Model-based Reinforcement Learning with Scalable Composite Policy Gradient Estimators

Paavo Parmas [1] **Takuma Seno** [2] **Yuma Aoki** [3]

Abstract

In model-based reinforcement learning (MBRL), policy gradients can be estimated either by derivative-free RL methods, such as likelihood ratio gradients (LR), or by backpropagating through a differentiable model via reparameterization gradients (RP). Instead of using one or the other, the Total Propagation (TP) algorithm in prior work showed that a combination of LR and RP estimators averaged using inverse variance weighting (IVW) can achieve orders of magnitude improvement over either method. However, IVW-based composite estimators have not yet been applied in modern RL tasks, as it is unclear if they can be implemented scalably. We propose a scalable method, Total Propagation X (TPX) that improves over TP by changing the node used for IVW, and employing coordinate wise weighting. We demonstrate the scalability of TPX by applying it to the state of the art visual MBRL algorithm Dreamer. The experiments showed that Dreamer fails with long simulation horizons, while our TPX works reliably for only a fraction of additional computation. One key advantage of TPX is its ease of implementation, which will enable experimenting with IVW on many tasks beyond MBRL.

1. Introduction

Reinforcement learning (RL) deals with optimizing the behavioral parameters, $\boldsymbol{\theta}$, of an agent in an environment, so as to maximize the expected rewards $\mathbb{E}_{p(\boldsymbol{\tau};\boldsymbol{\theta})}\left[R(\boldsymbol{\tau})\right]$ (Sutton & Barto, 2018) of the trajectory, $\boldsymbol{\tau}$. One main approach to RL is stochastic gradient ascent using the policy gradient (PG), $\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}}\mathbb{E}_{p(\boldsymbol{\tau};\boldsymbol{\theta})}\left[R(\boldsymbol{\tau})\right]$. Computing this expectation analytically is intractable—it can only be estimated by sampling—and, thus, the gradient must be statistically approximated.

This gradient estimation is often performed with the likelihood ratio (LR) method that only requires the value of R to create an unbiased estimator (Glynn, 1990; Williams, 1992; Sutton et al., 2000).[1] While LR does not require knowledge of the system, if we *do* have access to such a differentiable model, we can use the reparameterization gradient (RP)[2] that uses $\frac{\mathrm{d}R}{\mathrm{d}\boldsymbol{\tau}}$ to perform a more efficient estimate (Rezende et al., 2014; Jordan & Rumelhart, 1992; Werbos, 1990).

As $\frac{\mathrm{d}R}{\mathrm{d}\boldsymbol{\tau}}$ contains D bits of information (one for each dimension of $\boldsymbol{\tau}$), while R is a single scalar, one can construct examples where the accuracy of RP scales D times better than LR (Rezende et al., 2014). This improved scalability is enjoyed by several impressive RL results, such as differentiable simulation (Xu et al., 2022), visual MBRL (Hafner et al., 2023) or inventory management (Madeka et al., 2022).

Unfortunately, derivative-based methods such as RP are not always effective. Parmas et al. (2018); Parmas (2020) decisively showed in MBRL that RP has a major flaw in chaotic systems causing the gradient estimation variance to explode. This issue is related to the exploding gradient problem (Pascanu et al., 2013), and it is common in machine learning tasks with long chains of non-linear computations.

The LR method was robust to the problem with chaos; however, LR is in general inefficient, so there is a risk that the solutions to many problems of interest are out of the reach of standard algorithms. Fortunately, Parmas also proposed an algorithm to alleviate the issue: Total Propagation (TP).

Total Propagation combines the best of RP and LR: it achieves efficiency while being robust to chaos. The basic idea is that it performs weighted averaging of LR and RP, with the weights estimated by inverse variance weighting (IVW), without requiring additional hyperparameters. Moreover, TP includes a step-wise algorithmic approach that can even give 100 times improvement in gradient accuracy (Parmas & Seno, 2022), particularly at the edge of chaos. TP also influenced follow-up works that employed IVW in varied fields such as meta-learning (Metz et al., 2019) and differentiable simulation (Suh et al., 2022).

