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, $\hat{g}$, s.t. $E[\hat{g}] = \frac{1}{K} \sum_{i=1}^{K} g(x(i))$.

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

$$\frac{d}{d\beta} E_{x \sim p(x; \beta)} [f(x)] = E_{\varepsilon \sim p(\varepsilon)} \left[ \frac{d}{d\beta} f(T(\varepsilon; \beta)) \right] = E_{\varepsilon \sim p(\varepsilon)} \left[ \frac{df}{dx} \frac{dT(\varepsilon; \beta)}{d\beta} \right].$$

(3)

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

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

$$\frac{d}{d\beta} E_{x \sim p(x; \beta)} [f(x)] = E_{\varepsilon \sim p(\varepsilon)} \left[ \frac{d\log p(x; \beta)}{d\beta} (f(x) - b) \right],$$

(4)

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

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 $\nabla f$, but does not scale well with the dimensionality. The two estimators were recently unified by Parmas and Sugiyama (2021) based on an intuitive 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, $\nabla [\hat{g}]$. The variance can be reduced by a factor $\frac{1}{K}$ 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.\footnote{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.}

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...
at nodes is deterministic. Therefore, by denoting \( \zeta_{x_i} \) 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)

\[
\frac{d\zeta_y}{d\beta} = \sum_{\text{Path} \in \text{Paths}[\beta \rightarrow y]} \prod_{\text{Edge} \in [l \rightarrow k] \in \text{Path}} \frac{\partial \zeta_{x_k}}{\partial \zeta_{x_l}}
\]

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

\[
\frac{d\zeta_y}{d\beta} = \sum_{n \in \mathcal{N}} \left( \sum_{\text{Path} \in \text{Paths}[n \rightarrow y]} \prod_{\text{Edge} \in [l \rightarrow k] \in \text{Path}} \frac{\partial \zeta_{x_k}}{\partial \zeta_{x_l}} \right) \left( \sum_{\text{Path} \in \text{Paths}[\beta \rightarrow n] \setminus \mathcal{N}} \prod_{\text{Edge} \in [l \rightarrow k] \in \text{Path}} \frac{\partial \zeta_{x_k}}{\partial \zeta_{x_l}} \right),
\]

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

\[
\frac{d\zeta_y}{d\beta} = \sum_{n \in \mathcal{N}} \frac{d\zeta_y}{d\zeta_n} \left( \sum_{\text{Path} \in \text{Paths}[\beta \rightarrow n] \setminus \mathcal{N}} \prod_{\text{Edge} \in [l \rightarrow k] \in \text{Path}} \frac{\partial \zeta_{x_k}}{\partial \zeta_{x_l}} \right),
\]

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 \( \frac{d\zeta_y}{d\zeta_n} \) terms are estimated by total derivative estimators, such as LR or value gradient methods (Fairbank, 2008), whereas the effect of the local partial derivatives, \( \frac{\partial \zeta_{x_k}}{\partial \zeta_{x_l}} \), 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 \( \hat{g}_{TP} = k \hat{g}_{LR} + (1 - k) \hat{g}_{RP} \), where \( k = \nabla \left( \frac{\nabla [g_{RP}] / (\nabla [g_{LR}]) + \nabla [g_{RP}]}{\nabla [g_{LR}]} \right) \), i.e. it picks \( k \propto \frac{1}{\nabla [g_{LR}]} \) and \((1 - k) \propto \frac{1}{\nabla [g_{RP}]} \). 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 \( \nabla f \) 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 \( \mu \) and \( \Sigma \) at a distal node \( n \), then construct an LR-type gradient estimator for these parameters, estimating the gradients \( \frac{d\mu}{dx} \) and \( \frac{d\Sigma}{dx} \). Assuming a cost function \( c(x_n) \) that depends on \( x_n \), we can resample points on the approximated Gaussian distribution, estimate the gradients \( \frac{dc(x_n)}{d\mu} \) and \( \frac{dc(x_n)}{d\Sigma} \) and apply the chain rule to obtain the total gradient from \( x \) 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 \( x \), with \( \frac{df(x_n)}{d\mu} \) and \( \frac{df(x_n)}{d\Sigma} \). 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 **RPPProp** propagator can be used.

**RPPProp**

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., RPPProp(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: \([(\text{tensors, grad_tensors}), \text{sum of losses}]\)

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: \((\text{tensors=log p(x), grad_tensors=loss})\)

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

**Sequence:** \[\text{Optional(BackPropagator)}, \text{LRBase}, \text{Optional(BaselineProp)}\]

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: \((\text{tensors}=[\log p(x), x], \text{grad_tensors}=[k \times \text{loss}, (1-k) \times \text{grads}])\)

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 backpropagation, so there is no issue.

