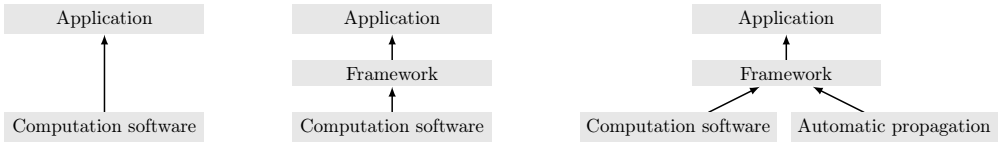
Proppo: a Message Passing Framework for Customizable and Composable Learning Algorithms

Anonymous Author(s)
Affiliation
Address
email

Abstract

While existing automatic differentiation (AD) frameworks allow flexibly composing model architectures, they do not provide the same flexibility for composing learning algorithms—everything has to be implemented in terms of backpropagation. To address this gap, we invent Automatic Propagation (AP) software, which generalizes AD, and allows custom and composable construction of complex learning algorithms. The framework allows packaging custom learning algorithms into propagators that automatically implement the necessary computations, and can be reused across different computation graphs. We implement Proppo, a prototype AP software package built on top of the Pytorch AD framework. To demonstrate the utility of Proppo, we use it to implement Monte Carlo gradient estimation techniques, such as reparameterization and likelihood ratio gradients, as well as the total propagation algorithm and Gaussian shaping gradients, which were previously used in model-based reinforcement learning, but do not have any publicly available implementation. Finally, in minimalistic experiments, we show that these methods allow increasing the gradient estimation accuracy by orders of magnitude, particularly when the machine learning system is at the edge of chaos.

1 Introduction



(a) Direct approach (b) Framework approach (c) Automatic propagation software approach

Figure 1: A simplified taxonomy of machine learning (ML) software development methodologies.

When a machine learning (ML) researcher comes up with a new algorithm, they are faced with a dilemma: do they directly implement an application of their algorithm using existing ML tools (Fig. 1a); or do they implement a framework for their algorithm, then use the framework to implement the desired application (Fig. 1b). The first approach is faster, because the researcher only has to worry about the details of the specific application; the second approach may take longer to implement, but will enable reuse of the code and speed up implementing future applications. Clearly, the second approach is preferable in the long-term. Our work aims to overcome this dilemma by simplifying the intermediate stage of implementing reusable code for the algorithm. The approach is illustrated in Fig. 1c, where, our invention, *automatic propagation software*, ameliorates the shortcomings of existing tools that we utilize concurrently to implement reusable code for a new algorithm. This interusability was a key design choice, as it allows benefitting from the rich ML software ecosystem.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

In ML a vast amount of effort has already been placed in developing computational software, e.g., established automatic differentiation (AD) libraries such as PyTorch (Paszke et al., 2019), Chainer (Tokui et al., 2015), TensorFlow (Abadi et al., 2016) or JAX (Bradbury et al., 2018); or newer research such as randomized AD (Oktay et al., 2020) or AD for pointful array programming in Dex (Paszke et al., 2021). There exist libraries that add functionality on top of these frameworks, such as Keras (Chollet et al., 2015) that allows composing models in TensorFlow, BackPACK (Dangel et al., 2020) that allows automatically computing additional quantities besides gradients in PyTorch, or Einops (Rogozhnikov, 2021) that provides a new interface for Einstein notation operations. Other frameworks use these AD libraries as tools to implement functionality, e.g., Storchastic (Krieken et al., 2021), a library for Monte Carlo (MC) gradient estimation in PyTorch. Finally, there are libraries that do not directly build on these frameworks, but can be combined together, e.g., Ray (Moritz et al., 2018) that implements distributed computing. Creating each one of these libraries takes significant effort, and it makes sense to build on the work of others to simplify implementing new algorithms. Due to this, new implementations tend to follow the design paradigms of old software.

Barham and Isard (2019) described this situation with the sentence "Machine learning systems are stuck in a rut". They argued that current ML systems are optimized for computations encountered by popular algorithms, yet, other calculations may be slow, despite in principle requiring a similar amount of operations. Such an imbalance would discourage researchers from trying new ideas that require novel forms of computation. Our own observation is that existing AD frameworks encourage implementing algorithms by designing a differentiable "surrogate loss" (Schulman et al., 2015; Foerster et al., 2018) often employed for implementations of MC gradient estimators (Krieken et al., 2021). However, this may not be the most natural way to express the algorithms, or indeed, it may not even be possible. Minka (2019) argued that a general-purpose ML language should "simplify implementation of *all* ML algorithms", and promoted a message passing approach. We aim for such a general-purpose message passing framework with our automatic propagation (AP) proposal.

We list our desiderata for a general-purpose framework, and the approach we take to meet these desired properties in Proppo, our prototype AP software package.

Ease of use. Proppo is implemented entirely in Python, a language favored by ML researchers.

Flexibility. Proppo allows implementing arbitrary message passing algorithms.

Computational speed. Proppo interfaces with existing computational frameworks such as PyTorch to achieve efficient computation. Proppo adds functionality to such frameworks; the computational speed will depend on how the user implements their algorithm.

