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

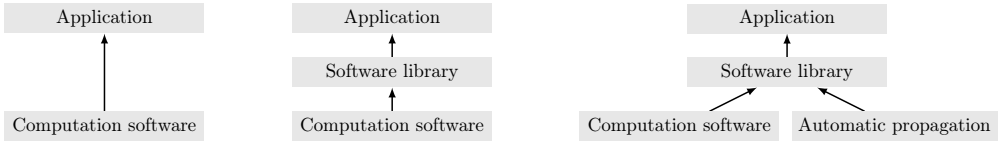
Paavo Parmas[*]
Kyoto University
paavo@sys.i.kyoto-u.ac.jp

Takuma Seno[†]
Keio University
seno@ailab.ics.keio.ac.jp

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.[3]

1 Introduction



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

Figure 1: A simplified taxonomy of machine learning (ML) software development methodologies. The *computation software* refers to existing ML tools such as PyTorch (Paszke et al., 2019) or NumPy (Harris et al., 2020); the *software library* is reusable code for a new ML method.

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 library for their algorithm, then use the library 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

[*]Affiliated with the Okinawa Institute of Science and Technology (OIST) during much of the project.
[†]Hired by OIST during his contribution to the project. Currently working at Sony AI.
[3]The code will be made available at https://github.com/proppo/proppo.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

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.

In ML a vast amount of effort has already been placed in developing computational software related to back propagation, ranging from established automatic differentiation (AD) libraries (Paszke et al., 2019; Tokui et al., 2015; Abadi et al., 2016; Bradbury et al., 2018); newer AD libraries(Oktay et al., 2020; Paszke et al., 2021); deep learning libraries (Chollet et al., 2015); extensions to AD libraries (Dangel et al., 2020; Rogozhnikov, 2021) to frameworks that use these AD libraries as tools to implement algorithms (Krieken et al., 2021; Tran et al., 2016; Gardner et al., 2018). Moreover, some libraries do not directly build on these frameworks, but can be combined together, e.g., Ray (Moritz et al., 2018) for 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 old design paradigms of existing 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 Monte Carlo (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.

Message passing (MP) is a promising candidate for a general ML framework because of the many already existing efficient algorithms that are based on it. Some of the prominent MP algorithms are belief propagation (Pearl, 1982) and expectation propagation (Minka, 2001) in probabilistic modeling, and turbo codes in information theory (McEliece et al., 1998). In modern deep learning, MP is used in graph neural networks (Wu et al., 2020). Also, back propagation can be seen as an MP algorithm that passes the gradient messages backwards. A closely related framework to MP is the Actor Model of computation (Hewitt et al., 1973), where programs operate by actors passing messages between each other. The Actor Model has been suggested as a promising framework for parallel AI systems (Hewitt and Lieberman, 1983) and has been noted to enable modularity (Agha, 1986). We aim to incorporate these strengths into a practical general-purpose message passing framework that enables conveniently implementing a variety of MP algorithms, and propose automatic propagation (AP).

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. 3.3) 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 without previously existing publicly available implementations, 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, as it makes clear that MC gradient estimators can be implemented as MP programs passing information backwards. 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` corresponds 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. $\mathbb{E}\left[\hat{\boldsymbol{g}}\right] = \frac{\mathrm{d}J}{\mathrm{d}\beta}$. 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. The LR method uses $f(\boldsymbol{x})$ to construct an estimator, whereas RP uses $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. Total propagation (TP) (Parmas et al., 2018) combines LR and RP using inverse variance weighting. Moreover, TP does so at a step-wise level at each sampling node. The required step-wise computations interfere with the natural operation of back propagation, necessitating AP software to implement TP. We give more background on these methods in App. A.

3 Proppo explanation

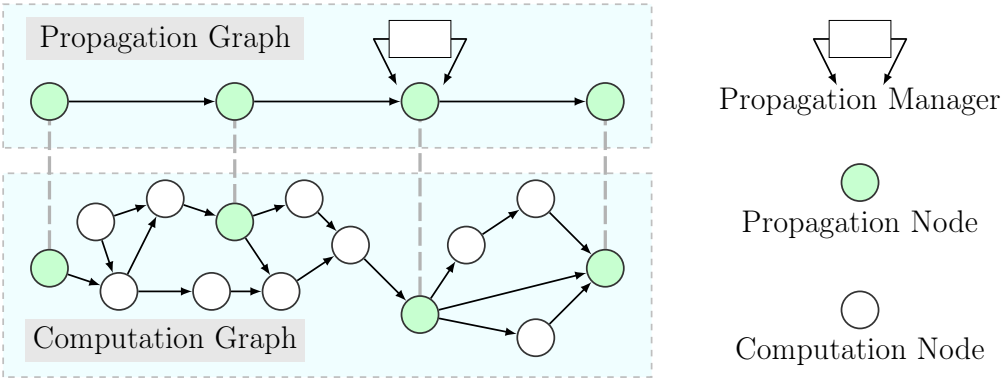


Figure 2: Automatic propagation framework (Sec. 3). A subset of the nodes ○ are designated as propagation nodes ○ forming the propagation graph. The propagation manager traverses the propagation graph and activates the propagators to ensure the correct functioning of the implemented algorithm. Here, the propagation graph is illustrated as a chain, but it may also have a graph structure. The gray dashed lines show the corresponding same nodes in the propagation and computation graphs.

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), then the processes by which they operate (Sec. 3.2). Finally, we explain AP from a more general perspective (App. B.1), loosening some of the specific choices in Proppo.

3.1 Components of Proppo

Computation nodes. 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 occurring 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.

3.2 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 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 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 backpropagate until the parameter node in the computation graph. 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 `config` without having to modify the program code. Finally, note that instead of changing the behavior to use MC gradient estimators by using `RPProp` or `LRProp`, one could also simply use standard back propagation by setting an appropriate different propagator in the configuration. 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. 3.3 we highlight common propagator types that can lead to composable and flexible algorithms.

3.3 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 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 creates 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. This example propagator is often composed with a *base propagator* using `ComboProp`. 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 the composition, 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 included 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.

3.4 Example Monte Carlo gradient estimator propagators

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. We provide detailed pseudo code for these propagator implementations in App. B.2. Here, we briefly outline some of the more prominent propagator implementations.

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. Alternatively, the output can be detached, and the gradients passed backwards using `BackPropagator`.

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. 3.3.

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. 3.3 is used for one last back propagation with the combined gradient.

3.5 Implementation and computation speed considerations

Primarily, Proppo does not perform any major computations on its own; it merely manages the operations performed by other computational tools (Sec. 3). This overhead is negligible for moderate size computations as confirmed by benchmarking experiments (App. B.3) and real applications (Anonymous, 2022). Experiments also confirmed that Proppo can scale to millions of propagation nodes (App. B.4). The scalability could be further increased by running multiple propagation managers on the propagation graph in parallel, and we plan to implement this in future work.

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

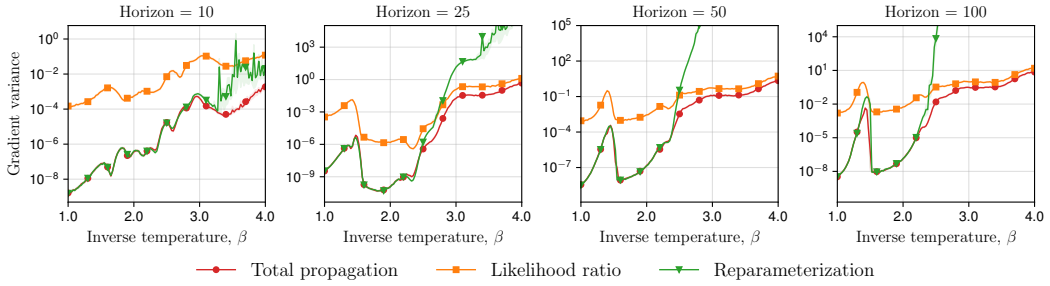
In our experiments, we estimate the gradient, $\boldsymbol{g}$, of an expected loss $\mathbb{E}_{p(\boldsymbol{x};\beta)}\left[f(\boldsymbol{x})\right]$, w.r.t. a parameter β, using various different MC gradient estimation methods. We repeat this estimation many times, and compute the empirical variance of the gradient estimators, $\hat{\mathbb{V}}\left[\boldsymbol{g}\right]$, 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

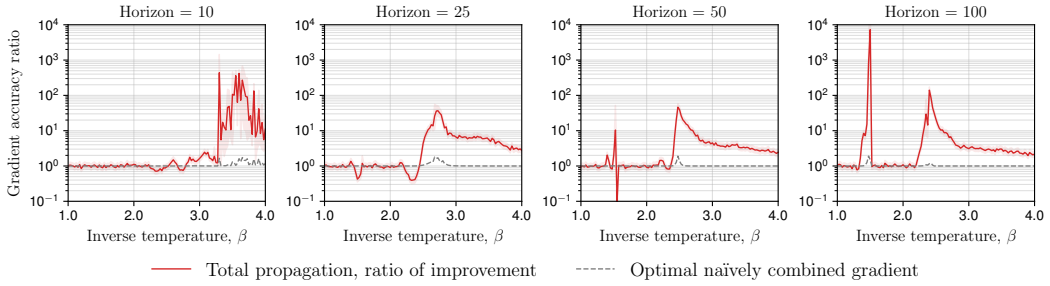
$$\boldsymbol{x}_{t+1} = \text{Sigmoid}(\beta W \boldsymbol{x}_t) + \varepsilon, \tag{2}$$

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,[4] $\sigma = 0.001$ 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] 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.



(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. 9 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 $\beta \sim 2.5$.

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 that the improvement may be particularly large at the edge of chaos. The simplicity of our RNN example allowed examining 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,[5] 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,

[5] 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 $\mathbb{V}[\boldsymbol{g}_{\text{RP}}]$ and $\mathbb{V}[\boldsymbol{g}_{\text{LR}}]$. In practice, one also has to estimate these variances and the mixing ratio from each batch of samples.

later Metz et al. (2019) "followed the insight" from Parmas et al. (2018) and used the naïve method in the context of metalearning. 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.

5 Related work

From the point of view of MC gradient estimation, the concurrent work, Storchastic (Krieken et al., 2021) is the most closely related to ours. Storchastic is a library for MC gradient estimation implemented in PyTorch based on the surrogate loss formalism (Schulman et al., 2015; Foerster et al., 2018). Their user interface of using a `.backward()` function is similar to ours, as both were inspired by the PyTorch interface that is the same. However, Storchastic is not an AP software, and for instance, it does not enable implementing Total Propagation.

From a conceptual point of view, the Actor Model (Hewitt et al., 1973) is related. In the Actor Model, everything in the program are actors and they communicate with each other via message passing. However, practically, Proppo is not implemented as an actor system. Unlike in an actor system, the propagators acting at the nodes can interact with outside objects directly without explicitly implemented messaging mechanisms. From a theoretical point of view, one could consider these interactions as "passing messages" but practically, these interactions are not implemented based on the actor formalism. Instead, AP software aims to be pragmatic and flexible to enable interusability with existing ML tools. A further difference is that AP automates the default messaging targets. Prominent Actor Model implementations, such as Akka (Boner et al., 2010) are typically used for server and web apps, and they are not primarily meant for implementing machine learning algorithms. Nevertheless, studying existing actor systems may inspire improvements to AP implementations.

6 Discussion & Conclusions

Recently, many impressive results were achieved by leveraging large scale data and computation together with basic algorithms (Brown et al., 2020; Ramesh et al., 2022; Chowdhery et al., 2022). We ask whether simple algorithms are used, simply because researchers and engineers have difficulty with properly implementing more complicated ones? There is no guarantee that merely scaling up our existing algorithms will be sufficient for satisfactory performance in the long term of ML research; we must also explore algorithmic improvements. We hope that automatic propagation will enable the proliferation of complex algorithms by providing a standard for sharing and building ML tools.

Author Contributions

Paavo Parmas invented automatic propagation software, designed and implemented most of the code including the key parts, ran the experiments, and wrote the paper.
Takuma Seno worked part-time to help write the code and give advice on the software engineering aspects in the early stages of the project. He also gave comments on the paper.

Acknowledgements

The project was started with funding provided by the Proof of Concept Program at the Okinawa Institute of Science and Technology (OIST). Most of the key contributions were made during this stage. PP finished the project while hired under the Cyborg-AI project supported by NEDO, Japan. PP would like to thank the Neural Computation Unit at OIST for administrative support while the project was being conducted at OIST.