[1]Kyoto University [2]Sony AI [3]University of Tokyo. Correspondence to: Paavo Parmas <paavo@sys.i.kyoto-u.ac.jp>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

[1]LR is often also called the score function estimator.

[2]RP is a type of pathwise gradient similar to back propagation.

While the idea in TP is promising, the original work (and follow-up works) only applied it on low-dimensional continuous control problems. Therefore, we investigate how to scale up the TP algorithm, so that it can be applied to modern machine learning tasks beyond toy problems.

The main problem with TP was that it could not be implemented easily due to its low compatibility with automatic differentiation (AD) frameworks. To overcome this issue, in our concurrent work (Parmas & Seno, 2022) we created a new machine learning framework, Proppo that allows automatically using TP on arbitrary computation graphs.

Using our framework, we implement a new more scalable variant of TP that we call Total Propagation X, which has the main differences: (1) instead of estimating the gradient variances at the $\boldsymbol{\theta}$ node, it estimates the variances at the $\boldsymbol{\tau}$ node, (2) in addition to the variance, it takes into account the covariances of the gradient estimates, (3) instead of a scalar mixing weight, k, it uses a tensor-valued weight, $\boldsymbol{k}$, with different weights for different dimensions of the gradients.

We perform experiments in two settings: we replicate the cart-pole task in the original TP paper (PIPPS, App. B), we combine TPX with the state of the art visual MBRL algorithm Dreamer (Sec. 6) (Hafner et al., 2020), and apply it on continuous control from pixels in the DMC environments (Tunyasuvunakool et al., 2020). The results showed that TPX can reliably match TP while requiring less computation and being easier to implement. Moreover, TPX improved the robustness to the hyperparameter choices, such as the length of the prediction horizon, H.

Our contributions are as follows:

- We propose Total Propagation X, the first composite gradient estimation algorithm using inverse variance weighting that is demonstrated to be applicable at scale.
- We combine TPX with Dreamer, and show that it improves robustness to the horizon hyperparameter.
- We analyze the design choices in Total Propagation, and shed light on their effect.
- We propose methodology for comparing the quality of gradient estimators that may be biased.

While advanced composite gradient estimators have shown promising results, the major barrier to their adoption is how to easily implement them at scale. The design choices in TPX appear generally applicable, and following them is likely to enable scaling up future composite estimators as well, not only in MBRL but also other tasks, defined on arbitrary stochastic computation graphs (Schulman et al., 2015).

2. Related Work

Following the original TP, several works also used IVW to combine gradient estimates. Metz et al. (2019) replicated the experimental analysis of Parmas et al. (2018) in the context of meta-learning to show that the same issues with chaos occur, and used IVW to combine RP and LR gradients to improve the performance, similarly to TP. Geffner & Domke (2018) used IVW to automatically combine control variates (Greensmith et al., 2004) for variance reduction in stochastic variational inference (adding the control variate can be shown to be equivalent to averaging the gradient estimates). In later work, Geffner & Domke (2020) proposed a technique for incorporating the computation time for estimator selection. In differentiable simulation, Suh et al. (2022) proposed an extension to IVW to attempt dealing with the bias in the RP estimate when there are discontinuities (Lee et al., 2018; Parmas & Sugiyama, 2021). All of the above works only dealt with small scale tasks or low-dimensional models. Moreover, they did not use the step-wise gradient combination scheme used in the full TP, as it is difficult to implement. In our concurrent work, we showed that the step-wise scheme can improve the performance by multiple orders of magnitude (Parmas & Seno, 2022).