Reusability of code. Proppo allows packaging algorithms into propagators—the key building blocks of AP software—that encapsulate an algorithm and can be reused across different computation graphs.

Composability. Proppo includes special sequence propagators (Sec. 3.2) that allow combining multiple propagators together to form new propagators. These new propagators may then be further composed, allowing to deal with arbitrary complexity.

The main functionality of Proppo is to provide base classes for the propagators, and to implement automatic message passing. When a user wants to implement a new algorithm, they need to create the propagators to represent their algorithm as a message passing program. When a user wants to use an existing algorithm, they can simply download the existing propagators, and apply them in their code.

To demonstrate the utility of Proppo, we use it to implement the total propagation (TP) and Gaussian shaping (GS) gradient algorithms (Parmas et al., 2018; Parmas, 2018). Both of these are techniques for MC gradient estimation that did not previously have any publicly available implementation, and they were one of our key motivations for this research. In particular, the probabilistic computation graph framework (Parmas, 2018, 2020) influenced our thinking on MC gradient estimation. Our experimental results in Sec. 4.2 and App. C.3 demonstrate orders of magnitude improvement in gradient accuracy for these methods. The improvement was particularly large when the system was at the *edge of chaos*. This implies that AP software can achieve fundamentally better computational results compared to the current AD software paradigm. To demonstrate the scalability and practicality of Proppo, in our concurrent work (Anonymous, 2022) we use Proppo to replicate the model-based RL results of (Parmas et al., 2018) and we apply TP to the high dimensional visual MBRL algorithm Dreamer (Hafner et al., 2020). All our results imply that Proppo is scalable, leads to more modular code, simplifies implementation of some algorithms and can lead to greatly improved performance.

2 Preliminaries

2.1 Automatic differentiation software

A recent review of AD in ML was written by Baydin et al. (2018). To explain the functionality of AD software, suppose that a user has written a program to compute the output of a function $y = f(\boldsymbol{x})$ given the input $\boldsymbol{x}$. In such a situation, AD software can automatically obtain the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. Such functionality is achieved by differentiating each intermediate computation and applying the chain rule. A series of computations can be represented as a directed acyclic graph with nodes and their associated variables $\boldsymbol{x}_i$, and with an edge from node i to j when the variable $\boldsymbol{x}_j$ is computed with a function $f_k(\boldsymbol{x}_i, \ldots)$ taking the variable at node i as an input. In this case, the total derivative from $\boldsymbol{x}$ to y can be obtained by summing the product of derivatives across all paths between the two nodes (Bauer, 1974),

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \sum_{\text{Path} \in \text{Paths}[\boldsymbol{x} \to y]} \prod_{\text{Edge}[l \to k] \in \text{Path}} \frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{x}_l}. \tag{1}$$

A naïve way to implement AD would be to compute the intermediate Jacobians $\frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{x}_l}$ by differentiating the operations $f_k(\boldsymbol{x}_l, \ldots)$. However, note that this will require storing full Jacobian matrices having size $K \times L$, where K and L are the respective dimensionalities of the variables. Moreover, if the gradient were computed by multiplying the Jacobians in the forward direction from $\boldsymbol{x}$ to y, this would require matrix-matrix multiplications at each step, and may incur a large computational cost. Reverse mode automatic differentiation (also known as back propagation), overcomes this issue by performing the computations backwards from the y node. As y is just a 1 dimensional scalar, its local Jacobian is a vector. Therefore the computation at the last step is a vector-matrix product, yielding another vector. This vector is propagated backwards, yielding a vector at each backward step. This allows computing the total gradient using only vector-matrix products, leading to a large computational saving compared to computing the gradients in the forward direction using matrix-matrix products.

Another subtle point is that it is not necessary to compute the intermediate Jacobians $\frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{x}_l}$, one merely needs a means to compute $\boldsymbol{v}^T \frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{x}_l}$, where $\boldsymbol{v}$ is the back propagated gradient vector. So in practice, AD software often only stores the variables necessary to evaluate, $\boldsymbol{v}^T \frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{x}_l}$, and never explicitly computes $\frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{x}_l}$. This leads to further computational and memory savings, as the full Jacobian matrices with size $K \times L$ need not be stored. In PyTorch, this is implemented in the `torch.autograd.backward(tensors, grad_tensors)` function, where `tensors` corresponds to $\boldsymbol{x}_k$ and `grad_tensors` corrsponds to $\boldsymbol{v}$. All in all, AD software implements *forward* functions to cache the necessary data at each node, and *backward* functions to evaluate the vector-Jacobian product at each node using the cache, resulting in efficient and automatic computation of the gradient.