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

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems*. 1

Anonymous (2022). Anonymous. In *Anonymous*, page 0000. 1, 3.5, B.3

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

Boner, J. et al. (2010). Akka. `https://akka.io/`. 5

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

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901. 6

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*. 6

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, 5

Gardner, J., Pleiss, G., Weinberger, K. Q., Bindel, D., and Wilson, A. G. (2018). Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. *Advances in neural information processing systems*, 31. 1

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

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

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362. 1

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. 1, 5

Hewitt, C. and Lieberman, H. (1983). Design issues in parallel architecture for artificial intelligence. In *In Proc. 28th IEEE Computer Society Int. Conf,, San Francisco CA*. Citeseer. 1

Ingraham, J., Riesselman, A., Sander, C., and Marks, D. (2019). Learning protein structure with a differentiable simulator. In *International Conference on Learning Representations*. C.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

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, 5

Maclaurin, D., Duvenaud, D., and Adams, R. (2015). Gradient-based hyperparameter optimization through reversible learning. In *International conference on machine learning*, pages 2113–2122. PMLR. C.1

McEliece, R. J., MacKay, D. J. C., and Cheng, J.-F. (1998). Turbo decoding as an instance of pearl's" belief propagation" algorithm. *IEEE Journal on selected areas in communications*, 16(2):140–152. 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.1

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

Minka, T. P. (2001). Expectation propagation for approximate bayesian inference. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 362–369. 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

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, 8

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. 1, 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*. 6

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

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, 3.4, 5, 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

Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., and Blei, D. M. (2016). Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*. 1

Wang, Q. (2013). Forward and adjoint sensitivity computation of chaotic dynamical systems. *Journal of Computational Physics*, 235:1–13. C.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, 4, C.1, 8

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

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24. 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 at `https://github.com/proppo` 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? [Yes] We included code in Appendix. D. The code will also be released upon publication at `https://github.com/proppo/proppo`.
   (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]

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

A Monte Carlo gradient estimators **16**

A.1 Basic Monte Carlo gradient estimation 16

A.2 Monte Carlo gradient estimation on Probabilistic Computation Graphs 16

B Automatic Propagation software: additional details and experiments **18**

B.1 A generalized view of automatic propagation software 18

B.2 Examples of propagators and their details: customizability and composability . . . 18

B.3 Computational time comparison 21

B.4 Scaling with the number of nodes 23

C Chaotic net: Additional details, explanations and experiments **24**

C.1 Explanation of chaotic dynamics of the recurrent neural network 24

C.2 Additional details of the single path chaotic net experiment 25

C.3 Multi path experiment: showing the advantage of Gaussian shaping gradients . . . 25

D Proppo prototype code listings **27**

A Monte Carlo gradient estimators

A.1 Basic Monte Carlo gradient estimation

In this section we provide additional background explanations on the two main gradient estimators: reparameterization gradients (RP) and likelihood ratio gradients (LR). Both are MC gradient estimation methods, i.e., they provide estimators, $\hat{\boldsymbol{g}}$, s.t. $\mathbb{E}\left[\hat{\boldsymbol{g}}\right] = \frac{\mathrm{d}}{\mathrm{d}\beta}\mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x};\beta)}\left[f(\boldsymbol{x})\right]$.

RP. In RP, one samples from a simple fixed distribution $\varepsilon \sim p(\boldsymbol{\varepsilon})$, and one defines a transformation, $\mathcal{T}$, dependent on β s.t. $\mathcal{T}(\boldsymbol{\varepsilon};\beta)$ has the same distribution as a sample from the original distribution $\boldsymbol{x} \sim p(\boldsymbol{x};\beta)$. Then, the derivative can be pushed inside the expectation, and the gradient can be estimated as

$$\frac{\mathrm{d}}{\mathrm{d}\beta}\mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x};\beta)}\left[f(\boldsymbol{x})\right] = \mathbb{E}_{\boldsymbol{\varepsilon}\sim p(\boldsymbol{\varepsilon})}\left[\frac{\mathrm{d}}{\mathrm{d}\beta}f\big(\mathcal{T}(\boldsymbol{\varepsilon};\beta)\big)\right] = \mathbb{E}_{\boldsymbol{\varepsilon}\sim p(\boldsymbol{\varepsilon})}\left[\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}}\frac{\mathrm{d}\mathcal{T}(\boldsymbol{\varepsilon};\beta)}{\mathrm{d}\beta}\right]. \tag{3}$$

For example, for a 1-dimensional Gaussian distribution $\mathcal{N}(x;\mu,\sigma)$, one possible reparameterization is $\mathcal{T}(\varepsilon;\mu,\sigma) = \mu + \varepsilon\sigma$, where $\varepsilon \sim \mathcal{N}(\varepsilon;0,1)$, and then $\frac{\mathrm{d}\mathcal{T}}{\mathrm{d}\mu} = 1$ and $\frac{\mathrm{d}\mathcal{T}}{\mathrm{d}\sigma} = \varepsilon$.

LR. While RP used the gradient of f to construct an estimator, LR, instead, uses its value. The LR gradient estimator is characterized by the equation

$$\frac{\mathrm{d}}{\mathrm{d}\beta}\mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x};\beta)}\left[f(\boldsymbol{x})\right] = \mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x};\beta)}\left[\frac{\mathrm{d}\log p(\boldsymbol{x};\beta)}{\mathrm{d}\beta}\big(f(\boldsymbol{x}) - b\big)\right], \tag{4}$$

where b is a baseline for variance reduction, often computed as the batch mean of the samples $b = \frac{1}{B}\sum_{i=1}^{B} f(\boldsymbol{x}^{(i)})$.

Both LR and RP estimators are interchangeable (as long as the gradients and value of f are available), and which one is better depends on the specific situation. While RP tends to handle high dimensional spaces better (Rezende et al., 2014), it often fails on long computation graphs (Parmas et al., 2018). On the other hand, LR has robust behavior, and does not require access to ∇f, but does not scale well with the dimensionality. The two estimators were recently unified by Parmas and Sugiyama (2021) based on an intuititve probability flow theory related to the work of Jankowiak and Obermeyer (2018).

A key metric to determine which gradient estimator is effective is the *variance* of the gradient estimator, $\mathbb{V}\left[\hat{\boldsymbol{g}}\right]$. The variance can be reduced by a factor $1/K$ by computing K samples of the same estimator and averaging; therefore, we could consider a reduction in the gradient variance by a factor K to be roughly equivalent to increasing the computation speed by a K factor.[6] For this reason, much research on MC gradient estimators has focused on reducing the variance, primarily by using control variates and baselines (Greensmith et al., 2004; Weaver and Tao, 2001); or conditioning and importance sampling (Owen, 2013). Another line of research takes advantage of the graph structure of the computations to obtain more accurate gradient estimates (Parmas et al., 2018; Parmas, 2018, 2020). While the former methods are readily implemented using the surrogate loss formalism (Schulman et al., 2015), the latter methods are not easily implemented, motivating the creation of Proppo.

In this section, we discussed how the basic estimators are implemented through a single sampling operation. In the next section, we discuss graphs with multiple stochastic operations, and also introduce the total propagation and Gaussian shaping gradient methods, which take advantage of this graph structure.

A.2 Monte Carlo gradient estimation on Probabilistic Computation Graphs

In the deterministic case, the total derivative intuitively decomposes into a sum across the paths as shown in Eq. (1); Parmas (2018) explained that a similar framework can be employed for stochastic graphs, using their probabilistic computation graph (PCG) formalism. In a PCG, we note that while the sampling operations themselves are *stochastic*, the relationship between the marginal distributions

[6] Note that in practice parallel computation may allow increasing the batch size without proportionally increasing the computation time, if the computational resources are not already maxed out. Moreover, reducing the gradient variance by a factor K may not guarantee proportionally faster optimization because increasing the gradient accuracy has diminishing returns once the gradient is sufficiently accurate.

at nodes is *deterministic*. Therefore, by denoting $\zeta_{\boldsymbol{x}_i}$ as the abstract parameters of the marginal distribution at the node i, and replacing the usual derivative with functional derivatives between the probability distributions, we can obtain a similar decomposition of the total gradient as in Eq. (1)

$$\frac{\mathrm{d}\zeta_y}{\mathrm{d}\beta} = \sum_{\text{Path}\in\text{Paths}[\boldsymbol{\beta}\to y]} \prod_{\text{Edge}[l\to k]\in\text{Path}} \frac{\partial \zeta_{\boldsymbol{x}_k}}{\partial \zeta_{\boldsymbol{x}_l}} \tag{5}$$

Parmas (2018) proposed a further decomposition by assigning a set of intermediate nodes $\mathcal{N}$, and considering the paths passing through the nodes $n \in \mathcal{N}$ giving the equation

$$\frac{\mathrm{d}\zeta_y}{\mathrm{d}\beta} = \sum_{n\in\mathcal{N}} \left(\sum_{\text{Path}\in\text{Paths}[n\to y]} \prod_{\text{Edge}[l\to k]\in\text{Path}} \frac{\partial \zeta_{\boldsymbol{x}_k}}{\partial \zeta_{\boldsymbol{x}_l}} \right) \left(\sum_{\text{Path}\in\text{Paths}[\beta\to n]\setminus\mathcal{N}} \prod_{\text{Edge}[l\to k]\in\text{Path}} \frac{\partial \zeta_{\boldsymbol{x}_k}}{\partial \zeta_{\boldsymbol{x}_l}} \right), \tag{6}$$

where $\text{Paths}[\beta \to n] \setminus \mathcal{N}$ denotes the paths going from β to n, but not passing through nodes in $\mathcal{N}$. Parmas (2018) further showed that Eqs. (5) and (6) can be combined to yield

$$\frac{\mathrm{d}\zeta_y}{\mathrm{d}\beta} = \sum_{n\in\mathcal{N}} \frac{\mathrm{d}\zeta_y}{\mathrm{d}\zeta_n} \left(\sum_{\text{Path}\in\text{Paths}[\beta\to n]\setminus\mathcal{N}} \prod_{\text{Edge}[l\to k]\in\text{Path}} \frac{\partial \zeta_{\boldsymbol{x}_k}}{\partial \zeta_{\boldsymbol{x}_l}} \right), \tag{7}$$

and they explained that this equation generalizes the deterministic and stochastic policy gradient theorems (Sutton et al., 2000; Silver et al., 2014). In particular, the $\frac{\mathrm{d}\zeta_y}{\mathrm{d}\zeta_n}$ terms are estimated by total derivative estimators, such as LR or value gradient methods (Fairbank, 2008), whereas the effect of the local partial derivatives, $\frac{\partial \zeta_{\boldsymbol{x}_k}}{\partial \zeta_{\boldsymbol{x}_l}}$, is estimated by pathwise estimators, such as direct differentiation or RP. They also proposed more advanced estimators: total propagation that combines RP and LR (Parmas et al., 2018) and Gaussian shaping gradients that use a different decomposition of the paths to obtain a gradient estimator (Parmas, 2018).

Total propagation (TP). Inverse variance weighting is the optimal method to take a weighted average of two uncorrelated statistical estimators, as is well known in statistics. The TP method (Parmas et al., 2018) uses this weighting scheme to obtain a weighted average of LR and RP, based on the observation that both estimators are interchangeable. In particular, it performs the computation $\hat{\boldsymbol{g}}_{\text{TP}} = k\hat{\boldsymbol{g}}_{\text{LR}} + (1-k)\hat{\boldsymbol{g}}_{\text{RP}}$, where $k = \mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{RP}}\right] / (\mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{LR}}\right] + \mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{RP}}\right])$, i.e. it picks $k \propto \frac{1}{\mathbb{V}[\hat{\boldsymbol{g}}_{\text{LR}}]}$ and $(1-k) \propto \frac{1}{\mathbb{V}[\hat{\boldsymbol{g}}_{\text{RP}}]}$. The gradient variances for computing the weights are obtained from the empirical variances of the gradient samples. Moreover, TP is not a simple combination of the two estimators computed separately on the whole graph, instead, it combines the two estimates at each sampling node, and propagates the combined gradient backwards, potentially leading to much increased accuracy (the advantage is experimentally clear in Sec. 4.2). However, this kind of gradient variance estimation during the backward computation poses problems for existing AD software, as it cannot be implemented by differentiating a surrogate loss.

