

Online Real-Time Recurrent Learning Using Sparse Connections and Selective Learning

Khurram Javed¹
Haseeb Shah¹
Richard Sutton^{1, 2}
Martha White¹

KJAVED@UALBERTA.CA
HSHAH1@UALBERTA.CA
RSUTTON@UALBERTA.CA
WHITEM@UALBERTA.CA

¹ *University of Alberta, Edmonton, Canada*

² *DeepMind, Edmonton, Canada*

Abstract

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

Keywords: Scalable recurrent learning, Online learning, RTRL, Cascade correlation networks, Agent-state construction

1. Introduction

Learning by interacting with the world is a powerful framework for building systems that can autonomously achieve goals in complex worlds. A key ingredient for building autonomous systems is agent-state construction—learning a compact representation of the history of interactions that helps in predicting and controlling the future. One solution for state construction is to use differentiable recurrent neural networks (RNNs) learned to minimize prediction errors (Kapturovski *et al.* 2018 and Vinyals *et al.* 2019).

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

We define online state construction as the ability of the system to learn the agent-state in real-time while interacting with the world. Both BPTT and RTRL are not suitable for online state construction. BPTT requires storing all past activations and does sequential computation proportional to the length of the data-stream seen so far for estimating the gradient. As a result, it is a poor choice for online state construction. RTRL, on the other hand, can estimate the gradient on the go, and does not require more computation per-step for longer sequences. However, RTRL scales poorly with an increase in the number of parameters of the RNN. Both BPTT and RTRL can be approximated, to make them more suitable for online learning.

A promising direction to scale gradient-based credit assignment to large networks is to approximate the gradient. Elman (1990) proposed to ignore the influence of parameters on future predictions entirely for training RNNs. This resulted in a computationally cheap but biased algorithm. Williams and Peng (1990) proposed a more general algorithm called Truncated-BPTT (T-BPTT). T-BPTT tracks the influence of all parameters on predictions made up to k steps in the future, where k is a hyperparameter. T-BPTT limits the BPTT computation to the last k steps and assumes the gradient to be zero beyond k steps. It works well for a range of problems (Mikolov *et al.*, 2009, 2010; Sutskever, 2013 and Kapturowski *et al.*, 2018). The main limitation of T-BPTT is that the resultant gradient is blind to long-range dependencies. Mujika *et al.* (2018) showed that on a simple copy task, T-BPTT failed to learn dependencies beyond the truncation window. Tallec *et al.* (2017) demonstrated T-BPTT can even diverge when a parameter has a negative long-term effect on a target and a positive short-term effect.

Hochreiter and Schmidhuber (1997) used a diagonal approximation to RTRL that scales linearly with the number of parameters. The same approximation is also a special case of the SnAp- k algorithm recently proposed by Menick *et al.* (2021), when $k = 1$. Diagonal-RTRL is not blind to all long-term dependencies, but introduces significant bias in the gradient estimate for dense recurrent networks. It assumes that changing a recurrent feature will not change the values of other features, an assumption that does not hold in densely connected recurrent networks. Another algorithm to make RTRL scalable is UORO. It computes unbiased samples of the gradient, instead of the actual gradient (Tallec *et al.* 2017). However, the resulting samples can be highly noisy and are only effective for learning with very small step-sizes. Menick *et al.* 2021 showed that UORO works poorly even on simple benchmarks.

Existing methods for scalable gradient-based recurrent learning approximate the gradient but do not make assumptions about the function class of the recurrent network. In this work, we propose a different strategy: instead of introducing bias or noise in the gradient estimate, we limit the function class of the RNNs to enable scalable, unbiased, and noise-free gradient estimation. We introduce three methods: Columnar, Constructive, and Constructive-Columnar Networks. The Columnar networks constraints the recurrent network such that the gradient of a recurrent feature w.r.t other recurrent features is zero. The Constructive networks learns recurrent features one by one, instead of simultaneously, and constraints the network such that a feature learned later cannot influence features learned earlier. Finally, the Constructive-Columnar networks (CCNs) combines both the Columnar and the Constructive approach by grouping features together and learning the groups one by one.

2. Problem Formulation

We formulate the goal of a learner as predicting the discounted sum of a cumulant from an online stream of experience. The agent sees a feature vector $\mathbf{x}_t \in \mathcal{R}^n$ at time step t and predicts the discounted sum of the future value of a cumulant c_t , where c_t is a fixed index of the vector \mathbf{x}_t . The goal of the agent is to minimize the sum of squared error between the prediction and the empirical return incurred over time, *i.e.*, the agent aims to minimize:

$$L(k, T) = \frac{1}{T} \sum_{t=k}^{T+k} (y_t - \sum_{j=t+1}^T \gamma^{j-t} c_j)^2 \quad (1)$$

where T is the horizon over which the prediction error is accumulated, and y_t is the prediction made at time step t . Note that the error is measured w.r.t the predictions made over time and not using a final set of weights. The first prediction, y_1 , would be inaccurate because the network has just started learning. Over time, we can expect the predictions to improve. If T is large enough, the prediction performance at convergence dictates which algorithm is better.

Our problem formulation can capture various online temporal-prediction and supervised-learning problems. For example, setting the cumulant to be the reward turns our problem formulation into policy evaluation (Sutton & Barto 2018). Similarly, by setting $\gamma = 0$, the problem formulation can represent online supervised recurrent learning benchmarks.

2.1 Learning under Resource Constraints

We focus on the under-parameterized setting with limited compute and memory.

The learners have a fixed compute and memory budget per-step, that they can allocate however they choose. For instance, a learner can choose to spend the budget on an expensive learning algorithm, such as RTRL, and satisfy the compute constraint by using a smaller recurrent network. Alternatively, it can choose to use a larger recurrent network, and learn the network using a cheaper learning algorithm, such as T-BPTT with a small truncation window.

The focus on the under-parameterized setting, limited resources, and per-step learning emphasizes developing learning algorithms that are cheap and can be applied continually. Moreover, since the real-world is significantly more complex compared to even the largest recurrent networks, the under-parameterized setting is arguably a better proxy for understanding how different algorithms will behave on real-world problems.

2.2 Learning with Recurrent Architectures

In our temporal prediction setting it is natural to assume that the learner will face a partially observable environment. The learner will want to leverage histories to improve prediction accuracy. Throughout this work, we assume that learners attempt to summarize this history by learning recurrent neural networks (RNNs).

The dynamics of an RNN can be written as

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t, \theta) \quad (2)$$

where $\mathbf{h}_t \in \mathbb{R}^d$ is the hidden state of the network at time t , $\mathbf{x}_t \in \mathbb{R}^n$ is the feature vector seen by the learner, f is the dynamics function of the recurrent network, and θ are the parameters of the RNN. The hidden state \mathbf{h}_t of the recurrent network is linearly weighted with weights $\mathbf{w}_t \in \mathbb{R}^d$ to make a prediction y_t as:

$$y_t = \sum_{k=0}^{d-1} h_{t,k} w_{t,k} \tag{3}$$

where $h_{t,k}$ and $w_{t,k}$ are the k th element of vectors \mathbf{h}_t and \mathbf{w}_t , respectively.

To update the parameters θ_t at time t , we need to be able to compute the gradient of this prediction with respect to θ . Using the chain rule, we have

$$\frac{\partial y_t}{\partial \theta} = \frac{\partial y_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \theta}. \tag{4}$$

The key question is how to compute $\frac{\partial \mathbf{h}_t}{\partial \theta}$. We can obtain a recursive formula for this expression, which is used by RTRL and by the algorithms we introduce in this work. To make it clear how we can use the multivariable chain rule, let us explicitly write $\mathbf{h}_t(\theta) = f(\mathbf{h}_{t-1}(\theta), \mathbf{x}_t, \mathbf{g}_t(\theta))$ where $\mathbf{g}_t(\theta) \doteq \theta$. Then the multivariable chain rule gives us

$$\frac{\partial \mathbf{h}_t}{\partial \theta} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{g}_t} \frac{\partial \mathbf{g}_t}{\partial \theta} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \theta}, \tag{5}$$

where the first term corresponds to the gradient of the RNN given \mathbf{h}_{t-1} and inputs \mathbf{x}_t and the second term corresponds to the indirect impact of θ on $\frac{\partial \mathbf{h}_t}{\partial \theta}$ due to its impact on $\frac{\partial \mathbf{h}_{t-1}}{\partial \theta}$.

This recursive relationship is exploited by two algorithms: BPTT and RTRL. BPTT stores the network activations and inputs from prior steps and uses the expansion

$$\frac{\partial y_t}{\partial \theta} = \frac{\partial y_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{g}_t} \frac{\partial \mathbf{g}_t}{\partial \theta} + \frac{\partial y_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \theta} \tag{6}$$

to compute the gradient. It unrolls the formula back in time, computing and accumulating gradient until the start of the recursion at $t = 0$. RTRL, on the other hand, maintains the Jacobian $\frac{\partial \mathbf{h}_t}{\partial \theta}$ using equation 5 at every step. It starts at time $t = 0$ and at each timestep adds the new term to update the Jacobian. To get the gradient w.r.t the prediction, it plugs in the computed Jacobian in equation 4.

These two algorithms both compute the same gradient, but make different compromises in terms of computation and memory. RTRL does not require storing past activations and inputs, as it can update the Jacobian using only the most recent input. However, computing the Jacobian using equation 5 requires $O(j\mathbf{h}^2j\theta j)$ operations and $O(j\mathbf{h}j\theta j)$ memory. The size of the parameters $j\theta j$ in a fully connected RNN is $j\mathbf{h}^2$. RTRL is therefore often said to have quartic complexity in terms of the size of the hidden state, and scales poorly to large networks. BPTT, on the other hand, requires $O(j\theta j t)$ memory and compute, where t is the length of the sequence. It avoids the bigger memory cost by computing the product $\frac{\partial y_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{g}_t} \frac{\partial \mathbf{g}_t}{\partial \theta}$ directly, rather than separately computing the Jacobian and then multiplying by $\frac{\partial y_t}{\partial \mathbf{h}_t}$. For sequences shorter than $j\mathbf{h}^2$, BPTT is cheaper than RTRL for fully connected RNNs.