2.2 Basics of Monte Carlo gradient estimation

In ML, the computations may be stochastic, and the objective is an expectation $J = \mathbb{E}_{\boldsymbol{x} \sim p(\boldsymbol{x};\beta)} [f(\boldsymbol{x})]$. In this case, it is not clear how to directly differentiate the stochastic operations, and we use Monte Carlo (MC) methods to obtain an estimator, $\hat{\boldsymbol{g}}$, for the gradient $\frac{\mathrm{d}J}{\mathrm{d}\beta}$, s.t. $\mathbb{E}[\hat{\boldsymbol{g}}] = \frac{\mathrm{d}J}{\mathrm{d}\beta}$. Such methods have been reviewed in ML by Mohamed et al. (2020). The two major MC gradient estimators are the reparameterization (RP) (Rezende et al., 2014; Kingma and Welling, 2013) and likelihood ratio (LR) (Williams, 1992; Glynn, 1990) gradient estimators. Total propagation (TP) (Parmas et al., 2018) efficiently combines LR and RP. We give more background on these methods in App. A.

3 Proppo explanation

3.1 Overview

A schematic illustration of automatic propagation software is provided in Fig. 2. Automatic propagation is *not* an alternative to existing frameworks, such as AD packages, rather, it adds functionality on top of them, and can be used in conjunction. We first explain the Proppo framework, in its simplest form, which is one specific implementation of AP software; we explain the different components of the framework (Sec. 3.1.1), then the processes by which they operate (Sec. 3.1.2). Finally, we explain AP from a more general perspective (App. B.1), loosening some of the specific choices in Proppo.

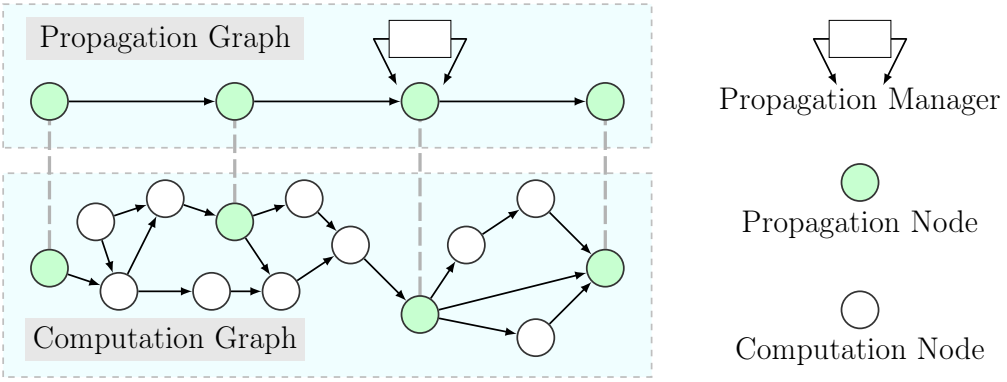


Figure 2: Illustration of the automatic propagation framework. A subset of the nodes ○ are designated as propagation nodes ○ forming the propagation graph. The propagation manager ⬠ traverses the propagation graph and activates the propagators to ensure the correct functioning of the implemented algorithm. Here, the propagation graph is illustrated as a chain, but it may also have a complicated graph structure. Moreover, there may be multiple propagation managers working in parallel.

3.1.1 Components of Proppo

○ **Computation nodes.** While in principle AP software does not require using any external computational software, in practice, it will usually be convenient to take advantage of existing computation software, such as AD packages, and use them in conjunction. Automatic propagation software can be combined together with any other computation package or program code. These external packages can be used regularly as one would usually write code. However, at certain locations in the code, the user can specify special computations that will determine what algorithm is implemented using the AP software. These special computations will create *propagation nodes* ○.

○ **Propagation nodes.** The propagation nodes determine the structure of the operations performed in the AP software. In Proppo, each propagation node includes the three components: (1) a pointer to a propagator ←○→ that determines the forward and backward computations belonging to the node, (2) a cache/memory that stores necessary information for the computations occuring at the node, (3) a message box for incoming messages ⊠ from other propagation nodes.

←○→ **Propagators.** The propagators are the key components of AP software. The propagators determine the computations performed at each propagation node. In Proppo, each propagator implements a forward and backward mode of computation. The forward function is used when the node is first created; it may take an input from the program code, perform computations, store data in the memory of the node, and produce an output. The backward function is used in later processing; it performs computations using the stored data in the memory and incoming messages to the node, and sends messages ⊠ to other nodes in the propagation graph.

✉ **Messages.** Globally, the algorithms are implemented by passing messages between the propagation nodes. Each message includes the two components: (1) the contents of the message, (2) an address determining where the message should be sent. A typical address may be a pointer to the parent node of the current propagation node, so that the messages are passed backwards.

⬠ **Propagation manager.** In Proppo, all of the messages are first passed to the propagation manager, then the propagation manager decides where to send the message based on the address in the message. The propagation manager also has other important roles: it constructs the propagation graph, keeps track of the nodes, activates the propagators and passes messages.

