of timesteps. This fails to learn dependencies present beyond the fixed window.

Real time recurrent learning (RTRL) (Williams & Zipser, 1989), a BPTT alternative, proposes to propagate the partials $\frac{\partial h_t}{\partial h_{t-1}}$ and $\frac{\partial h_t}{\partial W}$ from timestep t to $t+1$, by noting that there is significant overlap in the product term (see Eq. 3) from time t to $t+1$, allowing for recursive computation. Early attempts suffered large memory overhead limiting its usage, and while recent attempts (Mujika et al., 2018; Menick et al., 2021; Tallec & Ollivier, 2018; Ollivier & Charpiat, 2015) have been more successful, these methods still fall short of BPTT performance, and so trainability of RNNs is still a significant issue.

In contrast to these gradient based methods, FPTT is based on directly updating RNN parameters, and the updates optimize an instantaneous loss. Thus, RNN parameters are allowed to vary after each time-step, and over time our updates converge to stationary solution. As a result, our method has low memory footprint, small computational cost/iteration, and most importantly, exhibits empirical performance, dominating BPTT training on many long-term dependency datasets.

RNN Architectures. The choice of architecture has a significant impact on trainability with BPTT. To this end gated variants have been introduced: Long Short Term Memory networks (LSTMs) (Hochreiter & Schmidhuber, 1997) and Gated Recurrent Units (GRUs) (Cho et al., 2014) have considerably improved performance, but have been shown to fare poorly on tasks involving long range dependencies due to vanishing gradients (Zhang et al., 2018; Chang et al., 2019; Kusupati et al., 2018).

Unitary RNNs (Arjovsky et al., 2016; Jing et al., 2017; Zhang et al., 2018; Mhammedi et al., 2017; Kerg et al., 2019; Lezcano-Casado & Martínez-Rubio, 2019) focus on designing well-conditioned state transition matrices, attempting to enforce unitary-property, during training. This helps mitigate the gradient issues, but are less popular due to overhead involved in imposing unitary/orthogonal constraints. There are other designs, such as those based on ODEs (Chang et al., 2019; Kag et al., 2020; Kag & Saligrama, 2021; Kusupati et al., 2018; Erichson et al., 2021). These RNNs enforce the partials between the hidden states to be near identity, thus mitigating gradient issues faced by architectures like LSTMs/GRUs.

Our view is that architecture and training methods are complementary. Our ablative results suggest that FPTT improves upon BPTT method on different architectures.

Miscellaneous. (Miller & Hardt, 2019) analyzes LSTMs from the stability perspective and show that in an unconstrained form LSTMs are unstable. They show that under some conditions on the non-linearity in transition, stable recurrent neural networks can be represented by a feed-forward network. While stability reasons about the exploding gradients, it does not eliminate the vanishing gradients issue during training. (Linsley et al., 2020) addresses the large memory cost of BPTT which scales linearly with the number of time steps. They modify the training process by replacing the BPTT algorithm with Recurrent BackProp (RBP) algorithm which optimizes the parameters to achieve steady state dynamics. We point out that their scheme is orthogonal to FPTT and we can leverage RBP to provide similar benefits

3. Method

First, we describe our proposed algorithm and our learning objective. We describe a general pseudo code that can be leveraged to train any RNN architecture. Finally, we will describe a method for dealing with terminal prediction tasks.

Notation. The training set $B = \{x^i, y^i\}_{i=1}^N$ consists of N examples. Each input $x^i \in \mathcal{X}$ is a $T-$length sequence which can be written as $\{x_1^i, x_2^i, \cdots, x_T^i\}$. For sequence-to-sequence modelling tasks, the label $y^i \in \mathcal{Y}$ is a $T-$length sequence written as $\{y_1^i, y_2^i, \cdots, y_T^i\}$. While in the terminal prediction case, the label $y^i \in \mathcal{Y}$ is provided for a single timestep T as the feedback for the entire input sequence x^i. We will drop the superscript i to denote a data point wherever it can be inferred from the context. With initial hidden state $h_0 = 0 \in \mathcal{H}$, an RNN with parameters $W \in \mathcal{W}$ and transition function $f : \mathcal{W} \times \mathcal{X} \times \mathcal{H} \to \mathcal{H}$ generates the hidden state sequence $\{h_1, h_2, \cdots, h_T\}$ for the data point $\{x, y\}$. We use $\ell_t(W) = \ell(y_t, v^\top f(W, x_t, h_{t-1}))$ to denote the loss incurred at time step t, where $v \in \mathcal{V}$ is a linear classifier.

To simplify the equations, (a) we will only use one example and drop the $\sum_i$ and superscript i used in Eq. 2, since the motivation behind the proposal remains same, and (b) we will assume v is constant, while in practice to learn v, the

Algorithm 1 Training RNN with BackProp

Input: Training data $B = \{x^i, y^i\}_{i=1}^N$, Timesteps T
Input: Learning rate η, #Epochs E
Initialize: W_1 randomly in the domain $\mathcal{W}$
for $e = 1$ **to** E **do**
  Randomly Shuffle B
  for $i = 1$ **to** N **do**
    Set: $(x, y) = (x^i, y^i)$ and $h_0 = 0$
    for $t = 1$ **to** T **do**
      Update : $h_t = f(W, x_t, h_{t-1})$
    end for
    Loss: $\ell(W) = \sum_{t=1}^T \ell(y_t, v^\top h_t)$
    Set: : $W_{i+1} = W_i - \eta \nabla_W \ell(W)|_{W=W_i}$
  end for
  Reset: $W_1 = W_{N+1}$
end for
Return : W_{N+1}

operations applied on W, are applied on v as well.