Gaussian shaping gradients (GS). The GS method (Parmas, 2018) is interesting because it allows obtaining an LR type gradient estimator while using ∇f instead of f as is usual in LR gradients. This allows better scalability with the dimensionality compared to a regular LR method, as we show experimentally in Sec. C.3. The basic idea is to assume a Gaussian density with parameters $\boldsymbol{\mu}$ and Σ at a distal node n, then construct an LR-type gradient estimator for these parameters, estimating the gradients $\frac{\mathrm{d}\boldsymbol{\mu}}{\mathrm{d}\boldsymbol{x}}$ and $\frac{\mathrm{d}\Sigma}{\mathrm{d}\boldsymbol{x}}$. Assuming a cost function $c(\boldsymbol{x}_n)$ that depends on $\boldsymbol{x}_n$, we can resample points on the approximated Gaussian distribution, estimate the gradients $\frac{\mathrm{d}\mathbb{E}[c]}{\mathrm{d}\boldsymbol{\mu}}$ and $\frac{\mathrm{d}\mathbb{E}[c]}{\mathrm{d}\Sigma}$ and apply the chain rule to obtain the total gradient from $\boldsymbol{x}$ to c. The final algorithm resembles LR, except that the usual f multiplier is replaced with a different scalar given by the dot product of some statistics of the distribution at $\boldsymbol{x}_n$ with $\frac{\mathrm{d}\mathbb{E}[c]}{\mathrm{d}\boldsymbol{\mu}}$ and $\frac{\mathrm{d}\mathbb{E}[c]}{\mathrm{d}\Sigma}$. Similarly to TP, this method is also cumbersome to implement by differentiating a surrogate loss, but can be intuitively expressed as a message passing program that can be implemented in Proppo.

B Automatic Propagation software: additional details and experiments

B.1 A generalized view of automatic propagation software

In the main contents, we introduced AP software from the viewpoint of Proppo, our prototype implementation of an AP library. We did this to provide a concrete example to aid in clarity. More generally, AP software may have many more features than introduced, or on the contrary, it may also be more minimalistic than our example with Proppo. In this section we aim to clarify what type of additional features we foresee to be useful for AP software, and also to clarify what are the minimum requirements for something to qualify as an AP library.

First, we note that it may be useful for the `forward` computations to also be able to send messages. Moreover, it can be beneficial for the forward methods to be able to pass information to the propagation manager, so as to automate the choice of propagators for subsequent nodes. In fact, Proppo already implements such automation in the choice of the configuration for the loss propagators depending on what type of MC gradient estimator was previously used. This observation points toward generalizations concerning the methods of the propagators as well as the managers.

Regarding the propagators, in our discussion we limited the computations to `forward` and `backward` computations. However, there is no need for such a restriction, and in general we require propagators to implement *types* of computations, where `forward` and `backward` are just two possible types. The different types of computations can in general access and modify the memory in the node, and may send messages to other nodes. We note that the distinction between types of computations is superficial because the multiple types of computations may also be embedded into a single type of computation, where an input can be passed to switch between the different embedded computations.

Regarding the propagation manager, this was merely our choice of implementation to keep track of the propagation graph, automatically decide on the order of activating the propagators, and to deal with passing the messages. Any other method to implement such functionality would also be allowed.

In summary the crucial aspects to automatic propagation software are:

1. Nodes that can store information.
2. Propagators that can be associated to the nodes. The propagators have user programmable methods that can directly modify the information in the nodes that they are linked to, and can indirectly influence other nodes by sending them messages containing general information.
3. The methods of the propagators may directly interact with processes and information external to the nodes.
4. When a message is sent from a propagator operating at a node, if a target for the message was not explicitly specified by the user, a default target is determined.
5. It is possible to trigger the system so that the propagators at multiple nodes are automatically activated in sequence.

Regarding points 4 and 5, we envision that typically the nodes will be arranged in an acyclic directed graph structure, and the system can be triggered to traverse the graph backwards, activating the `backward` method of the propagator at each node, and sending the messages to the parent nodes. Even if the contents of a message are not intended to be used by a direct parent node, but they are meant for a node earlier in the graph, as long as this message is continually routed backwards it will eventually reach the target destination. Thus we envision that such a default setting will allow implementing a wide range of algorithms. In fact, there already exist many prominent machine learning algorithms that operate purely based on sending messages backwards locally, e.g. back propagation or belief propagation (Pearl, 1982).

B.2 Examples of propagators and their details: customizability and composability

In this section we aim to illustrate the composability and customizability that can be achieved in AP software. These properties are achieved by using the sequence propagators described in Sec. 3.3. These propagators allow combining multiple propagators together into a sequence to compose new propagators. The forward and backward methods of the propagators will be activated in sequence.

One of our main design patterns to effectively use these sequence propagators is to create a base propagator for each non-trivial functionality that we want to implement. Then we plug these base propagators together into a chain to achieve all of the desired properties. Some of these base propagators may be reused across many different composite propagators, allowing to create compact code. For example the `BackPropagator` in Sec. 3.3 can be combined with many different MC gradient estimation propagators to initialize the back propagation of the gradients.

Another feature of the sequence propagators is that they allow optional propagators in the sequence. These optional propagators can be turned on or off using keyword arguments when instantiating the propagator from the class. Essentially, this allows creating a factory for a composite propagator with a rich configuration space. In the following, we give pseudocode and details of several MC gradient estimation propagators, their constituent propagators, and explain how they interact.

BackPropagator

This propagator commences backpropagating the gradients at a computation node.

Forward:
`pass`

The forward method does not do anything, as this propagator is designed to be combined together with other propagators that will perform the necessary forward computations.

Backward:
```
get from message:  tensors, grad_tensors
call:  torch.autograd.backward(tensors, grad_tensors)
```

The backward method retrieves the tensors and gradients, and backpropagates them using AD software as explained in Sec. 2. One of the propagators that can be combined together with `BackPropagator` is `RPBase` for reparameterization gradients explained next.

RPBase

This is the base propagator for reparameterization gradients. It injects reparameterized noise into an input variable, and allows backpropagating through this stochasticity.

Forward:
```
input from program code:  x
inject reparameterized noise into x
store into node:  output, detached output
```

Backward:
```
get grads from detached output
message:  (output, grads)
```

Note that the backward method does not backpropagate the gradients on the computation graph. It merely forms the pair of the output node and its corresponding gradient. To have this gradient be backpropagated as well, the following compound `RPProp` propagator can be used.

RPProp

This propagator applies the reparameterization transformations and also automatically commences the backpropagation on the computation graph.

Sequence: `[Optional(BackPropagator), RPBase]`

Here we used the `Optional()` notation to mean that the `BackPropagator` can be either included or omitted using a keyword, i.e., `RPProp(backprop=False)` would omit the `BackPropagator`. If it is included, then having simply plugged these two propagators together causes the gradients to be backpropagated on the computation graph as well—`RPBase` will form the `(tensors, grad_tensors)` pair, and `BackPropagator` will commence the back propagation.

LossBase

In many machine learning tasks, there are loss nodes that we want to propagate gradients from. Moreover, for likelihood ratio gradient estimators, we may also want to send the value of the loss itself backwards as well. This propagator implements the required base functionality.

Configuration: whether gradients are needed or not

Forward:
```
compute loss
store loss in the node
```

Backward:
```
get from message:  incoming sum of losses
optional:  create tensors and grad_tensors for the loss
optional:  sum new loss with incoming losses
message:  [(tensors, grad_tensors), sum of losses]
```

LossProp

To add automatic gradient backpropagation, or baseline computations to the loss node, we create a compound propagator with the following sequence.

Sequence: `[Optional(BackPropagator), Optional(BaselineProp), LossBase]`

BaselineProp

In `LossProp`, one part of the sequence was the baseline propagator. The baseline propagator subtracts a baseline from the sum of the loss to reduce the likelihood ratio gradient variance.

Configuration: the type of baseline function to use, e.g., subtract the mean

Forward:
```
pass
```

Like the `BackPropagator`, the baseline propagator is also designed to be used together with other propagators, and it does not need its own forward method.

Backward:
```
compute baseline
subtract baseline from losses
message:  loss with baseline subtracted
```

LRBase

This propagator implements the base functionality for using likelihood ratio gradient estimators.

Forward:
```
input from program code:  x
inject noise into x
compute log p(x)
store into node:  log p(x)
```

Backward:
```
get from message:  incoming loss with the baseline subtracted if available
get from node:  log p(x)
message:  (tensors=log p(x), grad_tensors=loss)
```

LRProp

This propagator adds optional backpropagation and baseline functionality to the base propagator for LR gradients.

Sequence: `[Optional(BackPropagator), LRBase, Optional(BaselineProp)]`

Notice that in `LRProp` the `BaselineProp` is at the right side of the sequence, whereas for `LossProp` it is at the left side. The backward sequence commences from right to left. In the `LRProp` case, we want to subtract the baseline before estimating the LR gradient, whereas in the `LossProp` case, we want to subtract the baseline in the end, after having computed the loss. Which one is better is problem dependent—if there is a single loss node, but many LR nodes, it may be better to subtract the baseline at the loss node, and *vice versa*.

TPBase

This is the base propagator for the most complicated propagator we show in our examples here—it implements the computations for the total propagation algorithm.

Configuration: what node to use for inverse variance weighting

Forward:
```
input from program code:  x
inject reparameterized noise into x
compute log p(x)
store into node:  log p(x), output, detached output
```

Backward:
```
get from message:  incoming loss with the baseline subtracted if available
get from node:  log p(x), output, detached output
get grads from detached output
compute LR and RP gradients until the inverse gradient variance node
perform inverse variance weighting and compute the mixing ratio, k
message:  (tensors=[log p(x), x], grad_tensors=[k*loss, (1-k)*grads])
```

Note that here we are packing two sets of `tensors` and `grad_tensors` together into a list. The `torch.autograd.backward()` function that is called in `BackPropagator` can handle lists of such pairs, and simultaneously invoke the backpropagtion, so there is no issue.

TotalProp

Finally, this propagator adds the back propagation and baseline functionalities to the base propagator for TP gradients.

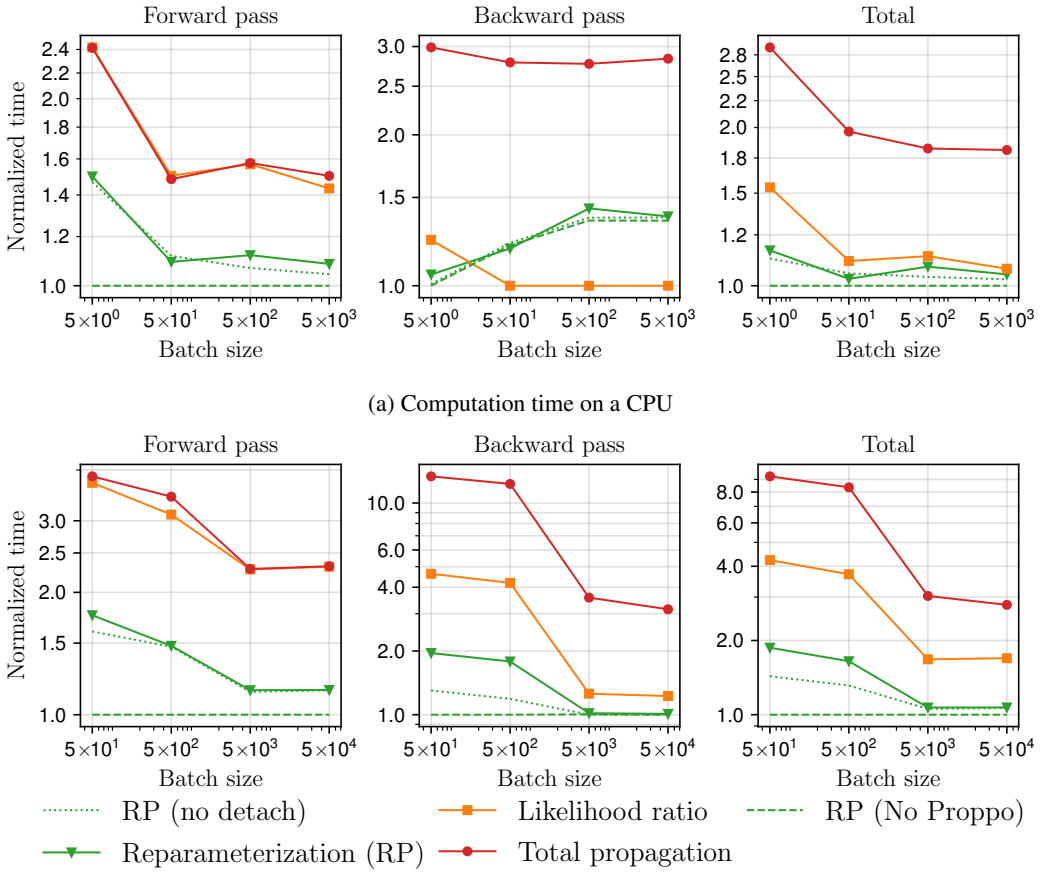
Sequence: `[Optional(BackPropagator), TPBase, Optional(BaselineProp)]`

B.3 Computational time comparison

Setup. To give an indication of the overhead in computational time caused by using Proppo, we perform experiments with a similar recurrent neural network as in Sec. 4.1, but while varying the batch size (Fig. 5) or dimensionality (Fig. 6). When varying the batch size, the dimensionality was fixed to 500; and when varying the dimensionality, the batch size was fixed to 1000. The horizon was 10. We performed the forward and backward computations 100 times and estimated the average computation time both on a CPU and on a GPU. The computation times were normalized with the minimum computation time at the corresponding setting to better highlight the ratio difference. We compared the computation times of RP, LR, TP and when implementing RP without using Proppo. Moreover, for RP, we test two implementations: one which detaches the tensors at each propagation node, and manually back propagates the gradients; and another that does not detach the tensors, and allows PyTorch to handle the gradient back propagation.