3.1.2 Typical program flow and usage of automatic propagation software

Usage of AP software can be broadly categorized into two: (1) implementing a new algorithm into an AP package, (2) using an algorithm already implemented in AP software. We start by explaining how one would typically use an already existing algorithm packaged into AP software. We begin by explaining a minimalistic code snippet in Fig. 3 of MC gradient estimation with Proppo.

```
1
2
3
4
5 state = torch.randn(batch_size,
6                     num_dimensions)
7 for t in range(horizon_length):
8     state = rnn(state)
9
10
11 loss = loss_func(state)
12 loss.backward()
```

(a) PyTorch code

```
1 # Configuration code
2 prop = RPProp(config) # Choose RPProp or LRProp
3 manager = PropagationManager(
4                     default_propagator=prop)
5 # Program code
6 state = torch.randn(batch_size, num_dimensions)
7 for t in range(horizon_length):
8     state = rnn(state)
9     state = manager.forward(state)
10
11 loss = loss_func(state)
12 manager.backward(loss)
```

(b) Proppo code combined with PyTorch code

Figure 3: Code for estimating the gradient w.r.t. the parameters of a recurrent neural network (`rnn`).

Minimalistic code example. In the code example in Fig. 3a, a batch of initial states (`state`) is sampled from a Gaussian distribution (lines 5–6), this batch is pushed through the recurrent neural network (`rnn`) for `horizon_length` times, then a loss is computed at the final step using the function `loss_func`, finally the gradients are backpropagted from the loss node by calling `loss.backward()`. As explained in the preliminaries (Sec. 2.1), the gradients will propagate until the parameters in the computation graph, e.g. if the parameters are stored in `rnn.weight`, the gradient could be access by `rnn.weight.grad`. The regular PyTorch code in Fig. 3a computes the gradient using back propagation; the Proppo code in Fig. 3b modifies the computation by adding noise into the forward computation, and estimating the gradient using either RP gradients or LR gradients. Conveniently, only the definition of the propagator on line 2 has to be changed to switch between using LR or RP gradients. Moreover, settings such as whether the LR gradient should use a baseline or not can be specified in the configuration without having to modify the program code. This was a minimalistic example to give a taste of how Proppo might be used. In practice both the configuration and program code may be more complicated with multiple types of propagators. We will explain how existing AP algorithms would typically be used with reference to this minimalistic example.

Using an AP algorithm. The code for using AP software can be broadly separated into two: the *configuration code* that sets up the AP software and selects the algorithm to be used, and the *program code* that determines how and where the algorithm is applied. It may be beneficial to keep these sections of the code distinct to improve the readability and modularity.

In the minimalistic example in Fig. 3b the *configuration code* is on lines 2–4. When configuring the code, the user instantiates the propagator and propagation manager objects from their existing class code, while giving them the appropriate configuration. A propagator may be created using

```
propagator = Propagator(config).
```

There are different propagator classes, and the user would use a different command for the different types of propagators. Some propagators may allow for a configuration input to make small tweaks to the behavior. The propagation manager may be created using

```
manager = PropagationManager(config).
```

Here the configuration input may define default propagators to be used for specified computations, default message passing styles, etc. The configuration in the RNN example was simple, but in general the configuration may be more complicated and include setting up many different types of propagators and equipping them to the manager by assigning them names by which the manager identifies them.

After selecting the desired algorithm by properly configuring the AP software, the user must modify the *program code* to make it compatible with the AP software and properly implement the algorithm. In the minimalistic example in Fig. 3b, the program code is on lines 6–12, and the modifications to use Proppo are on lines 9 and 12. We emphasize that the same modifications may be compatible with multiple algorithms, so that only the configuration code has to be modified to switch between the algorithms. The main purpose of the modifications to the program code are to activate the propagators and construct the propagation graph. A propagator is activated using the command

```
output = manager.forward(input, arguments).
```

This command will add a node into the propagation graph of the manager. In the current case, the `manager` has been equipped with the default propagator `prop`. When `manager` calls the forward function, it will create a new node, and assign the `prop` propagator to it. Then, the forward method of `prop` is called with the `input`. The forward method applies some modification to the input, stores data in the memory of the node, and produces an output. Note that the data stored in the memory of the node will be used later during the backward pass of the algorithm.

The `arguments` input to the `forward` function serves multiple purposes. One of the main purposes is to modify the behavior of the propagator in some way. For example, in the minimalistic code in Fig. 3b the propagator injects noise into `state` to produce the output, so one part of the `arguments` may be the type of noise and the parameters of the noise distribution. Note that often these settings can be defined in the configuration without having to modify the `arguments`, e.g. the type of distribution may just be fixed in the configuration code. Another role of the `arguments` is to modify the targets where the messages should be sent if the default setting is unsuitable. Furthermore, if multiple types of propagators are necessary to implement the algorithm, the `arguments` may also be used to select between different propagators equipped to the manager.

Finally, the backward pass of the algorithm is invoked by calling

```
manager.backward().
```

When the `manager.backward` function is called, the manager will typically call the `backward` function of the propagator of the last node, pass the message to its target node, call the `backward` function of the previous node, and iterate this process from the end of the graph until the beginning. If the propagators are correctly implemented, and the propagation graph is correctly constructed, this framework allows automatically using an algorithm available in the AP software.