3. Background

Notation. Bold letters, $\boldsymbol{x}$, are tensors. The notation $\hat{\mathbb{E}}_i^N[\cdot] = \frac{1}{N}\sum_{i=1}^N (\cdot)^{(i)}$ refers to the empirical mean of a set of N elements, where the elements are indexed by i. The empirical variance has an analogous notations $\hat{\mathbb{V}}_i^N[\cdot]$. The similar covariance estimator $\hat{\text{cov}}_i^N[\cdot,\cdot]$ computes the dimension-wise covariances, but not the off-diagonal terms of the covariance matrix. The notation $\sum \boldsymbol{x}$ means to sum all elements of the tensor, unless the index is specified. Some tensors $\boldsymbol{x}$ may have a batch axis, and we index the i^{th} member of the batch with the notation $\boldsymbol{x}^{(i)}$. Other dimensions are indexed with the $\boldsymbol{x}_{(\cdot)}$ notation, e.g., $\boldsymbol{x}_{(ij)}$ denotes the element in the i^{th} row and j^{th} column of the matrix $\boldsymbol{x}$. The notation $\boldsymbol{x}_i$ does not index the tensor, but we use it to discern different tensors that are somehow related, e.g., states at different time-steps in a sequence may be denoted by $\boldsymbol{x}_t$. For two tensors $\boldsymbol{x}$ and $\boldsymbol{y}$ where, they have the same shape except for $\boldsymbol{x}$ having an additional batch axis (or vice versa), we define the broadcast product $\boldsymbol{x} \odot \boldsymbol{y}$ that multiplies $\boldsymbol{y}$ element-wise with each member of $\boldsymbol{x}$. The gradient tensor $\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}}$ has $Shape\left(\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}}\right) = [Shape(\boldsymbol{y}), Shape(\boldsymbol{x})]$, and the product between two gradient tensors, $\frac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}\boldsymbol{y}}\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}} = \frac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}\boldsymbol{x}}$, is defined such that the chain rule holds, i.e., for fixed indexes of $\boldsymbol{z}$ and $\boldsymbol{x}$, all of the corresponding elements at indexes of $\boldsymbol{y}$ are multiplied and summed. Division between two tensors denotes an element-wise division. Note that the notations here are for mathematical convenience, and need not represent how the methods are implemented in code.

Elementary gradient estimators. An unbiased gradient estimator, $\hat{\boldsymbol{g}}$, is a random variable, such that

$$\mathbb{E}\left[\hat{\boldsymbol{g}}\right] = \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}} \mathbb{E}_{p(\boldsymbol{\tau};\boldsymbol{\theta})}\left[R(\boldsymbol{\tau})\right]. \tag{1}$$

The LR gradient estimator is $\hat{\boldsymbol{g}}_{LR} = \frac{\mathrm{d}\log p(\boldsymbol{\tau};\boldsymbol{\theta})}{\mathrm{d}\boldsymbol{\theta}}(R-b)$, where b is a constant baseline for variance reduction. We use the mean baseline throughout, $b = \hat{\mathbb{E}}_i^N\left[\boldsymbol{R}^{(i)}\right]$.

The RP gradient estimator is $\hat{\boldsymbol{g}}_{RP} = \frac{\mathrm{d}R}{\mathrm{d}\boldsymbol{\tau}}\frac{\mathrm{d}\mathcal{T}(\boldsymbol{\varepsilon})}{\mathrm{d}\boldsymbol{\theta}}$, where $\mathcal{T}_{\boldsymbol{\theta}}$ is a reparameterization transform, such that sampling $\boldsymbol{\varepsilon} \sim p(\boldsymbol{\varepsilon})$ and transforming $\boldsymbol{\tau} = \mathcal{T}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon})$ is equivalent to $\boldsymbol{\tau} \sim p(\boldsymbol{\tau};\boldsymbol{\theta})$.

It is often useful to rewrite $\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}}(\cdot)$ in the estimator by the chain rule to an intermediate distribution parameter $\boldsymbol{\zeta}$, i.e.,

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}}(\cdot) = \frac{\mathrm{d}\boldsymbol{\zeta}}{\mathrm{d}\boldsymbol{\theta}}\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\zeta}}(\cdot). \tag{2}$$

For example, $\boldsymbol{\zeta} = [\boldsymbol{\mu}, \boldsymbol{\sigma}]$ may be the parameters of a Gaussian distribution, while $\boldsymbol{\theta}$ are the parameters of a neural network that output $\boldsymbol{\zeta}$. This way, the total derivative estimator can be rewritten as a sum over gradient estimations at intermediate nodes, from which we backpropagate the estimated gradients to $\boldsymbol{\theta}$ (Parmas, 2018).

