Journal of Machine Learning Research 21 (2020) 1-34 Submitted 7/19; Revised 3/20; Published 6/20

A Unified Framework of Online Learning Algorithms for Training Recurrent Neural Networks

Owen Marschall OEM214@NYU.EDU
Center for Neural Science
New York University
New York, NY 10003, USA

Kyunghyun Cho[*] KYUNGHYUN.CHO@NYU.EDU
New York University
CIFAR Azrieli Global Scholar

Cristina Savin CSAVIN@NYU.EDU
Center for Neural Science
Center for Data Science
New York University

Editor: Yoshua Bengio

Abstract

We present a framework for compactly summarizing many recent results in efficient and/or biologically plausible online training of recurrent neural networks (RNN). The framework organizes algorithms according to several criteria: (a) past vs. future facing, (b) tensor structure, (c) stochastic vs. deterministic, and (d) closed form vs. numerical. These axes reveal latent conceptual connections among several recent advances in online learning. Furthermore, we provide novel mathematical intuitions for their degree of success. Testing these algorithms on two parametric task families shows that performances cluster according to our criteria. Although a similar clustering is also observed for pairwise gradient alignment, alignment with exact methods does not explain ultimate performance. This suggests the need for better comparison metrics.

Keywords: real-time recurrent learning, backpropagation through time, approximation, biologically plausible learning, local, online

1. Introduction

Training recurrent neural networks (RNN) to learn sequence data is traditionally done with stochastic gradient descent (SGD), using the backpropagation through time algorithm (BPTT, Werbos et al., 1990) to calculate the gradient. This requires "unrolling" the network over some range of time steps T and performing backpropagation as though the network were feedforward under the constraint of sharing parameters across time steps ("layers"). BPTT's success in a wide range of applications (Mikolov et al., 2010; Graves, 2013; Bahdanau et al., 2016, 2014; Cho et al., 2015; Graves et al., 2016; Vinyals et al., 2015; Luong et al., 2015) has made it the industry standard; however, there exist alternative **online** algorithms

[*]. This work was done while the author was a part-time research scientist at Facebook AI Research.

for training RNNs. These compute gradients in real time as the network runs forward, without explicitly referencing past activity or averaging over batches of data. There are two reasons for considering online alternatives to BPTT. One is practical: computational costs do not scale with T. The other is conceptual: human brains are able to learn long-term dependencies without explicitly memorizing all past brain states, and understanding online learning is a key step in the larger project of understanding human learning.

The classic online learning algorithm is real-time recurrent learning (RTRL, Williams and Zipser, 1989), which is equivalent to BPTT in the limit of a small learning rate (Murray, 2019). RTRL recursively updates the total derivative of the hidden state with respect to the parameters, eliminating the need to reference past activity but introducing an order n (hidden units) $\times n^2$ (parameters) $= n^3$ memory requirement. In practice, this is often more computationally demanding than BPTT (order nT in memory), hence not frequently used in applications. Nor is RTRL at face value a good model of biological learning, for the same reason: no known biological mechanism exists to store—let alone manipulate—a float for each synapse-neuron pair. Thus RTRL and online learning more broadly have remained relatively obscure footnotes to both the deep learning revolution and its impact on computational neuroscience.

Recent advances in recurrent network architectures have brought the issue of online learning back into the spotlight. While vanishing/exploding gradients used to significantly limit the extent of the temporal dependencies that an RNN could learn, new architectures like LSTMs (Hochreiter and Schmidhuber, 1997) and GRUs (Cho et al., 2014) as well as techniques like gradient clipping (Pascanu et al., 2013) have dramatically expanded this learnable time horizon. Unfortunately, taking advantage of this capacity requires an equally dramatic expansion in computational resources, if using BPTT. This has led to an explosion of novel online learning algorithms (Tallec and Ollivier, 2017; Mujika et al., 2018; Roth et al., 2019; Murray, 2019; Jaderberg et al., 2017) which aim to improve on the efficiency of RTRL, in many cases using update rules that might be implemented by a biological circuit.

The sheer number and variety of these approaches pose challenges for both theory and practice. It is not always clear what makes various algorithms different from one another, how they are conceptually related, or even why they might work in the first place. There is a pressing need in the field for a cohesive framework for describing and comparing online methods. Here we aim to provide a thorough overview of modern online algorithms for training RNNs, in a way that provides a clearer understanding of the mathematical structure underlying different approaches. Our framework organizes the existing literature along several axes that encode meaningful conceptual distinctions:

a) **Past facing** vs. **future facing**

b) The **tensor structure** of the algorithm

c) **Stochastic** vs. **deterministic** update

d) **Closed form** vs. **numerical** solution for update

These axes will be explained in detail later, but briefly: the past vs. future axis is a root distinction that divides algorithms by the type of gradient they calculate, while the other three describe their representations and update principles. Table 1 contains (to our knowledge) all recently published online learning algorithms for RNNs, categorized according to these

Algorithm	**Facing**	**Tensor**	**Update**		**Bias**	**Mem.**	**Time**
RTRL	Past	M_{kij}	Determ.	Closed-form	No	n^3	n^4
UORO	Past	$A_k B_{ij}$	Stoch.	Closed-form	No	n^2	n^2
KF-RTRL	Past	$A_j B_{ki}$	Stoch.	Closed-form	No	n^2	n^3
R-KF	Past	$A_i B_{jk}$	Stoch.	Closed-form	No	n^2	n^3
r-OK	Past	$\sum_{l=1}^{r} A_{lj} B_{lki}$	Stoch.	Numerical	No	rn^2	rn^3
KeRNL	Past	$A_{ki} B_{ij}$	Determ.	Numerical	Yes	n^2	n^2
RFLO	Past	$\delta_{ki} B_{ij}$	Determ.	Closed-form	Yes	n^2	n^2
E-BPTT	–	–	Determ.	Closed-form	No	nT	n^2
F-BPTT	Future	–	Determ.	Closed-form	No	nT	$n^2 T$
DNI	Future	A_{li}	Determ.	Numerical	Yes	n^2	n^2

Table 1: A list of learning algorithms reviewed here, together with their main properties. The indices k, i and j reference different dimensions of the "influence tensor" of RTRL (§3.1.1); l references components of the feedback vector $\tilde{\mathbf{a}}$ in DNI (§4.2).

criteria. We can already see that many combinations of these characteristics manifest in the literature, suggesting that new algorithms could be developed by mixing and matching properties. (We provide a concrete example of this in §3.4.)

Here we describe each algorithm in unified notation that makes clear their classification by these criteria. In the process, we generate novel intuitions about why different approximations can be successful and discuss some of the finer points of their biological plausibility. We confirm these intuitions numerically by evaluating different algorithms' ability to train RNNs on a common set of synthetic tasks with parameterized, interpretable difficulty. We find that both performance metrics and pairwise angular gradient alignments cluster according to our criteria (a)–(d) across tasks, lending credence to our approach. The class of approximations that proves best depends on the nature of the task: stochastic methods are better when the task involves long time dependencies while deterministic methods prove superior in high dimensional problems. Curiously, gradient alignment with exact methods (RTRL and BPTT) does not always predict performance, despite its ubiquity as a tool for analyzing approximate learning algorithms.

2. Past- and Future-Facing Perspectives of Online Learning

Before we dive into the details of these algorithms, we first articulate what we mean by past- and future-facing, related to the "reverse/forward accumulation" distinction described by Cooijmans and Martens (2019). Consider a recurrent neural network that contains, at each time step t, a state[1] $\mathbf{a}^{(t)} \in \mathbb{R}^n$. This state is updated via a function $F_{\mathbf{w}} : \mathbb{R}^m \rightarrow \mathbb{R}^n$, which is parameterized by a flattened vector of parameters $\mathbf{w} \in \mathbb{R}^P$. Here $m = n + n_{in} + 1$ counts the total number of input dimensions, including the recurrent inputs $\mathbf{a}^{(t-1)} \in \mathbb{R}^n$, task inputs $\mathbf{x}^{(t)} \in \mathbb{R}^{n_{\text{in}}}$, and an additional input clamped to 1 (to represent bias). For some initial state

1. An element of $\mathbb{R}^n$ is a column vector unless it appears in the denominator of a derivative; $\partial(\cdot)/\partial\mathbf{a}$ would be a row vector.

$\mathbf{a}^{(0)}$, $F_{\mathbf{w}}$ defines the network dynamics by

$$\mathbf{a}^{(t)} = F_{\mathbf{w}}(\mathbf{a}^{(t-1)}, \mathbf{x}^{(t)}).$$

At each time step an output $\mathbf{y}^{(t)} \in \mathbb{R}^{n_{\text{out}}}$ is computed by another function $F^{\text{out}}_{\mathbf{w}_{\text{o}}} : \mathbb{R}^{n} \to \mathbb{R}^{n_{out}}$, parameterized by $\mathbf{w}_{\text{o}} \in \mathbb{R}^{P_{\text{o}}}$. We will typically choose an affine-softmax readout for $F^{\text{out}}_{\mathbf{w}_{\text{o}}}$, with output weights/bias $\mathbf{W}^{\text{out}} \in \mathbb{R}^{n_{\text{out}} \times (n+1)}$. A loss function $L(\mathbf{y}^{(t)}, \mathbf{y}^{*(t)})$ calculates an instantaneous loss $L^{(t)}$, quantifying to what degree the predicted output $\mathbf{y}^{(t)}$ matches the target output $\mathbf{y}^{*(t)}$.

The goal is to train the network by gradient descent (or other gradient-based optimizers such as ADAM from Kingma and Ba, 2014) on the total loss $\mathcal{L} = \sum L^{(t)}$ w.r.t. the parameters $\mathbf{w}$ and $\mathbf{w}_{\text{o}}$. It is natural to learn $\mathbf{w}_{\text{o}}$ online, because only information at present time t is required to calculate the gradient $\partial L^{(t)}/\partial \mathbf{w}_{\text{o}}$. So the heart of the problem is to calculate $\partial \mathcal{L}/\partial \mathbf{w}$.

The parameter $\mathbf{w}$ is applied via $F_{\mathbf{w}}$ at every time step, and we denote a particular application of $\mathbf{w}$ at time s as $\mathbf{w}^{(s)}$.[2] Of course, a recurrent system is constrained to share parameters across time steps, so a perturbation $\delta\mathbf{w}$ is effectively a perturbation across all applications $\delta\mathbf{w}^{(s)}$, i.e., $\partial \mathbf{w}^{(s)}/\partial \mathbf{w} = \mathbf{I}_P$. In principle, each application of the parameters affects all future losses $L^{(t)}$, $t \geq s$. The core of any recurrent learning algorithm is to estimate the influence $\partial L^{(t)}/\partial \mathbf{w}^{(s)}$ of one parameter application $\mathbf{w}^{(s)}$ on one loss $L^{(t)}$, since these individual terms are necessary and sufficient to define the global gradient

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \sum_t \frac{\partial L^{(t)}}{\partial \mathbf{w}} = \sum_t \sum_{s \leq t} \frac{\partial L^{(t)}}{\partial \mathbf{w}^{(s)}} \frac{\partial \mathbf{w}^{(s)}}{\partial \mathbf{w}} = \sum_t \sum_{s \leq t} \frac{\partial L^{(t)}}{\partial \mathbf{w}^{(s)}}. \tag{1}$$

This raises the question of how to sum these components to produce individual gradients to pass to the optimizer. In truncated BPTT, one unrolls the graph over some range of time steps and sums $\partial L^{(t)}/\partial \mathbf{w}^{(s)}$ for all t, s in that range with $t \geq s$ (see §4.1.1). This does not qualify as an "online" learning rule, because it requires two independent time indices—at most one can represent "real time" leaving the other to represent the future or the past. If we can account for one of the summations via dynamic updates, then the algorithm is **online** or **temporally local**, i.e. not requiring explicit reference to the past or future. As depicted in Fig. 1, there are two possibilities. If t from Eq. (1) corresponds to real time, then the gradient passed to the optimizer is

$$\nabla_{\mathbf{w}} \mathcal{L}(t) = \sum_{s=0}^{t} \frac{\partial L^{(t)}}{\partial \mathbf{w}^{(s)}} = \frac{\partial L^{(t)}}{\partial \mathbf{w}}. \tag{2}$$

In this case, we say learning is **past facing**, because the gradient is a sum of the influences of *past* applications of $\mathbf{w}$ on the current loss. On the other hand, if s from Eq. (1) represents real time, then the gradient passed to the optimizer is

$$\nabla_{\mathbf{w}} \mathcal{L}(s) = \sum_{t=s}^{\infty} \frac{\partial L^{(t)}}{\partial \mathbf{w}^{(s)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(s)}}. \tag{3}$$

Here we say learning is **future facing**, because the gradient is a sum of influences by the current application of $\mathbf{w}$ on *future* losses.

2. Many authors use total and partial derivative operators to make this distinction when differentiating. For us, $d/d\mathbf{w} \to \partial/\partial \mathbf{w}$ and $\partial/\partial \mathbf{w} \to \partial/\partial \mathbf{w}^{(s)}$.