A keen reader may have noticed that in the minimalistic example in Fig. 3b an additional `loss` input was given to the `backward` call. This is a convenience implemented in Proppo, where the manager will automatically add a propagator corresponding to the loss as the final node before starting to iterate backwards on the propagation graph. Alternatively, the same operations could have been implemented by the commands `manager.append_loss(loss)` followed by `manager.backward()`. Internally, `append_loss(·)` simply calls the forward method of a special propagator designed for adding a node corresponding to a loss. In Proppo, yet another way how the lines 11–12 for adding the loss could have been implemented is the following: `manager.append_loss(state, lossfunc=loss_func)` followed by `manager.backward()`. In this case, the loss function `loss_func` is passed to the propagator directly via the `lossfunc` keyword, and the propagator will compute the loss by inputting `state` into `loss_func`. The last method of implementation is beneficial for compatibility between different types of loss propagators, for example, it is useful for easily switching between the loss operations necessary for Gaussian shaping gradients and a regular loss computation. We see that there are many different ways to implement the same algorithm using propagators.

Creating an AP algorithm. In the previous section, we saw that the functionality of an AP algorithm is determined by the implementation of the propagators. Consequently, creating a new AP algorithm requires implementing the appropriate propagators corresponding to the desired algorithm. In Proppo, several base classes for new propagators are already provided, and the user must only implement the `forward` and `backward` methods of the propagators.

In principle, the `forward` and `backward` methods can contain arbitrary code, and there is no restriction on what they can do. The programmer must figure out how to implement their algorithm through a process of storing data in the `forward` pass, and further computations and message passing in the `backward` pass. The programmer must also decide how many different types of propagators are necessary for their algorithm, and also how they are intended to be used in the *program code*. Like with any software, good design of the code and clear documentation are crucial for usability.

Throughout our discussion of AP software, we have promised composability and customizability of the algorithms. We note that merely using AP software to implement an algorithm does not guarantee that the algorithm will be composable. It is up to the creator of the algorithm to design the propagators so as to facilitate such desirable properties. However, we find that the paradigm of propagators is a good framework by which to construct an algorithm in a composable way. In particular, in Sec. 3.2 we highlight common propagator types that can lead to composable and flexible algorithms.

3.2 Example elementary propagators

In AP software, composability is achieved by deriving new propagators from already existing propagators. The simplest form of this may be to connect propagators into a chain, and activate their methods in sequence. In Proppo, we provide two propagators with such functionality, `ComboProp` and `ChainProp`. While both of these methods activate the forward and backward methods of the constituent propagators sequentially, they differ in how the messages and nodes are handled.

ComboProp. This propagator is a simple way to connect propagators into a chain. The output of the forward method of one propagator is fed into the input of the next propagator, and each propagator operates on the same node. Likewise, the backward methods use the data in the same node, and the messages are simply combined together and passed to the backward method of the previous propagator. `ComboProp` is suitable for simple combinations of propagators; however, it can run into problems when trying to implement more complex behaviors. Namely, the different propagators may overwrite data stored by other propagators, and the messages may also interfere with one another leading to unexpected consequences.

ChainProp. This propagator provides a more systematic way to combine propagators, so as to overcome the sometimes unpredictable behavior of `ComboProp`. To rigorously define the behavior, `ChainProp` internally create its own propagation graph, connecting nodes with propagators into a chain, and assigning a propagation manager to handle the message passing. The constituent propagators may themselves also be of the `ChainProp` type. This way, propagation graphs can be embedded into other propagation graphs, creating a hierarchy of connected propagators all handled by their own managers. This system allows safely composing propagators to define complex behaviors.

BackPropagator. We introduce one example propagator that is often composed together with other propagators in Proppo. In Pytorch, the `torch.autograd.backward(tensors, grad_tensors)` method allows inputting lists of tensors and gradients, so that the backpropagation can be commenced at multiple tensors at the same time using a single call. In Proppo, this propagator is often composed with another base propagator using `ComboProp`. The base propagator produces a tensor and the gradients, and `BackPropagator` commences the back propagation on the AD computation graph. Importantly, usually the config of the propagator allows determining whether `BackPropagator` should be include in the combo sequence or not. This is useful, if one wants to omit immediately backpropagating gradients in favor of grouping multiple tensors together and initiating each backprop simultaneously with a single call.

3.3 Example Monte Carlo gradient estimator implementations

One of our main motivations for developing AP software was to enable easy implementation of advanced MC gradient estimation methods, such as total propagation or Gaussian shaping gradients. Here, we provide details on how one might implement these methods using propagators. For reference, we also explain how simpler MC gradient estimators, such as RP or LR can be implemented.

Reparameterization. In PyTorch, RP gradients are already implemented, and it is possible to obtain reparameterized samples from a distribution, so that the gradients automatically flow through.

Likelihood ratio. Typically, LR gradients are implemented in AD software by constructing a surrogate loss (Schulman et al., 2015) of the form $\log p(x;\theta) f(x)$, and then differentiating this w.r.t. θ to obtain the LR gradient. In Proppo, we do not need to construct such a surrogate loss. We instead store $\log p(x;\theta)$ in the node as a tensor, then during the backward pass, we set $f(x)$ as the input gradient, and back propagate it into $\log p(x;\theta)$, using the `BackPropagator` in Sec. 3.2.