Total Propagation. The idea in TP is to construct a new gradient estimator by the weighted average of LR and RP:

$$\hat{\boldsymbol{g}}_{\text{TP}} = k\hat{\boldsymbol{g}}_{\text{LR}} + (1-k)\hat{\boldsymbol{g}}_{\text{RP}}. \tag{3}$$

An appropriate weight, k, yields a lower variance compared to either estimator alone, by averaging the randomness. The TP method chooses k by the well-known optimal weighting scheme (assuming uncorrelated random variables) called inverse variance weighting (IVW). In IWV, k is chosen inversely proportional to the variance of the estimator, i.e.,

$$k \propto \frac{1}{\mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{LR}}\right]} \quad \text{and} \quad (1-k) \propto \frac{1}{\mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{RP}}\right]}. \tag{4}$$

In practice, one computes a batch of N gradient estimator samples for both RP and LR, and performs the weighting using the empirical sample variance of the batch:

$$\hat{k} = \frac{\sum \hat{\mathbb{V}}_i^N\left[\hat{\boldsymbol{g}}_{\text{RP}}^{(i)}\right]}{\sum \hat{\mathbb{V}}_i^N\left[\hat{\boldsymbol{g}}_{\text{LR}}^{(i)}\right] + \sum \hat{\mathbb{V}}_i^N\left[\hat{\boldsymbol{g}}_{\text{RP}}^{(i)}\right]}. \tag{5}$$

The naïve IVW method would perform the combination based on separate estimates of RP and LR through the whole computation graph. However, this approach is limited, as it can reduce the variance at most by a factor 2 until the lower bound $\mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{naïve}}\right] \geq \frac{1}{2}\min\left(\mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{LR}}\right], \mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{RP}}\right]\right)$ (with equality when the variances are equal, $\mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{LR}}\right] = \mathbb{V}\left[\hat{\boldsymbol{g}}_{\text{RP}}\right]$, App. A). The TP method overcomes this limitation by computing LR and RP together, and combining the estimators at a node-wise level in the graph. This procedure enabled reducing the variance by over 100 times in a minimalistic recurrent neural network experiment (Parmas & Seno, 2022)!

An illustration of the TP algorithm is shown in Fig. 1. This example is based on a multi-layer neural network (NN) in supervised learning, with an input $\boldsymbol{x}$, hidden variables $\boldsymbol{z}$, an output $\boldsymbol{y}$, a target $\boldsymbol{t}$, the sample-wise losses $\boldsymbol{L}$ and the neural network weight parameters of each layer $\boldsymbol{W}$. Moreover, this is a stochastic NN, and the hidden variables are sampled $\boldsymbol{z}_n \sim p(\boldsymbol{z}_n;\boldsymbol{\zeta}_n)$, where $\boldsymbol{\zeta}_{\boldsymbol{z}_n} = f(\boldsymbol{z}_{n-1}, \boldsymbol{W}_n)$ are the distribution parameters at the node that depend on the NN weights and the previous hidden variable. Note that sampling is necessary for gradient estimation. The algorithm