3.1. FPTT : Forward Propagation Through Time

Given one example $(x, y) = (\{x_t\}_{t=1}^T, \{y_t\}_{t=1}^T)$ and initial parameter estimate W_0, BPTT (see Algorithm 1) updates the parameter once by taking the gradient of the T length loss $\sum_{t=1}^T \ell_t(W)$. In contrast, we update parameters at every time step t by utilizing (x_t, y_t) to avoid getting penalized by T length gradient dependence. Since the parameters update very frequently, we need to incorporate two mechanisms in parameter updates: (a) *stability* in updates so that a single step does not stray, (b) since our training does not follow standard RNN transition (i.e. keep a single parameter W through the input sequence), our updates should ensure that the iterates converge to a single parameter. This in turn guarantees that towards the end of the training sequence we will mimic an RNN.

To build motivation into our method Algorithm 2, we refer to the sequence of updates on a single instance, $i \in [N]$ at time step t. The first update is a gradient step of the loss $\ell_t(W)$ for a fixed value of $\bar{W}_t$. As such, we track one additional copy of the parameter, W_t, namely $\bar{W}_t \in \mathcal{W}$. At time step t, we update parameters using the supervision (x_t, y_t) and previous iterates $W_t, \bar{W}_t$. Following T updates, on a new instance, we set the initial weight parameter $W_0 \leftarrow W_{T+1}$, and the iteration follows subsequently.

To understand our scheme, let us consider the situation where the number of gradient steps of the loss approaches infinity. In this case, our equations read as:

$$W_{t+1} = \arg\min_W \ell_t(W) + \frac{\alpha}{2}\|W - \bar{W}_t - \frac{1}{2\alpha}\nabla\ell_{t-1}(W_t)\|^2 \tag{4}$$

$$\bar{W}_{t+1} = \frac{1}{2}(\bar{W}_t + W_{t+1}) - \frac{1}{2\alpha}\nabla\ell_t(W_{t+1}) \tag{5}$$

These update equations are loosely inspired by consensus in distributed optimization problems over a star-network[1]

Intuition. The basic concept here is that $\bar{W}_t$ represents, in principle, the running average of all the W_t's seen so far, with a small correction term (Eq. 5). Therefore, the updates impose proximity to the running average in the update step. However, this alone is not sufficient to converge to stationary points of Eq. 3. We will show this later. As such $\bar{W}_t$ is a vector that summarize past losses. Eq. 5 is also the first order condition for $\ell_t(W_{t+1}) + \frac{\alpha}{2}\|W_{t+1} - \bar{W}_t - \frac{1}{\alpha}\nabla\ell_{t-1}(W_t)\|^2$. Taken together the scheme resembles an alternative optimization method for a joint risk function over $W, \bar{W}$, namely, we hold $\bar{W}_t$ fixed and optimize W, and after the update, optimize $\bar{W}$ with fixed W. However, notice that unlike conventional setting, here the risk functions are time-varying. Note that Eq. 5 requires gradient of the loss ℓ_t at the new iterate W_{t+1}. Computational cost for this step can be eliminated by keeping a running estimate λ_t with update equation $\lambda_{t+1} = \lambda_t - \alpha(W_{t+1} - \bar{W}_t)$ and initial value $\lambda_0 = 0$.

Observe that for large α, we expect W_{t+1} to be close to the previous W_t, and this would result in the hidden state sequence $\{h_t\}_{t=1}^T$ to be essentially very close to the one generated by a single $W \approx W_{t+1} \approx W_t$. In effect this would simulate hidden state trajectories with a static time-invariant RNN parameter.

Pseudo Code. Algorithms 1 and 2 enumerates the learning schemes for BPTT and FPTT respectively. These procedures can be utilized to train an RNN architecture in any popular deep learning framework with minimal efforts. Note that for simplicity we write the algorithms with batch size 1, this is relaxed to the conventional choice of larger batch size in our experiments. In FPTT , starting with small values of α we gradually increase α to enforce the constraint. We explore the impact of this hyper-parameter in the ablative experiments (see Sec. 4.2).

Remarks. (a) Note that even though we have separate W_t for each timestep, we do not suffer additional storage overhead of the factor T. This follows from the fact that we solve these sub-problems forward in time and only solve for W_{t+1} at timestep t. (b) We show that the iterates converge in our ablative experiments (see supplementary), below we provide an explanation for the convergence.

Convergence. Let us focus on the $\arg\min$ step in Eq. 4.

[1] Distributed agents connected over a star-network seek to solve a joint optimization problem, which requires seeking consensus on the decision variables (Boyd et al., 2011). A master agent coordinates with the agents to communicate and synchronize decision variables in an iterative fashion. Eq. 2 could be viewed in a number of ways, as a single-agent network, a T node network, or an N node network etc. Each of these in turn lead to different coordinating mechanisms.