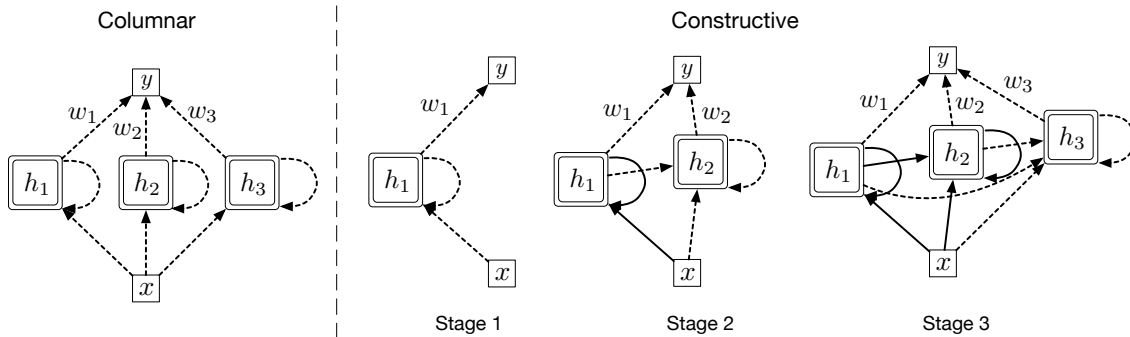


Figure 1: Two structures of recurrent neural networks for which gradients can be estimated in a scalable way without significant bias or noise. Dotted lines represent parameters that are updated at every step, whereas solid lines are weights that are fixed. Once the weight is fixed, the feature is fixed and gradient estimates do not need to propagate past the feature. This allows us to compute unbiased gradients, without having to backpropagate far back in time. Recurrent networks with a columnar structure can be trained end-to-end using gradients without any truncation, only requiring $O(n)$ operations and memory per step. However, columnar networks do not have hierarchical recurrent features—recurrent features made out of other recurrent features. Constructive networks have hierarchical recurrent features; however, they must be trained incrementally to avoid bias in the gradient estimate. Incremental learning is achieved by initializing all w_i to zero, and learning h_1 , h_2 , and h_3 in three stages. In the second and third stages, parameters represented by solid lines are fixed

Both approaches are generally too expensive, and typically approximations are used. One such approximation is truncated-BPTT which only computes the gradients back in time for a fixed number of steps that might be shorter than t . Approximations to RTRL involve approximating the Jacobian, typically with low-rank approximations (Tallec *et al.* 2017 and Menick *et al.* 2021). In this work, we explore an alternative approach: instead of approximating the gradients, we modify the recurrent architectures that we consider.

3. Constructive Columnar Networks

In this section we develop a new approach to recurrent learning, called Constructive Columnar Networks (CCNs). CCNs leverage two key ideas: First, RTRL is computationally efficient for modular recurrent networks where each module has a scalar hidden state; we call these modular networks Columnar Networks. Second, RTRL is computationally efficient if the recurrent units of a recurrent networks are learned incrementally, as opposed to learning them all simultaneously. We call the the incremental learning approach Constructive Networks. Figure 1 visualizes the central ideas behind Columnar and Constructive Networks

Both Columnar and Constructive networks, on their own, show promising results but also have limitations. Columnar Networks perform poorly on difficult tasks, and Constructive Networks learns slowly. We show that their weaknesses can be largely overcome by combining the two ideas to create a third learning system that we call the Constructive Columnar Networks (CCNs).

3.1 Columnar Networks

Columnar networks organize the recurrent network such that each scalar recurrent feature is independent of other recurrent features. Let $h_{t;k}$ be the k th index of the state vector \mathbf{h}_t . Then, in columnar networks,

$$h_{t;k} = f_k(h_{t-1;k}, \mathbf{x}_t, \theta_{t;k}). \tag{7}$$

Each f_k outputs a scalar recurrent feature and is called a column.¹ $\theta_{t;k}$ is the set of parameters of the k th column. For any $i \neq j$, the set $\theta_{t;i}$ and $\theta_{t;j}$ are disjoint. A columnar network consists of d columns. The output of all columns are concatenated to get the d -dimensional hidden-state vector \mathbf{h}_t . Figure 1 (left) shows a graphical representation of a Columnar network. Note that changing h_1 has no influence on the value of h_2 or h_3 .

Because recurrent features in a columnar network are independent of each other, we can apply RTRL to each of them individually. To better understand why, let us rederive our recursive formula for the gradient. For θ_k the parameters for the k th column, we have

$$\frac{\partial y_t}{\partial \theta_k} = \frac{\partial y_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \theta_k} = \sum_{j=1}^d \frac{\partial y_t}{\partial h_{t;j}} \frac{\partial h_{t;j}}{\partial \theta_k} = \frac{\partial y_t}{\partial h_{t;k}} \frac{\partial h_{t;k}}{\partial \theta_k}$$

where most of the terms in the sum are zero because θ_k does not influence them. Therefore, we only have to compute $\frac{\partial h_{t;k}}{\partial \theta_k}$ with RTRL. Like before, we can write this recursively using $h_{t;k}(\theta_k) = f(h_{t-1;k}(\theta_k), \mathbf{x}_t, \mathbf{g}_t(\theta_k))$ where $\mathbf{g}_t(\theta_k) \doteq \theta_k$, giving

$$\frac{\partial h_{t;k}}{\partial \theta_k} = \frac{\partial h_{t;k}}{\partial \mathbf{g}_t} \frac{\partial \mathbf{g}_t}{\partial \theta_k} + \frac{\partial h_{t;k}}{\partial h_{t-1;k}} \frac{\partial h_{t-1;k}}{\partial \theta_k} \tag{8}$$

Computing and storing this Jacobian only costs $O(j\theta_{t;k})$ for each column k because $j\theta_{t;i} = 1$ for a single column. The cost for all the columns is

$$O(j\theta_{t,1}) + O(j\theta_{t,2}) + \dots + O(j\theta_{t,n}) = O(j\theta_t). \tag{9}$$

Therefore, RTRL for Columnar Networks scales linearly in the size of the parameters. In this work, we implement each column as a single LSTM cell with a hidden size of one. We provide the explicit gradients in Appendix 6.

3.2 Constructive Networks

In constructive networks, we learn the recurrent network one feature at a time. A feature that is learned later can take as input features that have already been learned. However, the

1. This terminology comes from the connection to structure observed in brains (Mountcastle 1957).

opposite is not allowed—feature learned earlier cannot take as input a feature that would be learned later. We elucidate the multi-stage learning process in a three-feature constructive network using Figure 1 (right). Dotted lines represent parameters that are being updated at every step, whereas solid lines represent weights that are fixed.

In the first stage, the learner learns the incoming weights of h_1 , which is connected to the input features x , but not to h_2 or h_3 . Note that we are omitting the time index for brevity, and h_1 is the same as $h_{t,1}$. Once the incoming and the recurrent weights of h_1 are learned, the learner freezes those weights and goes to stage 2. In stage 2, it learns the incoming weights of h_2 . h_2 can use both x and h_1 as the inputs. The outgoing weight of h_1 — w_1 —is not fixed and continues to be updated. Similarly in the 3rd stage, both h_1 and h_2 are fixed and fed to h_3 as input features. In each stage, the newly introduced feature can be connected to all prior features.

In this staged approach, the learner never learns more than one feature at a time. As a result, the effective size of the hidden state of the learning system is just one, and RTRL can be applied cheaply. In fact, since only a small subset of the network is being learned at any given time, constructive networks use even less per-step computation than columnar networks. Constructive networks introduce one additional hyperparameter—steps-per-stage—that controls the number of steps after which the learner moves from one stage to the next.

This constructive approach is similar to and inspired by related work on cascade correlation for recurrent neural networks (Fahlman, 1990). The biggest differences are that (1) in the cascade correlation work, new recurrent units are trained to maximize correlation with the error whereas we use the gradient w.r.t the prediction error to learn the incoming weights of the new units and (2) the cascade correlation work learns on a batch of data, whereas our network learns from an online stream of data. The two differences are arguably minor. Rather, the bigger departure from this older idea is when we combine it with columnar networks in the next section. This lets us move beyond adding only one recurrent feature in each stage, to learning multiple recurrent features (columns) in each stage.

3.3 Constructive Columnar Networks

Constructive columnar networks (CCNs), as the name suggests, are a combination of columnar and constructive networks. In CCNs, we keep the multi-stage approach of the constructive network; however, instead of learning a single feature in every stage, the learner can learn multiple features that are independent of each other. It leverages the fact that RTRL is efficient for columnar structures to grow the network more quickly.

A two-stage CCN is shown in Figure 2. In stage one, the learner learns the incoming weights of h_1 and h_2 . Note that since h_1 and h_2 are independent of each other, they are equivalent to a columnar network with two features, and can be learned cheaply together. In the second stage, the learner freezes the incoming and recurrent weights of h_1 and h_2 , and learns the incoming weights of h_3 and h_4 , which take as input both h_1 and h_2 . Once again, h_3 and h_4 are independent of each other.

CCNs have two hyperparameters—features-per-stage, and steps-per-stage. Features-per-stage controls how many features are learned in each stage.

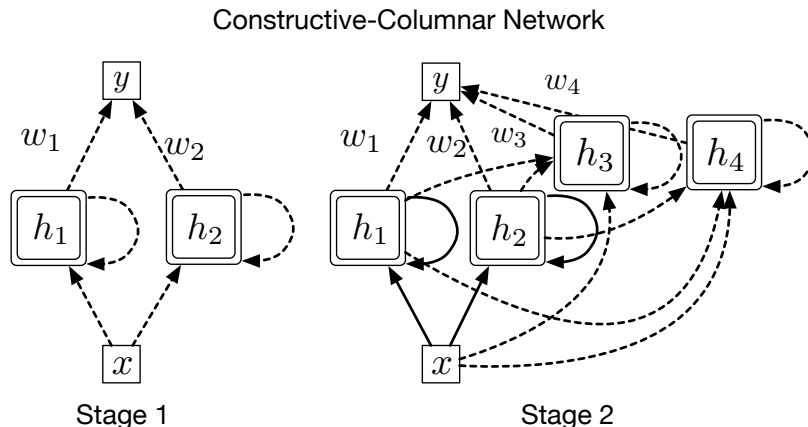


Figure 2: Constructive-Columnar networks (CCNs) combines the ideas from both the columnar and the constructive approach. In each stage, the learner learns multiple features that are independent of each other, just like a columnar network. Across stages, the learner can learn hierarchical feature, similar to the constructive approach.