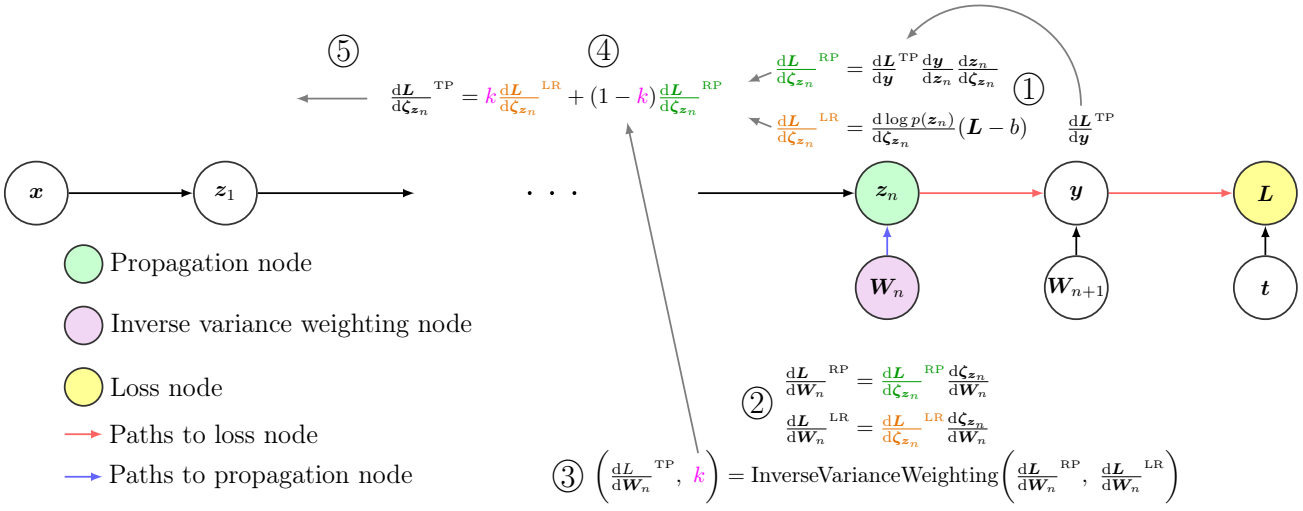


Figure 1: Illustration of the Total Propagation algorithm. The explanation is in Sec. 3

works by iterating backwards through the *propagation nodes* (Parmas & Seno, 2022) where RP and LR are combined via IVW. We explain the procedure at each propagation node:

① In the first step, we compute the RP and LR estimators w.r.t. the distribution parameters at the propagation node, $\boldsymbol{\zeta}_{\boldsymbol{z}_n}$. The RP estimator is computed by the chain rule via the incoming gradients $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{y}}^{\mathrm{TP}}$, and by estimating $\frac{\mathrm{d}\boldsymbol{z}_n}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}$ via the RP trick. The LR estimator is computed in the standard way using the the losses, $\boldsymbol{L}$. Both RP and LR are unbiased estimates of the total derivative from the propagation node to the loss node.

② Next, we separately backpropagate the RP and LR gradient estimates from $\boldsymbol{\zeta}_{\boldsymbol{z}_n}$ (the propagation node) to the weight parameters of our model, $\boldsymbol{W}_n$ (the gradients along the blue lines). Note that in typical AD software, the incoming batch gradients are automatically summed at the $\boldsymbol{W}_n$ node, meaning that the sample-wise gradients, $\hat{\boldsymbol{g}}^{(i)} = \frac{\mathrm{d}\boldsymbol{L}^{(i)}}{\mathrm{d}\boldsymbol{W}_n}$, are not accessible. However, TP requires the sample-wise gradients to be able to estimate the batch gradient variance, $\hat{\mathbb{V}}_i^N\left[\hat{\boldsymbol{g}}^{(i)}\right]$, for use in IVW (Eq. 5). Therefore, TP requires a non-trivial implementation that estimates sample-wise gradients.

③ In the third step, we use the obtained sample-wise gradients, $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{W}_n}^{\mathrm{RP}}$ and $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{W}_n}^{\mathrm{LR}}$ to compute the empirical variances $\hat{\mathbb{V}}_i^N\left[\frac{\mathrm{d}\boldsymbol{L}^{(i)}}{\mathrm{d}\boldsymbol{W}_n}^{\mathrm{LR}}\right]$ and $\hat{\mathbb{V}}_i^N\left[\frac{\mathrm{d}\boldsymbol{L}^{(i)}}{\mathrm{d}\boldsymbol{W}_n}^{\mathrm{RP}}\right]$ that are used for IVW to compute the scalar weight ratio k and the combined gradient $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{W}_n}^{\mathrm{TP}}$. Notice that in the TP gradient, the batch of gradients are averaged, and the losses are collapsed to a scalar $L = \hat{\mathbb{E}}_i^N\left[\boldsymbol{L}^{(i)}\right]$. This is the gradient that will be applied to the weights $\boldsymbol{W}_n$ in the optimization algorithm.

④ Now, the scalar weight k is used to combine the gradient estimates together at the $\boldsymbol{z}_n$ node obtaining $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}^{\mathrm{TP}} = k\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}^{\mathrm{LR}} + (1-k)\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}^{\mathrm{RP}}$. This operation, combines the two gradient estimators for each member of $\boldsymbol{\zeta}_{\boldsymbol{z}_n}$ together into a single one.

⑤ Finally, the combined gradient $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}^{\mathrm{TP}}$ is backpropagated to the preceding node, where it is used for RP gradient estimation, equivalently to $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{y}}^{\mathrm{TP}}$ in step ①.