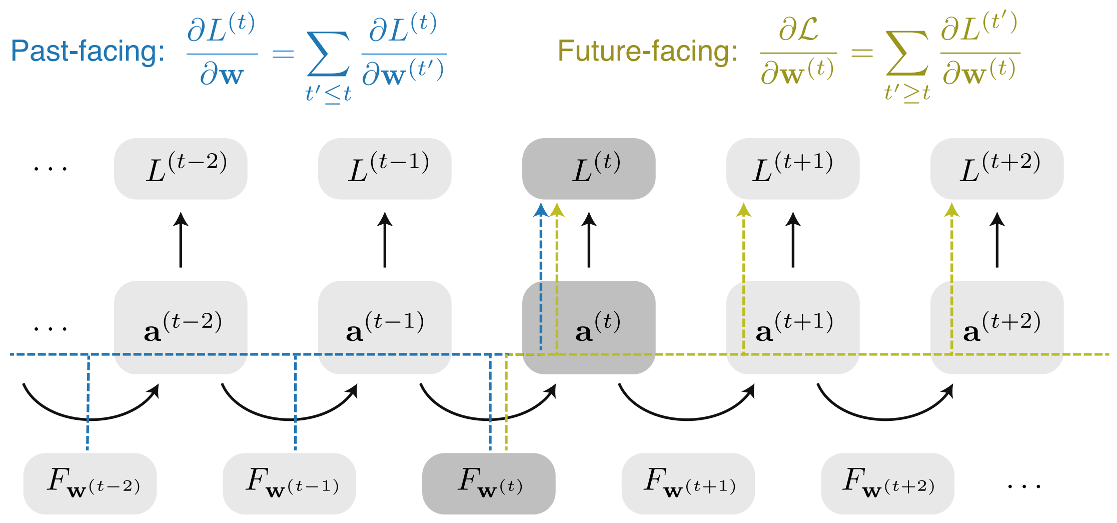
Past-facing: $\frac{\partial L^{(t)}}{\partial \mathbf{w}} = \sum_{t' \leq t} \frac{\partial L^{(t)}}{\partial \mathbf{w}^{(t')}}$
Future-facing: $\frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(t)}} = \sum_{t' \geq t} \frac{\partial L^{(t')}}{\partial \mathbf{w}^{(t)}}$
$L^{(t-2)}$
$L^{(t-1)}$
$L^{(t)}$
$L^{(t+1)}$
$L^{(t+2)}$
$\mathbf{a}^{(t-2)}$
$\mathbf{a}^{(t-1)}$
$\mathbf{a}^{(t)}$
$\mathbf{a}^{(t+1)}$
$\mathbf{a}^{(t+2)}$
$F_{\mathbf{w}^{(t-2)}}$
$F_{\mathbf{w}^{(t-1)}}$
$F_{\mathbf{w}^{(t)}}$
$F_{\mathbf{w}^{(t+1)}}$
$F_{\mathbf{w}^{(t+2)}}$

Figure 1: Cartoon depicting the past- and future-facing perspectives of online learning, for an RNN unrolled over time. Each $\mathbf{a}$ represents the RNN hidden state value, while $F_{\mathbf{w}}$ denotes applications of the recurrent update; the instantaneous losses L implicitly depend on the hidden state through $L^{(t)} = L\left(F^{\text{out}}_{\mathbf{w}_\text{o}}(\mathbf{a}^{(t)}), \mathbf{y}^{*(t)}\right)$. The blue (yellow) arrows show the paths of influence accounted for by the past-facing (future-facing) gradient described in the corresponding equation.

2.1. Past-Facing Online Learning Algorithms

Here we derive a fundamental relation leveraged by past-facing (PF) online algorithms. Let t index real time, and define the **influence matrix** $\mathbf{M}^{(t)} \in \mathbb{R}^{n \times P}$, where n and P are respectively the number of hidden units and the number of parameters defining $F_{\mathbf{w}}$. $\mathbf{M}^{(t)}$ tracks the derivatives of the current state $\mathbf{a}^{(t)}$ with respect to each parameter w_p:

$$M^{(t)}_{kp} = \frac{\partial a^{(t)}_k}{\partial w_p}. \tag{4}$$

Let's rewrite Eq. (4) with matrix notation and unpack it by one time step:

$$\begin{aligned}
\mathbf{M}^{(t)} = \frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{w}} &= \sum_{s \leq t} \frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{w}^{(s)}} = \sum_{s \leq t-1} \frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{w}^{(s)}} + \frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{w}^{(t)}} \\
&= \sum_{s \leq t-1} \frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{a}^{(t-1)}} \frac{\partial \mathbf{a}^{(t-1)}}{\partial \mathbf{w}^{(s)}} + \frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{w}^{(t)}} \\
&= \frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{a}^{(t-1)}} \frac{\partial \mathbf{a}^{(t-1)}}{\partial \mathbf{w}} + \frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{w}^{(t)}} \\
&\equiv \mathbf{J}^{(t)} \mathbf{M}^{(t-1)} + \overline{\mathbf{M}}^{(t)}.
\end{aligned} \tag{5}$$

A simple recursive formula emerges, wherein the influence matrix is updated by multiplying its current value by the Jacobian $\mathbf{J}^{(t)} = \partial \mathbf{a}^{(t)} / \partial \mathbf{a}^{(t-1)} \in \mathbb{R}^{n \times n}$ of the network and then adding the **immediate influence** $\overline{\mathbf{M}}^{(t)} = \partial \mathbf{a}^{(t)}/\partial \mathbf{w}^{(t)} \in \mathbb{R}^{n\times P}$. To compute the gradient that ultimately gets passed to the optimizer, we simply use the chain rule over the current hidden state $\mathbf{a}^{(t)}$:

$$\frac{\partial L^{(t)}}{\partial \mathbf{w}} = \frac{\partial L^{(t)}}{\partial \mathbf{a}^{(t)}}\frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{w}} \equiv \overline{\mathbf{c}}^{(t)}\mathbf{M}^{(t)}, \tag{6}$$

where the **immediate credit assignment vector** $\overline{\mathbf{c}}^{(t)} \in \mathbb{R}^n$ is defined to be $\partial L^{(t)}/\partial \mathbf{a}^{(t)}$ and is calculated by backpropagating the error $\boldsymbol{\delta}^{(t)}$ through the derivative of the output function $F^{\text{out}}_{\mathbf{w}_\text{o}}$ (or approximated by Feedback Alignment, see Lillicrap et al., 2016). In the end, we compute a derivative in Eq. (6) that is implicitly a sum over the many terms of Eq. (2), using formulae that depend explicitly only on times t and $t-1$. For this reason, such a learning algorithm is **online**, and it is **past facing** because the gradient computation is of the form in Eq. (2).

2.2. Future-Facing Online Learning Algorithms

Here we show a symmetric relation for future-facing (FF) online algorithms. The **credit assignment vector** $\mathbf{c}^{(t)} \in \mathbb{R}^n$ is a row vector defined as the gradient of the loss $\mathcal{L}$ with respect to the hidden state $\mathbf{a}^{(t)}$. It plays a role analogous to $\mathbf{M}^{(t)}$ and has a recursive update similar to Eq. (5):

$$\begin{aligned}\mathbf{c}^{(t)} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(t)}} &= \sum_{s\geq t}\frac{\partial L^{(s)}}{\partial \mathbf{a}^{(t)}} = \frac{\partial L^{(t)}}{\partial \mathbf{a}^{(t)}} + \sum_{s\geq t+1}\frac{\partial L^{(s)}}{\partial \mathbf{a}^{(t)}}\\ &= \frac{\partial L^{(t)}}{\partial \mathbf{a}^{(t)}} + \sum_{s\geq t+1}\frac{\partial L^{(s)}}{\partial \mathbf{a}^{(t+1)}}\frac{\partial \mathbf{a}^{(t+1)}}{\partial \mathbf{a}^{(t)}}\\ &= \frac{\partial L^{(t)}}{\partial \mathbf{a}^{(t)}} + \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(t+1)}}\frac{\partial \mathbf{a}^{(t+1)}}{\partial \mathbf{a}^{(t)}}\\ &= \overline{\mathbf{c}}^{(t)} + \mathbf{c}^{(t+1)}\mathbf{J}^{(t+1)}.\end{aligned} \tag{7}$$

As in the PF case, the gradient is ultimately calculated using the chain rule over $\mathbf{a}^{(t)}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(t)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(t)}}\frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{w}^{(t)}} \equiv \mathbf{c}^{(t)}\overline{\mathbf{M}}^{(t)}. \tag{8}$$

The recursive relations for PF and FF algorithms are of identical form given the following changes: (1) swap the roles of $\mathcal{L}$ and $\mathbf{w}$, (2) swap the roles of $t-1$ and $t+1$, and (3) flip the direction of all derivatives. This clarifies the fundamental trade-off between the PF and FF approaches to online learning. On the one hand, memory requirements favor FF because $\mathcal{L}$ is a scalar while $\mathbf{w}$ is a matrix. On the other, only PF can truly be run online, because the time direction of the update in FF is opposite the forward pass. Thus, efficient PF algorithms must *compress* $\mathbf{M}^{(t)}$, while efficient FF algorithms must *predict* $\mathbf{c}^{(t+1)}$.

3. Past-Facing Algorithms

3.1. Real-Time Recurrent Learning

The Real-Time Recurrent Learning (RTRL, Williams and Zipser, 1989) algorithm directly applies Eqs. (5) and (6) as written. We call the application of Eq. (5) the "update" to the learning algorithm, which is **deterministic** and in **closed form**. Implementing Eq. (5) requires storing $nP \approx \mathcal{O}(n^3)$ floats in $\mathbf{M}^{(t)}$ and performing $\mathcal{O}(n^4)$ multiplications in $\mathbf{J}^{(t)}\mathbf{M}^{(t)}$, which is neither especially efficient nor biologically plausible. However, several efficient (and in some cases, biologically plausible) online learning algorithms have recently been developed, including Unbiased Online Recurrent Optimization (UORO; Tallec and Ollivier, 2017; §3.2), Kronecker-Factored RTRL (KF-RTRL; Mujika et al., 2018; §3.3), Kernel RNN Learning (KeRNL; Roth et al., 2019; §3.5), and Random-Feedback Online Learning (RFLO; Murray, 2019; §3.6). We claim that these learning algorithms, whether explicitly derived as such or not, are all implicitly approximations to RTRL, each a special case of a general class of techniques for compressing $\mathbf{M}^{(t)}$. In the following section, we clarify how each of these learning algorithms fits into this broad structure.

3.1.1. Approximations to RTRL

To concretely illuminate these ideas, we will work with a special case of $F_{\mathbf{w}}$, a time-continuous vanilla RNN:

$$\mathbf{a}^{(t)} = F_{\mathbf{w}}(\mathbf{a}^{(t-1)}, \mathbf{x}^{(t)}) = (1-\alpha)\mathbf{a}^{(t-1)} + \alpha\phi(\mathbf{W}\hat{\mathbf{a}}^{(t-1)}), \tag{9}$$

where $\hat{\mathbf{a}}^{(t-1)} = \text{concat}(\mathbf{a}^{(t-1)}, \mathbf{x}^{(t)}, 1) \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{n\times m}$, $\phi : \mathbb{R}^n \to \mathbb{R}^n$ is some point-wise nonlinearity (e.g. tanh), and $\alpha \in (0, 1]$ is the network's inverse time constant. The trainable parameters w_p are folded via the indexing $p = i \times n + j$ into the weight matrix W_{ij}, whose columns hold the recurrent weights, the input weights, and a bias. By reshaping w_p into its natural matrix form W_{ij}, we can write the influence matrix as an order-3 **influence tensor**

$$M_{kij}^{(t)} = \partial a_k^{(t)} / \partial W_{ij}.$$

Thus $M_{kij}^{(t)}$ specifies the effect on the k-th unit of perturbing the direct connection from the j-th unit to the i-th unit. The immediate influence can also be written as a tensor. By differentiating Eq. (9), we see it takes the sparse form

$$\overline{M}_{kij}^{(t)} = \partial a_k^{(t)} / \partial W_{ij}^{(t)} = \alpha\delta_{ki}\phi'(h_i^{(t)})\hat{a}_j^{(t-1)},$$

because W_{ij} can affect the k-th unit directly only if $k = i$. Many approximations of RTRL involve a decomposition of $M_{kij}^{(t)}$ into a product of lower-order tensors. For example, UORO represents $M_{kij}^{(t)}$ by an outer product $A_k^{(t)} B_{ij}^{(t)}$, which has a memory requirement of only $\mathcal{O}(n^2)$. Similarly, KF-RTRL uses a Kronecker-product decomposition $A_j^{(t)} B_{ki}^{(t)}$. We can generalize these cases into a set of six possible decompositions of $M_{kij}^{(t)}$ into products of lower-order tensors $A^{(t)}$ and $B^{(t)}$:

$$M_{kij}^{(t)} \approx \begin{cases} A_k^{(t)} B_{ij}^{(t)} & \text{UORO, §3.2} \\ A_j^{(t)} B_{ki}^{(t)} & \text{KF-RTRL, §3.3} \\ A_i^{(t)} B_{kj}^{(t)} & \text{``Reverse'' KF-RTRL, §3.4} \\ A_{ki}^{(t)} B_{ij}^{(t)} & \text{KeRNL/RFLO, §3.5/§3.6} \\ A_{kj}^{(t)} B_{ij}^{(t)} & \text{Unexplored} \\ A_{ki}^{(t)} B_{kj}^{(t)} & \text{Unexplored} \end{cases}.$$

Each such decomposition has a memory requirement of $\mathcal{O}(n^2)$. Of course, it is not sufficient to write down an idealized decomposition for a particular time point; there must exist some efficient way to *update* the decomposition as the network runs forwards. We now go through each algorithm and show the mathematical techniques used to derive update equations and categorize them by the criteria outlined in Table 1.