3.4 Feature Normalization

A key to making our system work is online feature normalization. Unlike dense recurrent networks, features in our constructive and CCN networks can have varying number of incoming weights. This discrepancy can change the scale of each feature, making learning difficult using a uniform step-size. Our feature normalization is similar to an online version of batch normalization (Ioffe and Szegedy 2015), and prior work has shown feature normalization to be helpful for recurrent networks in the batch setting (Cooijmans *et al.* 2017).

Our specific feature normalization approach is as follows. We estimate the mean and variance of each feature online, and normalize the features to have zero mean and unit variance. Additionally, if the variance of a feature goes below a threshold, we set it to a small number ϵ to prevent the normalized feature from getting too large, where ϵ is a hyperparameter. Capping the maximum value of the feature is important to prevent unstable behavior. The formula for the normalized feature \hat{f}_i , given the unnormalized feature f_i , is

$$\hat{f}_{t,i} = \frac{f_{t,i} - \mu_{t,i}}{\max(\epsilon, \sigma_{t,i})} \tag{10}$$

$$\text{where } \mu_{t,i} = \mu_{t-1,i}\beta + (1 - \beta)f_{t,i}$$

$$\sigma_{t,i}^2 = \sigma_{t-1,i}^2\beta + (1 - \beta)(\mu_{t,i} - f_{t,i})^2$$

where $\beta = 0.99999$ for all our experiments. $\mu_{0,i}$ and $\sigma_{0,i}^2$ are initialized to be 0 and 1 respectively. ϵ is tuned; the values used in this work are shown in Table 1 in Appendix 6.

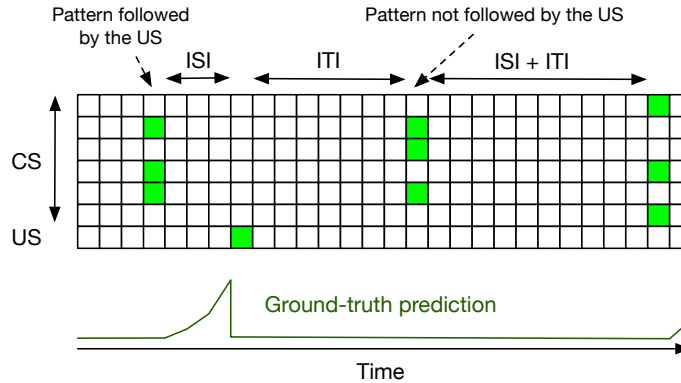


Figure 3: Visualization of the stream of experience for the trace patterning task. At each step, the learner sees a vector with seven values. The first six are the CS, and the last is the US. CS is either a vector of zeros, or three of them are one. Twenty possible patterns can be represented by the CS. Ten of these patterns activate the US after ISI number of steps, whereas others do not. The learner has to predict the discounted sum of the value of US in the future. The CS is present every ISI + ITI number of steps. The bottom part of the figure shows the ground-truth prediction for the task.

4. Experiments on an Animal Learning Benchmark

We start by evaluating the methods on a recently proposed benchmark inspired by animal learning (Rafiee *et al.*, 2022). The trace patterning task is an online prediction task that requires the learner to discriminate between patterns—conditional stimuli (CS)—that are followed by a scalar—unconditional stimuli (US)—after a time delay. The goal is to predict the discounted sum of the US. Correct predictions require the ability to discriminate between patterns that lead to US from those that do not. The time delay between the CS and US requires remembering information from the past. The delay between the CS and US is uniformly randomly sampled to be between 14 and 26 steps in our experiments and is called the inter-stimulus interval (ISI). The delay between the US and next CS is uniformly randomly sampled to be between 80 and 120 steps and is called the inter-trial interval (ITI). The CS consists of 6 features. When CS is present, three of the six features in the CS vector are one. Since $\binom{6}{3}$ is twenty, the CS vector can represent twenty different patterns. Ten randomly chosen patterns are followed by US=1 after ISI steps, whereas the remaining ten do not activate the US. The learner has to learn to discriminate between the patterns that lead to the US from those that do not.

A visual representation of experience from the trace patterning benchmark with ISI of 3 and ITI of 7 is shown in Figure 3. The vertical dimension represents the features, and the horizontal the time. At time-step 4, 3 of the 6 features are one. After 3 more (ISI = 3) steps, the US becomes active. Then no features are active for ITI number of steps. After ITI steps, the CS again becomes active. The second pattern of the CS is not followed by

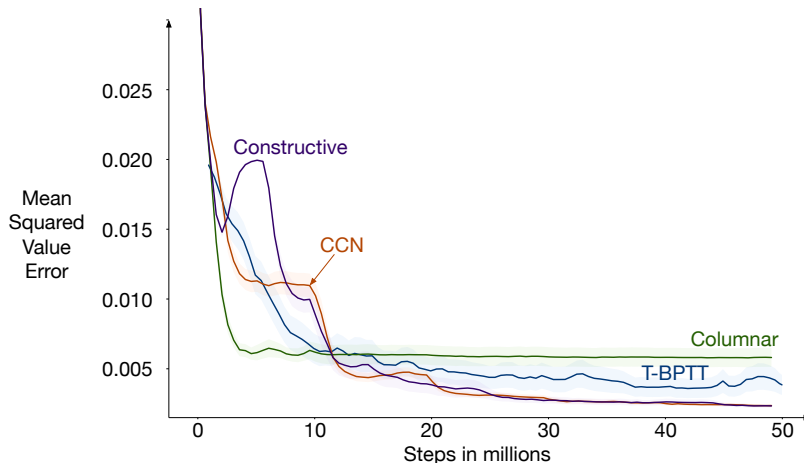


Figure 4: Results of the proposed algorithms, and the best performing T-BPTT on the trace patterning task for 50 million learning steps. All methods can learn to make accurate predictions. Columnar learns quickly, but converges to a worse solution because it is unable to build hierarchical representations. Both CCN and constructive converge to the almost optimal solution. The best T-BPTT performs worse than both the constructive and CCN. All plots are averaged over 30 seeds, and the shaded area is \pm standard error.

US. At the bottom of Figure 3, we show the ground truth return that the learner has to make to minimize the prediction error.

4.1 Experimental Setup

We compare CCNs to T-BPTT, Columnar networks and Constructive Networks. All networks use the LSTM cell architecture (Hochreiter and Schmidhuber 1997) for recurrence. For T-BPTT, we use a fully connected LSTM network. T-BPTT introduces another hyperparameter—the truncation window k . To keep the per-step computation constant, a learner using a larger truncation window has to use fewer features.

We set the per-step compute budget to 4,000 floating point operations and use TD(λ) (Sutton 1984, 1988) for learning for all methods. We use $\lambda = 0.99$, and $\gamma = 0.90$ and report the learning curves for 50 million steps. For each method, we individually tune the step-size, ϵ , steps-per-stage, features-per-stage, and the truncation window; we report the results for the best performing configuration. Details of hyperparameter tuning are in Appendix 6. The columnar networks, constructive networks, and CCNs have 5, 10, and 20 features respectively. T-BPTT uses a truncation window of 30, and has two features. Note that even though T-BPTT only has two features, it uses the same amount of per-step computation as CCNs because it uses a more expensive learning algorithm.

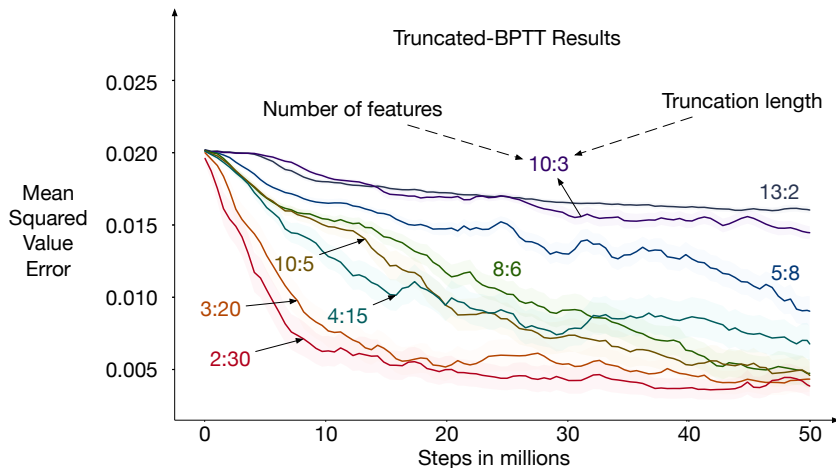


Figure 5: Different combinations of T-BPTT on the trace patterning task. Each curve is denoted by two numbers a:b. The first number indicates the number of features in the learner, and the second number indicates the truncation length of T-BPTT. For example, 2:30 means an LSTM with two features trained with T-BPTT with a truncation window of 30. All lines use about the same amount of computation for learning and prediction. By choosing a small value of truncation, the learner can afford to have more features. We see that different values of truncation results in very different performance. Large networks trained with small truncation lengths—13:2 and 10:3—perform the worst showing the impact of the bias introduced by T-BPTT. All lines are averaged over 30 random seeds.

4.2 Results

We start by looking at the learning curves for all four methods in Figure 4. All three approaches learn to reduce the prediction error over time. Columnar networks perform the worst, demonstrating the need for hierarchical recurrent features. Both CCNs and constructive networks reliably converge to a good solution. There is a clear structure in their learning curves, where there are plateaus followed by sudden steep declines in error when new features are added, which occurs every 5 million steps. T-BPTT achieves error in-between columnar networks and CCNs.

We further investigate the sensitivity of T-BPTT to the value of truncation. We first consider the impact of reallocating resources, allowing T-BPTT to have bigger networks with shorter truncation length k . The results above used T-BPTT with two features and $k = 30$. To maintain the same level of computation and increase the number of features, k has to be correspondingly decreased. We can see from the learning curves in Figure 5 that when the truncation length is much smaller than the longest dependency in the learning problem—26—the performance drops significantly. T-BPTT performs the best when it selects a smaller network (two features) and longer truncation ($k = 30$).