Total propagation. TP requires storing both data for RP and LR in its node. In addition, it requires setting a target tensor in the AD computation graph where the variances of LR and RP will be computed for the inverse variance weighting. We implement this by storing a pointer to the target tensor, and use `torch.autograd.grad` to compute the LR and RP gradients. After computing the gradient variances, the mixture weight can be computed, and `BackPropagator` from Sec. 3.2 is used for one last back propagation with the combined gradient.

4 Chaotic net minimalistic experiments for comparing gradient estimators

In this section we show minimalistic easily reproducible experiments using the Proppo framework. The main aim is to demonstrate that there exist situations where TP and GS gradients can improve the gradient accuracy by multiple orders of magnitude compared to conventional gradient estimation methods. The main result for TP is in Fig. 4, while the experiments for GS are in App. C.3.

4.1 Setup: gradient variance of a recurrent neural network with chaotic behavior

In our experiments, we estimate the gradient, $\boldsymbol{g}$, of an expected loss $\mathbb{E}_{p(\boldsymbol{x};\beta)}[f(\boldsymbol{x})]$, w.r.t. a parameter β, using various different MC gradient estimation methods. We repeat this estimation many times, and compute the empirical variance of the gradient estimators, $\hat{\mathbb{V}}[\boldsymbol{g}]$, from the set of estimates $\{\boldsymbol{g}^{(k)}\}_{k=1}^{K}$. In terms of the particular situation we consider, we were motivated by PIPPS (Parmas et al., 2018), where they showed a situation where chaos-like properties of the dynamics caused the RP gradient variance to explode, while LR gradients were robust. Parmas et al. (2018) considered a cart-pole swing-up and balancing task; while this task appears simple, there is still redundant complexity as the control policy had a high-dimensional parameter. Moreover, while the fractal input-output patterns that they visualized are strongly suggestive that the system was chaotic, it falls short of a rigorous proof. Therefore, we devised a similar simplified system based on a recurrent neural network (RNN).

We consider a particular 2-dimensional sigmoid recurrent neural network that was mathematically proven to exhibit chaotic behavior by Wang (1991). Formally, the system dynamics are given by

$$\boldsymbol{x}_{t+1} = \text{Sigmoid}(\beta W \boldsymbol{x}_t) + \varepsilon, \tag{2}$$

where $\text{Sigmoid}(y) \coloneqq \frac{1}{1+\exp(-y)}$, $W = \begin{bmatrix} -5 & 5 \\ -25 & 25 \end{bmatrix}$, β is the inverse temperature, a variable parameter of the network, $\varepsilon \sim \mathcal{N}(0, I\sigma^2)$ is Gaussian noise added to the state at each time step,[1] $\sigma = 0.001$ is the standard deviation of the added activation noise, and the initial state is $\boldsymbol{x}_0 = [0.35; 0.55]$. We use a batch size, $B = 1000$, and simulate this system for H time steps. We compute a quadratic loss at the final time step $L(\boldsymbol{x}_H) = \frac{1}{2}(\boldsymbol{x}_H - \mathbf{1})^{\intercal}(\boldsymbol{x}_H - \mathbf{1})$ and estimate the influence through all time steps to obtain the gradient w.r.t. β. Wang (1991) explained that β controls the dynamics of the system. When β is small, the system is well-behaved, but as β is increased, the system becomes chaotic through a period-doublings mechanism. We discuss this system in greater depth in App. C.1.

4.2 Single path experiment: showing the advantage of total propagation

Experimental setup. We consider an experiment on the simple RNN system (Sec. 4.1). We compute the gradient variances of RP, LR and TP gradients, and plot these against β. We considered the horizons $H \in \{10, 25, 50, 100\}$. The batch size was $B = 1000$ for 1 gradient estimate, $\boldsymbol{g}$ (averaged over the batch), and we performed this estimated $K = 800$ times. We compute the empirical gradient variance of the K estimates, $\{\boldsymbol{g}^{(k)}\}_{k=1}^{K}$. We use the percentile bootstrap method to estimate 95% confidence intervals. The results are in Fig. 4.

Results. The gradient variances are plotted in Fig. 4a. We see that RP is much more accurate than LR at low β, but the gradient variance of RP explodes as the system transitions to chaos around $\beta = 2.5$. This effect is stronger as the horizon is increased. Fig. 4b shows the ratio of improvement of TP over the best between RP and LR at each β. We see that around the edge of chaos at $\beta = 2.5$ TP improves by up to around 100 times compared to the best choice between RP and LR.