Discussion. In the results in Figs. 5 and 6 we see that for very small problem sizes, Proppo causes a significant overhead, but as the problem size becomes larger, the overhead becomes negligible. For large problem sizes, typically TP required 2 times more computational time. Compared to a 100 time reduction in gradient variance (Sec. 4.1), this additional computational time is negligible (note that the naïve way to reduce gradient variance by a factor K is to increase the batch size by the same factor K roughly requiring K times more computational time). We also note that in a typical full implementation of an ML algorithm, Proppo may be used in only some section of the computations. In this case, the overhead caused by Proppo may be only a small fraction of the total computational time associated with the algorithm. We observed this point in our concurrent work in model-based reinforcement learning (Anonymous, 2022), where the total change in computational time was typically less than 50% extra.

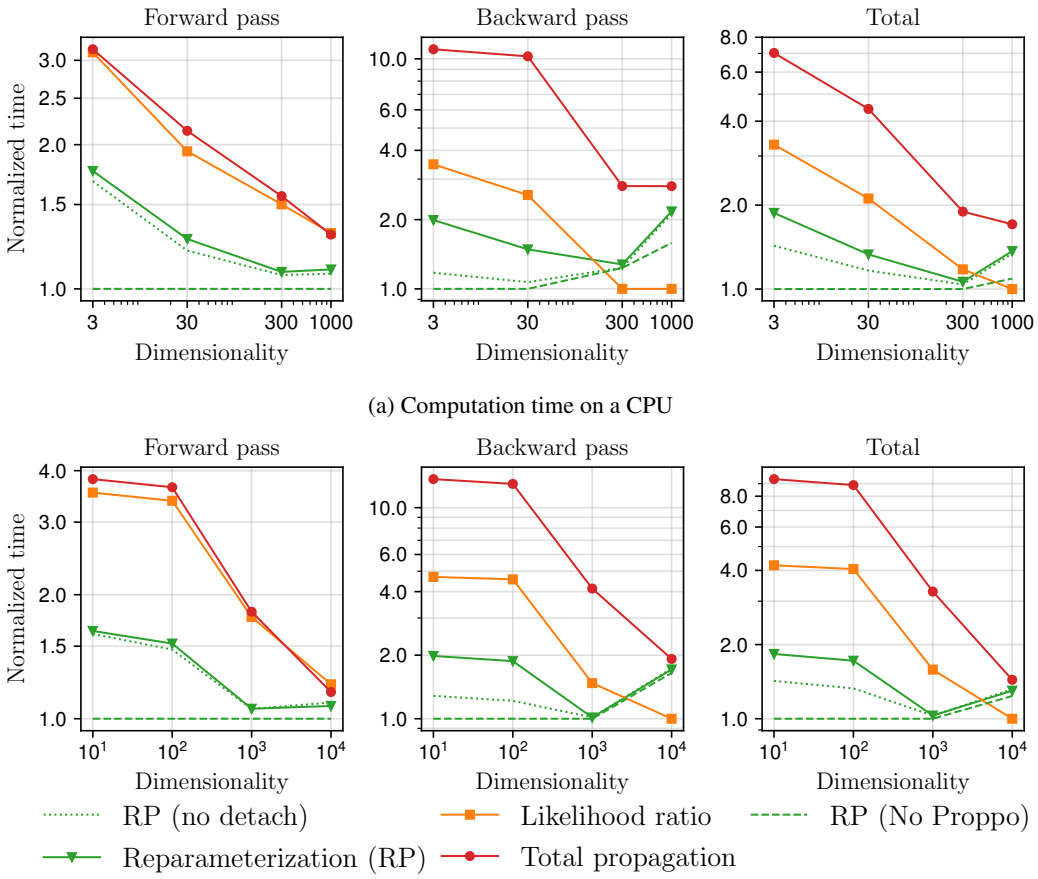
Another point to note is the difference between the used *framework* and *implementation* of an algorithm. The performance will primarily depend on the implementation. Proppo allows creating multiple implementations, fast or slow ones (e.g., compare the two implementations of RP). The main point is not so much that the computation time might increase a bit if Proppo is used, but rather that using Proppo enables implementing algorithms that would be cumbersome to implement otherwise. The current implementations are also not fully tuned, and can be implemented to run faster. In particular, the implementation of TP has issues with scalability if the number of parameters used for the inverse variance weighting becomes large, but we have found sensible solutions to this issue in our concurrent work (Anonymous, 2022).



(a) Computation time on a CPU

(b) Computation time on a GPU

Figure 5: Computation times of algorithms in Proppo when varying the batch size (Sec. B.3).



(a) Computation time on a CPU

(b) Computation time on a GPU

Figure 6: Computation times of algorithms in Proppo when varying the dimensionality (Sec. B.3).

B.4 Scaling with the number of nodes

Setup. We perform an experiment to test how Proppo scales with the number of propagation nodes. In the experiment, we connect nodes into a chain with a dummy propagator that performs no computations, but merely stores an empty node in the forward pass, and sends empty messages in the backwards pass. This experiment allows testing the overhead associated with the "bookkeeping" performed by Proppo. We ran 10 experiments, and averaged the elapsed time. We tested values between 10 and 10^7 nodes, and the experiment was performed on an Intel i9 CPU. In Fig. 7, we show the results separately for the forward and backwards pass, as well as the total elapsed time.

Discussion. The results in Fig. 7 show that the computation time increases linearly. In particular, we see that the time spent per node is roughly 10^{-6} seconds. This means that if the computations performed by the propagator take longer than 10^{-6} seconds, we would expect the additional overhead for storing nodes and passing messages caused by Proppo to be negligible. This overhead will be non-negligible for extremely simple calculations such as summing scalars together; however, the aim of Proppo is to facilitate implementing complicated algorithms. For complicated algorithms, we expect the computations at each node to take longer than 10^{-6} seconds, so in practice, the overhead caused by Proppo will be negligible. Moreover, we see that Proppo can scale to millions of propagation nodes, while, in our experience applying Proppo to practical problems, we have so far not needed more than a thousand nodes. Finally, we note that this experiment does not guarantee that the implementation utilizing Proppo will have the same computation time as one without it. Typically, when Proppo is used, it would tweak the operation of some underlying computational software. These tweaks may interfere with the normal operation of the computational software, making it perform

slightly slower. For example, when Proppo is used to override the standard back propagation in PyTorch, it would manually pass gradients backwards at the propagation nodes, instead of letting PyTorch automatically back propagate. This may slow down the computation due to the interference. To obtain optimal performance, one should only use Proppo when it is needed, or when it increases convenience due to greater modularity.

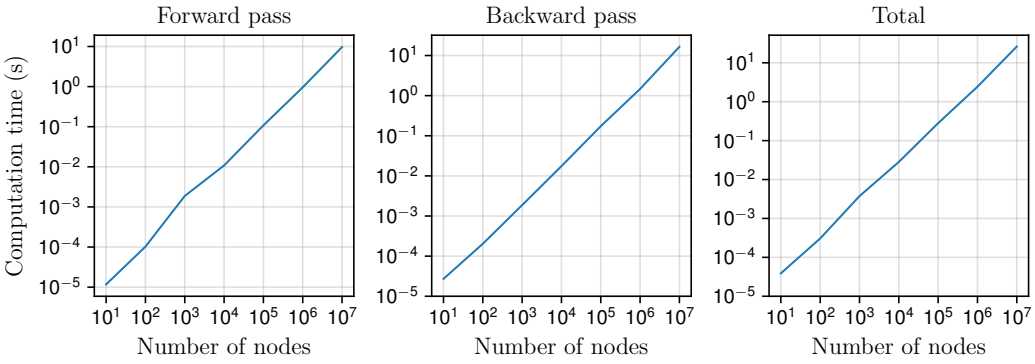


Figure 7: Scaling of computation time with the number of propagation nodes (Sec. B.4).

C Chaotic net: Additional details, explanations and experiments

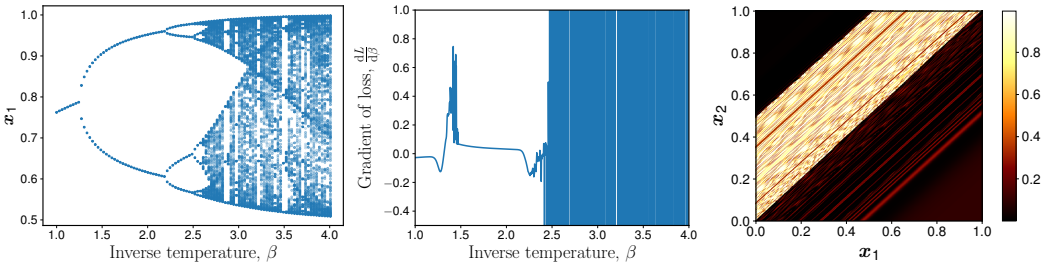
In Sec. C.1, we provide additional explanations about the chaotic net experiments (Sec. 4); in Sec. C.2, we provide additional explanations of the single path experiment (Sec. 4.2); in Sec. C.3, we present an additional experiment designed to show the utility and better scalability w.r.t. the dimensionality of Gaussian shaping gradients.

C.1 Explanation of chaotic dynamics of the recurrent neural network

Wang (1991) explained that the inverse temperature parameter β controls the dynamics of the RNN system introduced in Sec. 4.1. When β is small, the system is well-behaved, but as β is increased, the system becomes chaotic through a period-doublings mechanism. They illustrated this via a bifurcation diagram that we have replicated in Fig. 8a. In this figure, for each β we simulate the system for 10000 steps, and plot the first dimension $\boldsymbol{x}_1$ of the last 500 steps onto the figure as dots (note that one obtains a similar diagram when considering $\boldsymbol{x}_2$, the second dimension of the system). We see that when β is small (around 1), all of the dots are at the same position; hence the system converged to a fixed state. As β is increased, the system starts oscillating between two states. As β is further increased, the states further split, with a phase transition happening around $\beta = 2.5$ leading to chaotic oscillation.

This chaotic behavior causes the gradient to be ill-behaved as illustrated in Fig. 8b. In this figure, we followed the experimental protocol of Parmas et al. (2018), and plotted the RP derivative of the loss function w.r.t. β while keeping the random number seed fixed. We see that the derivative is well-behaved in the non-chaotic region, but starts rapidly oscillating up and down with a large magnitude as the system becomes chaotic at the phase transition around $\beta = 2.5$. Moreover, the gradient variance also explodes. Parmas et al. (2018) explained this behavior by plotting the loss landscape of their system w.r.t. the start position. We have replicated a similar result for the RNN system in Fig. 8c. Recall from the preliminaries (Sec. 2, see also Eq. (7)) that the total derivative sums the terms $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{x}}\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\beta}$ for each time step. As the loss landscape L has a fractal structure due to the chaotic properties of the system, the gradient $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{x}}$ is oscillating rapidly. Thus, if one tries to average the gradients together over some region of this landscape by sampling the gradients in said region, the variance of this estimate will explode, and it is impossible to compute a sensible gradient direction using the RP method. The LR gradient, on the other hand, does not use $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{x}}$, it only uses L to estimate the gradient. Thus, small amplitude fluctuations of L do not affect the LR gradient, and it is robust to the issues with chaos.

Similar chaotic properties occur in many ML tasks. We already introduced the work of Parmas et al. (2018) in model-based reinforcement learning that was the basis for much of our discussion. Their work appears to be the first to discuss the explosion of the gradient variance due to chaos when the computations are stochastic, and they also suggested to incorporate LR methods to tackle the issue. Similar chaotic properties have also been discussed in metalearning (Maclaurin et al., 2015; Metz et al., 2019) and protein folding software (Ingraham et al., 2019). Moreover, the effect of chaos on estimating the sensitivity of fluid simulations has also been studied in many works, e.g., the work by Wang (2013). We believe that our minimalistic experiment captures interesting characteristics of such challenging ML tasks.