3.2. Unbiased Online Recurrent Optimization (UORO)

Tallec and Ollivier (2017) discovered a technique for approximating $\mathbf{M}^{(t)} \in \mathbb{R}^{n\times P}$ as an outer product $\mathbf{A}^{(t)}\mathbf{B}^{(t)}$, where $\mathbf{A}^{(t)} \in \mathbb{R}^{n\times 1}$ and $\mathbf{B}^{(t)} \in \mathbb{R}^{1\times P}$. The authors proved a crucial lemma (see Appendix A or Tallec and Ollivier, 2017) that gives, in closed form, an unbiased rank-1 estimate[3] of a given matrix over the choice of a random vector $\boldsymbol{\nu} \in \mathbb{R}^n$ with $\mathbb{E}[\nu_i\nu_j] \propto \delta_{ij}$ and $\mathbb{E}[\nu_i] = 0$. They leverage this result to derive a closed-form update rule for $\mathbf{A}^{(t)}$ and $\mathbf{B}^{(t)}$ at each time step, without ever having to explicitly (and expensively) calculate $\mathbf{M}^{(t)}$. We present an equivalent formulation in terms of tensor components, i.e.,

$$M_{kij}^{(t)} \approx A_k^{(t)} B_{ij}^{(t)},$$

where $B_{ij}^{(t)}$ represents the "rolled-up" components of $\mathbf{B}^{(t)}$, as in W_{ij} w.r.t. $\mathbf{w}$. Intuitively, the kij-th component of the influence matrix is constrained to be the product of the k-th unit's "sensitivity" $A_k^{(t)}$ and the ij-th parameter's "efficacy" $B_{ij}^{(t)}$. Eqs. (10) and (11) show the form of the update and why it is unbiased over $\boldsymbol{\nu}$, respectively:

$$\begin{aligned}
A_k^{(t)} B_{ij}^{(t)} &= \left(\rho_0 \sum_{k'} J_{kk'}^{(t)} A_{k'}^{(t-1)} + \rho_1 \nu_k\right)\left(\rho_0^{-1} B_{ij}^{(t-1)} + \rho_1^{-1}\sum_{k'} \nu_{k'} \overline{M}_{k'ij}^{(t)}\right) \\
&= \sum_{k'} J_{kk'}^{(t)} A_{k'}^{(t-1)} B_{ij}^{(t-1)} + \sum_{k'} \nu_k \nu_{k'} \overline{M}_{k'ij}^{(t)} \\
&\quad + \sum_{k'} \nu_{k'}\left[\rho_1\rho_0^{-1}\delta_{kk'} B_{ij}^{(t-1)} + \rho_0\rho_1^{-1}\overline{M}_{k'ij}^{(t)} \sum_{k''} J_{k'k''}^{(t)} A_{k''}^{(t-1)}\right] \qquad (10)
\end{aligned}$$

$$\begin{aligned}
\implies \mathbb{E}\left[A_k^{(t)} B_{ij}^{(t)}\right] &= \sum_{k'} J_{kk'}^{(t)} \mathbb{E}\left[A_{k'}^{(t-1)} B_{ij}^{(t-1)}\right] + \sum_{k'} \mathbb{E}[\nu_k\nu_{k'}] \overline{M}_{k'ij}^{(t)} \\
&\quad + \sum_{k'} \mathbb{E}[\nu_{k'}]\,(\text{cross terms}) \\
&= \sum_{k'} J_{kk'}^{(t)} M_{k'ij}^{(t-1)} + \sum_{k'} \delta_{kk'} \overline{M}_{k'ij}^{(t)} + \sum_{k'} 0 \times (\text{cross terms}) \\
&= \sum_{k'} J_{kk'}^{(t)} M_{k'ij}^{(t-1)} + \overline{M}_{kij}^{(t)} \\
&= M_{kij}^{(t)}. \qquad (11)
\end{aligned}$$

The cross terms vanish in expectation because $\mathbb{E}[\nu_k] = 0$. Thus, by induction over t, the estimate of $M_{kij}^{(t)}$ remains unbiased at every time step. The constants $\rho_0, \rho_1 \in \mathbb{R}^{>0}$ are

3. In this instance we mean "rank" in the classical linear algebra sense, not "tensor rank," which we refer to as "order."

chosen at each time step to minimize total variance of the estimate by balancing the norms of the cross terms. This algorithm's update is **stochastic** due to its reliance on the random vector $\boldsymbol{\nu}$, but it is in **closed form** because it has an explicit update formula (Eq. 10). Both its memory and computational complexity are $\mathcal{O}(n^2)$.

3.3. Kronecker-Factored RTRL (KF-RTRL)

Mujika et al. (2018) leverage the same lemma as in UORO, but using a decomposition of $\mathbf{M}^{(t)}$ in terms of a Kronecker product $\mathbf{A}^{(t)} \otimes \mathbf{B}^{(t)}$, where now $\mathbf{A}^{(t)} \in \mathbb{R}^{1\times m}$ and $\mathbf{B}^{(t)} \in \mathbb{R}^{n\times n}$. This decomposition is more natural, because the immediate influence $\overline{\mathbf{M}}^{(t)}$ factors *exactly* as a Kronecker product $\hat{\mathbf{a}}^{(t)} \otimes \mathbf{D}^{(t)}$ for vanilla RNNs, where $D^{(t)}_{ki} = \alpha\delta_{ki}\phi'(h^{(t)}_i)$. To derive the update rule for UORO, one must first generate a rank-1 estimate of $\overline{\mathbf{M}}^{(t)}$ as an intermediate step, introducing more variance, but in KF-RTRL, this step is unnecessary. In terms of components, the compression takes the form

$$M^{(t)}_{kij} \approx A^{(t)}_j B^{(t)}_{ki},$$

which is similar to UORO, up to a cyclic permutation of the indices. Given a sample $\boldsymbol{\nu} \in \mathbb{R}^2$ of only 2 i.i.d. random variables, again with $\mathbb{E}[\nu_i\nu_j] = \delta_{ij}$ and $\mathbb{E}[\nu_i] = 0$, the update takes the form shown in Eqs. (12) and (13):

$$\begin{aligned}
A^{(t)}_j &= \left(\nu_0\rho_0 A^{(t-1)}_j + \nu_1\rho_1\hat{a}^{(t-1)}_j\right) && (12)\\
B^{(t)}_{ki} &= \left(\nu_0\rho_0^{-1}\sum_{k'} J^{(t)}_{kk'}B^{(t-1)}_{k'i} + \nu_1\rho_1^{-1}\alpha\delta_{ki}\phi'(h^{(t)}_i)\right) && (13)\\
\implies A^{(t)}_j B^{(t)}_{ki} &= \nu_0^2\sum_{k'} J^{(t)}_{kk'}A^{(t-1)}_j B^{(t-1)}_{k'i} + \nu_1^2\alpha\delta_{ki}\phi'(h^{(t)}_i)\hat{a}^{(t-1)}_j + \text{cross-terms}\\
\implies \mathbb{E}\left[A^{(t)}_j B^{(t)}_{ki}\right] &= \sum_{k'} J^{(t)}_{kk'}\mathbb{E}\left[A^{(t-1)}_j B^{(t-1)}_{k'i}\right] + \alpha\delta_{ki}\phi'(h^{(t)}_i)\hat{a}^{(t-1)}_j\\
&= \sum_{kk'} J^{(t)}_{kk'}M^{(t-1)}_{k'ij} + \overline{M}^{(t)}_{kij}\\
&= M^{(t)}_{kij}.
\end{aligned}$$

As in UORO, the cross terms vanish in expectation, and the estimate is unbiased by induction over t. This algorithm's updates are also **stochastic** and in **closed form**. Its memory complexity is $\mathcal{O}(n^2)$, but its computation time is $\mathcal{O}(n^3)$ because of the matrix-matrix product in Eq. (13).

3.4. Reverse KF-RTRL (R-KF)

Our exploration of the space of different approximations naturally raises a question: is an approximation of the form

$$M^{(t)}_{kij} \approx A^{(t)}_i B^{(t)}_{kj} \qquad (14)$$

also possible? We refer to this method as "Reverse" KF-RTRL (R-KF) because, in matrix notation, this would be formulated as $\mathbf{M}^{(t)} \approx \mathbf{B}^{(t)} \otimes \mathbf{A}^{(t)}$, where $\mathbf{A}^{(t)} \in \mathbb{R}^{1\times n}$ and $\mathbf{B}^{(t)} \in$

$\mathbb{R}^{n\times m}$. We propose the following update for $A_i^{(t)}$ and $B_{kj}^{(t)}$ in terms of a random vector $\boldsymbol{\nu} \in \mathbb{R}^n$:

$$\begin{aligned}
A_i^{(t)} B_{kj}^{(t)} &= \left(\rho_0 A_i^{(t-1)} + \rho_1 \nu_i\right)\left(\rho_0^{-1}\sum_{k'} J_{kk'}^{(t)} B_{k'j}^{(t-1)} + \rho_1^{-1}\sum_{i'} \nu_{i'} \overline{M}_{ki'j}^{(t)}\right) \quad (15)\\
&= \sum_{k'} J_{kk'}^{(t)} A_i^{(t-1)} B_{k'j}^{(t-1)} + \sum_{i'} \nu_i \nu_{i'} \overline{M}_{ki'j}^{(t)} + \text{cross-terms}\\
\implies \mathbb{E}\left[A_i^{(t)} B_{kj}^{(t)}\right] &= \sum_{k'} J_{kk'}^{(t)} \mathbb{E}\left[A_i^{(t-1)} B_{k'j}^{(t-1)}\right] + \overline{M}_{kij}^{(t)}\\
&= \sum_{k'} J_{kk'}^{(t)} M_{k'ij}^{(t-1)} + \overline{M}_{kij}^{(t)}\\
&= M_{kij}^{(t)}. \quad (16)
\end{aligned}$$

Eq. (16) shows that this estimate is unbiased, using updates that are **stochastic** and in **closed form**, like its sibling algorithms. Its memory and computational complexity are $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively. R-KF is actually more similar to UORO than KF-RTRL, because $\overline{M}_{kij}^{(t)}$ does not naturally factor like Eq. (14), introducing more variance. Worse, it has the computational complexity of KF-RTRL due to the matrix-matrix multiplication in Eq. (15). KF-RTRL stands out as the most effective of these 3 algorithms, because it estimates $\mathbf{M}^{(t)}$ with the lowest variance due to its natural decomposition structure. (See Mujika et al., 2018 for variance calculations.)

3.4.1. Optimal Kronecker-Sum Approximation (OK)

We briefly mention an extension of KF-RTRL by Benzing et al. (2019), where the influence matrix is approximated not by 1 but rather a sum of r Kronecker products, or, in components

$$M_{kij}^{(t)} \approx \sum_{l=1}^{r} A_{lj}^{(t)} B_{lki}^{(t)}.$$

On the RTRL update, the k index of $B_{lki}^{(t)}$ is propagated forward by the Jacobian, and then the immediate influence—itself a Kronecker product—is added. Now $M_{kij}^{(t)}$ is approximated by $r+1$ Kronecker products

$$M_{kij}^{(t)} \approx \sum_{l=1}^{r} A_{lj}^{(t-1)} J_{kk'}^{(t)} B_{lk'i}^{(t-1)} + \alpha \hat{a}_j^{(t-1)} \delta_{ki} \phi'(h_i^{(t)}),$$

but the authors developed a technique to optimally reduce this sum back to r Kronecker products, keeping the memory complexity $\mathcal{O}(rn^2)$ and computational complexity $\mathcal{O}(rn^3)$ constant. This update is **stochastic** because it requires explicit randomness in the flavor of the above algorithms, and it is **numerical** because there is no closed form solution to the update. We leave the details to the original paper.

3.5. Kernel RNN Learning (KeRNL)

Roth et al. (2019) developed a learning algorithm for RNNs that is essentially a compression of the influence matrix of the form $M_{kij}^{(t)} \approx A_{ki} B_{ij}^{(t)}$. We will show that this algorithm is also an implicit approximation of RTRL, although the update rules are fundamentally different than those for UORO, KF-RTRL and R-KF. The **eligibility trace** $\mathbf{B}^{(t)} \in \mathbb{R}^{n \times m}$ updates by temporally filtering the immediate influences $\alpha\phi'(h_i^{(t)})\hat{a}_j^{(t-1)}$ with unit-specific, learnable timescales α_i:

$$B_{ij}^{(t)} = (1-\alpha_i) B_{ij}^{(t-1)} + \alpha\phi'(h_i^{(t)})\hat{a}_j^{(t-1)}. \tag{17}$$

The **sensitivity matrix** $\mathbf{A} \in \mathbb{R}^{n \times n}$ is chosen to approximate the multi-step Jacobian $\partial a_k^{(t)} / \partial a_i^{(t')}$ with help from the learned timescales:

$$\frac{\partial a_k^{(t)}}{\partial a_i^{(t')}} \approx A_{ki}(1-\alpha_i)^{(t-t')}. \tag{18}$$

We will describe how $\mathbf{A}$ is learned later, but for now we assume this approximation holds and use it to show how the KeRNL update is equivalent to that of RTRL. We have dropped the explicit time-dependence from $\mathbf{A}$, because it updates too slowly for Eq. (18) to be specific to any one time point. If we unpack this approximation by one time step, we uncover the consistency relation

$$A_{ki}(1-\alpha_i) \approx \sum_{k'} J_{kk'}^{(t)} A_{k'i}. \tag{19}$$

By taking $t = t'$ in Eq. (18) and rearranging Eq. (19), we see this approximation implicitly assumes both

$$A_{ki} \approx \begin{cases} \delta_{ki} \\ (1-\alpha_i)^{-1} \sum_{k'} J_{kk'}^{(t)} A_{k'i} \end{cases}. \tag{20}$$

Then the eligibility trace update effectively implements the RTRL update, assuming inductively that $M_{kij}^{(t-1)}$ is well approximated by $A_{ki} B_{ij}^{(t-1)}$:

$$\begin{aligned} A_{ki} B_{ij}^{(t)} &= A_{ki}\left[(1-\alpha_i) B_{ij}^{(t-1)} + \alpha\phi'(h_i^{(t)})\hat{a}_j^{(t-1)}\right] \\ &= A_{ki}(1-\alpha_i) B_{ij}^{(t-1)} + \alpha A_{ki} \phi'(h_i^{(t)})\hat{a}_j^{(t-1)} \\ &\approx \sum_{k'} J_{kk'}^{(t)} A_{k'i} B_{ij}^{(t-1)} + \alpha\delta_{ki}\phi'(h_i^{(t)})\hat{a}_j^{(t-1)} \\ &= \sum_{kk'} J_{kk'}^{(t)} M_{k'ij}^{(t-1)} + \overline{M}_{kij}^{(t)} \\ &= M_{kij}^{(t)}. \end{aligned} \tag{21}$$

In Eq. (21), we use each of the special cases from Eq. (20). Of course, the A_{ki} and α_i have to be learned, and Roth et al. (2019) use gradient descent to do so. We leave details to the original paper; briefly, they run in parallel a perturbed forward trajectory to estimate the LHS of Eq. (18) and then perform SGD on the squared difference between the LHS and RHS, giving gradients for A_{ki} and α_i.

KeRNL uses **deterministic** updates because it does not need explicit random variables. While the $B_{ij}^{(t)}$ update is in closed form via Eq. (17), the updates for A_{ki} and α_i are **numerical** because of the need for SGD to train them to obey Eq. (18). Both its memory and computational complexities are $\mathcal{O}(n^2)$.