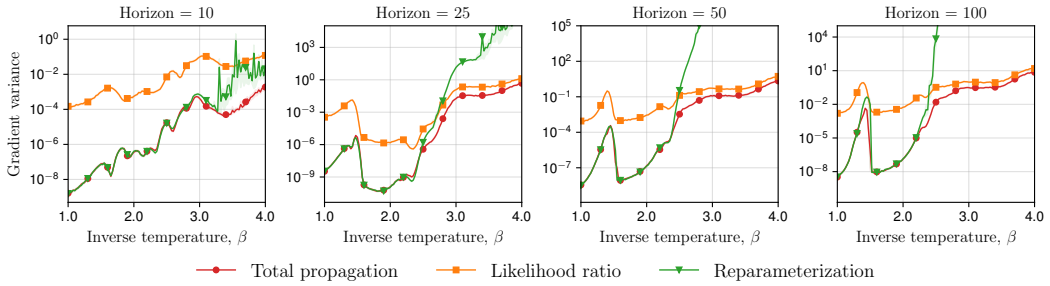
Discussion. Previously (Parmas et al., 2018) already showed an example where the RP gradient variance explodes. They also showed that LR was robust to this issue, and demonstrated that TP improves the performance. However, in their results, while the improvement of both TP and LR was huge compared to RP in the chaotic regime, the improvement of TP over LR was comparatively small, up to around a 3 times improvement. Here, on the other hand, we showed that TP can simultaneously improve over both RP and LR by multiple orders of magnitude. We also elucidated

[1] Note that Wang (1991) did not add any noise to the system, but the system still shows similar behavior even when such noise is added. We added this noise to be able to use LR gradients.

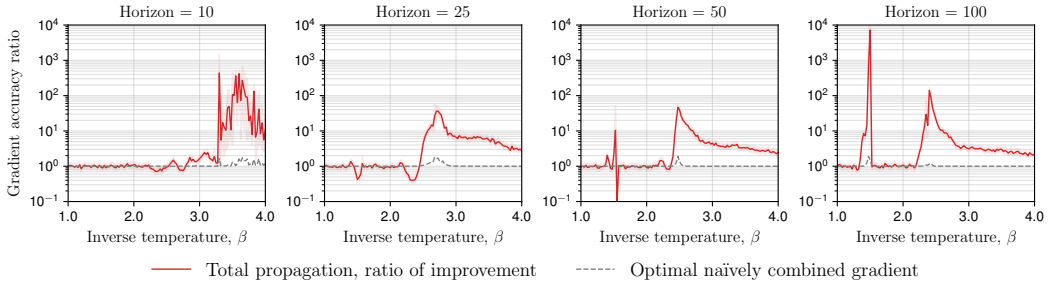
that the improvement may be particularly large at the edge of chaos. The simplicity of our RNN example allowed examinining the advantage of TP in greater depth with better reproducibility.

In Fig. 4b, we have additionally plotted the performance for an optimal "naïve" gradient combination based on applying inverse variance weighting to LR and RP computed separately,[2] without taking advantage of the graph structure of the computations. Such a naïve method was first mentioned by Parmas et al. (2018) who immediately skipped it for TP, because of its conceptual issues; however, later Metz et al. (2019) "followed the insight" from Parmas et al. (2018) and used the naïve method. In Fig. 4b, we see that the naïve method performs poorly, leading to hardly any improvement in gradient accuracy compared to the best between RP and LR.

Indeed, when combining statistical estimators, the theoretical maximum accuracy is the sum of the accuracies of the individual estimators. If two estimators are equally good, this would maximally lead to a doubling of the accuracy. However, in Fig. 4b we see improvements far exceeding this limit. Such a result was possible because TP takes advantage of the graph structure of the computations. It not only improves the accuracy of the final result, it also improves the accuracy of the intermediate gradient computations that are propagated backwards. The improvements in the accuracy of the intermediate calculations have a compounding effect, leading to orders of magnitude improvement of the final result. This advantage of TP highlights that automatic propagation software such as Proppo can produce results unattainable by standard back propagation procedures.



(a) Variance of RP, LR and TP gradient estimators against the inverse temperature β for various horizons.



(b) Improvement of TP over the best between LR and RP for each β.

Figure 4: Comparison of the RP, LR and TP gradient estimators on the chaotic RNN task. See Sec. 4.1 and Fig. 4 for the description of the task, and Sec. 4.2 for the discussion of the experiment. The main result is that TP provides a large improvement at the edge of chaos regime around $\beta \sim 2.5$.

5 Discussion & Conclusions

Recently, many impressive results were achieved by scaling up simple algorithms (Ramesh et al., 2022; Chowdhery et al., 2022). We note that there is no guarantee that simple algorithms will be sufficient for satisfactory performance. We ask instead, are simple algorithms used, simply because researchers and engineers fail at properly implementing more complicated ones? We hope that AP software will enable complex algorithms by providing a standard for sharing and building ML tools.

[2] Strictly speaking, we did not actually combine the gradient estimates, but only computed the variance of an idealized version of this estimator by using the precise variances $\mathbb{V}[\boldsymbol{g}_{\mathrm{RP}}]$ and $\mathbb{V}[\boldsymbol{g}_{\mathrm{LR}}]$. In practice, one also has to estimate these variances and the mixing ratio from each batch of samples.