Algorithm 2 Training RNN with FPTT

Input: Training data $B = \{x^i, y^i\}_{i=1}^N$, Timesteps T
Input: Learning rate η, Hyper-parameter α, # Epochs E
Initialize: W_1 randomly in the domain $\mathcal{W}$
Initialize: $\bar{W}_1 = W_1$
for $e = 1$ **to** E **do**
  Randomly Shuffle B
  for $i = 1$ **to** N **do**
    Set: $(x, y) = (x^i, y^i)$ and $h_0 = 0$
    for $t = 1$ **to** T **do**
      Update : $h_t = f(W, x_t, h_{t-1})$
      $\ell_t(W) = \ell_t(y_t, v^\top h_t)$
      $\ell(W) = \ell_t(W) + \frac{\alpha}{2}\|W - \bar{W}_t - \frac{1}{2\alpha}\nabla\ell_{t-1}(W_t)\|^2$
      $W_{t+1} = W_t - \eta\nabla_W \ell(W)|_{W=W_t}$
      $\bar{W}_{t+1} = \frac{1}{2}(\bar{W}_t + W_{t+1}) - \frac{1}{2\alpha}\nabla\ell_t(W_{t+1})$
    end for
    Reset: $W_1 = W_T$ and $\bar{W}_1 = \bar{W}_T$
  end for
end for
Return : W_T

Taking gradient w.r.t. W results in the following dynamics:

$$\nabla\ell_t(W_{t+1}) - \nabla\ell_{t-1}(W_t) + \alpha(W_{t+1} - \bar{W}_t) = 0$$
$$\bar{W}_{t+1} = \frac{1}{2}(\bar{W}_t + W_{t+1}) - \frac{1}{2\alpha}\nabla\ell_t(W_{t+1}) \quad (6)$$

Let us see why these equations allow for reaching a stationary point of Eq. 2. For now suppose the sequence W_t converges to a limit point W_∞. One way to ensure this happens is to view Eq. 6 as a map from $[W_t, \bar{W}_t]^\top \to [W_{t+1}, \bar{W}_{t+1}]^\top$, and show that this map is contractive, and as such invoke the Banach fixed point theorem. Nevertheless, this is difficult to show and we assume that it is true for now.

Proposition 1. *In the Algorithm 2, suppose, the sequence W_t is bounded and converges to a limit point W_∞. Further assume the loss function ℓ_t is smooth and Lipschitz. Let the cumulative loss be $F = \frac{1}{T}\sum_{t=1}^T \nabla\ell_t(W_\infty)$ after T iterations [2]. It follows that W_∞ is a stationary point of Eq. 3, i.e., $\lim_{T\to\infty} \frac{\partial F}{\partial W}(W_\infty) = 0$.*

We sketch the proof below (see Sec. 6.7 for detailed proof). Rewriting the first equation in Eq. 6 as: $W_{t+1} = \bar{W}_t + \frac{1}{\alpha}(\ell_{t-1}(W_t) - \ell_t(W_{t+1}))$, we note that if $W_{t+1} \to W_\infty$, then invoking Cesaro mean[3] argument, the corresponding averages do as well: $\frac{1}{T}\sum_{t=1}^T W_{t+1} \overset{T\to\infty}{\longrightarrow} W_\infty$. In turn, we note that the second term in the above expression telescopes, and consequently, it follows that $\frac{1}{T}\sum_{t=1}^T W_{t+1} = \frac{1}{T}\sum_{t=1}^T \bar{W}_t - \frac{1}{T}\nabla\ell_T(W_{T+1})$. Now under smoothness and Lipschitz conditions, we can assume that $\frac{1}{T}\nabla\ell_T(W_{T+1}) \to 0$ in all of its components. As a result, we have $\frac{1}{T}\sum_{t=1}^T \bar{W}_t \to W_\infty$ as well. Plugging these facts into the second equation, we get $\frac{1}{2\alpha T}\sum_{t=1}^T \nabla\ell_t(W_{t+1}) \approx 0$. Now we also know that $W_{t+1} \approx W_\infty$ for sufficiently large T, and using standard arguments it follows that $\frac{1}{2\alpha T}\sum_{t=1}^T \nabla\ell_t(W_\infty)$ also approaches zero. This is the proposed stationarity condition, and our claim follows.

[2]For simplicity in exposition, we concatenate all the losses ℓ_t into a single online stream and get rid of the index N, that gets repeated to provide T iterations of the gradient updates.

[3]https://www.ee.columbia.edu/~vittorio/CesaroMeans.pdf

Computational Complexity. BPTT gradient cost scales as $\Omega(T)$ as seen from Eq. 3. Although FPTT for T times steps leads to $\Omega(T)$ gradient computations, but it is worth noting that the constants involved in taking gradient for the full length T are higher than computing single step gradients. However, FPTT has more arithmetic operations per gradient step, and as such a tradeoff exists. BPTT has higher memory overhead since it stores intermediate hidden states for the full-time horizon T. In contrast, since FPTT only optimizes instantaneous loss functions, it does not require storing hidden states for the full-time horizon. We list computational complexities of different algorithms in Table 1.

FPTT -K. Instead of updating parameters at every timestep, we could perform updates in Eq. 4 only K times for the sequence length T. To do this, for each example, we consider a window of size $\omega = \lfloor \frac{T}{K} \rfloor$, and define a windowed loss, $\bar{\ell}_{t,\omega}(W) = \frac{1}{\omega}\sum_{\tau=t-\omega}^t \ell_t(W)$. We then loop this over K steps instead of T. Setting $K = 1$ is the same as learning RNN through BPTT, while $K = T$ results in the FPTT 2. We provide ablative experiments to study the effect of this parameter on learning efficiency in Section 4.

Table 1. Per-instance computational cost for gradient, parameter update & memory storage overhead. Parameter update involves several arithmetic operations (see Algo. 2), exceeding cost of gradient update by a constant factor. Note that constant associated with gradient computation is a monotonically increasing function $c(\cdot)$ of the sequence length, i.e. $c(1) < c(K) < C(T)$.