The key step of TP is the combination of the gradients in the state space in ④. This step has two main advantages over a naïve combination of LR and RP: (a) It reduces the total amount of computation, as a single averaged gradient is backpropagted instead of two, (b) It allows overcoming the bound of $2\times$ improvement over the best between RP and LR. By improving the gradient estimation accuracy at each backward step, the accuracy improvements compound leading to a potentially much more accurate estimate.

4. Proposal: Variants of Total Propagation

The basic version of TP described in Sec. 3 has shown promising results in small scale tasks, but some design elements have made it difficult to apply at scale. The main issue is that it was difficult to implement, but we solved this in our concurrent work by providing a software framework for automatically using TP (Parmas & Seno, 2022). However, there are still several points on which TP could be improved.

The issues:

(i) Total propagation requires computing the individual gradient samples in the batch $\frac{\mathrm{d}\boldsymbol{L}^{(i)}}{\mathrm{d}\boldsymbol{\theta}}$, or at least it requires modifying typical AD to obtain $\hat{\mathbb{V}}_i^N\left[\frac{\mathrm{d}\boldsymbol{L}^{(i)}}{\mathrm{d}\boldsymbol{\theta}}\right]$ (as described in ② in Sec. 3). Such a modification to the code will reduce the efficiency of the computations. Moreover, if the parameter space $\boldsymbol{\theta}$ is high-dimensional, computing the individual gradients will require a lot of memory, and may not be practical.

(ii) In step ② in Fig. 1, the gradients for RP and LR are propagated separately through the same paths in the computation graph. If k were known beforehand, this computation would be redundant as the RP and LR gradients could be averaged before backpropagating, i.e., step ④ could be done first to obtain $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}^{\mathrm{TP}}$, which could be used to obtain $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{W}_n}^{\mathrm{TP}}$ using the chain rule.

(iii) In step ④, the gradients at the propagation node are combined using a scalar k. This may be inefficient, as the different dimensions of $\boldsymbol{\zeta}_{\boldsymbol{z}_n}$ may require different mixing ratios to achieve efficient estimation. Using a tensor $\boldsymbol{k}$ instead (one k for each dimension of $\boldsymbol{\zeta}_{\boldsymbol{z}_n}$) may reduce the variance, but it is non-trivial to estimate such a tensor in the current setup, as the dimensionality of the parameter space ($\boldsymbol{\theta} = \boldsymbol{W}_n$ in the example) may have a different dimensionality compared to $\boldsymbol{\zeta}_{\boldsymbol{z}_n}$.

(iv) The IVW used in TP is optimal when combining statistically *independent* random variables, but does not take into account possible *correlations* in the LR and RP estimators (App. A). The true optimal mixing weight when combining random variables, $kX + (1-k)Y$, takes into account the covariance σ_{XY}, and is given by

$$k = \frac{\sigma_Y^2 - \sigma_{XY}}{\sigma_X^2 + \sigma_Y^2 - 2\sigma_{XY}}. \tag{6}$$

Proposed solutions: Our first proposal is to perform the IVW using only a subset of the parameters, $\boldsymbol{\theta}$, of the model. We call this Total Propagation S (TPS)—while it is algorithmically the same as TP, the setup is not the same as the vanilla version using all of the parameters. This proposal