Acknowledgements

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). {TensorFlow}: A system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283. 1

Anonymous (2022). Anonymous. In *Anonymous*, page 0000. 1

Barham, P. and Isard, M. (2019). Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183. 1

Bauer, F. L. (1974). Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96. 2.1

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43. 2.1

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs. 1

Chollet, F. et al. (2015). Keras. `https://keras.io`. 1

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. (2022). Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*. 5

Dangel, F., Kunstner, F., and Hennig, P. (2020). BackPACK: Packing more into backprop. In *International Conference on Learning Representations*. 1

Fairbank, M. (2008). Reinforcement learning by value gradients. *arXiv preprint arXiv:0803.3539*. A.2

Foerster, J., Farquhar, G., Al-Shedivat, M., Rocktäschel, T., Xing, E., and Whiteson, S. (2018). Dice: The infinitely differentiable Monte Carlo estimator. In *International Conference on Machine Learning*, pages 1529–1538. 1

Glynn, P. W. (1990). Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM*, 33(10):75–84. 2.2, A.1

Greensmith, E., Bartlett, P. L., and Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530. A.1

Hafner, D., Lillicrap, T., Ba, J., and Norouzi, M. (2020). Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*. 1

Jankowiak, M. and Obermeyer, F. (2018). Pathwise derivatives beyond the reparameterization trick. In *International Conference on Machine Learning*, pages 2240–2249. A.1

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*. 2.2, A.1

Krieken, E., Tomczak, J., and Ten Teije, A. (2021). Storchastic: A framework for general stochastic automatic differentiation. *Advances in Neural Information Processing Systems*, 34. 1

Metz, L., Maheswaranathan, N., Nixon, J., Freeman, C. D., and Sohl-Dickstein, J. (2019). Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*. 4.2, C.2

Minka, T. (2019). From automatic differentiation to message passing. Slides for Presentation at Advances and challenges in machine learning languages workshop. [Online; https://tminka.github.io/papers/acmll2019/minka-acmll2019-slides.pdf]. 1

Mohamed, S., Rosca, M., Figurnov, M., and Mnih, A. (2020). Monte Carlo gradient estimation in machine learning. *Journal of Machine Learning Research*, 21:1–62. 2.2, A.1

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. (2018). Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577. 1

Oktay, D., McGreivy, N., Aduol, J., Beatson, A., and Adams, R. P. (2020). Randomized automatic differentiation. In *International Conference on Learning Representations*. 1

Owen, A. B. (2013). *Monte Carlo theory, methods and examples*. A.1

Parmas, P. (2018). Total stochastic gradient algorithms and applications in reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 10204–10214. 1, A.1, A.2, A.2, A.2, A.2, A.2, C.3

Parmas, P. (2020). *Total stochastic gradient algorithms and applications to model-based reinforcement learning*. PhD thesis, Okinawa Institute of Science and Technology Graduate University. 1, A.1

Parmas, P., Rasmussen, C. E., Peters, J., and Doya, K. (2018). PIPPS: Flexible model-based policy search robust to the curse of chaos. In *International Conference on Machine Learning*, pages 4062–4071. 1, 2.2, 4.1, 4.2, A.1, A.2, A.2, C.1, C.1, 5, C.2

Parmas, P. and Sugiyama, M. (2021). A unified view of likelihood ratio and reparameterization gradients. In *International Conference on Artificial Intelligence and Statistics*, pages 4078–4086. PMLR. A.1

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32. 1

Paszke, A., Johnson, D. D., Duvenaud, D., Vytiniotis, D., Radul, A., Johnson, M. J., Ragan-Kelley, J., and Maclaurin, D. (2021). Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29. 1

Pearl, J. (1982). Reverend bayes on inference engines: a distributed hierarchical approach. In *Proceedings of the Second AAAI Conference on Artificial Intelligence*, pages 133–136. B.1

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., and Chen, M. (2022). Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*. 5

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, pages 1278–1286. 2.2, A.1, A.1

Rogozhnikov, A. (2021). Einops: Clear and reliable tensor manipulations with einstein-like notation. In *International Conference on Learning Representations*. 1

Schulman, J., Heess, N., Weber, T., and Abbeel, P. (2015). Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3528–3536. 1, 3.3, A.1

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR. A.2

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063. A.2

Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6. 1

Wang, X. (1991). Period-doublings to chaos in a simple neural network: An analytical proof. *Complex Systems*, 5(4):425–441. 4.1, 4.1, 1, C.1, C.1, 3, 5

Weaver, L. and Tao, N. (2001). The optimal reward baseline for gradient-based reinforcement learning. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 538–545. Morgan Kaufmann Publishers Inc. A.1

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256. 2.2, A.1

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] Our paper is about a framework, not a direct application.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A] No novel theoretical results.
   (b) Did you include complete proofs of all theoretical results? [N/A] No novel theoretical results.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] The experiments are simple, well documented and easy to reproduce. We will release the code upon publication.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [No] The experiments in this paper were small scale run on a CPU that anyone can replicate.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A] The used assets are well-known, e.g., PyTorch, so this is redundant.
   (c) Did you include any new assets either in the supplemental material or as a URL? [No] The code will be released upon publication.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] We did not use or curate data.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] We did not use or curate data.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]