(a) Bifurcation diagram for $\boldsymbol{x}_1$ (b) Gradient (c) Loss landscape w.r.t. start location

Figure 8: Illustration of the behavior of the chaotic sigmoid recurrent neural network (Sec. 4.1). (a) Bifurcation diagram of the RNN. The activation noise was removed, $\sigma = 0$, to replicate previous results with a deterministic RNN (Wang, 1991). Note that the result does not change much when noise is added, the dots are just spread out around the location on the current figure. (b) Gradient of the objective plotted against β, similarly to (Parmas et al., 2018). The horizon was $H = 100$. (c) Fractal loss landscape. The parameters were $\beta = 3.5$, resolution of the grid: 500×500, horizon: $H = 15$. We also added a fixed perturbation on W_{11} sampled from a Gaussian with standard deviation $\sigma_w = 0.1$. This was done for aesthetic reasons, and to show that the system stays chaotic even when perturbed.

C.2 Additional details of the single path chaotic net experiment

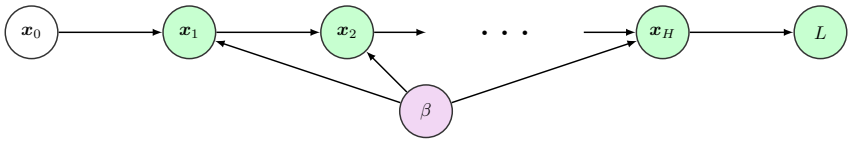


Figure 9: Single path chaotic RNN probabilistic computation graph diagram.

Figure 9 includes a schematic diagram of the RNN computations in the experiments in Sec. 4.2. The computations start from an initial state $\boldsymbol{x}_0$, and are simulated forwards for H steps. A loss, L, is computed at the last step. All of the nodes when $t \geq 1$ are propagation nodes ◯. The $\boldsymbol{x}$ nodes are propagation nodes due to the sampling of noise, and gradient estimation, while the loss node, L, is implemented as a loss propagator. The β node ◯ is the parameter node where the inverse variance weights are computed for use in the TP algorithm.

C.3 Multi path experiment: showing the advantage of Gaussian shaping gradients

Experimental setup. In this section, the main aim is to show the better scalability of Gaussian shaping gradients compared to regular LR gradients. To this end, we modify the simple RNN in Sec. 4.1 by replicating multiple instances of this RNN, and computing them in parallel; thus, increasing the dimensionality of the system. The setup is illustrated in Fig. 10. The additional parallel dimensions act as nuisance variables on the final loss, increasing the variance of the LR gradient estimator. Formally, the state is modified into $\tilde{\boldsymbol{x}} := [\boldsymbol{x}^{(1)}; \ldots; \boldsymbol{x}^{(D)}]$, and the evolution of each $\boldsymbol{x}^{(d)}$ is computed separately according to Eq. (2). While the initial state is the same for each d, the added activation noise ε is different, so the trajectories for each parallel path are different. The loss is computed at the final step as $\tilde{L}(\tilde{\boldsymbol{x}}_H) = \frac{1}{2}(\tilde{\boldsymbol{x}}_H - \mathbf{1})^{\mathrm{T}}(\tilde{\boldsymbol{x}}_H - \mathbf{1})$. Note that the parameters of the

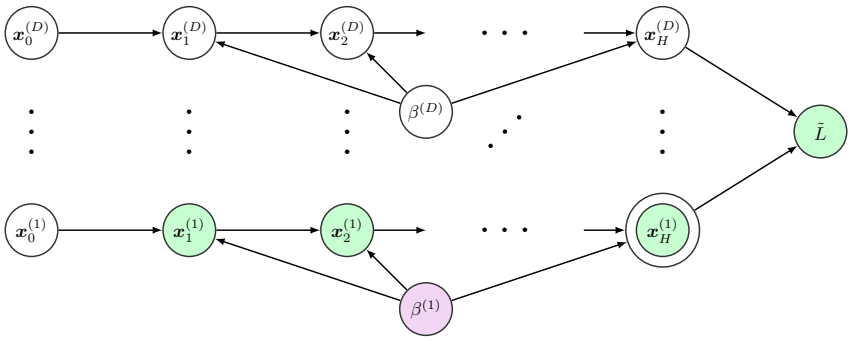


Figure 10: Multi path chaotic RNN probabilistic computation graph diagram.

networks W and β are *not* shared across the paths, and we estimate the gradient only through the first path according to $\beta^{(1)}$. If instead the β parameter were shared, the problems of LR demonstrated in this section would not appear. Gaussian shaping modifies the gradient estimation method by resampling the batch in the first path at the last step $\{\boldsymbol{x}_H^{(1,m)}\}_{m=1}^{B}$ from a fitted Gaussian distribution (this node is illustrated as ◎ in Fig. 10). The other experimental details are the same as before in Sec. 4.2. We estimate the gradient when $\beta \in \{2.0, 2.5, 3.5\}$, and plot the gradient variance against the number of parallel paths D (Fig. 11).

Results. The results are in Fig. 11. We see that the regular LR has a linearly increasing variance as the dimensionality D is increased, whereas the variance of GS stays constant. The variance of RP also stays constant with the dimensinality irrespective of whether GS is used or not; however, RP is inaccurate in the chaotic regime with $\beta \in \{2.5, 3.5\}$. The variance of TP follows a similar pattern to LR; however, for $\beta = 2.0$ the variance of the regular TP does not increase, because RP gradients are accurate in that scenario. We also see that TP outperformed the other estimators in all cases.

Discussion. Previously, Parmas (2018) explained a potential advantage of GS in terms of the bias that it adds—smoothing the loss with a Gaussian may promote unimodality simplifying some optimization problems. However, they did not demonstrate a fundamental advantage in terms of computational complexity. Here, on the other hand, we have demonstrated a fundamental advantage of GS in terms of its scalability with the dimensionality D. This newly shown effect is particularly important when the system is near-chaotic, and regular back propagation gradients are ill-behaved.

Our newly shown advantage of GS may pose it beneficial for optimizing complex model architectures. When there are several modules independently influencing the behavior of downstream components of the system, GS may disentangle the individual contributions, and enable efficient optimization. As GS is not practical to implement using standard automatic differentiation software, our result highlights that automatic propagation software such as Proppo may enable training previously untrainable machine learning systems composed of complex networks of connected modules.

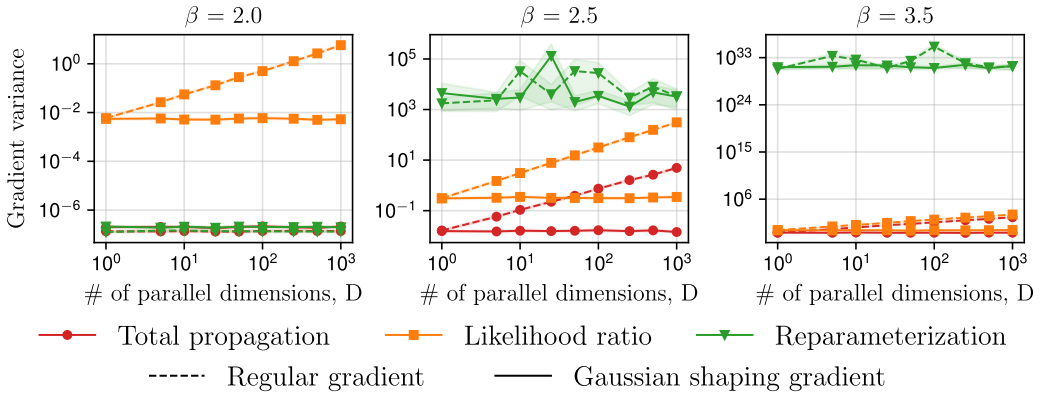


Figure 11: Comparison of Gaussian shaping (GS) gradient estimators with their regular counterparts in terms of scalability w.r.t. the dimensionality of the system D. The discussion of the experiment is in Sec. C.3. The main result is that the variance of the regular LR increases linearly as D is increased, whereas the GS variant has a constant variance.

D Proppo prototype code listings

This section contains prototype code for Proppo, as well as its application to create MC gradient estimator propagators. The code describes one possible prototype of automatic propagation software, but there may also be other implementations. The code here is not intended for use—it is provided for archival purposes. Code for use will be uploaded to https://github.com/proppo/proppo.

Propagation manager.

```
# propagation_manager.py
import torch
import proppo.back_methods as back_methods
import proppo.propagators as propagators
from proppo.containers import Node, Message


def _reversed_enumerate(l):
    count = len(l)
    for value in reversed(l):
        count -= 1
        yield count, value


class PropagationManager:
    """ Propagation Manager class.
    This class enables custom forward and backward propagations for flexibly
    designing new gradient estimation and learning algorithms for computational
    graphs, e.g. neural networks.

    """

    def __init__(self,
                 default_propagator=propagators.BackPropagator(),
                 loss_propagator=None,
                 terminal_propagator=propagators.BackPropagator()):
        self.nodes = []
        self.node_pointer = 0  # Pointer for the current position on the tape.
        self.default_propagator = default_propagator
        self.propagators = {}
        if loss_propagator:
            self.loss_propagator = loss_propagator
        elif default_propagator:
            self.loss_propagator = self.default_propagator.loss_propagator()
        # The terminal propagator exists to handle any remaining messages
        # once the backward pass has finished. For example, a common use
        # case is to use BackPropagator() to call backprop once all
        # outputs and gradients have been assembled for the backprop call,
        # if these can be performed in parallel.
        self.terminal_propagator = terminal_propagator
        if self.terminal_propagator != None:
```

```
            self.forward(x=None, local_propagator=terminal_propagator)
            self.nodes[0]['clear'] = False

    def add_propagator(self, name, propagator):
        self.propagators[name] = propagator

    def forward(
            self,
            x,
            force_targets=None,
            local_propagator=None,
            get_node=False,
            **kwargs):
        """ Register input as a node, and returns output through local forward
            function.

        """
        # Do modification to x, e.g. add noise
        if local_propagator != None:
            if isinstance(local_propagator, str):
                local_propagator = self.propagators[local_propagator]
            node = local_propagator.forward(x, **kwargs)
        else:
            node = self.default_propagator.forward(x, **kwargs)

        if not isinstance(node, Node):
            node = Node.from_container(node)

        # set local backward propagator for later use with the
        # propagator.backward method
        node.assign_propagator(local_propagator)

        if force_targets != None:
            node['force_targets'] = force_targets

        if 'output' in node:
            output = node['output']
        else:
            output = None

        if node['register_node']:
            if self.node_pointer > (len(self.nodes) - 1):
                self.nodes.append(node)
            else:
                self.nodes[self.node_pointer] = node
            self.node_pointer += 1

        # Return output as well as pointers to the messages and
        # node content.
        # The node pointer is given if one wants to retrospectively change
        # something in the node, e.g. if one wants to target messages to
        # future nodes.

        if get_node:
            return (output, self.nodes[self.node_pointer])
        else:
            return output


    def backward(self, loss=None, clear_nodes=False, message=None):
        """ Execute backward propagation at the registered nodes one by one.

        """
        # backward until the last node
        if loss != None:
            self.append_loss(loss)

        if message != None:  # TODO, send message to node instead.
            self._send_message(
                message=message
            )  # message should be of Message type and include the target

        # Loop through all nodes in reverse order, calling the
        # custom backward method of that node
        for i, node in _reversed_enumerate(self.nodes):
            self.node_pointer = i
            if node.propagator != None:
                messages = node.backward()
            else:
                messages = self.default_propagator.backward(
                    node, node.messages)
```

```python

            # Clear node and message tape content, then send the message;
            # this allows to send a message to the propagators own slot
            # as well (i.e. to keep a history between different
            # manager.backward() calls). Node can optionally be not cleared,
            # but message_tape is always cleared. (To keep a history, in
            # message tape, the propagator should send a message to itself).
            if node['clear']:
                self.nodes[i].clear()
            self.nodes[i].messages.clear()
            if messages != None:  # TODO: loop through the message items
                for target, message_container in messages.messages():
                    # choose the priority target, transform the target, send the message
                    targets = self._target_conflict_resolution(target, node)
                    target_nodes = self._find_nodes(targets)
                    self._send_messages(message_container, target_nodes)

        # refresh node history
        if clear_nodes:  # TODO: remove message tape
            if self.terminal_propagator:
                self.nodes = [self.nodes[0]]
            else:
                self.nodes = []

        if self.terminal_propagator:
            self.node_pointer = 1
        else:
            self.node_pointer = 0

        return messages  # The last remaining messages are returned if desired.

    def _send_message(self, message, target_node):
        if target_node != None:
            target_node.receive(message)

    def _send_messages(self, messages, targets):
        if isinstance(targets, list):
            for target in targets:
                self._send_message(messages, target)
        else:
            self._send_message(messages, targets)

    def _target_conflict_resolution(self, target, node):
        if 'force_targets' in node:
            targets = node['force_targets']
            return targets
        elif target != None:
            return target
        elif 'targets' in node:
            return node['targets']
        else:
            targets = -1
            return targets

    def _find_node(self, target):
        if isinstance(target, int):
            node_index = target + self.node_pointer
            if (node_index < 0) or (node_index > (len(self.nodes) + 1)):
                target = None  # If out of bounds, don't send
            else:
                target = self.nodes[node_index]
        return target

    def _find_nodes(self, targets):
        if isinstance(targets, (list, tuple)):
            targets = [self._find_node(t) for t in targets]
            return targets
        else:
            targets = self._find_node(targets)
            return targets

    def append_loss(self, loss_node, loss_propagator=None, **kwargs):
        if loss_propagator is None:
            loss_propagator = self.loss_propagator
        out = self.forward(x=loss_node,
                           targets=None,
                           local_propagator=loss_propagator,
                           get_message_box=False,
                           get_node=False,
                           **kwargs)
        return out
```

```python

    def size(self):
        """ Returns the number of the registered nodes.

        Returns:
            int: the number of nodes.

        """
        return len(self.nodes)
```