3.6. Random-Feedback Online Learning (RFLO)

Coming from a computational neuroscience perspective, Murray (2019) developed a beautifully simple and biologically plausible learning rule for RNNs, which he calls Random-Feedback Online Learning (RFLO).[4] He formulates the rule in terms of an eligibility trace $B_{ij}^{(t)}$ that filters the non-zero immediate influence elements $\phi'(h_i^{(t)})\hat{a}_j^{(t-1)}$ by the network inverse time constant α:

$$B_{ij}^{(t)} = (1-\alpha)B_{ij}^{(t-1)} + \alpha\phi'(h_i^{(t)})\hat{a}_j^{(t-1)}.$$

Then the approximate gradient is ultimately calculated[5] as

$$\frac{\partial L^{(t)}}{\partial W_{ij}} \approx \overline{c}_i^{(t)} B_{ij}^{(t)}.$$

By observing that

$$\overline{c}_i^{(t)} B_{ij}^{(t)} = \sum_k \overline{c}_k^{(t)} \delta_{ki} B_{ij}^{(t)},$$

we see that RFLO is a special case of KeRNL, in which we fix $A_{ki} = \delta_{ki}$, $\alpha_i = \alpha$. Alternatively, and as hinted in the original paper, we can view RFLO as a special case of RTRL under the approximation $J_{kk'}^{(t)} \approx (1-\alpha)\delta_{kk'}$, because the RTRL update reduces to RFLO with $M_{kij}^{(t)} = \delta_{ki}B_{ij}^{(t)}$ containing $B_{ij}^{(t)}$ along the diagonals:

$$\begin{aligned} M_{kij}^{(t)} &= \sum_{k'} J_{kk'}^{(t)} M_{k'ij}^{(t-1)} + \overline{M}_{kij}^{(t)} \\ &= (1-\alpha)\sum_{k'} \delta_{kk'} M_{k'ij}^{(t-1)} + \overline{M}_{kij}^{(t)} \\ &= (1-\alpha)M_{kij}^{(t-1)} + \alpha\delta_{ki}\phi'(h_i^{(t)})\hat{a}_j^{(t-1)}. \end{aligned} \tag{22}$$

Fig. 2 illustrates how $\mathbf{B}^{(t)}$ is contained in the influence matrix $\mathbf{M}^{(t)}$. This algorithm's update is **deterministic** and in **closed form**, with memory and computational complexity $\mathcal{O}(n^2)$.

4. Future-Facing Algorithms

4.1. Backpropagation Through Time (BPTT)

For many applications, a recurrent network is unrolled only for some finite number of time steps, and backpropagation through time (BPTT) manifests as the computation of the sum

4. An essentially equivalent algorithm, termed *e-prop 1*, was applied to biologically inspired spiking networks in Bellec et al. (2019).

5. As the "random feedback" part of the name suggests, Murray goes a step further in approximating $\overline{c}_k^{(t)}$ by random feedback weights á la Lillicrap et al., 2016, but we assume exact feedback in this paper for easier comparisons with other algorithms.

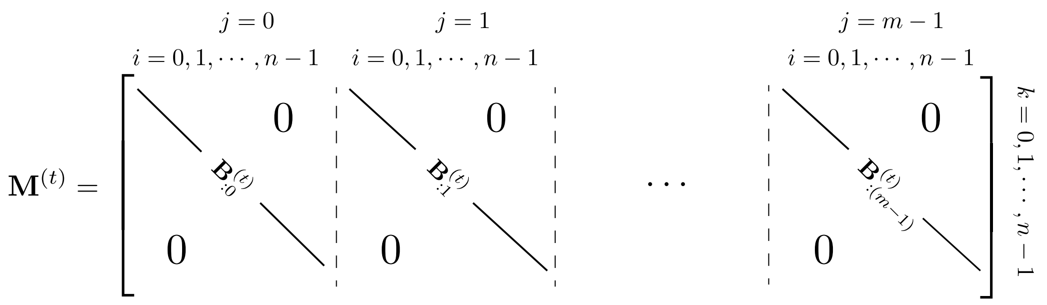

Figure 2: A visualization of the influence matrix and its 3 indices k, i, and j. In RFLO, the filtered immediate influences, stored in $B^{(t)}_{ij}$, sparsely populate the influence matrix along the diagonals.

$\partial L^{(t)}/\partial \mathbf{w}^{(s)}$ over every $s \leq t$ in the graph. This can be efficiently accomplished using

$$\mathbf{c}^{(t)} = \overline{\mathbf{c}}^{(t)} + \mathbf{c}^{(t+1)}\mathbf{J}^{(t+1)} \tag{23}$$

(see Eq. 7) to propagate credit assignment backwards. However, in our framework, where a network is run on an infinite-time horizon, there are two qualitatively different ways of unrolling the network. We call them "efficient" and "future-facing" BPTT.

4.1.1. Efficient Backpropagation Through Time (E-BPTT)

For this method, we simply divide the graph into non-overlapping segments of truncation length T and perform BPTT between $t - T$ and t as described above, using Eq. (23). It takes $\mathcal{O}(n^2T)$ computation time to compute one gradient, but since this computation is only performed once every T time steps, the computation time is effectively $\mathcal{O}(n^2)$, with memory requirement $\mathcal{O}(nT)$. A problem with this approach is that it does not treat all time points the same: an application of $\mathbf{w}$ occurring near the end of the graph segment has less of its future influence accounted for than applications of $\mathbf{w}$ occurring before it, as can be visualized in Fig. 3. And since any one gradient passed to the optimizer is a sum across both t and s, it is not an online algorithm by the framework we presented in §2. Therefore, for the purpose of comparing with online algorithms, we also show an alternative version of BPTT that calculates a future-facing gradient (up to truncation) $\partial \mathcal{L}/\partial \mathbf{w}^{(t)}$ for every t.

4.1.2. Future-Facing Backpropagation Through Time (F-BPTT)

In this version of BPTT, we keep a dynamic list of truncated credit assignment estimates $\hat{\mathbf{c}}^{(s)}$ for times $s = t - T, \cdots, t-1$:

$$\left[\hat{\mathbf{c}}^{(t-T)}, \cdots, \hat{\mathbf{c}}^{(t-1)}\right],$$

where each truncated credit assignment estimate includes the influences of $\mathbf{a}^{(s)}$ only up to time $t - 1$:

$$\hat{\mathbf{c}}^{(s)} = \sum_{t'=s}^{t-1} \frac{\partial L^{(t')}}{\partial \mathbf{a}^{(s)}}.$$

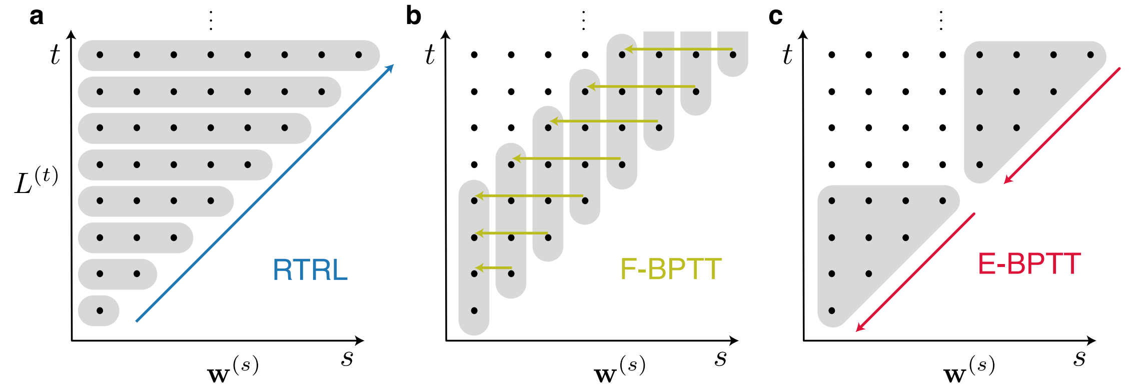

Figure 3: A visualization of various exact gradient methods. Each plot contains a lattice of points, representing derivatives $\partial L^{(t)}/\partial \mathbf{w}^{(s)}$ for $s \leq t$, with gray boxes representing individual gradients passed to the optimizer. **a)** RTRL sums these derivatives into gradients for fixed t, using the PF relation (Eq. 5, §2) to efficiently derive successive gradients (blue arrow). **b)** F-BPTT sums these derivatives into gradients for fixed s by backpropagating through time (yellow arrows). **c)** E-BPTT creates a triangular gradient for non-overlapping subgraphs, using the FF relation (Eq. 7, §2) for efficient computation (red arrows). Here, the truncation horizon is $T = 4$.

At current time t, every element $\hat{\mathbf{c}}^{(s)}$ is extended by adding $\partial L^{(t)}/\partial \mathbf{a}^{(s)}$, calculated by backpropagating from the current loss $L^{(t)}$, while the explicit credit assignment $\overline{\mathbf{c}}^{(t)}$ is appended to the front of the list. To compensate, the oldest credit assignment estimate $\hat{\mathbf{c}}^{(t-T)}$ is removed and combined with the immediate influence to form a (truncated) gradient

$$\hat{\mathbf{c}}^{(t-T)}\overline{\mathbf{M}}^{(t-T)} = \sum_{t'=t-T}^{t} \frac{\partial L^{(t')}}{\partial \mathbf{a}^{(t-T)}} \frac{\partial \mathbf{a}^{(t-T)}}{\partial \mathbf{w}^{(t-T)}} = \sum_{t'=t-T}^{t} \frac{\partial L^{(t')}}{\partial \mathbf{w}^{(t-T)}} \approx \frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(t-T)}},$$

which is passed to the optimizer to update the network. This algorithm is "online" in that it produces strictly future-facing gradients at each time step, albeit delayed by the truncation time T and requiring memory of the network states from $t - T$. Each update step requires $\mathcal{O}(n^2 T)$ computation, but since the update is performed at every time step, computation remains a factor of T more expensive than E-BPTT. Memory requirement is still $\mathcal{O}(nT)$. Fig. 3 illustrates the differences among these methods and RTRL, using a triangular lattice as a visualization tool. Each point in the lattice is one derivative $\partial L^{(t)}/\partial \mathbf{w}^{(s)}$ with $t \geq s$, and the points are grouped together into discrete gradients passed to the optimizer.

4.2. Decoupled Neural Interfaces (DNI)

Jaderberg et al. (2017) developed a framework for online learning by **predicting** credit assignment. Whereas PF algorithms face the problem of a large influence tensor $M_{kij}^{(t)}$ that needs a compressed representation, FF algorithms face the problem of incomplete information: at time t, it is impossible to calculate $\mathbf{c}^{(t)}$ without access to future network variables. The approach of Decoupled Neural Interfaces (DNI) is to simply make a linear prediction of $\mathbf{c}^{(t)}$ (Czarnecki et al., 2017) based on the current hidden state $\mathbf{a}^{(t)}$ and the current labels $\mathbf{y}^{*(t)}$:

$$c_i^{(t)} \approx \sum_l \tilde{a}_l^{(t)} A_{li},$$

where $\tilde{\mathbf{a}}^{(t)} = \text{concat}(\mathbf{a}^{(t)}, \mathbf{y}^{*(t)}, 1) \in \mathbb{R}^{m'}$, $m' = n + n_{\text{out}} + 1$, and A_{li} are the components of a matrix $\mathbf{A} \in \mathbb{R}^{m' \times n}$, which parameterizes what the authors call the **synthetic gradient** function. The parameters A_{li} are trained to minimize the loss

$$L_{\text{SG}}^{(t)} = \frac{1}{2} \left\| \sum_l \tilde{a}_l^{(t)} A_{li} - c_i^{(t)} \right\|^2 \tag{24}$$

via gradient descent, similar to KeRNL's treatment of A_{ki} and α_i (and we drop the time dependence of A_{li} for the same reason). Of course, this begs the question—the whole point is to avoid calculating $\mathbf{c}^{(t)}$ explicitly, but calculating the error in Eq. (24) requires access to $\mathbf{c}^{(t)}$. So the authors propose a "bootstrapping" technique analogous to the Bellman equation in Reinforcement Learning (Sutton and Barto, 2018). If we take the FF relation we derived in Eq. (7)

$$\mathbf{c}^{(t)} = \overline{\mathbf{c}}^{(t)} + \mathbf{c}^{(t+1)} \mathbf{J}^{(t+1)} \tag{25}$$

and approximate the appearance of $\mathbf{c}^{(t+1)}$ with the synthetic gradient estimate $\tilde{\mathbf{a}}^{(t+1)} \mathbf{A}$, then Eq. (25) provides an estimate of $c_i^{(t)}$ to use in Eq. (24). Then the update for $\mathbf{A}$ can be written as

$$\Delta A_{li} \propto -\tilde{a}_l^{(t)} \left[\sum_{l'} \tilde{a}_{l'}^{(t)} A_{l'i} - \left(\overline{c}_i^{(t)} + \sum_m \sum_{l'} \tilde{a}_{l'}^{(t+1)} A_{l'm} J_{mi}^{(t+1)} \right) \right] \tag{26}$$

with learning rate chosen as a hyperparameter. As in Eq. (8), the gradient is calculated by combining the estimated credit assignment for the i-th unit with the explicit influence by the ij-th parameter:

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{(t)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(t)}} \frac{\partial a_i^{(t)}}{\partial W_{ij}^{(t)}} = \overline{c}_i^{(t)} \phi'(h_i^{(t)}) \hat{a}_j^{(t-1)} \approx \left(\sum_l \tilde{a}_l^{(t)} A_{li} \right) \phi'(h_i^{(t)}) \hat{a}_j^{(t-1)}$$

This algorithm is **future facing** because it ultimately estimates the effect of applying $\mathbf{w}$ at current time t on *future* losses. Its updates are **deterministic**, because no explicit randomness is required, and **numerical**, because the minimization problem over A_{li} implied by Eq. (24) is approximated via gradient descent rather than solved in closed form. It requires $\mathcal{O}(n^2)$ memory for $\mathbf{A}$ and $\mathcal{O}(n^2)$ computation for the matrix-vector multiplications in Eq. (26).

4.2.1. Biological Approximation to DNI

While many of the algorithms we have presented are biologically plausible in the abstract, i.e. temporally/spatially local and requiring no more than $\mathcal{O}(n^2)$ memory, we have not yet discussed any explicit biological implementations. There are a handful of additional considerations for evaluating an algorithm with respect to biological plausibility:

i Any equation describing synaptic strength changes (weight updates) must be **local**, i.e. any variables needed to update a synapse connecting the i-th and j-th units must be physically available to those units.

ii Matrix-vector multiplication can be implemented by network-wide neural transmission, but input vectors must represent **firing rates** (e.g. post-activations $\mathbf{a}$) and not membrane potentials (e.g. pre-activations $\mathbf{h}$), since inter-neuron communication is mediated by spiking.

iii **Feedback weights** used to calculate $\bar{\mathbf{c}}$ cannot be perfectly symmetric with $\mathbf{W}^{\text{out}}$, since there is no evidence for biological weight symmetry (see Lillicrap et al., 2016).

iv Matrices (e.g. $\mathbf{J}$ or $\mathbf{A}$) must represent a set of synapses, whose strengths are determined by some local update.