Algorithm	Gradient Updates	Parameter Updates	Memory Storage
BPTT	$\Omega(c(T)T)$	$\Omega(1)$	$\Omega(T)$
FTRL	$\Omega(c(T)T^2)$	$\Omega(T)$	$\Omega(T)$
FPTT	$\Omega(c(1)T)$	$\Omega(T)$	$\Omega(1)$
FPTT -K	$\Omega(c(K)T)$	$\Omega(K)$	$\Omega(T/K)$

Intermediate Losses for Terminal Prediction. In our exposition so far, we assumed that we have access to an instantaneous loss ℓ_t at timestep t. While this is true for Seq-to-Seq modelling tasks, for terminal prediction tasks we only get one label y for the entire input sequence x. Let $\hat{P} = \text{softmax}(v^\top h_t)$ be our current estimate of label distribution and Q be our estimate in last training epoch. We use cross-entropy for the classification loss. We construct

intermediate losses ℓ_t for anytime step t as a convex combination of two terms : (a) cross-entropy using the current label distribution $\hat{P}$, and (b) divergence like term to enforce $\hat{P}$ and Q to stay close by. This results in the following loss:

$$\ell_t = \beta \ell_t^{CE} + (1-\beta)\ell_t^{Div}$$
$$\ell_t^{CE} = -\sum_{\bar{y}\in\mathcal{Y}} \mathbf{1}_{\bar{y}=y} \log \hat{P}(\bar{y}); \;\; \ell_t^{Div} = -\sum_{\bar{y}\in\mathcal{Y}} Q(\bar{y}) \log \hat{P}(\bar{y})$$

where $\beta \in [0, 1]$. Our intuition is that for timesteps near T, classification loss is weighted more and less to the divergence term. In the beginning, we should have much less confidence in the classification loss and more on the divergence term. This leads to a natural choice of $\beta = \frac{t}{T}$ achieving the desired effect.

Extensions to Stacked/Hierarchical RNNs. There can be various extensions of our scheme to multi-layered RNNs. One simple scheme is to treat the transition in the stacked RNN as a multi-layered function which transforms hidden states from one time step to the next. Our language modelling experiments (Sec. 4.3) on PTB-word and character level uses this extension for the 3-layered stacked LSTM models. Note that ideally such an extension should work for hierarchical RNNs as well as state transitions can be seen at the most frequent update equation. We leave this as a potential future direction.

4. Experiments

In this section we empirically demonstrate that the proposed algorithm outperforms BPTT. First, we provide ablative experiments to justify our default choice of hyper-parameters and the chosen architecture. Next, we run FPTT on sequence-to-sequence modelling tasks. Finally, we benchmark FPTT on terminal prediction tasks which provide the true label only for the full input sequence.

4.1. Experimental Setup

We implement FPTT in Pytorch using the pseudo code given by Algorithm 2. We perform our experiments on single GTX 1080 Ti GPU. The benchmark datasets used in this study are publicly available along with a train and test split. For hyper-parameter tuning, we set aside a validation set on tasks where a validation set is not available. Wherever applicable we use grid search for tuning hyper-parameters (details in supplementary). In our experiment, we use LSTM(Hochreiter & Schmidhuber, 1997) as the default RNN architecture for evaluation purpose. They have been shown to suffer from vanishing/exploding gradients on many tasks (Zhang et al., 2018; Chang et al., 2019; Kag et al., 2020). Our reasoning follows from the fact that they are widely available with most efficient CUDA implementation on many popular deep learning libraries. This also reduces our experimentation cost. Although we use LSTMs to show that FPTT works on many benchmark datasets, we provide ablative study to show that FPTT works on many RNN architectures (see Sec. 4.2).

Since many RTRL algorithms do not scale to the large-scale benchmark tasks, we do not show their performance in our results. TBPTT has been shown to perform poorly in comparison to BPTT (Trinh et al., 2018). Hence, we will only consider BPTT as the baseline and add the prefix FPTT whenever RNNs are trained with the proposed algorithm, otherwise the training algorithm is assumed to be BPTT. Wherever applicable we will include known results from the literature to compare our BPTT implementation.

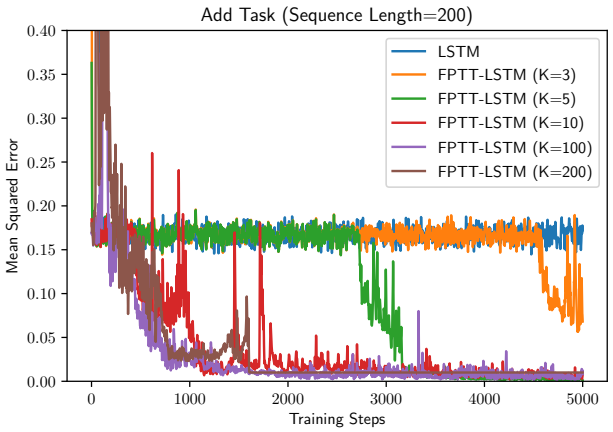


Figure 2. Ablative Experiment: Add Task ($T = 200$) solved by splitting in multiple parts. Note that FPTT with $K = 1$ corresponds to BPTT for LSTM while $K = 200$ updates W_t at every timestep. This figure demonstrates as K increases the performance of the algorithm improves.

4.2. Ablative experiments

Below ablative experiments highlight key aspects of FPTT .