Listing 1: Example prototype Python code for a Propagation Manager class

Smart containers and contents.

```python
# containers.py
from typing import List

class Content():

    __slots__ = ('_content', )

    def __init__(self, content):
        if isinstance(content, Content):
            self._content = content.get()
        else:
            self._content = content

    def get(self):
        return self._content

    def set(self, value):
        if isinstance(value, Content):
            v = value.get()
        else:
            v = value

        self._content = v

    def update(self, value):
        if hasattr(self.get(), 'update'):
            if isinstance(value, Content):
                v = value.get()
            else:
                v = value
            self.get().update(v)
        else:
            self.set(value)

    def __repr__(self):
        return 'Content(' + str(self._content) + ')'

    def __str__(self):
        return str(self._content)

    def __add__(self, value):
        if isinstance(value, Content):
            return self.get() + value.get()
        else:
            return self.get() + value

    def __mul__(self, value):
        return self.get() * value.get()

    def __rmul__(self, value):
        return self.get() * value.get()

    def __matmul__(self, value):
        return self.get() @ value.get()


class Summed(Content):

    def update(self, value):
        self.set(self + value)

    def __repr__(self):
        return 'Summed(' + str(self._content) + ')'


```

```
class Locked(Content):

    def set(self, value):
        raise RuntimeError(
            'Attempting to call ".set()" on a Locked type'
            ' Content. Locked type Content objects are used'
            ' for contents that are never supposed to be updated.')

    def __repr__(self):
        return 'Locked(' + str(self._content) + ')'


class Listed(Content):

    def __init__(self, *args):
        list_content = []
        for content in args:
            if isinstance(content, Content):
                list_content.append(content.get())
            else:
                list_content.append(content)
        super().__init__(list_content)

    def __repr__(self):
        return 'Listed(' + str(self._content) + ')'

    def update(self, *args):
        if len(args) == 1:
            if not isinstance(args[0], Listed):
                listed_content = Listed(args[0])
            else:
                listed_content = args[0]
        else:
            list_vals = []
            for arg in args:
                if isinstance(arg, Content):
                    if isinstance(arg, Listed):
                        list_vals += arg.get()
                    else:
                        list_vals.append(arg.get())
                else:
                    list_vals.append(arg)
            listed_content = Listed(*list_vals)

        self._content = self + listed_content

    def set(self, *args):
        listed_content = []
        for value in args:
            if isinstance(value, Content):
                v = value.get()
                if not isinstance(value, Listed):
                    v = [v]
            else:
                v = [value]
            listed_content += v

        self._content = listed_content


class Container():

    __slots__ = ('_contents', )

    def __init__(self, cont_dict=None, **kwargs):
        self._contents = {}
        if cont_dict != None:
            kwargs.update(cont_dict)
        for k, v in kwargs.items():
            if not isinstance(v, Content):
                val = Content(v)
            else:
                val = v
            self._contents[k] = val

    def clear(self):
        self._contents.clear()

    def get_contents(self):
        return self._contents

```

```
    def set_content(self, key, value):
        if not isinstance(value, Content):
            v = Content(value)
        else:
            v = value

        if key in self._contents:
            self._contents[key].set(v)
        else:
            self._contents[key] = v

    def get(self, key):
        return self[key]

    def item_iter(self):
        for k in self.keys():
            yield (k, self[k])

    def items(self):
        return self.item_iter()

    def keys(self):
        return self._contents.keys()

    def _update_keys(self):
        return self._contents.keys()

    def value_iter(self):
        for k in self.keys():
            yield self[k]

    def values(self):
        return self.value_iter()

    def pop(self, key):
        return self._contents.pop(key).get()

    def __str__(self):
        return 'Contents: ' + str(self._contents)

    def __iter__(self):
        return iter(self.keys())

    def __getitem__(self, k):
        return self._contents[k].get()

    def __setitem__(self, k, v):
        self.set_content(k, v)

    def __contains__(self, k):
        return k in self._contents

    def update(self, container):
        # Works for both dict or container.
        if isinstance(container, dict):
            c = Container(cont_dict=container)
        else:
            c = container

        for k in c._update_keys():
            if k in self._update_keys():
                self._contents[k].update(c._contents[k])
            else:
                self.set_content(k, c._contents[k])


class Node(Container):
    __slots__ = ('_contents', 'messages', 'propagator')

    def __init__(self,
                 cont_dict=None,
                 box_class=Container,
                 propagator=None,
                 **kwargs):
        super().__init__(cont_dict=cont_dict, **kwargs)
        self.messages = box_class()
        self.propagator = propagator

    @classmethod
    def from_container(cls, container):
        kwargs = {}
```

```
        if 'box_class' in container:
            kwargs['box_class'] = container.pop('box_class')
        if 'propagator' in container:
            kwargs['propagator'] = container.pop('propagator')
        if isinstance(container, Container):
            cont_dict = container.get_contents()
        elif isinstance(container, dict):
            cont_dict = container
        else:
            raise TypeError('container must be of Container or dict type.')

        return Node(cont_dict=cont_dict, **kwargs)

    def forward(self, x, **kwargs):
        return self.propagator.forward(x, **kwargs)

    def backward(self):
        return self.propagator.backward(self, self.messages)

    def receive(self, message):
        # TODO: add more message box classes
        if isinstance(message, Message):
            for m in message.containers():
                self.messages.update(m)
        else:
            self.messages.update(message)

    def assign_propagator(self, propagator):
        self.propagator = propagator


class Message(Container):
    __slots__ = ('_contents', 'multi_message')

    def __init__(self, cont_dict=None, target=-1, container=None, **kwargs):
        if container == None:
            super().__init__(
                cont_dict={target: Container(cont_dict=cont_dict, **kwargs)})
        else:
            super().__init__(cont_dict={target: container})
        self.multi_message = False

    def _switch_multi(self):
        if len(self.targets()) > 1:
            self.multi_message = True
        else:
            self.multi_message = False

    def _get_main_message(self):
        return self._contents[next(iter(self.targets()))].get()

    def get_message(self, target):
        return self._contents[target].get()

    def __str__(self):
        return 'Message: ' + str(self._contents)

    def targets(self):
        return Container.keys(self)

    def iter_containers(self):
        for t in self.targets():
            yield self.get_message(t)

    def containers(self):
        return self.iter_containers()

    def iter_messages(self):
        for t in self.targets():
            yield (t, self.get_message(t))

    def messages(self):
        return self.iter_messages()

    def iter_keys(self):
        for container in self.containers():
            for k in container.keys():
                yield k

    def keys(self):
        if self.multi_message == False:
```

```
            return self._get_main_message().keys()
        else:
            # return self.targets()

            return self.iter_keys()

    def iter_items(self):
        for container in self.containers():
            for item in container.items():
                yield item

    def items(self):
        if self.multi_message == False:
            return super().items()
        else:
            return self.iter_items()

    def iter_values(self):
        for container in self.containers():
            for value in container.values():
                yield value

    def values(self):
        if self.multi_message == False:
            return super().values()
        else:
            return self.iter_values()

    def pop(self, k):
        if self.multi_message == True:
            out = super().pop(k)
            self._switch_multi()
            return out
        else:
            return self._get_main_message().pop(k)

    def pop_message(self, k):
        out = super().pop(k)
        self._switch_multi()
        return out

    def update(self, message):
        if isinstance(message, Message):
            super().update(message)
            self._switch_multi(
            )  # check whether a message with a new target was added
        elif isinstance(message, (Container, dict)):
            if self.multi_message == False:
                self._get_main_message().update(message)
            else:
                raise TypeError(
                    'Current Message contains multiple targets. Updating'
                    ' with Container is disabled due to ambiguity'
                    ' in the target. Turn container into Message'
                    ' type with a specified target, then update'
                    ' the current message.')
        else:
            raise TypeError('Message can only be updated with a Message'
                            ' or Container type.')

    def __getitem__(self, k):
        if self.multi_message == False:
            return self._get_main_message()[k]
        else:
            return super().__getitem__(k)

    def __setitem__(self, k, v):
        if self.multi_message == False:
            self._get_main_message().set_content(k, v)
        else:
            super().__setitem__(k, v)

    def __contains__(self, k):
        if self.multi_message == False:
            return k in self._get_main_message()
        return super().__contains(k)
```

Listing 2: Example prototype Python code for smart contents and containers

Smart initializers.

```python
# initializers.py
from collections import OrderedDict
from functools import partial
import copy


class Init():

    def __init__(self, cls, *args, defaults=None, kwdefaults=None, **kwargs):
        self._cls = cls
        if kwdefaults != None:
            self.kwdefaults = {**kwdefaults, **kwargs}
        elif kwargs != {}:
            self.kwdefaults = kwargs
        if defaults != None:
            self.defaults = (*args, *defaults)
        elif args != ():
            self.defaults = args

    def __call__(self, *args, **kwargs):
        if hasattr(self, 'kwdefaults'):
            input_kwargs = {**self.kwdefaults, **kwargs}
        else:
            input_kwargs = kwargs
        # Note that concatenating defaults and args is not allowed
        # due to ambiguity of when to overwrite.
        # TODO: fix the default arguments so that they are first
        # converted into keyword arguments, and then merged with the
        # keyword arguments. If there is a conflict, e.g., the same
        # keyword exists in both sets, then raise an error.
        if hasattr(self, 'defaults') and len(args) == 0:
            input_args = self.defaults
        else:
            input_args = args

        return self._call(*input_args, **input_kwargs)

    def _call(self, *args, **kwargs):
        return self._cls(*args, **kwargs)

    @classmethod
    def init(cls, *args, **kwargs):
        return Init(cls, *args, **kwargs)

class Lock(Init):

    def __init__(self, *args, allow_unused=False, **kwargs):
        super().__init__(*args, **kwargs)
        self.allow_unused = allow_unused

    def __call__(self, *args, allow_unused=False, **kwargs):
        if allow_unused or self.allow_unused:
            return super().__call__()
        elif len(args) == 0 and len(kwargs) == 0:
            return super().__call__()
        else:
            raise RuntimeError('allow_unused permission is False.'
                               ' The Lock initalizer allows only'
                               ' initializing with the default parameters.'
                               ' Remove args and kwargs inputs from'
                               ' initialization.')

class Optional(Init):
    """ Initializer that only initializes when the on flag is
    True. Otherwise, an instance of the class is not created, it returns None.
    """
    def _call(self, on, *args, **kwargs):
        if on:
            return super()._call(*args, **kwargs)
        else:
            return None

class Choice(Init):
    def __init__(self, cls, *args, **kwargs):
        if not isinstance(cls, (tuple, dict)):
            raise TypeError('cls input must be of type tuple or dict.')
        else:
            super().__init__(cls, *args, **kwargs)