With a few modifications, many of the presented algorithms can satisfy these requirements. We briefly illustrate one particular case with DNI, as shown in Marschall et al. (2019). To address (i), the result of the synthetic gradient operation $\sum_l \tilde{a}_l^{(t)} A_{li}$ can be stored in an electrically isolated neural compartment, in a manner similar to biological implementations of feedforward backpropagation (Guerguiev et al., 2017; Sacramento et al., 2018), to allow for local updates to W_{ij}. For (ii), simply pass the bootstrapped estimate of $\bar{\mathbf{c}}^{(t+1)}$ from Eq. (26) through the activation function ϕ so that it represents a neural firing rate. For (iii), one can use fixed, random feedback weights $\mathbf{W}^{\text{fb}}$ instead of the output weights to calculate $\bar{\mathbf{c}}^{(t)}$, as in Lillicrap et al. (2016). And for (iv), one can train a set of weights $\mathcal{J}_{ij}$ online to approximate the Jacobian by performing SGD on $L_J^{(t)} = \left\| a_i^{(t)} - \sum_j \mathcal{J}_{ij} a_j^{(t-1)} \right\|^2$, which encodes the error of the linear prediction of the next network state by $\mathcal{J}_{ij}$. The update rule manifests as

$$\Delta \mathcal{J}_{ij} \propto -\left(a_i^{(t)} - \sum_{j'} \mathcal{J}_{ij'} a_{j'}^{(t-1)} \right) a_j^{(t-1)},$$

essentially a “perceptron” learning rule, which is local and biologically realistic. Although this approximation brings no traditional computation speed benefits, it offers a plausible mechanism by which a neural circuit can access its own Jacobian for learning purposes. This technique could be applied to any other algorithm discussed in this paper. We refer to this altered version of DNI as DNI(b) in the experiments section.

5. Experiments

The ultimate goal of empirical evaluation of different algorithms would be to assess their performance in real-world tasks. However, a full algorithmic comparison in large real-world problems would likely require a complex architecture (LSTM/GRU), batching, and sophisticated optimization. For each algorithm, each of these choices and associated hyperparameters would have to be individually optimized in a principled way for a fair comparison, which goes beyond the scope of this paper. Instead, we use the simplest possible setup for RNN learning, i.e. the simplest architecture (vanilla RNN), a true online setting (batch size 1), and basic SGD optimization, so that we can focus on the effects of different gradient approximations, without having to consider how they interact with all the additional techniques required to make learning work in real-world problems. This setup restricts us to simple tasks, but allows us to systematically investigate how key task features, such as time horizon and dimensionality, affect the comparison outcome.

5.1. Setup

We evaluate each algorithm's ability to learn two different synthetic tasks: an **additive dependencies** task ("Add") borrowed from Pitis (2016) and a **mimic target RNN** task ("Mimic"). Each of these tasks is strictly online, i.e. a continuous sequence of data, as opposed to discrete subsequences presented in batches. Each admits a parametrizable notion of difficulty, either time horizon of dependencies (in Add) or computational complexity (in Mimic).

In the Add task, a stream of i.i.d. Bernoulli inputs $x^{(t)} \in \{0,1\}$ is provided to the RNN. The label $y^{*(t)}$ has a baseline value of 0.5 that increases (or decreases) by 0.5 (or 0.25) if $x^{(t-t_1)} = 1$ (or $x^{(t-t_2)} = 1$), for specified lags t_1 and t_2. This task is similar in spirit to the more popular "Adding Problem" from Hochreiter and Schmidhuber (1997), but our Add task is truly online and can be learned by vanilla RNNs. Task difficulty is naturally parameterized by the two delays; here we chose to incrementally vary t_1, with fixed $t_2 - t_1$.

In the Mimic task, the inputs are multi-dimensional i.i.d. Bernoulli inputs $\mathbf{x}^{(t)} \in \{0,1\}^{n_{\text{in}}}$, with labels $\mathbf{y}^{*(t)} \in \mathbb{R}^{n_{\text{out}}}$ determined by the outputs of an untrained target RNN (with n_{h} hidden units and randomly generated, unitary recurrent weight matrix) that is fed the same input stream $\left\{\mathbf{x}^{(t')} : t' \leq t\right\}$. We typically use $n_{\text{in}} = n_{\text{out}} = 32$ in Mimic, chosen so that learning $\mathbf{W}$ is necessary for strong performance. We parameterically vary n_{h} to modulate task difficulty—a target RNN with more hidden units will induce more complex dependencies between $\mathbf{x}$ and $\mathbf{y}^*$.

For each task, we consider two versions on different time scales: when the network is perfectly discrete ($\alpha = 1$, see Eq. 9) and when the network update has some time continuity ($\alpha = 0.5$). For the $\alpha = 0.5$ case, the tasks are stretched over time by a factor of 2 to compensate (i.e. each input/label is duplicated for one time step).

We implement each algorithm in a custom NumPy-based Python module.[6] We use gradient descent with a learning rate of 10^{-4}, the fastest learning rate for which all algorithms are able to converge to a stable steady-state performance. We restrict ourselves to using a batch size of 1, because, in an online setting, the network must learn as data arrive in real time. Most algorithms demand additional configuration decisions and hyperparameter choices: the truncation horizon T (F-BPTT), the initial values of the tensors $\mathbf{A}$ and $\mathbf{B}$ (all approximations), the initial values of the learned timescales α_i (KeRNL), the distribution from which $\boldsymbol{\nu}$ is sampled for stochastic updates (UORO, KF-RTRL, R-KF), and the learning rate for the numerical updates (KeRNL, DNI). For each algorithm, we independently optimize these choices by hand (see Appendix B for details).

6. Link to public code repository: `https://github.com/omarschall/vanilla-rtrl/tree/release`

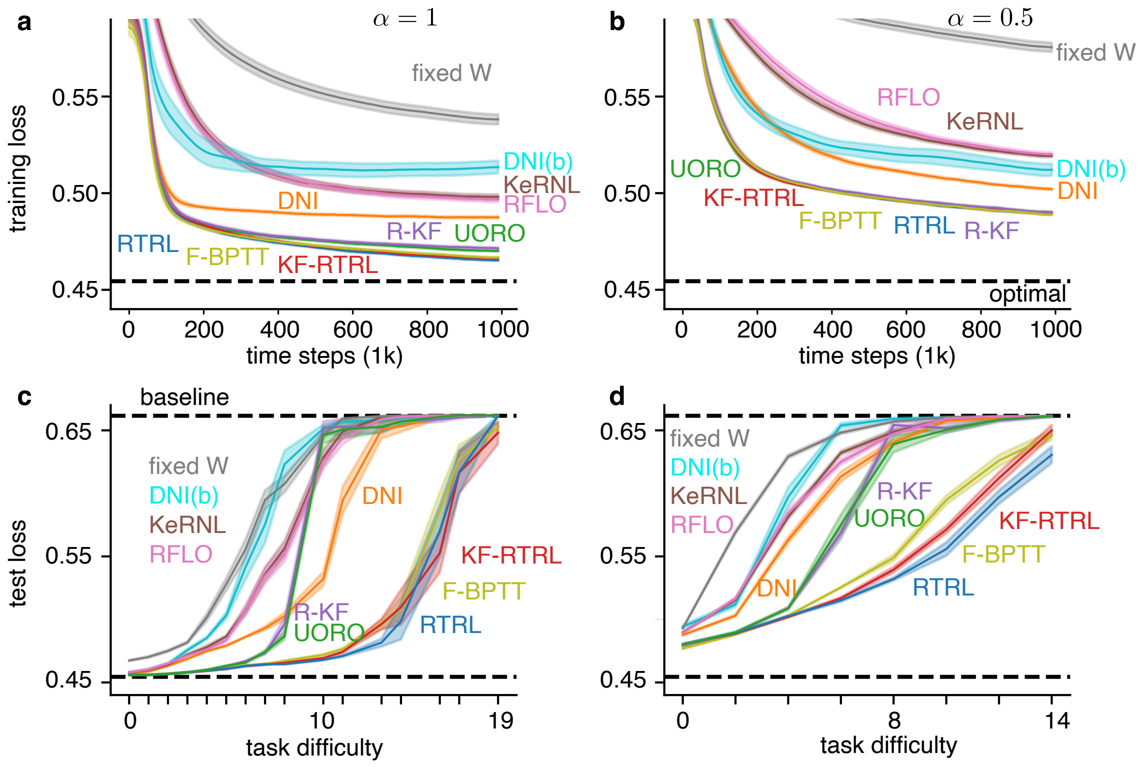

Figure 4: **a)** Cross-entropy loss for networks trained on Add task ($t_1 = 5, t_2 = 9$) with $\alpha = 1$ for various algorithms. Lines are means over 24 random seeds (weight initialization and training set generation), and shaded regions represent ± 1 S.E.M. Raw loss curves are first down-sampled by a factor of 10^{-4} (rectangular kernel) and then smoothed with a 10-time-step windowed running average. **b)** Same for $\alpha = 0.5$ and $t_1 = 2, t_2 = 4$. **c)** Test loss after training runs for the Add task with $\alpha = 1$ for a range of task difficulties, where "difficulty" is quantified as the horizon of the first dependency t_1, with $t_2 = t_1 + 2$. We used 12 random seeds per algorithm-difficulty pair. **d)** Same for $\alpha = 0.5$.

5.2. Add Task: Results and Analysis

Fig. 4 shows the performance of each learning algorithm on the Add task, in both $\alpha = 1$ and $\alpha = 0.5$ conditions. We also include a **fixed W** algorithm, where learning is restricted to the subset of parameters $\mathbf{W}^{\text{out}}$, as baseline. As expected, since they compute exact gradients up to truncation, RTRL and F-BPTT perform best, although KF-RTRL is a sufficiently robust approximation to virtually tie with them. All three approach the theoretical optimum, defined as the cross-entropy when the outputs perfectly match the labels.[7]

7. Counterintuitively, this is nonzero because the labels are *not* binary, so e.g. an output of 0.75 with a label of 0.75 is a perfect match, but carries a cross-entropy loss of $-0.75\log(0.75) - 0.25\log(0.25) = 0.56$.

R-KF and UORO perform similarly and worse than KF-RTRL does, as expected, since these approximations carry significantly more variance than KF-RTRL. However, in the $\alpha = 0.5$ condition, their performance is similar to that of KF-RTRL.

KeRNL and RFLO also cluster in performance across α conditions. KeRNL is theoretically a stronger approximation of RTRL than RFLO, because of its ability to learn optimal A_{ki} and α_i whereas RFLO has fixed $A_{ki} = \delta_{ki}$ and $\alpha_i = \alpha$. However, the numerical procedure for updating A_{ki} and α_i depends on several configuration/hyperparameter choices. Despite significant effort in exploring this space, we are not able to get KeRNL to perform better than RFLO, suggesting that the procedure for training A_{ki} and α_i does more harm than good in our setup (see Fig. S1 in Appendix C for a systematic analysis across tasks and α values). We have chosen a favorable setting for KeRNL, such that it performs almost as well as RFLO, but still learns A_{ki} values that are distinct from δ_{ki}. In the original paper, Roth et al. (2019) show promising results using RMSProp (Tieleman and Hinton, 2012) and batched training, which makes us suspect that the perturbation-based method for training A_{ki} is simply too noisy for online learning.

DNI sits somewhere between the RFLO-KeRNL cluster and the rest, with its biologically realistic sibling DNI(b) performing slightly worse than DNI, as to be expected, since it is an approximation on top of an approximation. As with KeRNL, DNI's numerical update of A_{li} introduces more hyperparameters and implementation choices, but there is a larger space of configurations in which the updates improve rather than hinder the algorithm's performance.

The above observations hold across task difficulties, as we vary t_1 with $t_2 = t_1 + 2$ (Fig. 4c,d). All algorithms perform near-optimally for easy versions of the task, while approaching the performance lower bound as t_1 increases. This bound is defined as the cross-entropy loss for a network that has learned the marginal output statistics but has no knowledge of the input dependencies. For a range of moderate difficulties the algorithms' performances are grouped similarly to Fig. 4a,b in both α settings.

5.3. Mimic Task: Results and Analysis

For the Mimic task (Fig. 5), we also see a clustering of the algorithms that reflects their conceptual similarity. For instance, stochastic algorithms UORO and R-KF perform similarly to each other (as in Add), and so do deterministic algorithms RFLO and KeRNL. However, the latter group performs *better* in Mimic than the former, unlike in Add.

Since in our setup all target networks have unitary weight matrices, Mimic has roughly the same time constant independent of task difficulty, as illustrated in the input-output cross-correlograms of the target networks (Fig. S2a,b in Appendix C). This time horizon is relatively short, as confirmed by the fact that the task can be learned to the same level of performance even with small BPTT truncation horizons (Fig. S2c,d in Appendix C). This explains why RFLO, KeRNL and DNI performance approaches that of BPTT and RTRL.