We conducted an additional experiment where we allowed T-BPTT to use more computation. We ignore our per-step resource constraint and fix the number of features to 10.

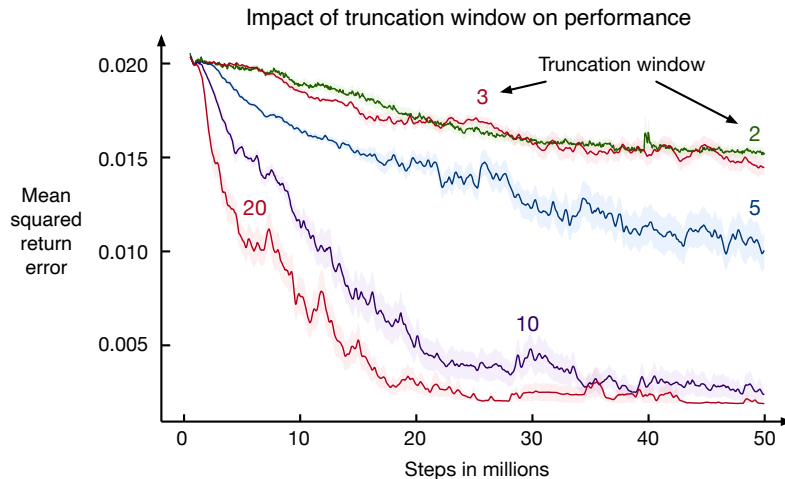


Figure 6: We train LSTMs with 10 hidden units using truncation windows of 2, 3, 5, 10, and 20. For each truncation window, we independently tuned the step-size parameter. We see that as the truncation window increases, the performance improves significantly at the expense of more computation: using a truncation window of 20 is ten times more computationally expensive as compared to a truncation window of two. The sensitivity of performance w.r.t truncation window highlights the degradation in performance due to the bias introduced by truncation. All lines are averaged over 30 random seeds with shaded regions corresponding to two standard error.

We then use T-BPTT with different truncation windows and report the results in Figure 6. We see that large networks with large truncation window—red line—performs almost as well as CCNs. However, it uses around six times more computation per-step.

5. Experiments in the Arcade Learning Environment

To evaluate the proposed algorithms on higher-dimensional image-based inputs, we introduce a new prediction learning benchmark based on the Arcade Learning Environment (Bellemare 2013). We first describe this benchmark, and then conduct similar experiments to above, now on these 50 games.

5.1 An Atari Prediction Benchmark for Recurrent Learning

Since our goal is to study state construction in the prediction setting, we do policy evaluation on expert Atari agents, as opposed to solving the control task. We use pre-trained Rainbow-DQN (Hessel et al 2018) agents from the model zoo of Chainer-RL (Fujita *et al.* 2021), and collect at least 200k samples following the greedy policy for each Atari environment. After 200k samples are collected for an environment, we keep collecting samples until the episode terminates. We clip the rewards to be in the range $(-1, +1)$.

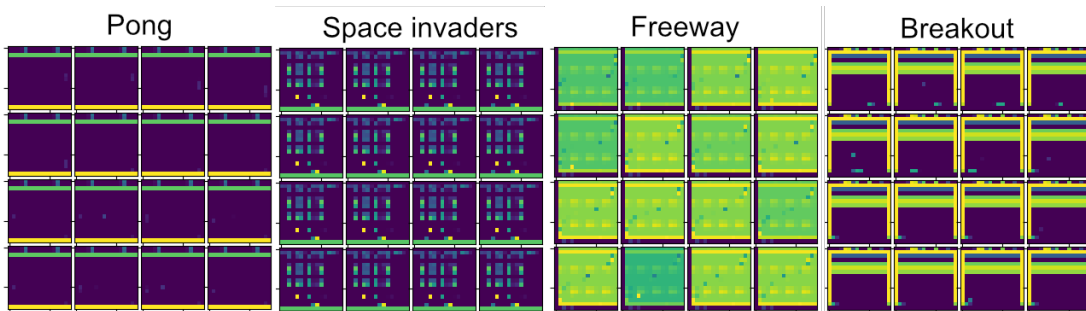


Figure 7: Environments down-scaled to 16 x 16. Looking at a single frame, it’s hard to figure out information about the environment. For instance in Pong, the ball is often not visible in a single frame. However, looking at the sequence of frames, we can tell the position and the direction of the ball. This partial observability due to down-scaling makes 16 x 16 Atari an interesting benchmark for studying state construction.

The input \mathbf{x} given to the learner is composed of the observation, action and reward from the previous step. The observation at each step of a Rainbow-DQN agent is 84 84 4, where each observation stacks the previous four frames to reduce partial observability. However, since our goal is to study how well our algorithms can construct agent-state, we only pass the single most-recent frame to the prediction learner. Additionally, we downscale the frames to be 16 16, resulting in 256 features. Downscaling and removing frame-stacking makes the problem much more challenging, and the learner has to look at the trajectory of observations to predict well. We visualize these downscaled observations in various games in Figure 7. We see that due to downscaling, it is not enough to look at a single frame to make an accurate prediction. For instance in the Pong environment, the ball or paddles are not visible in many frames. The only way to make accurate predictions on all frames is to remember information from the past. The expert Atari agent can take one of 20 actions. We one-hot encode the action and append it to the observation, giving 276 features at every step.

This dataset can generally be used to evaluate recurrent learning algorithms. We use it to simulate an online learning setting. If there are less than 200k learning steps, then the learner can simply iterate over the dataset in order, if they are seeing the data streaming in real-time. Otherwise, when learning for more than 200k steps, the dataset can be used like a simulator. After looping through all the episodes in the dataset, the order of episodes can be shuffled and the agent can do another epoch over the training set. We learn for 50 million steps, meaning we approximately loop over the entire dataset 500 times.

Traditionally, learning performance on a dataset is evaluated on a held-out test set to measure the generalization performance of the system. While the train and test distinction is important in offline learning, especially when the learning network is over-parameterized, it is unnecessary when the learner is an order of magnitude smaller than the dataset, as is the case in our experiments. A small learner that generalizes better can outperform a small learner that tries to memorize trajectories on a dataset.

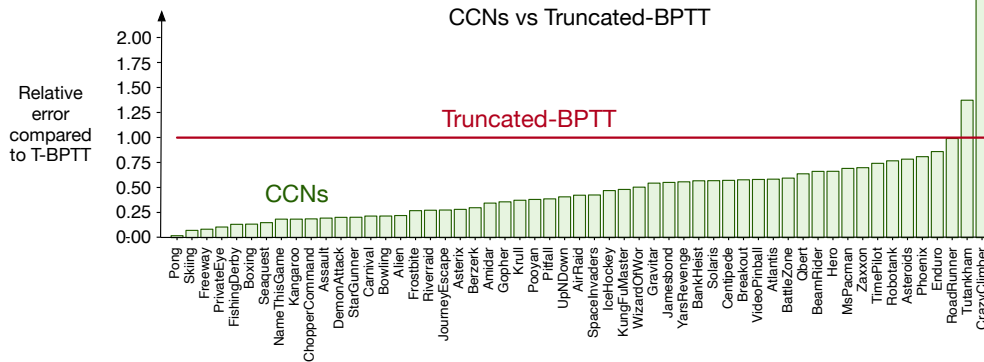


Figure 8: Comparing the CCN network with the best T-BPTT on the Atari Prediction Benchmark. For all but two games, CCN network achieves lower prediction error than T-BPTT. In many games, the CCN reduces the prediction error by many folds. All errors are averaged over 15 random seeds, and the error margins represent one standard error.

5.2 Experimental Setup

We compare our methods with T-BPTT, using LSTMs and TD(λ) for all algorithms. We set the per-step compute budget to 50k FLOPS and learn the value function for 50 million steps. We fix the discount factor γ to be 0.98, and λ to be 0.99 in all experiments. The remaining parameters— ϵ , steps-per-stage, truncation window, and step-size—are tuned for each method independently. The details of the hyperparameter tuning are in Table 6. We pick hyperparameters that give the best results averaged over all the environments.

For each environment, we report the average return error in the last 200k steps. Since the scale of the returns is very different for different environments, it is important to normalize the errors for visualization purposes. For each environment, we normalize the error of all methods by dividing by the error achieved by the T-BPTT baseline in that environment. This means that after normalization, the T-BPTT baseline has an error of one in all environments, whereas the error of other methods is relative to that achieved by T-BPTT. For instance, if a CCN network achieves an error of 0.5 in an environment, that means the error is half of what was achieved by T-BPTT. Similarly, an error of 2 for CCN means the error is twice as much as T-BPTT.

5.3 Overall Performance

We report error across all environments for the CCN and T-BPTT in Figure 8. The CCN performs better than T-BPTT in all but two environments. In many environments, it achieves 5x lower error, whereas even in the worst case of CrazyClimber, the error is only twice as much as T-BPTT.

We also look at errors achieved by constructive and columnar networks and report them in Figure 9. For brevity, we only report the error averaged over all environments. We see

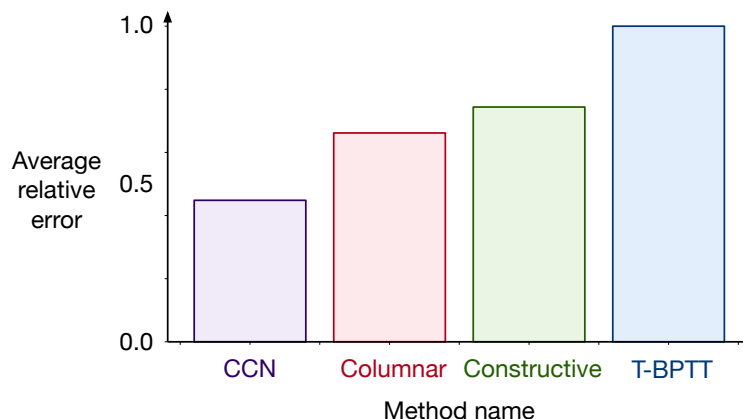


Figure 9: Average relative error of all methods on the Atari Prediction Benchmark averaged over 15 random seeds. Both the constructive and the columnar approach improve over T-BPTT on average. Combining both gives the best results. The average relative error achieved by CCN is less than half of what the best T-BPTT achieves at the same compute budget.

that all three of the proposed methods improve over T-BPTT. CCN performs the best, demonstrating that combining columnar and constructive approaches is useful.