Effect of the parameter K. As described in the Sec. 3 each round of RNN parameter update is more expensive than gradient update. We can address this issue by choosing a suitable K and run FPTT -K. Larger K decreases number of parameter updates, but can impact convergence. For the Add-Task with sequence length $T = 200$, we learn LSTMs with different K values (ranging from $K = 1, 3, 5, 10, 100, 200$). Note that $K = 1$ is essentially BPTT as we only update the RNN parameter after seeing the entire sequence. Figure 2 shows that higher K values result in better convergence. On the other hand higher values of K are more expensive since, as described in Table 1, cost of parameter updates exceeds gradient by a constant factor[4]. Suppose this factor is C^2, the highest computational efficiency is achieved by FPTT -K with $K = \lfloor \frac{T}{C} \rfloor$. On the other hand small values of K could

[4]This factor is difficult to pin-down since gradients leverage CUDA-pytorch, while parameter updates are not optimized.

lead to poor training as observed in Figure 2. As a rule of thumb we use $K \approx \sqrt{T}$ for all our other experiments. This results in meaningful performance (trainability), and computational efficiency matching GPU LSTM implementation.

Table 2. CIFAR-10 : Different RNN architectures.

	Accuracy	**#Params**
LSTM	60.11%	67K
FPTT LSTM	**71.03%**	67K
GRU	66.28%	51K
FPTT GRU	**71.37%**	51K
Antisymmetric	62.41%	37K
FPTT Antisymmetric	**72.13%**	37K

Choice of architecture. In this experiment we show that FPTT provides non-trivial gains for many RNN architectures. We train one layer LSTM, and GRU architectures on the CIFAR-10 dataset with the same setting as the described in section 4.4. Table 2 shows that RNNs trained with FPTT provide gains of about $5-10$ points in accuracy over the RNNs trained with BPTT. In the remaining experiments, we reduce experimentation cost by only performing evaluations on LSTMs as they are readily available in PyTorch with very efficient CUDA implementation. We also want to show that replacing BPTT with FPTT allows LSTMs to achieve performance near state-of-the-art performance achieved by recent architectural improvements.

Auxiliary Losses in BPTT vs FPTT . In this experiment we augment BPTT with the auxiliary losses (proposed for terminal prediction in section 3) similar to (Trinh et al., 2018) in order to isolate the gains from auxiliary losses in the BPTT routine. Table 3 shows that our auxiliary losses helps BPTT to improve the performance but still lack behind the proposed algorithm.

Table 3. CIFAR-10 : BPTT+Auxiliary Loss vs FPTT .

	Accuracy	**#Params**
LSTM	60.11%	67K
Aux-Loss+LSTM	65.65%	67K
FPTT LSTM	**71.03%**	67K

Sensitivity to α hyper-parameter. Note that very small value of α, i.e. $\alpha \to 0$ would lead FPTT to ignore the regularizer and would only optimize the instantaneous loss at every step resulting in diverging iterates. While very high value of α would lead FPTT to only optimize the regularizer and hence very poor generalization performance. We explore the sensitivity to the α hyper-parameter in the Algorithm 2. We use the PTB-300 language modelling dataset. In this experiment we train FPTT on the following α values: $\{1.0, 0.8, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001\}$. Best perplexity is reached at $\alpha = 0.5$ while $\alpha = 1.0$ fails to converge to a good solution. Also, the performance starts to decrease with $\alpha \leq 0.05$. We show the full result in the appendix (see Table 8 in Sec. 6.3).

4.3. Sequence Modelling

We perform experiments on three variants of the sequence-to-sequence benchmark Penn Tree Bank (PTB) dataset (McAuley & Leskovec, 2013). We provide full details of these experiments in the supplementary.

PTB-300 is a word level language modelling task with the difficult sequence length of 300 and has been studied in many previous works to study long range dependencies in language modeling (Zhang et al., 2018; Kusupati et al., 2018; Kag et al., 2020). Table 4 shows the test perplexity for our experimental runs along with results from earlier works. Note that improved architectures such as FastGRNN (Kusupati et al., 2018), SpectralRNNs(Zhang et al., 2018), IncrementalRNNs (Kag et al., 2020) show improvements over the LSTMs trained using backpropagation algorithm. By incorporating FPTT as the training algorithm we improve LSTM's test perplexity by nearly 11 points and thus outperforming the reported LSTM results.

Table 4. Results for PTB word level language modelling : Sequence length (300), 1-Layer LSTM.

Dataset	**PTB-w**	
	Perplexity	**#Params**
FastGRNN (Kusupati et al., 2018)	116.11	53K
IncrementalRNN (Kag et al., 2020)	**115.71**	30K
SpectralRNN (Zhang et al., 2018)	130.20	31K
LSTM (Zhang et al., 2018)	130.21	64K
LSTM (Kusupati et al., 2018)	**117.41**	210K
LSTM	117.09	210K
FPTT LSTM	**106.27**	210K

PTB-w is the traditional word level language modelling variant of the PTB dataset. It uses 70 as sequence length and we follow (Yang et al., 2018) to setup this experiment. We use three-layer LSTM model for this task with embedding dimensions 280 and hidden size 1150. We report the results with dynamic evaluation(Krause et al., 2018) on the trained model. We use the same architecture and training setup to train LSTMs with both BPTT and FPTT . Table 5 demonstrates that LSTM trained with FPTT result in better performance than the ones trained with BPTT.