```

```
    def _call(self, choice, *args, **kwargs):
        return self._cls[choice](*args, **kwargs)

class Empty(Init):
    def __init__(self):
        pass

    def __call__(self, *args, **kwargs):
        self._call()

    def _call(self, *args, **kwargs):
        raise RuntimeError('Trying to initialize a ChainInit containing'
                           ' an empty Init. First reconfigure the'
                           ' ChainInit to replace the Empty Init.'
                           ' Empty Inits are used in template'
                           ' ChainInits to indicate what slot has to be'
                           ' changed.')

class ChainInit():

    def __init__(self, regular_dict=False, **kwargs):

        if regular_dict:
            self._chaininit = {}
        else:
            self._chaininit = OrderedDict()

        for k, v in kwargs.items():
            if isinstance(v, Init):
                init = v
            else:
                init = Init(v)
            self._chaininit[k] = init

    def __call__(self, **kwargs):
        return self.init(**kwargs)

    def items(self):
        return self._chaininit.items()

    def values(self):
        return self._chaininit.values()

    def keys(self):
        return self._chaininit.keys()

    def __iter__(self):
        return self._chaininit.__iter__()

    def __contains__(self, key):
        return key in self._chaininit

    def __getitem__(self, key):
        return self._chaininit[key]

    def __len__(self):
        return len(self._chaininit)

    @staticmethod
    def _init_inputs(init, inputs=None):

        if inputs == None:
            obj = init()
        elif isinstance(inputs, tuple):
            arginputs = []
            kwinputs = {}
            for inp in inputs:
                if isinstance(inp, dict):
                    kwinputs.update(inp)
                else:
                    arginputs.append(inp)
            obj = init(*arginputs, **kwinputs)
        elif isinstance(inputs, dict):
            obj = init(**inputs)
        else:
            obj = init(inputs)
        return obj

    def init(self, dictionary=False, **kwargs):

        chainobjs = []
```

```python
        chainkeys = []

        # Check that all keys exist
        for key in kwargs:
            if key not in self._chaininit:
                raise KeyError(key, 'not included in keys of the ChainInit.')

        for k, init in self._chaininit.items():
            inputs = kwargs.get(k, None) # kwarg if exists, otherwise None

            obj = self._init_inputs(init, inputs)

            if obj != None:
                chainobjs.append(obj)
                chainkeys.append(k)
        if not dictionary:
            return chainobjs
        else:
            return {k: obj for k, obj in zip(chainkeys, chainobjs)}

    def reconf(self, **kwargs):
        # Create a new ChainInit by reconfiguring the current
        # ChainInit. For example, replace the Empty initializer
        # of the current ChainInit to create a functioning ChainInit.
        # You may also change the default parameters of the Inits.

        chaindict = copy.copy(self._chaininit)
        for k, v in kwargs.items():
            if k not in chaindict:
                raise KeyError('Key does not exist in the ChainInit'
                               ' that you are trying to reconfigure.')
            else:
                if isinstance(v, Init):
                    chaindict[k] = v
                else:
                    initializer = type(chaindict[k])
                    cls = chaindict[k]._cls
                    initializer = partial(initializer, cls)
                    chaindict[k] = self._init_inputs(initializer, v)

        return ChainInit(**chaindict)


class ChainInitTemplate(ChainInit):

    def __call__(self, **kwargs):
        initializers = super().__call__(dictionary=True, **kwargs)
        return ChainInit(**initializers)
```

Listing 3: Example prototype python code for smart initializers in Proppo

Propagators.

```python
# propagators.py
import proppo.forward_methods as fm
import proppo.back_methods as bm
import proppo.baseline_funcs as baselines
import proppo as pp
from proppo.utils import inverse_variance_weighting
from proppo.containers import Node, Message, Container
from proppo.initializers import (ChainInit, Optional, Init, Empty,
                                 ChainInitTemplate)

import copy


class Propagator:
    """ This pairs together the forward and backward methods.

    """

    def __init_subclass__(cls, **kwargs):
        cls.default_init_kwargs = kwargs

    def __init__(self, **kwargs):
        self.default_forward_kwargs = kwargs

    def forward(self, x, **kwargs):
        # Overwrite default arguments, then pass as input
```

```
        input_kwargs = {**self.default_forward_kwargs, **kwargs}

        # Must create a new node, and pass this to forward, otherwise
        # the propagators at different forward steps will overwrite, the
        # contents of the previous propagation.
        node = {}
        node = self.forward_impl(x, node, **input_kwargs)

        # Flag to store the node in manager.
        if 'register_node' not in node:
            node['register_node'] = True
        # Flag to clear node in manager after backwarding the node.
        if 'clear' not in node:
            node['clear'] = True

        if isinstance(node, dict):
            node = Container(cont_dict=node)

        return node

    def forward_impl(self, x, node={}, **kwargs):
        return node

    def backward(self, node, message):
        message_in = message
        message_out = self.backward_impl(node, message_in)

        # for backwards compatibility, convert dictionaries
        if not isinstance(message_out, Message):
            if isinstance(message_out, dict):
                if 'targets' in message_out:
                    target = message_out.pop('targets')
                    message_out = Message(cont_dict=message_out, target=target)
                else:
                    message_out = Message(cont_dict=message_out)
            elif isinstance(message_out, Container):
                message_out = Message(container=message_out)

        return message_out

    def backward_impl(self, node, message):
        message_out = Message(cont_dict=message.get_contents())
        return message_out

    def loss_propagator(self):
        """ Returns the default loss propagator that should
        be applied when appending a loss after having called
        manager.forward using the current propagator.

        """
        return LossProp()


class SequenceProp(Propagator):
    """ Base class for sequence based propagators, used to construct them.

    """

    def __init_subclass__(cls,
                          propagators=ChainInit(),
                          **kwargs):
        super().__init_subclass__(**kwargs)
        cls.propagators = propagators

    def _split_prop_kwargs(self, kwargs):
        prop_kwargs = {}
        for k in self.propagators:
            if k in kwargs:
                prop_kwargs[k] = kwargs.pop(k)
        return prop_kwargs, kwargs

    def __init__(self,
                 propagators=[], # A list of already initialized propagators.
                 **kwargs):

        default_init_kwargs = copy.copy(self.default_init_kwargs)

        if propagators != []:
            input_kwargs = {**default_init_kwargs, **kwargs}

            super().__init__(**input_kwargs)
```

```
            self.propagators = propagators
        else:
            def_prop_kwargs, def_init_kwargs = self._split_prop_kwargs(default_init_kwargs)
            prop_kwargs, init_kwargs = self._split_prop_kwargs(kwargs)
            input_kwargs = {**def_init_kwargs, **init_kwargs}

            super().__init__(**input_kwargs)

            input_prop_kwargs = {**def_prop_kwargs, **prop_kwargs}

            self.propagators = self.propagators(**input_prop_kwargs)


class ComboProp(SequenceProp):
    """ Combines propagators, and applies them in a sequence, updating
    message and node in-place.

    """

    def forward_impl(self, x, node={}, **kwargs):
        for prop in self.propagators:
            node_out = prop.forward_impl(x, node, **kwargs)
            node.update(node_out)
        return node

    def backward_impl(self, node, message):

        final_message = Message(cont_dict=message.get_contents())
        for prop in reversed(self.propagators):
            message_in = final_message._get_main_message()

            message_out = prop.backward_impl(node, message_in)

            # for backwards compatibility, convert dictionaries
            if isinstance(message_out, dict):
                if 'targets' in message_out:
                    target = message_out.pop('targets')
                    message_out = Message(cont_dict=message_out, target=target)
                else:
                    message_out = Message(cont_dict=message_out)

            final_message.update(message_out)

        return final_message


class BackPropagator(Propagator):
    """ Base propagator that will backprop gradient messages, if they
    are sent into this propagator.

    """

    def backward_impl(self, node, message):

        message_out = bm.backward(node, message)
        return message_out


class BaselineProp(Propagator):
    """ Class for adding a baseline subtraction to the local losses.

    """

    def __init__(self, baseline_func=baselines.mean_baseline, **kwargs):
        super().__init__(**kwargs)
        self.baseline_func = baseline_func  # Default baseline function

        # Note: if one wants to change the baseline function for just one
        # forward call compared to the default baseline function in a chain
        # of forward propagations, then they should define a new propagator
        # object for that new forward call. I could also allow giving an
        # additional argument in the forward call to specify a baseline
        # for just that node; however, this would not
        # give a key error if someone accidentally mistypes the key, and
        # may lead to bugs, so I avoid it.

    def backward_impl(self, node, message):
        local_loss = message.pop('local_loss')
        # Need to remove local_loss from previous message, and create
        # a new message to avoid duplicating loss in the ComboProp
```

```
        # backward_impl method.
        if isinstance(self.baseline_func, (list, tuple)):
            baselined_loss = copy.copy(local_loss)
            for func in reversed(self.baseline_func):
                baselined_loss = func(baselined_loss, node)
        else:
            baselined_loss = self.baseline_func(local_loss, node)

        message_out = {
            'baselined_loss': baselined_loss,
            'local_loss': local_loss
        }
        return message_out


mcgrad_temp = ChainInitTemplate(backprop=Optional(
    Optional.init(BackPropagator, True), False),
                                    base=Init,
                                    baseline=Optional(
    Optional.init(BaselineProp, True), False)
)


class PauseBase(Propagator):
    """ A propagator that pauses all incoming gradients, then sends the
    combined gradient backwards.

    """

    def forward_impl(self, x, node={}, **kwargs):
        node = fm.detached_output(x, **kwargs)
        return node

    def backward_impl(self, node, message):
        message_out = bm.rp_gradient(node, message)
        return message_out


class SkipProp(Propagator):
    """ A propagator that sends all incoming messages backward
    a determined length, skipping the nodes inbetween.

    """

    def __init__(self, skip=1, **kwargs):
        super().__init__(**kwargs)
        self.skip = skip

    def forward_impl(self, x, node={}, **kwargs):
        node['output'] = x
        return node

    def backward_impl(self, node, message):
        message['targets'] = -self.skip
        return message


class PauseProp(ComboProp, propagators=mcgrad_temp(backprop=True,
                                                   base=PauseBase)):
    pass


class SumBase(Propagator):
    """ Propagator that adds a local variable with a different variable in
    messages. This is usually used to accumulate a sum of variables during the
    backward pass, e.g. sum the rewards to obtain the return in reinforcement learning.

    """

    def __init__(self, sum_name, local_variable, **kwargs):
        self.sum_name = sum_name
        self.local_variable = local_variable
        super().__init__(**kwargs)

    def backward_impl(self, node, message):
        current_sum = message.pop(self.sum_name, 0)
        message = {self.sum_name: message[self.local_variable] + current_sum}

        return message


```

```python
class ChainProp(SequenceProp):
    """ Chains together a set of propagators into a single propagator.
    The implementation is based on creating a new PropagationManager object
    to correctly apply the propagators in sequence without any implementation
    errors. The propagators to chain together should be given as a list
    or tuple of Propagator instances during creation. The propagators
    themselves can also be Chain propagators, which allows for defining
    complex propagation strategies using nested propagation managers.

    """

    def forward_impl(self, x, node={}, chain_kwargs=[], **kwargs):
        manager = pp.PropagationManager(default_propagator=None,
                                        terminal_propagator=None)
        if chain_kwargs:
            for prop, kwarg in zip(self.propagators, chain_kwargs):
                kwarg.update(kwargs)
                x = manager.forward(x, local_propagator=prop, **kwarg)
        else:
            for prop in self.propagators:
                x = manager.forward(x, local_propagator=prop, **kwargs)

        node = {'output': x, 'manager': manager}
        if manager.size() == 0:
            node['register_node'] = False
        return node

    def backward_impl(self, node, message):
        message_out = node['manager'].backward(message=message)
        return message_out

    def loss_propagator(self):
        """ By default, usually the last one in the chain
        contains the correct loss propagator.

        """
        return self.propagators[-1].loss_propagator()


class RPBase(Propagator):
    """ Base class for RP propagator.

    """

    def forward_impl(self, x, node={}, detach=True, **kwargs):
        node = fm.rp_noise(x, detach=detach, **kwargs)
        return node

    def backward_impl(self, node, message):
        message_out = bm.rp_gradient(node, message)
        return message_out


class RPProp(ComboProp, propagators=mcgrad_temp(backprop=True,
                                                base=RPBase)):
    """ RP propagator combining the functionality from ComboProp.

    """
    pass


class LossBase(Propagator):
    """ Base class for loss nodes in the computational graph.

    """

    def __init__(self, loss_name='local_loss', **kwargs):
        super().__init__(**kwargs)
        self.loss_name = loss_name

    def forward_impl(self, x, node={}, lossgrad=True, lossfunc=None, **kwargs):
        if lossfunc:
            if isinstance(x, dict):
                losses = lossfunc(**x)
            else:
                losses = lossfunc(x)
        else:
            losses = x

        node = fm.loss_forward(losses, sum_loss=True, lossgrad=lossgrad)
        return node
```

```python

    def backward_impl(self, node, message):
        message_out = bm.loss_backward(node, message, loss_name=self.loss_name)
        return message_out


class LossProp(ComboProp,
               propagators=ChainInit(
                   backprop=Optional(BackPropagator, True),
                   baseline=Optional(BaselineProp,
                                     True,
                                     baseline_func=baselines.mean_baseline),
                   base=LossBase)):
    """ Propagator adding Baseline and ComboProp functionality to Loss nodes.

    """
    pass


class LRBase(Propagator):
    """ Base class for likelihood ratio gradient propagators.

    """

    def forward_impl(self, x, node, **kwargs):
        node = fm.lr_noise(x, **kwargs)
        return node

    def backward_impl(self, node, message):
        message_out = bm.lr_gradient(node, message)
        return message_out


class LRProp(ComboProp, propagators=mcgrad_temp(backprop=True,
                                                base=LRBase,
                                                baseline=True)):
    """ Class adding ComboProp functionality to LR gradient propagators.

    """

    def loss_propagator(self):
        return LossProp(backprop=False, lossgrad=False)


class TPBase(Propagator):
    """ Base class for total propagation gradient propagators.

    """

    def __init__(self,
                 var_weighting_func=inverse_variance_weighting,
                 **kwargs):
        super().__init__(**kwargs)
        self.var_weighting_func = var_weighting_func

    def forward_impl(self, x, node, **kwargs):
        node = fm.totalprop_noise(x, **kwargs)
        return node

    def backward_impl(self, node, message):
        message_out = bm.totalprop_gradient(
            node, message, var_weighting_func=self.var_weighting_func)
        return message_out


class TotalProp(ComboProp,
                propagators=mcgrad_temp(backprop=True,
                                        base=TPBase,
                                        baseline=True)
):
    """ Class adding ComboProp functionality to total propagation
    gradient propagation nodes.

    """
    pass
```