5.4 Visualizing the Predictions

We visualize the predictions made by the CCN and T-BPTT at end of learning in Figure 10. We can see that both methods can learn to make accurate predictions. Predictions made by the CCN are closer, on average, to the ground truth returns than the predictions made by the LSTM trained using T-BPTT.

5.5 Sensitivity to truncation for T-BPTT

As before, we more fully investigate the impact of the number of features and truncation length on the performance of T-BPTT, to give a more complete picture of our main comparator algorithm. We perform two experiments. In the first experiment, we fix the truncation window to 8 and vary the number of features from 2 to 15. In the second experiment, we fix the number of features to 8, and vary the truncation window from 2 to 15. We report both results in Figure 11. We see that both increasing the number of features, and the truncation window improve the performance of T-BPTT. The number of features has a bigger impact: going from 2 features to 15 halves the error, whereas going from a truncation window of 2 to 15 reduces the error by around 23%.

6. Conclusions and Future Directions

In this paper we showed that by either restricting connections between recurrent neurons—the columnar approach—or learning a recurrent network incrementally—the constructive

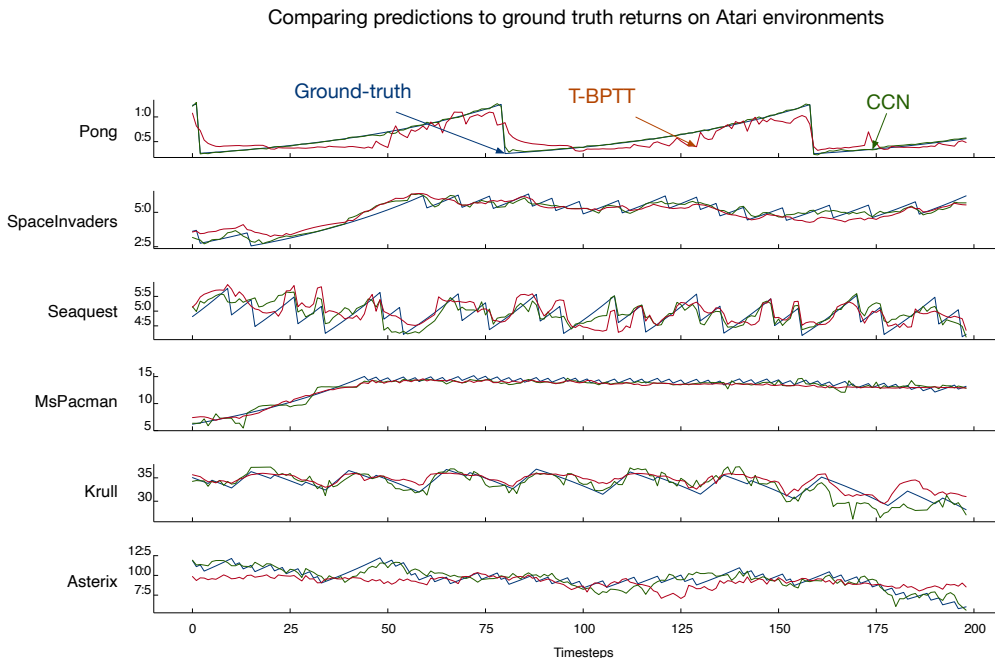


Figure 10: Visualizing predictions made by T-BPTT and CCN networks on five Atari environments. We plot predictions made at the end of the 50 million steps learning process. The green lines are predictions made by CCN, the orange are predictions made by the best T-BPTT, and the dotted blue line represents the ground truth return observed by the agent. We see that CCN makes qualitatively better predictions on most of the environments. The difference is most pronounced in Pong, in which CCN makes near perfect predictions. T-BPTT also learns the general trend of predictions correctly, but does not follow the ground truth return as closely.

approach—we can compute gradients of parameters of a recurrent network cheaply and without truncation. Moreover, unlike T-BPTT, our algorithm does not rely on sequential operations and can be fully parallelized. We show that in the under-parameterized setting, our methods out-perform T-BPTT when using a fixed compute budget. Moreover, the algorithms can be scaled to learn networks with billions of parameters using roughly the same amount of resources needed for inference of similarly sized models. Because the learning algorithms don’t use a lot of resources, there is no need to disable learning at the time of deployment and we can build system that continually learn and adapt to their data-stream.

One major limitation of our approach is that in both constructive and CCN approaches, most of the features are frozen as time goes by. As a result, the network loses its plasticity and the ability to adapt to changes.

We can address this limitation by combining our work with online weight and feature pruning. Instead of only adding features to grow the size of the network, we can instead

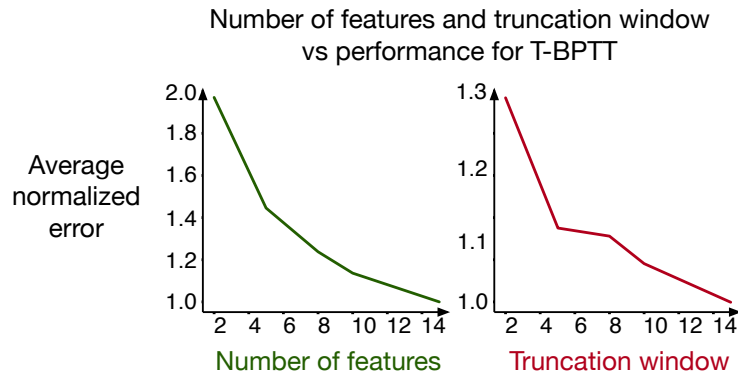


Figure 11: Impact of capacity and truncation window on the performance of T-BPTT on the Atari Prediction Benchmark. We report normalized error averaged over all atari environments. The error is normalized such that the average error is one when number of features, or truncation window is 15. For the graph on the left, we fix the truncation window to 8, and vary the number of features. We see that as the network gets larger, the performance improves. The error of an LSTM using two features is twice as much as an LSTM using 15 features. For the plot on the right, we fix the number of features to 8, and vary the truncation window of T-BPTT. Once again, we see that as the truncation window gets smaller, error increases.

continually replace the least useful features with new features, and learn them, as shown by Dohare *et al.* (2021). The continual pruning and generation assures that a frozen part of the network stays only as long as it is useful for prediction. Applying our ideas in this continual replacement setting is left as future work.

References

- Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *The journal of artificial intelligence research*.
- Cooijmans, T., Ballas, N., Laurent, C., Gülçehre, Ç., & Courville, A. (2017). Recurrent batch normalization. *International conference on learning representations*.
- Dohare, S., Sutton, R. S., & Mahmood, A. R. (2021). Continual backprop: Stochastic gradient descent with persistent randomness. *arXiv preprint arXiv:2108.06325*.
- Elman, J. L. (1990). Finding structure in time. *Cognitive science*.
- Fahlman, S. (1990). The recurrent cascade-correlation architecture. *Advances in neural information processing systems*.
- Fujita, Y., Nagarajan, P., Kataoka, T., & Ishikawa, T. (2021). Chainerrl: A deep reinforcement learning library. *The journal of machine learning research*.

- Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. AAAI conference on artificial intelligence.
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*.
- Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International conference on machine learning*.
- Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., & Dabney, W. (2019). Recurrent experience replay in distributed reinforcement learning. *International conference on learning representations*.
- Menick, J., Elsen, E., Evci, U., Osindero, S., Simonyan, K., & Graves, A. (2021). A practical sparse approximation for real time recurrent learning. *International conference on learning representations*.
- Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., & Khudanpur, S. (2010). Recurrent neural network based language model. *Interspeech*.
- Mikolov, T., Kopecky, J., Burget, L., & Glembek, O. (2009). Neural network based language models for highly inflective languages. *International conference on acoustics, speech and signal processing*.
- Mountcastle, V. B. (1957). Modality and topographic properties of single neurons of cat's somatic sensory cortex. *Journal of neurophysiology*.
- Mujika, A., Meier, F., & Steger, A. (2018). Approximating real-time recurrent learning with random kronecker factors. *Advances in neural information processing systems*.
- Rafiee, B., Abbas, Z., Ghiassian, S., Kumaraswamy, R., Sutton, R., Ludvig, E., & White, A. (2022). From Eye-blinks to State Construction: Diagnostic Benchmarks for Online Representation Learning. *Adaptive behavior*.
- Robinson, A. J., & Fallside, F. (1987). *The utility driven dynamic error propagation network*. University of Cambridge.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*.
- Sutskever, I. (2013). *Training recurrent neural networks*. University of Toronto.
- Sutton, R. S. (1988). *Learning to predict by the methods of temporal differences*. Machine learning.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.
- Sutton, R. S. (1984). *Temporal credit assignment in reinforcement learning*. University of Massachusetts Amherst.
- Sutton, R. S. (1988). *Learning to predict by the methods of temporal differences*. Machine learning.

- Tallec, C., & Ollivier, Y. (2018). Unbiased online recurrent optimization. International conference on learning representations.
- Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., ... & Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*.
- Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*.
- Williams, R. J., & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural computation*.
- Williams, R. J., & Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*.

Appendix A

Hyperparameter Settings

We tune the solution-specific hyperparameters for all the methods independently. For each configurations, we use five random seeds and look at the performance over all five seeds to pick the best hyperparameters. We then run the best hyperparameter configuration for 30 seeds for reporting the trace patterning results and 15 seeds for reporting the Atari results. List of all the hyperparameter, and their values are given in Table 1.