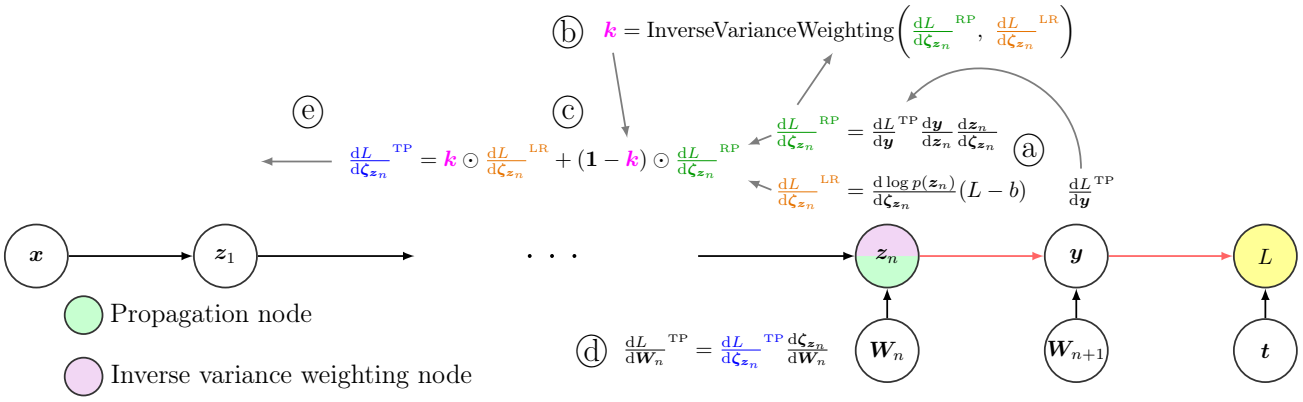


Figure 2: Illustration of the Total Propagation X algorithm. The explanation is in Sec. 4.

will remove the large memory requirements and will at least make it feasible to apply TP at larger scale. However, it only addresses the first issue (and only partially) without fixing the others. We apply TPS as a comparison in our later Dreamer experiments (Sec. 6). However, our second proposal is more practical and addresses all of the issues.

We propose Total Propagation X (TPX) a variant of TP that tackles all of the issues above. The algorithm is illustrated in Fig. 2. The main idea is to use a different node for IVW. Specifically, we estimate k using the local gradients, $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}$, at the propagation node (in step ⓑ), instead of using $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{W}_n}$ as in the standard TP (in step ③). This way, the individual gradients w.r.t. $\boldsymbol{W} = \boldsymbol{\theta}$ are not necessary, mitigating issue (i). Moreover, the weighted average of LR and RP can be computed immediately (in step ⓒ), and the combined gradient can be back propagated to the parameter node (in step ⓓ), mitigating issue (ii). Furthermore, as $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}$ has the same shape as $\boldsymbol{\zeta}_{\boldsymbol{z}_n}$, we can estimate a tensor $\boldsymbol{k}$, mitigating issue (iii). The gradient can be combined element-wise,

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}^{\mathrm{TP}} = \boldsymbol{k} \odot \frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}^{\mathrm{LR}} + (\mathbf{1} - \boldsymbol{k}) \odot \frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}_n}}^{\mathrm{RP}}, \quad (7)$$

and backpropagated to both the previous propagation node, as well as the parameter node. This order of operations allows skipping steps ② and ③ in Fig. 1, and saves much of the computational cost (i.e., in step ② we compute two gradients, but in step ⓓ only one gradient is necessary). Perhaps the biggest advantage of TPX is the mitigation of issue (i): there is no longer a need to modify the implementation to obtain the individual gradients $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{W}}$, but all gradients can use a scalar L, as is typical.

Why is this possible, and how do we estimate $\boldsymbol{k}$? Unlike the gradient w.r.t. the parameters $\boldsymbol{W}$, the gradient w.r.t. intermediate variables, such as $\boldsymbol{z}_n$ already contains the sample-wise gradients separated out. In particular $\frac{\mathrm{d}\boldsymbol{L}^{(i)}}{\mathrm{d}\boldsymbol{z}_n^{(i)}} = N\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{z}_n^{(i)}}$, where $L = \hat{\mathbb{E}}_i^N\left[\boldsymbol{L}^{(i)}\right]$, because the samples in the batch are independent, and $\boldsymbol{z}^{(j)}$, where $j \neq i$ does not affect the computation of $\boldsymbol{L}^{(i)}$.[3] The gradient $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{z}}$ contains one entry for each element of $\boldsymbol{z}$, and as $\boldsymbol{z}$ includes a batch axis, the gradient will be represented sample-wise. We denote the sample-wise gradients as $\hat{\boldsymbol{g}}^{(i)} = \frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\zeta}_{\boldsymbol{z}}^{(i)}}$,[4] and estimate $\boldsymbol{k}$ as