Listing 4: Example prototype python code for propagators for MC gradient estimation

Forward methods.

```python
# forward_methods.py
import torch
import collections


def detached_output(x, requires_grad=True, **kwargs):
    x_detached = x.detach()
    if requires_grad:
        x_detached.requires_grad_()
    node = {'output': x_detached, 'pre_output': x, 'register_node': True}
    return node


def loss_forward(x, lossgrad=False, sum_loss=True, **kwargs):
    """ Method to register a loss node. It will either
    just pass the loss in the local_loss slot, or will
    also backprop the gradient.

    """
    if x.dim() == 1:
        x = x.reshape([x.numel(), 1])

    node = detached_output(x, requires_grad=False)
    node['lossgrad'] = lossgrad
    node['sum_loss'] = sum_loss
    return node


def rp_noise(x, dist_class, dist_params, detach=True, **kwargs):
    """ Returns noisy node for reparametrization trick

    """
    if isinstance(dist_params, (tuple, list)):
        dist = dist_class(*dist_params)
    if isinstance(dist_params, dict):
        dist = dist_class(**dist_params)
    if isinstance(dist_params, collections.Callable):
        dist = dist_class(**dist_params(x))

    x_noisy = x + dist.rsample()

    if detach:
        node = detached_output(x_noisy)
    else:
        node = {'output': x_noisy, 'register_node': False}
    return node


def lr_noise(x, dist_class, dist_params, requires_grad=False, **kwargs):
    """ Returns noisy node for likelihood ratio

    """
    if isinstance(dist_params, (tuple, list)):
        dist = dist_class(*dist_params)
    if isinstance(dist_params, dict):
        dist = dist_class(**dist_params)
    if isinstance(dist_params, collections.Callable):
        dist = dist_class(**dist_params(x))

    x_noisy = x + dist.rsample()
    log_prob = dist.log_prob(x_noisy.detach() - x)
    node = detached_output(x_noisy, requires_grad=requires_grad)
    node['log_prob'] = log_prob
    return node


def totalprop_noise(x,
                    dist_class,
                    dist_params,
                    ivw_target,
                    k_interval=1,
                    **kwargs:)
    """ Returns noisy node for total propagation

    """
    node = lr_noise(x,
                    dist_class=dist_class,
                    dist_params=dist_params,
                    requires_grad=True)
    node['ivw_target'] = ivw_target
    node['k_interval'] = k_interval
```

```
    return node
```

Listing 5: Example prototype python code for the forward methods of the propagators

Backward methods.

```
# back_methods.py
import torch
from proppo.utils import inverse_variance_weighting
from proppo.containers import Listed


def backward(node, message, grad_name=None):
    """ This one is special for calling backward.
    Removes outputs and grads from message, calls backward,
    and passes the remaining message backwards.

    """
    if 'outputs' in message:
        outputs = message.pop('outputs')
        grads = message.pop('grads')
    else:
        return message

    torch.autograd.backward(tensors=outputs, grad_tensors=grads)
    return {}


def loss_backward(node, message_in, loss_name):
    local_loss = node['output']
    if node['sum_loss']:
        if loss_name in message_in:
            local_loss += message_in[loss_name]

    outputs = node['pre_output']
    # Grads is set so that the gradient of the average loss is computed.
    message = {loss_name: local_loss}

    if node['lossgrad']:
        ones_matrix = torch.tensor([1.0], device=outputs.device).expand(
            outputs.size())
        grads = {
            'outputs': Listed(outputs),
            'grads': Listed(ones_matrix / torch.numel(outputs))
        }
        message.update(grads)

    return message


def rp_gradient(node, message_in):
    """ Returns output tensor and its gradient for reparametrization trick.

    """
    detached_output = node['output']
    output = node['pre_output']
    message = {
        'outputs': Listed(output),
        'grads': Listed(detached_output.grad)
    }

    return message


def lr_gradient(node, message_in):
    """ Returns output tensor and its gradient for likelihood ratio

    """
    if 'baselined_loss' in message_in:
        local_loss = message_in['baselined_loss']
    else:
        local_loss = message_in['local_loss']

    lr_grad_outputs = local_loss / torch.numel(local_loss)
    log_prob = node['log_prob']

    lr_grad_outputs = lr_grad_outputs.expand(log_prob.shape)

    message = {'outputs': Listed(log_prob), 'grads': Listed(lr_grad_outputs)}
```

```python
    return message


def totalprop_gradient(node,
                       message_in,
                       var_weighting_func=inverse_variance_weighting):
    """ Returns output tensors and their gradients for total propagation.

    The total propagation is a combination of the reparametrization trick and
    the likelihood ratio.
    Each gradient will be combined based on inverse variance weighting.

    """
    detached_output = node['output']
    output = node['pre_output']
    log_prob = node['log_prob']
    ivw_target = node['ivw_target']
    k_interval = node['k_interval']

    if 'k_counter' in message_in:
        k_counter = message_in.pop('k_counter')
    else:
        k_counter = 0

    if 'baselined_loss' in message_in:
        local_loss = message_in['baselined_loss']
    else:
        local_loss = message_in['local_loss']
    # Make LR gradients compute mean gradient
    local_loss = local_loss / torch.numel(local_loss)

    lr_grad_outputs = local_loss.expand(log_prob.shape)

    if k_counter % k_interval == 0:
        rp_grads = torch.autograd.grad(outputs=output,
                                       inputs=ivw_target,
                                       grad_outputs=detached_output.grad,
                                       retain_graph=True)

        lr_grads = torch.autograd.grad(outputs=log_prob,
                                       inputs=ivw_target,
                                       grad_outputs=lr_grad_outputs,
                                       retain_graph=True)

        k_lr, k_rp = var_weighting_func(lr_grads, rp_grads)
    else:
        k_lr, k_rp = message_in['k_lr_k_rp']

    outputs = Listed(log_prob, output)
    grads = Listed(k_lr * lr_grad_outputs, k_rp * detached_output.grad)
    message_grads = {'outputs': outputs, 'grads': grads, 'targets': -1}

    message_k = {
        'k_counter': k_counter + 1,
        'k_lr_k_rp': (k_lr, k_rp),
        'targets': 0
    }

    if k_interval > 1:
        messages = (message_grads, message_k)
    else:
        messages = message_grads
    return messages
```

Listing 6: Example prototype python code for the backward methods of the propagators

Baseline functions.

```python
# baseline_funcs.py
import torch


def mean_baseline(local_losses, node=None):
    with torch.no_grad():
        batch_size = local_losses.numel()

        # leave-one-out baselines
        sum_loss = local_losses.sum()
        # The division by (batch_size - 1) instead of (batch_size)
```

```python
        # is algebraically equivalent to a leave-one-out baseline.
        sum_loss = (sum_loss - local_losses) / (batch_size - 1)
        losses = local_losses - sum_loss
    return losses


def no_baseline(local_losses, node=None):
    return local_losses
```

Listing 7: Example prototype python code for the baseline methods used in LR-based MC gradient estimators

Utilities.

```python
# utils.py
import torch


def expand(data, batch_size):
    """ Returns the tensor with batch dimension expanded.

    Arguments:
        data (torch.Tensor): input tensor
        batch_size (int): batch size for expansion

    Returns:
        torch.Tensor: output tensor

    """
    return data.expand((batch_size, ) + data.shape)


def inverse_variance_weighting(x1, x2, scalar_estimate=True):
    """ Returns weights of inverse variance weighting for each input.

    """

    if isinstance(x1, (list, tuple)) and isinstance(x2, (list, tuple)):
        x1_vars = []
        x2_vars = []
        c_list = []
        for v1, v2 in zip(x1, x2):
            assert v1.shape == v2.shape
            batch_size = v1.shape[0]

            d1 = v1 - v1.mean(dim=0, keepdims=True)
            d2 = v2 - v2.mean(dim=0, keepdims=True)

            c1 = torch.max(torch.abs(d1))
            c2 = torch.max(torch.abs(d2))
            c = torch.max(c1, c2)
            c_list.append(c)

            if c == 0:
                x1_vars.append(torch.tensor(0.0, device=c.device))
                x2_vars.append(torch.tensor(0.0, device=c.device))
            else:
                x1_vars.append(torch.sum((d1 / c)**2))
                x2_vars.append(torch.sum((d2 / c)**2))
        x1vec = torch.tensor(x1_vars)
        x2vec = torch.tensor(x2_vars)
        cvec = torch.tensor(c_list)
        cmax = torch.max(cvec)
        if cmax == 0:
            x1_var = torch.tensor(1.0, device=cmax.device)
            x2_var = torch.tensor(1.0, device=cmax.device)
        else:
            cvec = (cvec / cmax)**2
            x1_var = torch.sum(x1vec * cvec)
            x2_var = torch.sum(x2vec * cvec)
    else:
        assert x1.shape == x2.shape
        batch_size = x1.shape[0]

        d1 = x1 - x1.mean(dim=0, keepdims=True)
        d2 = x2 - x2.mean(dim=0, keepdims=True)

        c1 = torch.max(torch.abs(d1))
        c2 = torch.max(torch.abs(d2))
```

```
        c = torch.max(c1, c2)

        if c == 0:
            x1_var = torch.tensor(1.0, device=c.device)
            x2_var = torch.tensor(1.0, device=c.device)
        else:
            x1_var = torch.sum((d1 / c)**2)
            x2_var = torch.sum((d2 / c)**2)

    k_x1 = x2_var / (x1_var + x2_var)
    k_x1 = torch.clip(k_x1, 0, 1)

    if torch.isnan(k_x1):
        print('Warning: estimated k was nan. Automatically changed to 0.5.')
    k_x1[torch.isnan(k_x1)] = 0.5  # when 0/0 error occurs, take them equally.

    return k_x1, 1.0 - k_x1
```

Listing 8: Example prototype python code for utility functions used in Proppo

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

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems*. 1

Anonymous (2022). Anonymous. In *Anonymous*, page 0000. 1, 3.5, B.3

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

Boner, J. et al. (2010). Akka. https://akka.io/. 5

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

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901. 6

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*. 6

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, 5

Gardner, J., Pleiss, G., Weinberger, K. Q., Bindel, D., and Wilson, A. G. (2018). Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. *Advances in neural information processing systems*, 31. 1

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

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

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362. 1

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. 1, 5

Hewitt, C. and Lieberman, H. (1983). Design issues in parallel architecture for artificial intelligence. In *In Proc. 28th IEEE Computer Society Int. Conf,, San Francisco CA*. Citeseer. 1

Ingraham, J., Riesselman, A., Sander, C., and Marks, D. (2019). Learning protein structure with a differentiable simulator. In *International Conference on Learning Representations*. C.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

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, 5

Maclaurin, D., Duvenaud, D., and Adams, R. (2015). Gradient-based hyperparameter optimization through reversible learning. In *International conference on machine learning*, pages 2113–2122. PMLR. C.1

McEliece, R. J., MacKay, D. J. C., and Cheng, J.-F. (1998). Turbo decoding as an instance of pearl's" belief propagation" algorithm. *IEEE Journal on selected areas in communications*, 16(2):140–152. 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.1

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

Minka, T. P. (2001). Expectation propagation for approximate bayesian inference. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 362–369. 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

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, 8

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. 1, 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*. 6

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

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, 3.4, 5, 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

Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., and Blei, D. M. (2016). Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*. 1

Wang, Q. (2013). Forward and adjoint sensitivity computation of chaotic dynamical systems. *Journal of Computational Physics*, 235:1–13. C.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, 4, C.1, 8

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

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24. 1