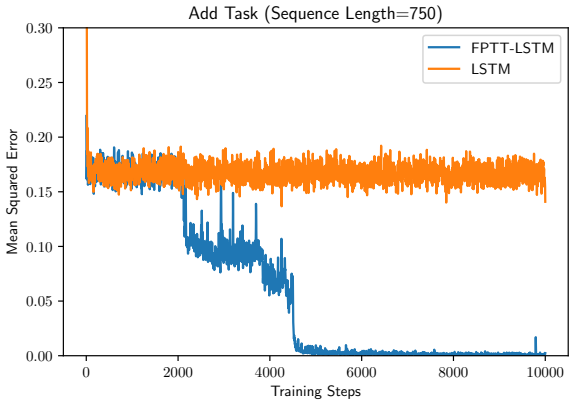
PTB-c is the character level modelling task that uses 150 sequence length. We utilize (Merity et al., 2018) to setup the character level task. We use 3-layer LSTM models as recommended with hidden size 1000 and embedding dimension 200. We train this model with both BPTT and FPTT with the same setting. As shown in Table 5, LSTM trained with FPTT results in better bits-per-characters and has comparable performance with existing state-of-the-art results present in this table.

Table 5. Results for PTB-w and PTB-c datasets. We use AWD-LSTM model in our PTB-c experiments and AWD-LSTM with Mixture-of-Softmaxes(Yang et al., 2018) in the PTB-w experiments. For PTB-w dataset, wherever applicable, all the baselines report the results with dynamiceval(Krause et al., 2018). It can be seen that training with FPTT outperforms the model trained with BPTT.

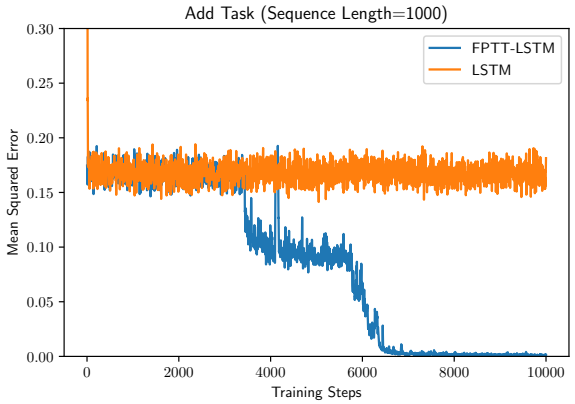
Dataset	**PTB-c**			**PTB-w**		
	Hidden Dimension	**BPC**	**#Params**	**Hidden Dimension**	**Perplexity**	**#Params**
Trellis-Net (Bai et al., 2019)	1000	**1.158**	13.4M	1000	54.19	34M
AWD-LSTM (Merity et al., 2018; Krause et al., 2018)	1000	1.175	13.8M	1150	51.1	24M
Dense IndRNN (Li et al., 2019)	2000	1.18	45.7M	2000	**50.97**	52M
LSTM	1000	1.183	13.8M	1150	51.9	22M
FPTT LSTM	1000	**1.165**	13.8M	1150	**50.96**	22M

4.4. Terminal Prediction

We benchmark FPTT on popular terminal prediction tasks to demonstrate that the proposed algorithm provides non-trivial gains over BPTT in this setting as well. For fair comparison, following previous works(Zhang et al., 2018; Kusupati et al., 2018; Kag et al., 2020), we use LSTMs with 128 dimensional hidden state and Adam as the choice of optimizer with initial learning rate $1e-3$ for both algorithms. We provide other hyper-parameter tuning details in the supplementary (see Sec. 6.1).



(a)



(b)

Figure 3. Results for Add Task with large sequence lengths : (a) $T = 750$, and (b) $T = 1000$.

Add-Task (Hochreiter & Schmidhuber, 1997) has been used to evaluate long range dependencies in RNN architectures. An example data point consists of two sequences (x_1, x_2) of length T and a target label y. x_1 contains real-valued entries drawn uniformly from $[0, 1]$, x_2 is a binary sequence with exactly two 1s, and the label y is the sum of the two entries in sequence x_1 where x_2 has 1s. For both the algorithms (BPTT and FPTT), we use episodic training where a train batch size of 128 is presented to the RNN to update its parameters and evaluated using an independently drawn test set. We use difficult sequence lengths $T = 750$ and $T = 1000$ in this task.

Figure 3 shows the convergence plots for both the algorithms on these two settings. This demonstrates that FPTT helps LSTMs solve this task while BPTT stays around the same loss value throughout the training phase. Note that this observation is consistent with previous works (Kag et al., 2020; Zhang et al., 2018).

Pixel & Permute MNIST, CIFAR-10 are sequential variants of the popular image classification datasets: MNIST (Lecun et al., 1998) and CIFAR-10 (Krizhevsky & Hinton, 2009). MNIST consists of images of 10 digits with shape $28 \times 28 \times 1$, while CIFAR-10 consists of images with shape $32 \times 32 \times 3$. The input images are flattened into a sequence (row-wise). At each time step, 1 and 3 pixels are presented as the input for MNIST and CIFAR datasets respectively. This construction results in Pixel MNIST and CIFAR datasets with 784 and 1024 length sequences respectively. While Permute-MNIST is obtained by applying a fixed permutation on the Pixel MNIST sequence. This creates a harder problem than the Pixel setting since there are no obvious patterns to explore.

Table 6 lists the performance of LSTMs trained using BPTT and FPTT , along with the known results in the literature on these datasets. This shows that LSTMs trained with FPTT outperforms the ones trained with BPTT. We point out that FPTT LSTMs performance is reasonably close to the best performance reported on this dataset with better architectures and higher complexity.