Table 1: Step-size

Symbol	Hyperparameter	Environment	Hyperparameter values
α	Step-size (T-BPTT)	Both	$1^{-2}, 3^{-3}, 1^{-3}, 3^{-4}, 1^{-4}$
α	Step-size (CCN and Constructive)	Both	$1^{-2}, 1^{-3}, 1^{-4}$
γ	Discount factor	Trace	0.90
γ	Discount factor	Atari	0.98
λ	Eligibility trace decay rate	Both	0.99
$k : d$	Truncation:Hidden features (T-BPTT)	Trace	2:13, 3:10, 5:8, 8:6, 10:5, 15:4, 20:3, 30:2
$k : d$	Truncation:Hidden features (T-BPTT)	Atari	15:2, 8:5, 5:8, 4:10, 2:25
	Hidden features (Columnar)	Trace	5
	Hidden features (Columnar)	Atari	7
	Features per stage (CCN)	Trace	4
	Features per stage (CCN)	Atari	5
	Steps per stage (CCN)	Trace	10 million
	Steps per stage (CCN)	Atari	16 million
	Steps per stage (Constructive)	Both	5 million
	Total steps	Both	50 million
	Seeds for parameter sweep	Both	$f0, 1, 2, 3, 4g$
	Seeds for best parameter configuration	Trace	$f0, 1, \quad, 29g$
	Seeds for best parameter configuration	Atari	$f0, 1, \quad, 14g$
ϵ	Min division term (CCN and Constructive)	Both	$f0.1, 0.01, 0.001g$

Implementation Details

We implement all methods in C++. For columnar, constructive, and CCN approaches, we use the update equations derived in Appendix B. We verify the correctness of the gradients computed by our derived equations, and our implementation of T-BPTT by comparing them to the gradients computed by PyTorch for networks initialized to have the same parameters. The gradients given by our implementation and those by PyTorch match exactly. Our C++ implementation avoids the overhead of Python and PyTorch, and is around 50x faster for small recurrent networks as compared to PyTorch for LSTMs that are trained using one

sample at a time. Having a fast and efficient implementation was crucial for performing large hyperparameter sweeps reporting statistically significant results using multiple seeds.

For batch learning, GPU implementation of PyTorch would be faster. That said, our algorithms are fully decentralized and when applied to recurrent networks with millions of features, constructive, columnar, and CCN can benefit from parallel compute units of GPUs. LSTM trained with T-BPTT, on the other hand, have to do sequential computation to compute the gradient, and are fundamentally limited in terms of benefiting from parallel compute units.

Compute infrastructure

We run all experiments on large CPU clusters. A single run of the trace patterning task for 50 million steps takes around 5 minutes on a single CPU, whereas a single run on Atari for 50 million steps takes around 50 minutes. Both experiments take less than 2 GB of ram per run. We used 1,000 CPUs to do the experiments in parallel.

Equations for Estimating Compute Used by Each Method

Every method uses roughly the same amount of computation per-step. We estimate the amount of compute used by each method by looking at its architecture and the learning algorithm. These estimates are not exact, and there may be some minor differences depending on how these methods are implemented in practice. However, the principle largely remains the same. And we have verified from our empirical observations that these estimates are close to what we observe.

Let jhj be the number of hidden features, jxj be the number of input features, k be the truncation window, and u be the features-per-stage parameter. Then the total amount of computation used by an LSTM cell for a single forward pass can be estimated using the following equation:

$$4jhj + 4jxj + 4$$

where the number four is due to the four gates used by an LSTM cell. In T-BPTT, we used a fully connected LSTM so the total number of features would be jhj . Forward pass of a fully connected LSTM would use:

$$h(4jhj + 4jxj + 4) = 4jhj^2 + 4jhjjxj + 4jhj$$

operations. Finally, T-BPTT requires k times more computation for computing the gradient, bringing the total cost to:

$$\begin{aligned} & 4jhj^2 + 4jhjjxj + 4jhj + k(4jhj^2 + 4jhjjxj + 4jhj) \\ & = (k + 1)(4jhj^2 + 4jhjjxj + 4jhj) \end{aligned}$$

For columnar, constructive and CCN, first we see from Appendix B that recursively computing the gradient is roughly six times more expensive than the forward pass of the LSTM, which, according to our empirical observations, is an overestimation. Total compute used by a single columnar cell for the forward pass, therefore, is:

$$4 + 4jxj + 4$$

since hidden state = 1 for a single column. Compute used by jhj cells is:

$$jhj(4jxj + 8).$$

Adding compute used by the learning algorithm, we get:

$$jhj(4jxj + 8) + 6jhj(4jxj + 8)$$

In the CCN approach, on average, an LSTM cell takes as input $\frac{jhj}{2}$ hidden states. As a result, the compute used for a single forward pass by a single recurrent feature is given by:

$$4\frac{jhj}{2} + 4jxj + 4,$$

and for jhj features it is:

$$jhj(2jhj + 4jxj + 4).$$

Since we learn u features at a time, the total estimated compute per step for CCN networks is given by:

$$jhj(2jhj + 4jxj + 4) + 6u(2jhj + 4jxj + 4).$$

For constructive networks, we can substitute $u = 1$ in the equation above.

Appendix B

Forward-mode gradient computation of an LSTM cell

Here we derive the update equations for recursively computing the gradients of a single LSTM based recurrent column. Each column has a single hidden unit. Because all columns are identical, the same update equations can be used for learning in columnar, constructive, and the CCN approach. We compared the gradients estimated using the derived equations with the gradient computed using BPTT in PyTorch without truncation on random trajectories, and found them to match exactly.

The state of an LSTM column is updated using following equations:

$$i(t) = \sigma(W_i^T x_k(t) + u_i h(t-1) + b_i) \tag{11}$$

$$f(t) = \sigma(W_f^T x_k(t) + u_f h(t-1) + b_f) \tag{12}$$

$$o(t) = \sigma(W_o^T x_k(t) + u_o h(t-1) + b_o) \tag{13}$$

$$g(t) = \phi(W_g^T x_k(t) + u_g h(t-1) + b_g) \tag{14}$$

$$c(t) = f(t)c(t-1) + i(t)g(t) \tag{15}$$

$$h(t) = o(t)\phi(c(t)) \tag{16}$$

where σ and ϕ are the sigmoid and tanh activation functions, $h(t)$ is the state of the column at time t and $W_i^T x_k(t) = \sum_{k=1}^m W_{i_k} x_k(t)$. The derivative of $\sigma(x)$ and $\phi(x)$ w.r.t to x are $\sigma(x)(1 - \sigma(x))$ and $(1 - \phi^2(x))$ respectively.

Let the length of input vector x be m . Then, W_i, W_f, W_o and W_g are vectors of length m whereas $u_i, b_i, u_f, b_f, u_o, b_o, u_g$ and b_g are scalars. We want to compute gradient of $h(t)$ with respect to all the parameters. We derive the update equations for $\frac{\partial h(t)}{\partial W_i}, \frac{\partial h(t)}{\partial u_i}, \frac{\partial h(t)}{\partial b_i}, \frac{\partial h(t)}{\partial W_f}, \frac{\partial h(t)}{\partial u_f}, \frac{\partial h(t)}{\partial b_f}, \frac{\partial h(t)}{\partial W_o}, \frac{\partial h(t)}{\partial u_o}, \frac{\partial h(t)}{\partial b_o}, \frac{\partial h(t)}{\partial W_g}, \frac{\partial h(t)}{\partial u_g}$, and $\frac{\partial h(t)}{\partial b_g}$ in the following sections.

$$\frac{\partial h(t)}{\partial W_i}$$

$W_i = (W_{i_1}, W_{i_2}, \dots, W_{i_m})$ is a vector of length m . Since all elements of W_i are symmetric, we show gradient derivation for W_{i_j} without loss of generality. Let

$$TH_{W_{i_j}}(t) := \frac{\partial h(t)}{\partial W_{i_j}} \quad (\text{By definition}) \quad (17)$$

$$TH_{W_{i_j}}(0) := 0 \quad (\text{By definition}) \quad (18)$$

$$TC_{W_{i_j}}(t) := \frac{\partial c(t)}{\partial W_{i_j}} \quad (\text{By definition}) \quad (19)$$

$$TC_{W_{i_j}}(0) := 0 \quad (\text{By definition}) \quad (20)$$

Then:

$$\begin{aligned} TH_{W_{i_j}}(t) &= \frac{\partial}{\partial W_{i_j}} (o(t)\phi(c(t))) && \text{From equation 16 and definition 17} \\ &= o(t) \frac{\partial \phi(c(t))}{\partial W_{i_j}} + \phi(c(t)) \frac{\partial o(t)}{\partial W_{i_j}} && \text{Product rule of differentiation} \\ &= o(t)(1 - \phi^2(c(t))) \frac{\partial c(t)}{\partial W_{i_j}} + \phi(c(t)) \frac{\partial o(t)}{\partial W_{i_j}} && \text{Derivative of } \phi(x) \text{ is } (1-\phi^2(x)) \\ &= o(t)(1 - \phi^2(c(t)))TC_{W_{i_j}}(t) + \phi(c(t)) \frac{\partial o(t)}{\partial W_{i_j}} && \text{From definition 19} \\ \frac{\partial o(t)}{\partial W_{i_j}} &= \frac{\partial}{\partial W_{i_j}} \sigma(W_o^T x(t) + u_o h(t-1) + b_o) && \text{From equation 13} \\ &= \sigma(y)(1 - \sigma(y))u_o TH_{W_{i_j}}(t-1) && \text{Where } y \text{ equals } W_o^T x(t) + u_o h(t-1) + b_o \\ TC_{W_{i_j}}(t) &= \frac{\partial c(t)}{\partial W_{i_j}} && \text{From definition 19} \\ &= \frac{\partial}{\partial W_{i_j}} (f(t)c(t-1) + i(t)g(t)) && \text{From equation 15} \\ &= f(t)TC_{W_{i_j}}(t-1) + c(t-1) \frac{\partial f(t)}{\partial W_{i_j}} && \text{Product rule and definition 19} \\ &\quad + \frac{\partial}{\partial W_{i_j}} (i(t)g(t)) \\ &= f(t)TC_{W_{i_j}}(t-1) + c(t-1) \frac{\partial f(t)}{\partial W_{i_j}} && \text{Product rule} \\ &\quad + i(t) \frac{\partial g(t)}{\partial W_{i_j}} + g(t) \frac{\partial i(t)}{\partial W_{i_j}} \end{aligned}$$

Where gradient of $g(t)$ w.r.t W_{i_j} is:

$$\begin{aligned} \frac{\partial g(t)}{\partial W_{i_j}} &= \frac{\partial}{\partial W_{i_j}} \phi(W_g^T x(t) + u_g h(t-1) + b_g) && \text{From equation 14} \\ &= (1 - \phi^2(y)) u_g T H_{W_{i_j}}(t-1) && \text{Where } y \text{ equals } W_g^T x(t) + u_g h(t-1) + b_g \end{aligned}$$