A noticeable difference is that stochastic algorithms perform systematically worse than they did on the Add task relative to deterministic ones. One potential explanation, in line with the above, is that when temporal demands of the task are small, the benefit of an unbiased gradient is outweighed by the drawback of gradient variance. It is also possible that the high task dimensionality of Mimic disproportionately affects the performance of

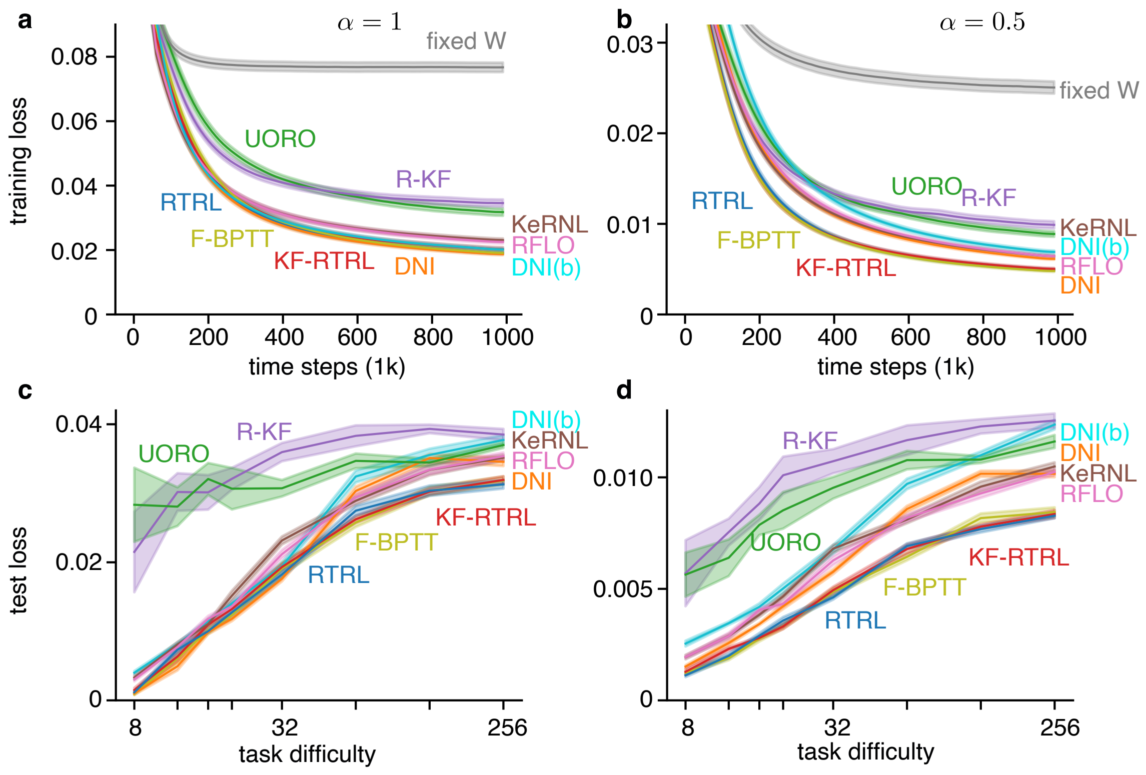

Figure 5: Same as Fig. 4, for Mimic task with mean-squared-error loss. Task difficulty is quantified as the number of hidden units n_h of the target RNN; $n_\text{h} = 32$ in a,b.

stochastic algorithms. In further experiments, we vary the input dimensionality, while duplicating inputs appropriately to keep $n_\text{in} = 32$, thereby fixing the total number of network parameters. Across latent dimensionalities, we find that UORO and R-KF perform significantly worse than all other algorithms. Only when latent dimensionality is small do they perform similarly (Fig. S3 in Appendix C). Our conjecture is that UORO and R-KF are more effective at maintaining information over time (cf. Add), but the variance of their gradients becomes problematic in high dimensions.

5.4. Gradient Similarity Analysis

We conduct an in-depth investigation looking beyond task accuracy by directly comparing the (approximate) gradients produced by each algorithm. Fig. 6 shows how a given pair of algorithms align on a time-step-by-time-step basis for the Add and Mimic tasks. Each subplot is a histogram giving the distribution of normalized dot products

$$\cos\left(\theta_{XY}^{(t)}\right) = \frac{\Delta_X^{(t)}\mathbf{W} \cdot \Delta_Y^{(t)}\mathbf{W}}{\left\|\Delta_X^{(t)}\mathbf{W}\right\| \left\|\Delta_Y^{(t)}\mathbf{W}\right\|} \tag{27}$$

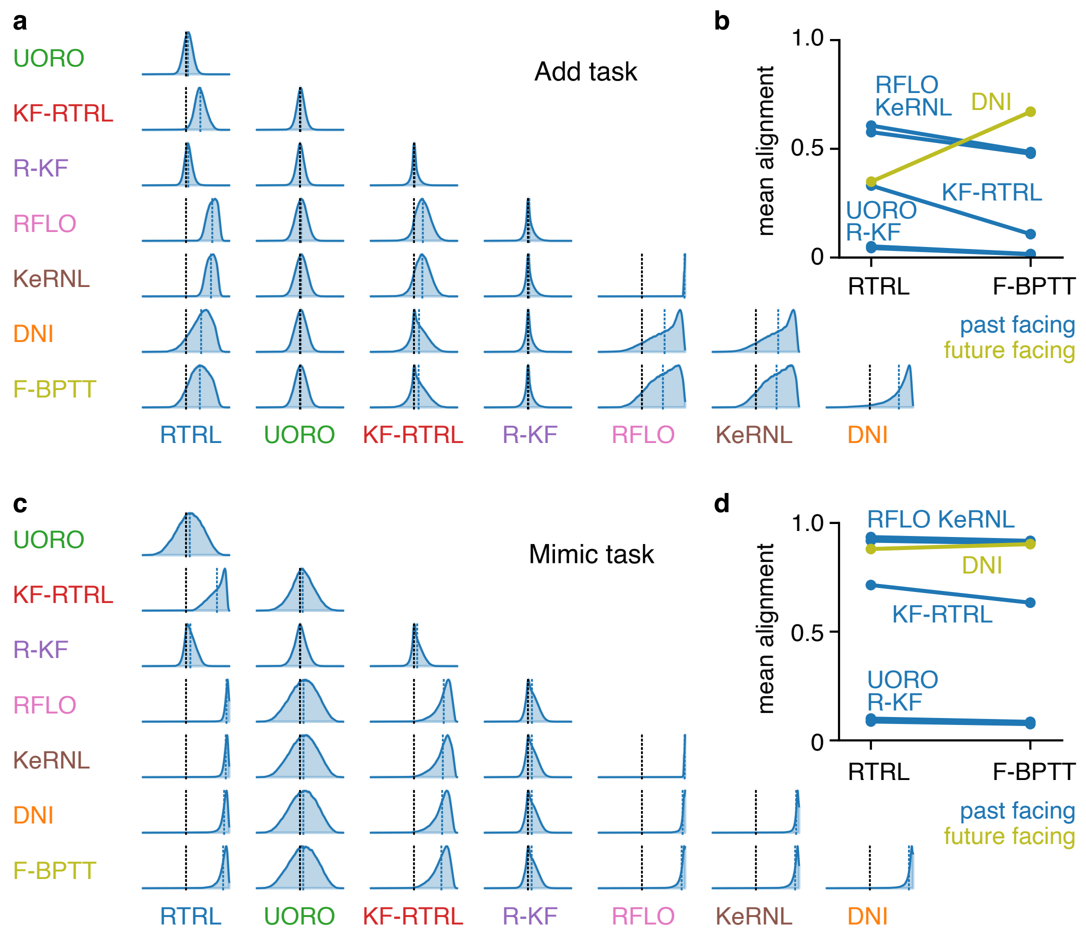

Figure 6: **a)** Histograms of normalized gradient alignments for each pair of algorithms. Gradients are calculated during a simulation of $100k$ time steps of the Add task (same hyperparameters as in Figs. 4a,b). Learning follows RTRL gradients, with other algorithms' gradients passively computed for comparison. Mean alignment (dashed blue line) and 0 alignment reference (dashed black line) shown. **b)** Mean alignments of each approximate algorithm with RTRL and F-BPTT, color-coded by past facing (UORO, KF-RTRL, R-KF, RFLO, KeRNL) vs. future facing (DNI). A single line segment corresponds to an approximate (i.e. non-F-BPTT, non-RTRL) algorithm, with its end points denoting mean alignment with each exact algorithm. **c)** Same as (a), for Mimic task, with hyperparameters as in Figs. 5a,b. **d)** Same as (b), for Mimic task.

for (flattened) weight updates $\Delta_X^{(t)}\mathbf{W}$ and $\Delta_Y^{(t)}\mathbf{W}$ prescribed by algorithms X and Y, respectively, at time t. Figs. 6a,c show qualitatively similar trends:

i As shown directly in Figs. 6b,d, PF algorithms (UORO, KF-RTRL, R-KF, RFLO, KeRNL) align better with RTRL than with F-BPTT, and vice versa for FF algorithms (DNI). This effect is subtler in Mimic—not unexpected since the time horizon required for solving the task is short, which obscures the distinction between PF and FF gradients.

ii The deterministic PF algorithms (RFLO and KeRNL) align better with RTRL than the stochastic algorithms (UORO, KF-RTRL, and R-KF) align with RTRL.

iii RFLO and KeRNL align more strongly with each other than any other pair (by construction since the learning rate for KeRNL is small).

iv UORO and R-KF do not align strongly with any other algorithms, despite their ability to train the RNN effectively. UORO and R-KF each have mean alignments with RTRL that are barely above 0 (Fig. 6b), yet they outperform other algorithms with much higher average alignment, in particular RFLO, KeRNL, and DNI.

Observations (i)–(iii) validate our categorizations, as similarity according to the normalized alignment corresponds to similarity by the past-facing/future-facing, tensor structure, and stochastic/deterministic criteria. Observation (iv) is puzzling, as it shows that angular alignment with exact algorithms is not predictive of learning performance.

How are UORO and R-KF able to learn at all if their gradients are almost orthogonal with RTRL on average? We address this question for both UORO and R-KF by examining the joint distribution of the gradient's alignment with RTRL and the gradient's norm (Fig. 7). All 4 cases show a statistically significant positive linear correlation between the normalized alignment and the common log of the gradient norm. This observation may partially explain (iv), because larger weight updates occur when UORO happens to strongly align with RTRL. However, these correlations are fairly weak even if statistically significant, and we argue that better algorithm similarity metrics are needed to account for observed differences in performance.

5.5. RFLO Analysis

Among all approximate algorithms, RFLO stands out as having the simplest tensor structure and update rule, and it has been empirically demonstrated to be able to train RNNs on long-term dependencies. This is such a severe approximation of RTRL, yet it works so well in practice—and there is no clear understanding why. Although Murray (2019) goes into detail showing how loss decreases despite the random feedback used to approximately calculate $\overline{\mathbf{c}}^{(t)}$, he does not address the more basic mystery of how RFLO is able to learn despite the significant approximation $\mathbf{J}^{(t)} \approx (1-\alpha)\mathbf{I}$. In this section, we provide some intuition for how this simple learning rule is so successful and empirically validate our claims.

We hypothesize that, rather than learning dynamics that actively retain useful bits of the past like RTRL and BPTT, RFLO works by training what is essentially a high-capacity feedforward model to predict labels from the **natural** memory traces of previous inputs contained in the hidden state. This is reminiscent of reservoir computing (Lukoševičius and Jaeger, 2009). We illustrate this idea in the special case of a perfectly discrete network ($\alpha = 1$), where the learning rule still performs remarkably well despite $B_{ij}^{(t)} = \phi'(h_i^{(t)})\hat{a}_j^{(t-1)}$

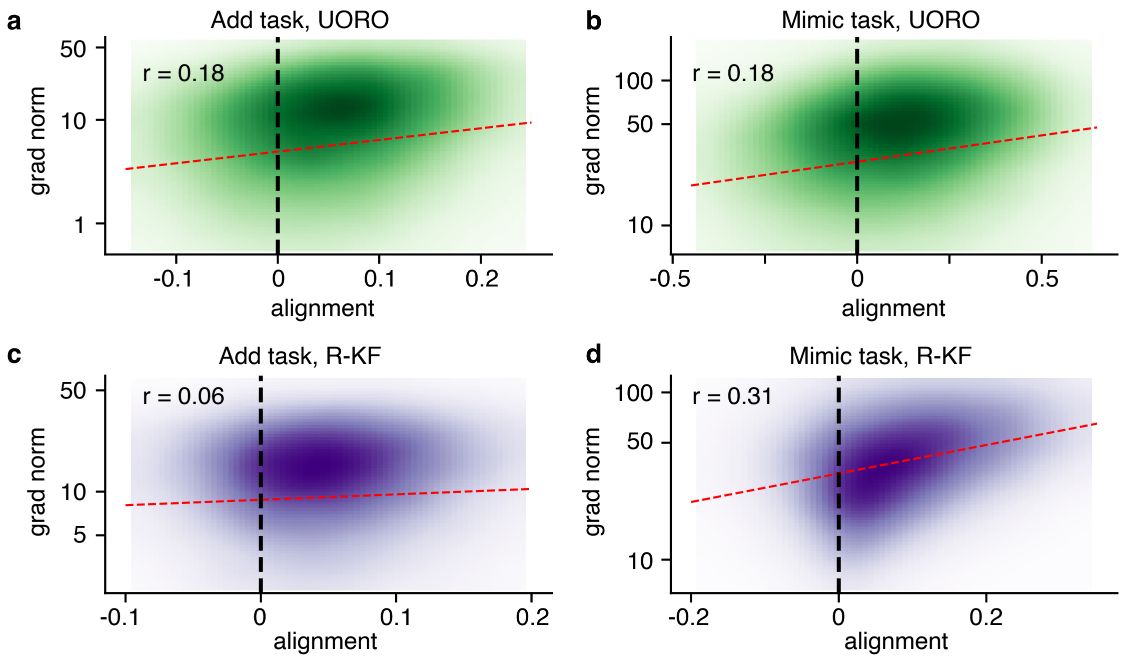

Figure 7: The joint distribution of normalized alignments and gradient norms (log scale). **a)** UORO on Add task, **b)** UORO on Mimic task, **c)** R-KF on Add task, **d)** R-KF on Mimic task. Color intensity represents (smoothed) observed frequency, based on a sample of 100k time steps, with least-squares regression line in red. The estimated correlation coefficients r are all significant.

containing no network history. As Fig. 8a depicts, $\mathbf{a}^{(t-1)}$ ultimately maps to $\mathbf{y}^{(t)}$ via a single-hidden-layer feedforward network parameterized by $\mathbf{W}$ and $\mathbf{W}^{\text{out}}$. The RFLO learning rule in the discrete case corresponds exactly to training $\mathbf{W}$ by backpropagation:

$$\frac{\partial L^{(t)}}{\partial W_{ij}} = \frac{\partial L^{(t)}}{\partial a_i^{(t)}} \frac{\partial a_i^{(t)}}{\partial W_{ij}} = \bar{c}_i^{(t)} \phi'(h_i^{(t)}) \hat{a}_j^{(t-1)} = \bar{c}_i^{(t)} B_{ij}^{(t)}. \tag{28}$$

While every learning algorithm additionally trains $\mathbf{W}^{\text{out}}$ online to best map $\mathbf{a}^{(t)}$ to $\mathbf{y}^{*(t)}$, this purely linear model cannot perfectly capture the complex ways that information about past inputs $\mathbf{x}^{(t')}$, $t' \leq t$ implicit in $\mathbf{a}^{(t)}$ relates to labels $\mathbf{y}^{*(t)}$. Adding a hidden layer improves the ability of the network to predict $\mathbf{y}^{*(t)}$ from whatever evidence of $\mathbf{x}^{(t')}$ is naturally retained in $\mathbf{a}^{(t-1)}$, analogous to how a single-hidden-layer feedforward network outperforms a simple softmax regression on MNIST (Deng, 2012).