Below we list benefits of the proposed algorithm.

Table 6. Results for Sequential MNIST, Permute MNIST and Sequential CIFAR-10. Models listed below use 1-Layer except IndRNN and TrellisNet as they are multi-layered architectures.

Dataset	Seq-MNIST		Permute-MNIST		CIFAR-10	
	Accuracy	**#Params**	**Accuracy**	**#Params**	**Accuracy**	**#Params**
AntisymmetricRNN (Chang et al., 2019)	98.8%	10K	93.1%	10K	62.20%	37K
IncrementalRNN (Kag et al., 2020)	98.13%	4K	95.62%	8K	-	-
IndRNN (6 Layers) (Li et al., 2018)	99.0%	-	96.0%	-	-	-
TrellisNet (16 Layers) (Bai et al., 2019)	**99.20%**	8M	**98.13%**	8M	**73.42%**	8M
r-LSTM (Trinh et al., 2018)	98.4%	100K	95.2%	100K	72.20%	101K
LSTM (Chang et al., 2019)	97.3%	68K	**92.6%**	68K	**59.70%**	69K
LSTM (Trinh et al., 2018)	**98.3%**	100K	89.4%	100K	58.80%	101K
LSTM (TBPTT-300) (Trinh et al., 2018)	11.3%	100K	88.8%	100K	49.01%	101K
LSTM	97.71%	66K	88.91%	66K	60.11%	67K
FPTT LSTM	**98.67%**	66K	**94.75%**	66K	**71.03%**	67K

(A) Better Generalization. *FPTT provides better generalization on many benchmark tasks compared to BPTT.* Tables 4, 5, and 6 shows that FPTT yields better test performance as compared to training with BPTT. Note that table 2 shows similar gains in architectures other than LSTMs.

(B) Learning Long Term Dependency tasks. *Training with FPTT enables LSTMs to solve LTD tasks.* Our experiments evaluate FPTT on many LTD datasets (Add-Task, Permute/Pixel MNIST, CIFAR-10 and PTB-300). Figure 3 shows that FPTT enables LSTMs to solve Add-Task while BPTT was unable to solve this task. Similarly, tables 4, and 6 shows that FPTT outperforms BPTT on LTD tasks.

(C) Better Model Efficiency. *FPTT trained LSTMs compete with higher complexity models.* Table 6 shows our 1-layer LSTMs are competitive with multi-layered deep RNN higher capacity models (TrellisNet(Bai et al., 2019), IndRNN(Li et al., 2018)). In retrospect it is worth considering that the higher complexity models have often stemmed from inability to train LSTMs on long-term dependency tasks. FPTT points to the fact that the issue is not capacity but the training method.

(D) Learning Short-term dependency tasks. *FPTT shows competitive performance on sequence modelling tasks.* Our experiments on language modelling tasks with shorter sequence lengths (PTB-w, PTB-c datasets) demonstrates the proposed method can learn short term dependencies present in these datasets. Table 5 shows that FPTT provides better test performance than BPTT.

(E) Computational Efficiency/Convergence. We discussed the computational trade-off of the proposed method in table 1. This trade-off allows FPTT to provide training complexity similar to BPTT while providing better statistical trainability and generalization. We show training time comparison in the supplementary (see Sec. 6.2). Additionally, FPTT reduces the memory overhead by only storing hidden states between two iterate updates, in contrast BPTT stores all the intermediate states in the time horizon T.

5. Conclusion

We proposed a novel forward-propagation-through-time (FPTT) method for training RNNs based on sequentially updating parameters forward through time. As such our method at each time t, involves taking a gradient step of an instantaneously constructed regularized risk, where the regularizer evolves dynamically, and is updated based on past history. Our method exhibits light-weight footprint and improves LSTM trainability for benchmark long-term dependency tasks, bypassing vanishing/exploding gradient issues encountered while training LSTMs with BPTT. As a result we show that LSTMs have sufficient capacity, and often realize results that are competitive with much higher capacity models.

Acknowledgements

We would like to thank the reviewers for their insightful comments. This research was supported by National Science Foundation grants CCF-2007350 (VS), CCF-2022446(VS), CCF-1955981 (VS), the Data Science Faculty and Student Fellowship from the Rafik B. Hariri Institute, the Office of Naval Research Grant N0014-18-1-2257 and by a gift from the ARM corporation.

References

Arjovsky, M., Shah, A., and Bengio, Y. Unitary evolution recurrent neural networks. In Balcan, M. F. and Weinberger, K. Q. (eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 1120–1128, New York, New York, USA, 20–22 Jun 2016. PMLR. URL http://proceedings.mlr.

press/v48/arjovsky16.html.

Bai, S., Kolter, J. Z., and Koltun, V. Trellis networks for sequence modeling. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HyeVtoRqtQ.

Bengio, Y., Boulanger-Lewandowski, N., and Pascanu, R. Advances in optimizing recurrent networks. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8624–8628, 2013.

Boyd, S., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3(1):1–122, January 2011. ISSN 1935-8237. doi: 10.1561/2200000016. URL https://doi.org/10.1561/2200000016.

Chang, B., Chen, M., Haber, E., and Chi, E. H. AntisymmetricRNN: A dynamical system view on recurrent neural networks. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryxepo0cFX.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014. doi: 10.3115/v1/D14-1179. URL http://www.aclweb.org/anthology/D14-1179.