, gradient of $f(t)$ w.r.t W_{i_j} is:

$$\begin{aligned} \frac{\partial f(t)}{\partial W_{i_j}} &= \frac{\partial}{\partial W_{i_j}} \sigma(W_f^T x(t) + u_f h(t-1) + b_f) && \text{From equation 12} \\ &= \sigma(y)(1 - \sigma(y)) u_f T H_{W_{i_j}}(t-1) && \text{Where } y \text{ equals } W_f^T x(t) + u_f h(t-1) + b_f \end{aligned}$$

and gradient of $i(t)$ w.r.t W_{i_j} is:

$$\begin{aligned} \frac{\partial i(t)}{\partial W_{i_j}} &= \frac{\partial}{\partial W_{i_j}} \sigma(W_i^T x(t) + u_i h(t-1) + b_i) && \text{From equation 11} \\ &= \sigma(y)(1 - \sigma(y)) \left(x_j(t) + u_i T H_{W_{i_j}}(t-1) \right) && \text{Where } y \text{ equals } W_i^T x(t) + u_i h(t-1) + b_i \end{aligned}$$

The derivation shows that using two traces per parameter of W_i , it is possible to compute the gradient of $h(t)$ w.r.t W_i recursively. We provide the derivations for parameters u_i and b_i below. We skip the step-by-step derivations for the remaining parameters as they are similar.

$$\frac{\partial h(t)}{\partial u_i}$$

$$T H_{u_i}(t) := \frac{\partial h(t)}{\partial u_i} \quad (\text{By definition}) \quad (21)$$

$$T H_{u_i}(0) := 0 \quad (\text{By definition}) \quad (22)$$

$$T C_{u_i}(t) := \frac{\partial c(t)}{\partial u_i} \quad (\text{By definition}) \quad (23)$$

$$T C_{u_i}(0) := 0 \quad (\text{By definition}) \quad (24)$$

$$\begin{aligned}
 TH_{u_i}(t) &= \frac{\partial}{\partial u_i} (o(t)\phi(c(t))) && \text{From equation 16} \\
 &= o(t)\frac{\partial\phi(c(t))}{\partial u_i} + \phi(c(t))\frac{\partial o(t)}{\partial u_i} && \text{Product rule} \\
 &= o(t)(1 - \phi^2(c(t)))\frac{\partial c(t)}{\partial u_i} + \phi(c(t))\frac{\partial o(t)}{\partial u_i} && \text{Derivative of } \phi(x) \text{ is } 1 - \phi^2(x) \\
 &= o(t)(1 - \phi^2(c(t)))TC_{u_i}(t) + \phi(c(t))\frac{\partial o(t)}{\partial u_i} && \text{Using definition 23} \\
 \frac{\partial o(t)}{\partial u_i} &= \frac{\partial}{\partial u_i} \sigma(W_o^T x(t) + u_o h(t-1) + b_o) && \text{Using equations 13} \\
 &= \sigma(x)(1 - \sigma(x))u_o TH_{u_i}(t-1) && \text{Where } x \text{ equal } W_o^T x(t) + u_o h(t-1) + b_o \\
 TC_{u_i}(t) &= \frac{\partial c(t)}{\partial u_i} && \text{Definition 23} \\
 &= \frac{\partial}{\partial u_i} (f(t)c(t-1) + i(t)g(t)) && \text{From equation 16} \\
 &= f(t)TC_{u_i}(t-1) + c(t-1)\frac{\partial f(t)}{\partial u_i} && \text{Product rule} \\
 &\quad + \frac{\partial}{\partial u_i} (i(t)g(t)) \\
 &= f(t)TC_{u_i}(t-1) + c(t-1)\frac{\partial f(t)}{\partial u_i} && \text{Product rule} \\
 &\quad + i(t)\frac{\partial g(t)}{\partial u_i} + g(t)\frac{\partial i(t)}{\partial u_i}
 \end{aligned}$$

Gradient of $g(t)$ w.r.t u_i is:

$$\begin{aligned}
 \frac{\partial g(t)}{\partial u_i} &= \frac{\partial}{\partial u_i} \phi(W_g^T x(t) + u_g h(t-1) + b_g) && \text{From equations 16} \\
 &= (1 - \phi^2(y))u_g TH_{u_i}(t-1) && \text{Where } y \text{ equals } W_g^T x(t) + u_g h(t-1) + b_g
 \end{aligned}$$

, gradient of $f(t)$ w.r.t u_i is:

$$\begin{aligned}
 \frac{\partial f(t)}{\partial u_i} &= \frac{\partial}{\partial u_i} \sigma(W_f^T x(t) + u_f h(t-1) + b_f) && \text{From equations 1} \\
 &= \sigma(y)(1 - \sigma(y))u_f TH_{u_i}(t-1) && \text{Where } y \text{ equals } W_f^T x(t) + u_f h(t-1) + b_f
 \end{aligned}$$

and the gradient of $i(t)$ w.r.t u_i is

$$\begin{aligned}
 \frac{\partial i(t)}{\partial u_i} &= \frac{\partial}{\partial u_i} \sigma(W_i^T x(t) + u_i h(t-1) + b_i) && \text{Using equations 1} \\
 &= \sigma(y)(1 - \sigma(y))(h(t-1) + u_i TH_{u_i}(t-1)) && \text{Where } y \text{ equals } W_i^T x(t) + u_i h(t-1) + b_i
 \end{aligned}$$

$$\frac{\partial h(t)}{\partial b_i}$$

$$TH_{b_i}(t) := \frac{\partial h(t)}{\partial b_i} \quad (\text{By definition}) \quad (25)$$

$$TH_{b_i}(0) := 0 \quad (\text{By definition}) \quad (26)$$

$$TC_{b_i}(t) := \frac{\partial c(t)}{\partial b_i} \quad (\text{By definition}) \quad (27)$$

$$TC_{b_i}(0) := 0 \quad (\text{By definition}) \quad (28)$$

$$\begin{aligned} TH_{b_i}(t) &= \frac{\partial}{\partial b_i} (o(t)\phi(c(t))) && \text{From equation 16} \\ &= o(t)\frac{\partial \phi(c(t))}{\partial b_i} + \phi(c(t))\frac{\partial o(t)}{\partial b_i} && \text{Product rule} \\ &= o(t)(1 - \phi^2(c(t)))\frac{\partial c(t)}{\partial b_i} + \phi(c(t))\frac{\partial o(t)}{\partial b_i} && \text{Derivative of } \phi(x) \text{ is } 1 - \phi^2(x) \\ &= o(t)(1 - \phi^2(c(t)))TC_{b_i}(t) + \phi(c(t))\frac{\partial o(t)}{\partial b_i} && \text{From definition 27} \\ \frac{\partial o(t)}{\partial b_i} &= \frac{\partial}{\partial b_i} \sigma(W_o^T x(t) + u_o h(t-1) + b_o) && \text{From equations 13} \\ &= \sigma(y)(1 - \sigma(y))u_o TH_{b_i}(t-1) && \text{Where } y \text{ equal } W_o^T x(t) + u_o h(t-1) + b_o \\ TC_{b_i}(t) &= \frac{\partial c(t)}{\partial b_i} && \text{From definition 27} \\ &= \frac{\partial}{\partial b_i} (f(t)c(t-1) + i(t)g(t)) && \text{From equation 15} \\ &= f(t)TC_{b_i}(t-1) + c(t-1)\frac{\partial f(t)}{\partial b_i} && \text{Product rule} \\ &\quad + \frac{\partial}{\partial b_i} i(t)g(t) \\ &= f(t)TC_{b_i}(t-1) + c(t-1)\frac{\partial f(t)}{\partial b_i} && \text{Product rule} \\ &\quad + i(t)\frac{\partial g(t)}{\partial b_i} + g(t)\frac{\partial i(t)}{\partial b_i} \end{aligned}$$

Where gradient of $g(t)$ w.r.t b_i is:

$$\begin{aligned} \frac{\partial g(t)}{\partial b_i} &= \frac{\partial}{\partial b_i} \phi(W_g^T x(t) + u_g h(t-1) + b_g) && \text{From equation 14} \\ &= (1 - \phi^2(y))u_g TH_{b_i}(t-1) && \text{Where } y \text{ equal } W_g^T x(t) + u_g h(t-1) + b_g \end{aligned}$$

, gradient of $f(t)$ w.r.t b_i is:

$$\begin{aligned} \frac{\partial f(t)}{\partial b_i} &= \frac{\partial}{\partial b_i} \sigma(W_f^T x(t) + u_f h(t-1) + b_f) && \text{From equation 12} \\ &= \sigma(y)(1 - \sigma(y)) u_f T H_{b_i}(t-1) && \text{Where } y \text{ equal } W_f^T x(t) + u_f h(t-1) + b_f \end{aligned}$$

and gradient of $i(t)$ w.r.t b_i is:

$$\begin{aligned} \frac{\partial i(t)}{\partial b_i} &= \frac{\partial}{\partial b_i} \sigma(W_i^T x(t) + u_i h(t-1) + b_i) && \text{From equation 11} \\ &= \sigma(y)(1 - \sigma(y)) (u_i T H_{b_i}(t-1) + 1) && \text{Where } y \text{ equal } W_i^T x(t) + u_i h(t-1) + b_i \end{aligned}$$

$$\frac{\partial h(t)}{\partial W_{f_j}}$$

The derivations for the remaining parameters is analogous to what previous derivations. The final equations are as follows.