**TotalProp**

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

**Sequence:** \[\text{Optional(BackPropagator)}, \text{TPBase}, \text{Optional(BaselineProp)}\]

### 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).

Figure 5: Computation times of algorithms in Proppo when varying the batch size (Sec. B.3).
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 $10^1$ 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.

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 $\beta$ controls the dynamics of the RNN system introduced in Sec. 4.1. When $\beta$ is small, the system is well-behaved, but as $\beta$ 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 $\beta$ we simulate the system for 10000 steps, and plot the first dimension $x_1$ of the last 500 steps onto the figure as dots (note that one obtains a similar diagram when considering $x_2$, the second dimension of the system). We see that when $\beta$ is small (around 1), all of the dots are at the same position; hence the system converged to a fixed state. As $\beta$ is increased, the system starts oscillating between two states. As $\beta$ 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. $\beta$ 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 $\frac{dL}{dx} \frac{dx}{d\beta}$ for each time step. As the loss landscape $L$ has a fractal structure due to the chaotic properties of the system, the gradient $\frac{dL}{dx}$ 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 $\frac{dL}{dx}$, 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.

Figure 7: Scaling of computation time with the number of propagation nodes (Sec. B.4).
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 (Metz et al., 2019), protein folding software, and differentiable simulation. Moreover, the effect of chaos on estimating the sensitivity of fluid simulations has also been studied in many works, e.g., the work by ?. We believe that our minimalistic experiment captures interesting characteristics of such challenging ML tasks.

![Bifurcation diagram for x_1](image1)

![Gradient of loss, \(dL/d\beta\)](image2)

![Loss landscape w.r.t. start location](image3)

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 \(\beta\), 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: \(500 \times 500\), 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

![Graph diagram](image4)

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 \(x_0\), 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 \(\bigcirc\). The \(x\) 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 \(\beta\) node \(\bigcirc\) 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 \(\tilde{x} := [x^{(1)}; \ldots; x^{(D)}]\), and the evolution of each \(x^{(d)}\) is computed separately according to Eq. (2). While the initial state is the same for each \(d\), the added activation noise \(\varepsilon\) is different, so the trajectories for each parallel path are different. The loss is computed at the final step as \(\tilde{L}(\tilde{x}_H) = \frac{1}{2}(\tilde{x}_H - 1)^T(\tilde{x}_H - 1)\). Note that the parameters of the
networks \( W \) and \( \beta \) are not shared across the paths, and we estimate the gradient only through the first path according to \( \beta^{(1)} \). If instead the \( \beta \) 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 \( \{x^{(1,m)}\}_{m=1}^B \) from a fitted Gaussian distribution (this node is illustrated as \( \odot \) 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 is uploaded to https://github.com/proppo/proppo/proppo.

Propagation manager.

```python
# 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=None, force_targets=None, local_propagator=None, get_node=False, **kwargs):
    """ Register input as a node, and returns output through local forward function. """
    if local_propagator != None:
        if isinstance(local_propagator, str):
            local_propagator = self.propagators[local_propagator]
        else:
            node = self.default_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. """
    if loss != None:
        self.append_loss(loss)

    if message != None:  # TODO, send message to node instead.
        self._send_message(message=message)

    # 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)
# 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

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
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.

# 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) + ')
    def __str__(self):
        return 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()
            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()
else:
    cont_dict = container

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

Smart initializers.
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 initializer 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)
```python
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 Chaints 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 is 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 = []
        chainkeys = []
```

36
# 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.

import proppo.forward_methods as fm
import proppo.back_methods as bm
import proppo.baseline_funcs as baselines
import 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.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=[], **kwargs):
        # 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

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.

```python
""

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

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.

# 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)
    elif isinstance(dist_params, dict):
        dist = dist_class(**dist_params)
    elif 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)
    elif isinstance(dist_params, dict):
        dist = dist_class(**dist_params)
    elif 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_params, ivw_target, k_interval=1, requires_grad=True)
    node['ivw_target'] = ivw_target
    node['k_interval'] = k_interval
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(detached_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)}
```
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']
    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.
def mean_baseline(local_losses, node=None):
    with torch.no_grad():
        batch_size = local_losses.numel()
        # leave-one-out baselines
        sun_loss = local_losses.sum()
        # The division by (batch_size - 1) instead of (batch_size)
# 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_var = torch.sum((d1 / c)**2)
                x2_var = torch.sum((d2 / c)**2)
                x1_vars.append(x1_var)
                x2_vars.append(x2_var)
        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