Erichson, N. B., Azencot, O., Queiruga, A., Hodgkinson, L., and Mahoney, M. W. Lipschitz recurrent neural networks. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=-N7PBXqOUJZ.

Gu, F., Askari, A., and Ghaoui, L. E. Fenchel lifted networks: A lagrange relaxation of neural network training. In Chiappa, S. and Calandra, R. (eds.), *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pp. 3362–3371. PMLR, 26–28 Aug 2020. URL http://proceedings.mlr.press/v108/gu20a.html.

Hochreiter, S. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91 (1), 1991.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Jing, L., Shen, Y., Dubcek, T., Peurifoy, J., Skirlo, S., LeCun, Y., Tegmark, M., and Soljačić, M. Tunable efficient unitary neural networks (EUNN) and their application to RNNs. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1733–1741. PMLR, 06–11 Aug 2017. URL http://proceedings.mlr.press/v70/jing17a.html.

Kag, A. and Saligrama, V. Time adaptive recurrent neural network, 2021. URL https://openreview.net/forum?id=VDUovuK0gV.

Kag, A., Zhang, Z., and Saligrama, V. Rnns incrementally evolving on an equilibrium manifold: A panacea for vanishing and exploding gradients? In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HylpqA4FwS.

Kerg, G., Goyette, K., Puelma Touzel, M., Gidel, G., Vorontsov, E., Bengio, Y., and Lajoie, G. Non-normal recurrent neural network (nnrnn): learning long time dependencies while improving expressivity with transient dynamics. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/9d7099d87947faa8d07a272dd6954b80-Paper.pdf.

Krause, B., Kahembwe, E., Murray, I., and Renals, S. Dynamic evaluation of neural sequence models. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2766–2775. PMLR, 10–15 Jul 2018. URL http://proceedings.mlr.press/v80/krause18a.html.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

Kusupati, A., Singh, M., Bhatia, K., Kumar, A., Jain, P., and Varma, M. Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/ab013ca67cf2d50796b0c11d1b8bc95d-Paper.pdf.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pp. 2278–2324, 1998.

Lezcano-Casado, M. and Martínez-Rubio, D. Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 3794–3803. PMLR, 09–15 Jun 2019. URL http://proceedings.mlr.press/v97/lezcano-casado19a.html.

Li, S., Li, W., Cook, C., Zhu, C., and Gao, Y. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5457–5466, 2018. doi: 10.1109/CVPR.2018.00572.

Li, S., Li, W., Cook, C., Gao, Y., and Zhu, C. Deep independently recurrent neural network (indrnn), 2019.

Linsley, D., Karkada Ashok, A., Govindarajan, L. N., Liu, R., and Serre, T. Stable and expressive recurrent vision models. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 10456–10467. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/766d856ef1a6b02f93d894415e6bfa0e-Paper.pdf.

McAuley, J. and Leskovec, J. Hidden factors and hidden topics: Understanding rating dimensions with review text. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pp. 165–172, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2409-0. doi: 10.1145/2507157.2507163. URL http://doi.acm.org/10.1145/2507157.2507163.

McMahan, H. B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., Nie, L., Phillips, T., Davydov, E., Golovin, D., Chikkerur, S., Liu, D., Wattenberg, M., Hrafnkelsson, A. M., Boulos, T., and Kubica, J. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2013.

Menick, J., Elsen, E., Evci, U., Osindero, S., Simonyan, K., and Graves, A. Practical real time recurrent learning with a sparse approximation. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=q3KSThy2GwB.

Merity, S., Keskar, N. S., and Socher, R. Regularizing and optimizing LSTM language models. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=SyyGPP0TZ.

Mhammedi, Z., Hellicar, A., Rahman, A., and Bailey, J. Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 2401–2409. PMLR, 06–11 Aug 2017. URL http://proceedings.mlr.press/v70/mhammedi17a.html.

Miller, J. and Hardt, M. Stable recurrent models. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Hygxb2CqKm.

Mujika, A., Meier, F., and Steger, A. Approximating real-time recurrent learning with random kronecker factors. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 31, pp. 6594–6603. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/dba132f6ab6a3e3d17a8d59e82105f4c-Paper.pdf.

Ollivier, Y. and Charpiat, G. Training recurrent networks online without backtracking. *CoRR*, abs/1507.07680, 2015. URL http://arxiv.org/abs/1507.07680.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. *Learning Internal Representations by Error Propagation*, pp. 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 026268053X.

Tallec, C. and Ollivier, Y. Unbiased online recurrent optimization. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJQDjk-0b.

Trinh, T., Dai, A., Luong, T., and Le, Q. Learning longer-term dependencies in RNNs with auxiliary losses. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4965–4974. PMLR, 10–15 Jul 2018. URL http://proceedings.mlr.press/v80/trinh18a.html.

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

Williams, R. J. and Peng, J. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Comput.*, 2(4):490–501, December 1990. ISSN 0899-7667. doi: 10.1162/neco.1990.2.4.490. URL https://doi.org/10.1162/neco.1990.2.4.490.

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

Yang, Z., Dai, Z., Salakhutdinov, R., and Cohen, W. W. Breaking the softmax bottleneck: A high-rank RNN language model. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=HkwZSG-CZ.

Zhang, J., Lei, Q., and Dhillon, I. Stabilizing gradients for deep neural networks via efficient SVD parameterization. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5806–5814. PMLR, 10–15 Jul 2018. URL http://proceedings.mlr.press/v80/zhang18g.html.