$$\begin{aligned} \frac{\partial g(t)}{\partial W_{f_j}} &= (1 - \phi^2(y)) (u_g T H_{W_{f_j}}(t-1)) \\ \frac{\partial f(t)}{\partial W_{f_j}} &= \sigma(y)(1 - \sigma(y)) (x_j + u_f T H_{W_{f_j}}(t-1)) \\ \frac{\partial i(t)}{\partial W_{f_j}} &= \sigma(y)(1 - \sigma(y)) (u_i T H_{W_{f_j}}(t-1)) \\ \frac{\partial o(t)}{\partial W_{f_j}} &= \sigma(y)(1 - \sigma(y)) (u_o T H_{W_{f_j}}(t-1)) \\ TC_{W_{f_j}} &= f(t) TC_{f_j}(t-1) + c(t-1) \frac{\partial f(t)}{\partial b_i} + i(t) \frac{\partial g(t)}{\partial b_i} + g(t) \frac{\partial i(t)}{\partial b_i} \\ TH_{W_{f_j}} &= o(t)(1 - \phi^2(c(t))) TC_{W_{f_j}}(t) + \phi(c(t)) \frac{\partial o(t)}{\partial W_{ij}} \end{aligned} \tag{29}$$

$$\frac{\partial h(t)}{\partial W_{o_j}}$$

$$\begin{aligned} \frac{\partial g(t)}{\partial W_{o_j}} &= (1 - \phi^2(y))(u_g TH_{W_{o_j}}(t-1)) \\ \frac{\partial f(t)}{\partial W_{o_j}} &= \sigma(y)(1 - \sigma(y))(u_f TH_{W_{o_j}}(t-1)) \\ \frac{\partial i(t)}{\partial W_{o_j}} &= \sigma(y)(1 - \sigma(y))u_i TH_{W_{o_j}}(t-1) \\ \frac{\partial o(t)}{\partial W_{o_j}} &= \sigma(x)(1 - \sigma(x))(x_j + u_o TH_{W_{o_j}}(t-1)) \\ TC_{W_{o_j}} &= f(t)TC_{o_j}(t-1) + c(t-1)\frac{\partial f(t)}{\partial b_i} + i(t)\frac{\partial g(t)}{\partial b_i} + g(t)\frac{\partial i(t)}{\partial b_i} \\ TH_{W_{o_j}} &= o(t)(1 - \phi^2(c(t)))TC_{W_{o_j}}(t) + \phi(c(t))\frac{\partial o(t)}{\partial W_{ij}} \end{aligned} \tag{30}$$

$$\frac{\partial h(t)}{\partial W_{g_j}}$$

$$\begin{aligned} \frac{\partial g(t)}{\partial W_{g_j}} &= (1 - \phi^2(y))(x_j + u_g TH_{W_{g_j}}(t-1)) \\ \frac{\partial f(t)}{\partial W_{g_j}} &= \sigma(y)(1 - \sigma(y))(u_f TH_{W_{g_j}}(t-1)) \\ \frac{\partial i(t)}{\partial W_{g_j}} &= \sigma(y)(1 - \sigma(y))(u_i TH_{W_{g_j}}(t-1)) \\ \frac{\partial o(t)}{\partial W_{g_j}} &= \sigma(x)(1 - \sigma(x))(u_o TH_{W_{g_j}}(t-1)) \\ TC_{W_{g_j}} &= f(t)TC_{g_j}(t-1) + c(t-1)\frac{\partial f(t)}{\partial b_i} + i(t)\frac{\partial g(t)}{\partial b_i} + g(t)\frac{\partial i(t)}{\partial b_i} \\ TH_{W_{g_j}} &= o(t)(1 - \phi^2(c(t)))TC_{W_{g_j}}(t) + \phi(c(t))\frac{\partial o(t)}{\partial W_{ij}} \end{aligned} \tag{31}$$

$$\frac{\partial h(t)}{\partial u_o}$$

$$\begin{aligned}
 \frac{\partial g(t)}{\partial u_o} &= (1 - \phi^2(y))(u_g TH_{u_o}(t-1)) \\
 \frac{\partial f(t)}{\partial u_o} &= \sigma(y)(1 - \sigma(y))(u_f TH_{u_o}(t-1)) \\
 \frac{\partial i(t)}{\partial u_o} &= \sigma(y)(1 - \sigma(y))(u_i TH_{u_o}(t-1)) \\
 \frac{\partial o(t)}{\partial u_o} &= \sigma(x)(1 - \sigma(x))(u_o TH_{u_o}(t-1) + h(t-1)) \\
 TC_{u_o} &= f(t)TC_{i_j}(t-1) + c(t-1)\frac{\partial f(t)}{\partial b_i} + i(t)\frac{\partial g(t)}{\partial b_i} + g(t)\frac{\partial i(t)}{\partial b_i} \\
 TH_{u_o} &= o(t)(1 - \phi^2(c(t)))TC_{u_o}(t) + \phi(c(t))\frac{\partial o(t)}{\partial W_{ij}}
 \end{aligned} \tag{32}$$

$$\frac{\partial h(t)}{\partial u_f}$$

$$\begin{aligned}
 \frac{\partial g(t)}{\partial u_f} &= (1 - \phi^2(y))(u_g TH_{u_f}(t-1)) \\
 \frac{\partial f(t)}{\partial u_f} &= \sigma(y)(1 - \sigma(y))(u_f TH_{u_f}(t-1) + h(t-1)) \\
 \frac{\partial i(t)}{\partial u_f} &= \sigma(y)(1 - \sigma(y))(u_i TH_{u_f}(t-1)) \\
 \frac{\partial o(t)}{\partial u_f} &= \sigma(x)(1 - \sigma(x))(u_o TH_{u_f}(t-1)) \\
 TC_{u_f} &= f(t)TC_{i_j}(t-1) + c(t-1)\frac{\partial f(t)}{\partial b_i} + i(t)\frac{\partial g(t)}{\partial b_i} + g(t)\frac{\partial i(t)}{\partial b_i} \\
 TH_{u_f} &= o(t)(1 - \phi^2(c(t)))TC_{u_f}(t) + \phi(c(t))\frac{\partial o(t)}{\partial W_{ij}}
 \end{aligned} \tag{33}$$

$$\frac{\partial h(t)}{\partial u_g}$$

$$\begin{aligned}
 \frac{\partial g(t)}{\partial u_g} &= (1 - \phi^2(y))(u_g TH_{u_g}(t-1) + h(t-1)) \\
 \frac{\partial f(t)}{\partial u_g} &= \sigma(y)(1 - \sigma(y))(u_f TH_{u_g}(t-1)) \\
 \frac{\partial i(t)}{\partial u_g} &= \sigma(y)(1 - \sigma(y))(u_i TH_{u_g}(t-1)) \\
 \frac{\partial o(t)}{\partial u_g} &= \sigma(x)(1 - \sigma(x))(u_o TH_{u_g}(t-1)) \\
 TC_{u_g} &= f(t)TC_{i_j}(t-1) + c(t-1)\frac{\partial f(t)}{\partial b_i} + i(t)\frac{\partial g(t)}{\partial b_i} + g(t)\frac{\partial i(t)}{\partial b_i} \\
 TH_{u_g} &= o(t)(1 - \phi^2(c(t)))TC_{u_g}(t) + \phi(c(t))\frac{\partial o(t)}{\partial W_{ij}}
 \end{aligned} \tag{34}$$

$$\frac{\partial h(t)}{\partial b_g}$$

$$\begin{aligned}
 \frac{\partial g(t)}{\partial b_g} &= (1 - \phi^2(y))(u_g TH_{b_g}(t-1) + 1) \\
 \frac{\partial f(t)}{\partial b_g} &= \sigma(y)(1 - \sigma(y))(u_f TH_{b_g}(t-1)) \\
 \frac{\partial i(t)}{\partial b_g} &= \sigma(y)(1 - \sigma(y))(u_i TH_{b_g}(t-1)) \\
 \frac{\partial o(t)}{\partial b_g} &= \sigma(x)(1 - \sigma(x))(u_o TH_{b_g}(t-1)) \\
 TC_{b_g} &= f(t)TC_{i_j}(t-1) + c(t-1)\frac{\partial f(t)}{\partial b_i} + i(t)\frac{\partial g(t)}{\partial b_i} + g(t)\frac{\partial i(t)}{\partial b_i} \\
 TH_{b_g} &= o(t)(1 - \phi^2(c(t)))TC_{b_g}(t) + \phi(c(t))\frac{\partial o(t)}{\partial W_{ij}}
 \end{aligned} \tag{35}$$

$$\frac{\partial h(t)}{\partial b_f}$$

$$\begin{aligned}
 \frac{\partial g(t)}{\partial b_f} &= (1 - \phi^2(y))(u_g TH_{b_f}(t - 1)) \\
 \frac{\partial f(t)}{\partial b_f} &= \sigma(y)(1 - \sigma(y))(u_f TH_{b_f}(t - 1) + 1) \\
 \frac{\partial i(t)}{\partial b_f} &= \sigma(y)(1 - \sigma(y))(u_i TH_{b_f}(t - 1)) \\
 \frac{\partial o(t)}{\partial b_f} &= \sigma(x)(1 - \sigma(x))(u_o TH_{b_f}(t - 1)) \\
 TC_{b_f} &= f(t)TC_{i_j}(t - 1) + c(t - 1)\frac{\partial f(t)}{\partial b_i} + i(t)\frac{\partial g(t)}{\partial b_i} + g(t)\frac{\partial i(t)}{\partial b_i} \\
 TH_{b_f} &= o(t)(1 - \phi^2(c(t)))TC_{b_f}(t) + \phi(c(t))\frac{\partial o(t)}{\partial W_{ij}}
 \end{aligned} \tag{36}$$

$$\frac{\partial h(t)}{\partial b_o}$$

$$\begin{aligned}
 \frac{\partial g(t)}{\partial b_o} &= (1 - \phi^2(y))(u_g TH_{b_o}(t - 1)) \\
 \frac{\partial f(t)}{\partial b_o} &= \sigma(y)(1 - \sigma(y))(u_f TH_{b_o}(t - 1)) \\
 \frac{\partial i(t)}{\partial b_o} &= \sigma(y)(1 - \sigma(y))(u_i TH_{b_o}(t - 1)) \\
 \frac{\partial o(t)}{\partial b_o} &= \sigma(x)(1 - \sigma(x))(u_o TH_{b_o}(t - 1) + 1) \\
 TC_{b_o} &= f(t)TC_{i_j}(t - 1) + c(t - 1)\frac{\partial f(t)}{\partial b_i} + i(t)\frac{\partial g(t)}{\partial b_i} + g(t)\frac{\partial i(t)}{\partial b_i} \\
 TH_{b_o} &= o(t)(1 - \phi^2(c(t)))TC_{b_o}(t) + \phi(c(t))\frac{\partial o(t)}{\partial W_{ij}}
 \end{aligned} \tag{37}$$