To empirically validate our explanation, we first show that the strength of natural memory traces in the RNN depends on its recurrent weights. We measure this "memory" by running an untrained RNN with fixed weights forwards for 20k time steps of the Add task and calculating the r^2 value of a linear regression of $\mathbf{a}^{(t+\Delta t)}$ onto $\mathbf{y}^{*(t)}$ for different values of the time shift Δt. The sudden jumps in information occur as Δt passes the time lags of

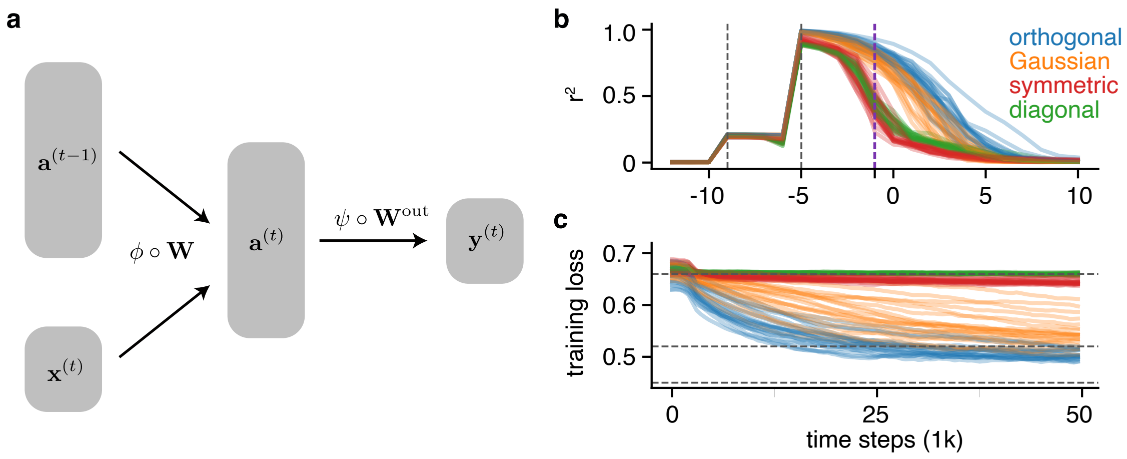

Figure 8: RFLO as a static multi-layer regression. **a)** One time step of recurrent dynamics combine with the output weights to form a single-hidden-layer feedforward network that is trained to predict $\mathbf{y}^{*(t)}$ from $\hat{\mathbf{a}}^{(t-1)}$ by the RFLO learning rule (ψ is softmax). **b)** Coefficient of determination r^2 between $\mathbf{a}^{(t+\Delta t)}$ and $\mathbf{y}^{*(t)}$ as a function of the time shift Δt for different methods of generating $\mathbf{W}$. Each trace is a different random initialization. The cyan dashed line shows the memory at $\Delta t = -1$, corresponding to the input $\mathbf{a}^{(t-1)}$ of the network in (b). The black dashed lines indicate the lags of the input-output dependencies explicitly included in the task, corresponding to sudden jumps in information about $\mathbf{y}^{*(t)}$ contained in the hidden state. **c)** Cross-entropy loss over learning by RFLO. The dashed lines indicate benchmarks for learning output statistics, the first dependency, and the second dependency, respectively (see Pitis, 2016 for details).

the input-output dependencies explicitly included in the task ($t_1 = 5, t_2 = 9$), followed by a slow decay as the information relevant for predicting $\mathbf{y}^{*(t)}$ gets corrupted by running the network forwards. The speed of this decay differs by the choice of $\mathbf{W}$. As Fig. 8b shows, orthogonal and Gaussian $\mathbf{W}$ seem to best preserve information over time, while symmetric and diagonal $\mathbf{W}$ lose information quite rapidly, likely due to their real spectra. In separate simulations (Fig. 8c), we trained networks initialized in these ways using the RFLO rule, and only the networks initialized with orthogonal or Gaussian $\mathbf{W}$ are able to learn at all with RFLO. This validates our hypothesis, that RFLO works by a static prediction of $\mathbf{y}^{*(t)}$ based on evidence in $\mathbf{a}^{(t-1)}$, because in cases where this evidence is absent (or at least weak) due to rapid decay, RFLO fails.

6. Discussion

We have presented a framework for conceptually unifying many recent advances in efficient online training of recurrent neural networks. These advances come from multiple perspectives: pure machine learning, biologically plausible machine learning, and computational neuroscience. We started by articulating a crucial distinction, (a) past facing vs. future facing, that divides algorithms by what exactly they calculate. We then presented a few other properties that characterize each new technique: (b) the tensor structure of the algorithm, (c) whether its update requires explicit randomness, and (d) whether its update can be derived in closed form from the relations (5)–(7) or must be approximated with SGD or some other numerical approach. Along the way, we clarified the relationship between some of the modern online approximations and exact methods. Further, we showed it is possible to create new algorithms by identifying unexplored combinations of properties (a)–(d).

We empirically validated our mathematical intuitions using simple synthetic tasks, solvable using a vanilla RNN architecture and basic gradient descent optimization. The advantage of using these tasks is that they succinctly capture two crucial challenges for training RNNs: calculating credit assignment over long delays (Add) and learning to implement complex, high-dimensional computations (Mimic). Moreover, they allow for parametric variation of the degree of these challenges. Using our synthetic tasks, we found that different algorithms responded to these two challenges differently: stochastic algorithms generally handle credit assignment challenges better (in Add) while deterministic algorithms behave better in high dimensions (in Mimic). We hypothesize that biased algorithms have difficulty propagating credit assignment information over long periods of time, due to error accumulation. Conversely, an unbiased stochastic estimate may perform poorly in high dimensions due to variability not averaging out as easily as in low dimensions.

Following common practice, we used the pairwise vector alignments of the (approximate) gradients calculated by each algorithm as a way to analyze the precision of different approximations. This similarity turned out to reflect the natural clustering of the algorithms along the axes proposed here. In particular, past-facing approximations had stronger alignment with the exact past-facing gradients calculated by RTRL compared to the exact (up to truncation) future-facing gradients calculated by F-BPTT, and vice versa for the future-facing approximation, DNI. Reassuringly, KeRNL and RFLO, which have the same tensor structure, featured strong alignment.

Importantly, the angular alignment of the gradients did *not* account for task performance: UORO and R-KF performed quite well despite their weak alignment with RTRL and BPTT, while KeRNL performed relatively poorly despite its strong alignment with RTRL and BPTT. Analyzing the magnitudes of the gradients partially explained this observation, as UORO and R-KF aligned with RTRL more strongly when their gradients were larger in norm. However, this effect was subtle, so probably not enough to account for the performance differences. This exposes a limitation of current ways of analyzing this class of algorithms. There clearly is a need for better similarity metrics that go beyond time-point-wise gradient alignment and instead compare long-term learning trajectories.

Notably, it is the *stochastic* algorithms that have particularly weak alignment with RTRL. UORO and R-KF barely align positively with RTRL on average despite outperforming many other algorithms on Add, and KF-RTRL has only modest average alignment with RTRL despite performing as well as the exact algorithms on both tasks. The answer to this puzzle may lie in how the averages are computed. Over many time steps of learning, corresponding to many samples of $\boldsymbol{\nu}$, these stochastic methods do contain complete information about $M_{kij}^{(t)}$ in expectation, but at any one time point the alignment is heavily corrupted by the explicit randomness. In contrast, deterministic approximations, such as KeRNL, RFLO and DNI, may partially align with exact methods by construction, but their

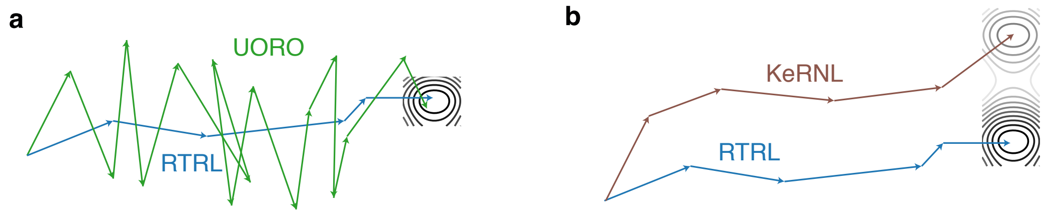

Figure 9: Cartoon illustrating how alignment with RTRL and performance might dissociate. **a)** UORO's noisy estimates of the true gradient are almost orthogonal with RTRL at each time point, but the errors average out over time and allow UORO to find a similar solution. **b)** KeRNL aligns more strongly with RTRL at each time point, but errors do not average out, so KeRNL converges to a worse solution.

errors have no reason to average out, hence their inability to find the same minima as exact methods (Fig. 9). This may also explain why stochastic approximations do not align with each other despite their conceptual similarity.

With the exception of KF-RTRL and R-KF, all of the algorithms discussed here can in principle be generalized to arbitrary RNN architectures, which makes them applicable to large-scale, real-world problems. For the purpose of our review, we did not include algorithms that are specific to a particular architecture, because mathematically comparing them to other methods is inherently difficult. However, such approaches hold great potential. As an example, Ororbia et al. (2017, 2018) propose a specialized neural architecture (Temporal Neural Coding Network), whose learning algorithm (Discrepancy Reduction) is naturally online and efficient, due to the network structure. As another example, Bellec et al. (2019) adapt what is effectively RFLO to a biologically motivated network architecture involving spiking units and time-varying thresholds. More generally, we believe the mathematical intuitions we provide for general-purpose methods can be leveraged to aid these efforts in designing new architecture-specific learning algorithms.

The search for special architectures that make learning easy is particularly relevant for computational neuroscience. In the brain, cortical architecture and synaptic plasticity rules have evolved together under physical constraints that require plasticity to be local. Still, the details of how local plasticity rules interact with neural circuits remain mysterious and are a current focus of research (Guerguiev et al., 2017; Sacramento et al., 2018; Bellec et al., 2019; Lillicrap and Santoro, 2019). Exploring which architectures allow locality to manifest as a consequence, rather than a constraint, of learning is a potentially fruitful point of interaction between artificial intelligence and computational neuroscience.

Acknowledgments

We thank Guangyu Yang and James Murray for helpful conversations. This work was supported by a Google faculty award, and **NRT-HDR** grant 1922658. KC thanks NVIDIA and eBay for their support.

Appendix A. Lemma for Generating Rank-1 Unbiased Estimates

For completeness, we state the Lemma from Tallec and Ollivier (2017) in components notation. Given a decomposition of a matrix $\mathbf{M} \in \mathbb{R}^{n \times m}$ into r rank-1 components

$$M_{ij} = \sum_{k=1}^{r} A_{ik} B_{kj}, \tag{29}$$

a vector of i.i.d. random variables $\boldsymbol{\nu} \in \mathbb{R}^r$ with $\mathbb{E}[\nu_k] = 1$, $\mathbb{E}[\nu_k \nu_{k'}] = \delta_{kk'}$, and a list of r positive constants $\rho_k > 0$, then

$$\tilde{M}_{ij} = \left(\sum_{k=1}^{r} \rho_k \nu_k A_{ik} \right) \left(\sum_{k=1}^{r} \rho_k^{-1} \nu_k B_{kj} \right) \tag{30}$$

is a rank-1, unbiased estimate of M_{ij} over the choice of $\boldsymbol{\nu}$.

Appendix B. Implementation details

For reproducibility, we describe in fuller detail our implementation configurations for each simulation. Table 2 shows hyperparameter/configuration choices that apply across all algorithms. Table 3 shows the algorithm-specific hyperparameter choices we made for each task. In Table 2, we reference sub-matrices of $\mathbf{W} = [\mathbf{w}^{\text{rec}}, \mathbf{w}^{\text{in}}, \mathbf{b}^{\text{rec}}]$ and $\mathbf{W}^{\text{out}} = [\mathbf{w}^{\text{out}}, \mathbf{b}^{\text{rec}}]$, since they are initialized differently.

hyperparameter	**value**	**explanation**
learning rate	10^{-4}	learning rate for SGD w.r.t. $\mathbf{W}$ and $\mathbf{W}^{\text{out}}$
n	32	number of hidden units in the network
ϕ	tanh	nonlinearity used in RNN forward dynamics
init. $\mathbf{w}^{\text{in}}$	$\sim \mathcal{N}(0, 1/\sqrt{n_{\text{in}}})$	initial value for input weights
init. $\mathbf{w}^{\text{rec}}$	rand. orth.	initial value for recurrent weights
init. $\mathbf{b}^{\text{rec}}$	0	initial value for recurrent bias
init. $\mathbf{w}^{\text{out}}$	$\sim \mathcal{N}(0, 1/\sqrt{n})$	initial value for output weights
init. $\mathbf{b}^{\text{out}}$	0	initial value for output bias
init. $\mathbf{W}^{\text{FB}}$	$\sim \mathcal{N}(0, 1/\sqrt{n_{\text{out}}})$	value for fixed feedback weights used in DNI(b)
init. $\mathbf{b}^{\text{rec}}_{\text{targ.}}$	$\sim \mathcal{N}(0, 0.1)$	initial value for target recurrent bias in Mimic
init. $\mathbf{b}^{\text{out}}_{\text{targ.}}$	$\sim \mathcal{N}(0, 0.1)$	initial value for target output bias in Mimic

Table 2: Default hyperparameter choices for the RNN independent of learning algorithm.

Some miscellaneous implementation details below:

- For the Add task in the $\alpha = 1$ condition, we changed the DNI/DNI(b) learning rate to 5×10^{-2} for A_{li} and 10^{-2} for $\mathcal{J}_{ij}$ (DNI(b)). In other cases, the learning rates for A_{li} and $\mathcal{J}_{ij}$ are identical.
- There are two appearances of the synthetic gradient weights A_{li} in Eq. (26). Although we wrote them as one matrix $\mathbf{A}$ for brevity, in implementation we actually keep two separate values, $\mathbf{A}$ and $\mathbf{A}^*$, the latter of which we use for for the right-hand appearance

algorithm	**initial values**	$\boldsymbol{\nu}$ **dist.**	**LR**	**pert.** σ	T
UORO	$A_k^{(0)} \sim \mathcal{N}(0,1), B_{ij}^{(0)} \sim \mathcal{N}(0,1)$	unif. $\{-1,1\}$			
KF-RTRL	$A_j^{(0)} \sim \mathcal{N}(0,1), B_{ki}^{(0)} \sim \mathcal{N}(0,1)$	unif. $\{-1,1\}$			
R-KF	$A_i^{(0)} \sim \mathcal{N}(0,1), B_{kj}^{(0)} \sim \mathcal{N}(0,1)$	unif. $\{-1,1\}$			
KeRNL	$A_{ki}^{(0)} = \delta_{ki}, B_{ij}^{(0)} = 0, \alpha_i^{(0)} = \alpha$		$1/\sigma$	10^{-7}	
DNI	$A_{li}^{(0)} \sim \mathcal{N}(0, 1/\sqrt{m'})$		10^{-3}		
DNI(b)	$A_{li}^{(0)} \sim \mathcal{N}(0, 1/\sqrt{m'}), \mathcal{J}_{ij}^{(0)} = W_{ij}^{\text{rec}}$		10^{-3}		
F-BPTT					10

Table 3: Hyperparameter choices specific to individual algorithms.

$A_{l'm}$ (specifically to calculate the bootstrapped estimate of the SG training label). We update $\mathbf{A}$ every time step but keep $\mathbf{A}^*$ constant, replacing it with the latest value of $\mathbf{A}$ only once per $\tau \in \mathbb{N}$ time steps. This integer τ introduces another hyperparameter, which we choose to be 5. (Inspired by an analogous technique used in deep Q-learning from Mnih et al., 2015.)

- In the original paper, Roth et al. (2019) use $(1 - \exp(-\gamma_i))$ rather than α_i as a temporal filter for $B_{ij}^{(t)}$. We made this change so that α_i makes sense in terms of the α in the forward dynamics of the network and RFLO. Of course, these are equivalent via $\gamma_i = -\log(1 - \alpha_i)$, but the gradient w.r.t. α_i must be rescaled by a factor of $1/(1 - \alpha_i)$ to compensate.

- For KeRNL, there is a choice for how to update the eligibility trace (Eq. 17): one can scale the right-hand term $\phi'(h_i^{(t)})\hat{a}_j^{(t-1)}$ by either the learned timescale α_i or the RNN timescale α. We chose the latter because it has stronger empirical performance and it theoretically recovers the RTRL equation under the approximating assumptions about $\mathbf{A}$.

- Perturbations for calculating gradients for A_{ki} and α_i in KeRNL are sampled i.i.d. $\zeta_i \sim \mathcal{N}(0, \sigma)$.

- In our implementation of the Add task, we use $n_{\text{in}} = n_{\text{out}} = 2$ for a "one-hot" representation of the input $x^{(t)} \in \{0, 1\}$ and label $y^{*(t)} \in \{0.25, 0.5, 0.75, 1\}$, such that $\mathbf{x}^{(t)} = [x^{(t)}, 1 - x^{(t)}]$ and $\mathbf{y}^{*(t)} = [y^{(t)}, 1 - y^{(t)}]$.

- In our implementation of Mimic, the target RNN was initialized in the same way as the RNNs we train, with the exception of the recurrent and output biases (see Table 2).

Appendix C. Supplementary Figures

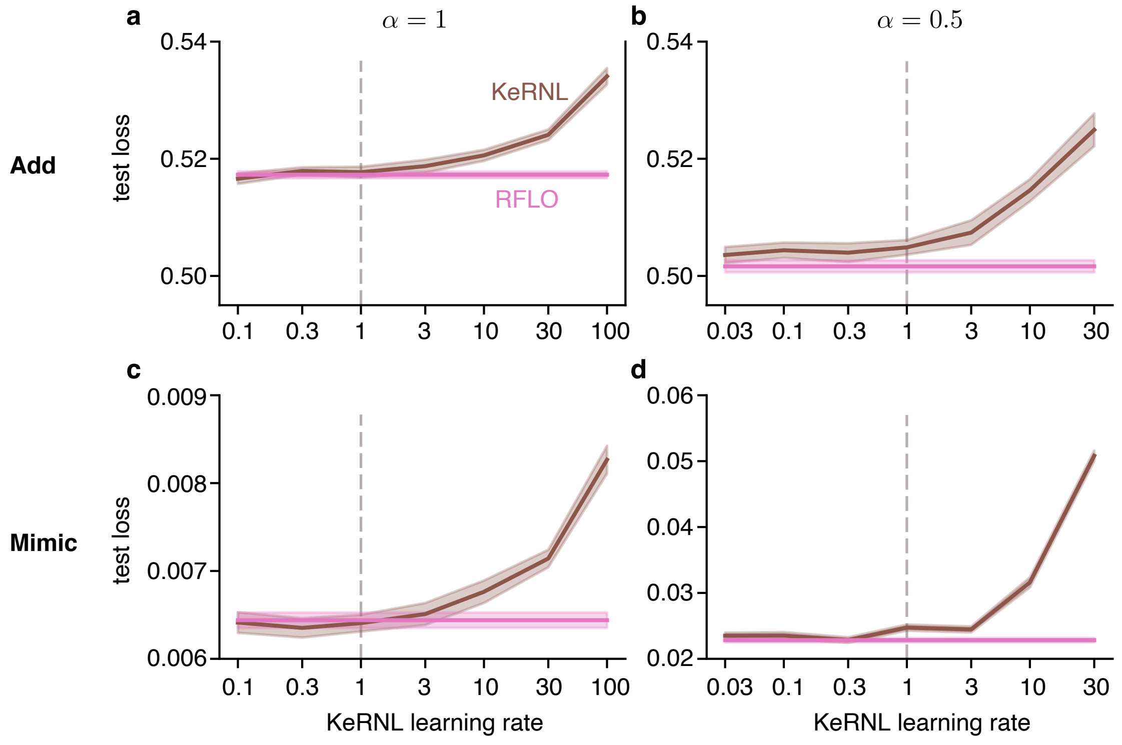

Figure S1: **a)** Test loss of KeRNL on Add task ($\alpha = 1, t_1 = 5, t_2 = 9$) and RFLO for different values of the learning rate for A_{ki} and α_i. The learning rate used in the rest of the paper is indicated by the grey dashed line. **b)** Same for Add with $\alpha = 0.5, t_1 = 2, t_2 = 4$. **c)** Same for Mimic with $\alpha = 1, n_\text{h} = 32$. **d)** Same for Mimic with $\alpha = 0.5, n_\text{h} = 32$.

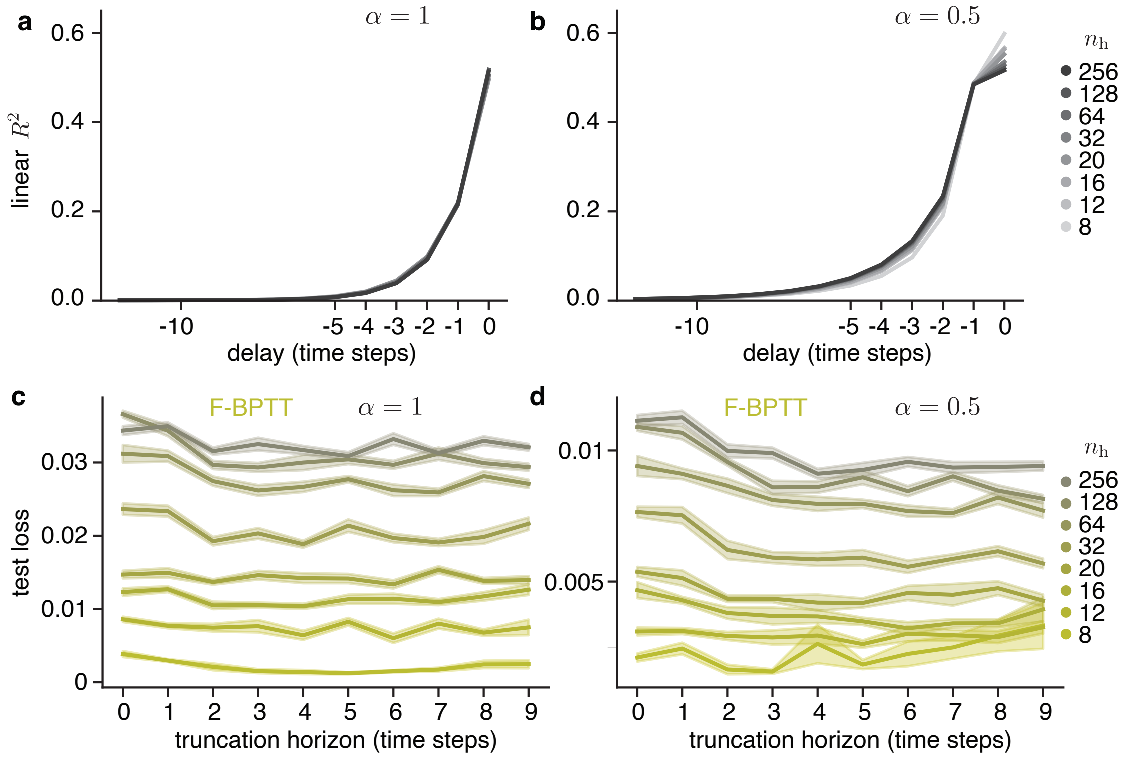

Figure S2: **a)** Linear R^2 coefficient between inputs $\mathbf{x}$ and labels $\mathbf{y}^*$ at a given time delay for the Mimic task ($\alpha = 1$) on various difficulties (shades of gray). **b)** Same for $\alpha = 0.5$. **c)** Test loss for F-BPTT algorithm on Mimic task ($\alpha = 1$) as a function of truncation horizon on various difficulties. **d)** Same for $\alpha = 0.5$. For low difficulties, performance gets worse with higher T. For higher difficulties, performance saturates around $T = 5$ or earlier.

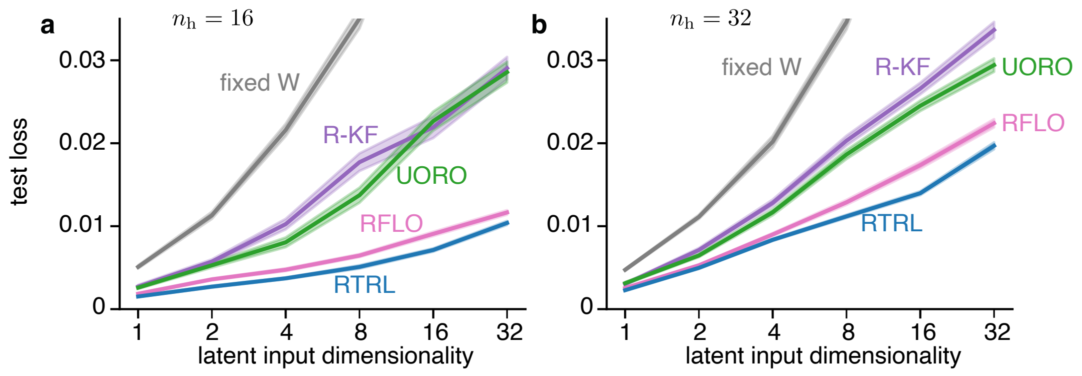

Figure S3: **a)** Test loss on Mimic ($\alpha = 1, n_h = 16$) as a function of the latent dimensionality of the inputs, while holding fixed the number of nominal input dimensions to 32 to keep $\mathbf{W}$ the same size. As task dimensionality gets larger, performance of UORO and R-KF diverges from rest of algorithms. DNI, F-BPTT, KF-RTRL and KeRNL omitted for readability but perform similarly to RTRL and RFLO. **b)** Same for $n_h = 32$.

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4945–4949. IEEE, 2016.

Guillaume Bellec, Franz Scherr, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass. Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets. *arXiv preprint arXiv:1901.09049*, 2019.

Frederik Benzing, Marcelo Matheus Gauy, Asier Mujika, Anders Martinsson, and Angelika Steger. Optimal kronecker-sum approximation of real time recurrent learning. *arXiv preprint arXiv:1902.03993*, 2019.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Kyunghyun Cho, Aaron Courville, and Yoshua Bengio. Describing multimedia content using attention-based encoder-decoder networks. *IEEE Transactions on Multimedia*, 17 (11):1875–1886, 2015.

Tim Cooijmans and James Martens. On the variance of unbiased online recurrent optimization. *arXiv preprint arXiv:1902.02405*, 2019.

Wojciech Marian Czarnecki, Grzegorz Swirszcz, Max Jaderberg, Simon Osindero, Oriol Vinyals, and Koray Kavukcuoglu. Understanding synthetic gradients and decoupled neural interfaces. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 904–912. JMLR. org, 2017.

Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

Jordan Guerguiev, Timothy P Lillicrap, and Blake A Richards. Towards deep learning with segregated dendrites. *ELife*, 6:e22901, 2017.

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

Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1627–1635. JMLR. org, 2017.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Timothy P Lillicrap and Adam Santoro. Backpropagation through time and the brain. *Current Opinion in Neurobiology*, 55:82 – 89, 2019. ISSN 0959-4388. doi: https://doi.org/10.1016/j.conb.2019.01.011. URL `http://www.sciencedirect.com/science/article/pii/S0959438818302009`. Machine Learning, Big Data, and Neuroscience.

Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications*, 7:13276, 2016.

Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Owen Marschall, Kyunghyun Cho, and Cristina Savin. Evaluating biological plausibility of learning algorithms the lazy way. 2019.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Asier Mujika, Florian Meier, and Angelika Steger. Approximating real-time recurrent learning with random kronecker factors. In *Advances in Neural Information Processing Systems*, pages 6594–6603, 2018.

James M Murray. Local online learning in recurrent networks with random feedback. *eLife*, 8:e43299, 2019.

Alexander Ororbia, Ankur Mali, C Lee Giles, and Daniel Kifer. Online learning of recurrent neural architectures by locally aligning distributed representations. *arXiv preprint arXiv:1810.07411*, 2018.

II Ororbia, G Alexander, Patrick Haffner, David Reitter, and C Lee Giles. Learning to adapt by minimizing discrepancy. *arXiv preprint arXiv:1711.11542*, 2017.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

Silviu Pitis. Recurrent neural networks in tensorflow 1. `r2rt.com/recurrent-neural-networks-in-tensorflow-i`, 2016. Accessed: 2018-11-13.

Christopher Roth, Ingmar Kanitscheider, and Ila Fiete. Kernel RNN learning (keRNL). In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=ryGfnoC5KQ`.

João Sacramento, Rui Ponte Costa, Yoshua Bengio, and Walter Senn. Dendritic cortical microcircuits approximate the backpropagation algorithm. In *Advances in Neural Information Processing Systems*, pages 8721–8732, 2018.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Corentin Tallec and Yann Ollivier. Unbiased online recurrent optimization. *arXiv preprint arXiv:1702.05043*, 2017.

T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015.

Paul J Werbos et al. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.