$$\hat{\boldsymbol{k}} = \frac{\hat{\mathbb{V}}_i^N\left[\hat{\boldsymbol{g}}_{\mathrm{RP}}^{(i)}\right] - \hat{\mathrm{cov}}_i^N\left[\hat{\boldsymbol{g}}_{\mathrm{RP}}^{(i)}, \hat{\boldsymbol{g}}_{\mathrm{LR}}^{(i)}\right]}{\hat{\mathbb{V}}_i^N\left[\hat{\boldsymbol{g}}_{\mathrm{LR}}^{(i)}\right] + \hat{\mathbb{V}}_i^N\left[\hat{\boldsymbol{g}}_{\mathrm{RP}}^{(i)}\right] - 2\hat{\mathrm{cov}}_i^N\left[\hat{\boldsymbol{g}}_{\mathrm{RP}}^{(i)}, \hat{\boldsymbol{g}}_{\mathrm{LR}}^{(i)}\right]}. \quad (8)$$

This equation creates an estimator for the ideal weighting in Eq. (6). In addition to including the covariance to address issue (iv), the major difference to the previous approach in Eq. (5) is the omission of the $\sum$ operator that sums the variances across all dimensions.

One could consider variants of TP that add a summation into Eq. (8) and use a scalar $\hat{k}$ at the propagation node. Another variant might still use a tensor $\hat{\boldsymbol{k}}$, but omit the covariance terms. The ideal $\boldsymbol{k}$ should include these features; however, there is a caveat that we cannot directly compute $\boldsymbol{k}$ but only estimate it. The other variants may be easier to estimate, and as a result lead to better overall performance. To clarify to what extent these design elements affect the performance, we evaluate them experimentally in later sections.

Summary of the advantage of TPX. The major advantage of TPX compared to TP is the ease of implementation. Specifically, while the modifications necessary to one's code to use TPX using Proppo are exactly the same as those for RP or LR, TP requires additional modifications to enable obtaining the individual gradients $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{\theta}}$. More accurately, in principle, we only need a means to compute the batch variance of the gradients, and in some situations there exist

[3] Note that the independence assumption is not true for some operations, e.g., batch normalization (Ioffe & Szegedy, 2015).

[4] In practice, as we are mostly using Gaussian distributions ($\boldsymbol{\zeta} = [\boldsymbol{\mu}, \boldsymbol{\sigma}]$), we estimate $\boldsymbol{k}$ from the mean parameters, $\boldsymbol{\mu}$, and use the same weighting for the scale, $\boldsymbol{\sigma}$. However, it would be straightforward to estimate separate weightings for both parameters.

also other efficient ways to do this (Dangel et al., 2020). However, it still requires non-standard modifications. Such requirements increase the barrier to adoption of TP. Another clear advantage of TPX is the reduction in computation time due to removing redundant operations. Regarding accuracy, whether TP or TPX is better is problem dependent. As we are ultimately interested in the gradient variance at the $\boldsymbol{\theta}$ node, TP has the advantage that it estimates the variances at this node. However, TPX has the advantage of using a tensor $\boldsymbol{k}$. Therefore, the winner in terms of accuracy will depend on which merit was more important for the given task. Nevertheless, in Sec. 5 we show a conceptual advantage of TPX when the behavior of the dynamics varies between the dimensions. In summary, TPX has clear advantages in terms of ease of implementation, scalability and computation speed, but whether it can match TP's accuracy has to be experimentally examined.

5. Minimalistic Experiment: the Advantage of Total Propagation X

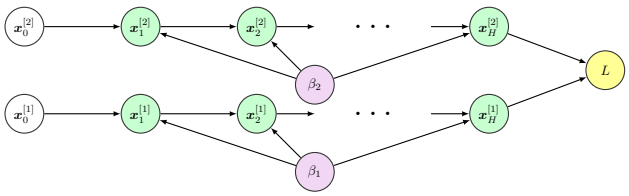


Figure 4: Parallel chaotic RNN problem setting.

In Sec. 4, we hypothesized an advantage of TPX over TP in the multi-dimensional setting—in TP, the scalar mixing weight, k, may lead to suboptimal gradient accuracy in some dimensions. As TPX uses a different mixing weight for each dimension, it may overcome such an issue. To demonstrate this advantage of TPX over TP, we perform a minimalistic experiment using recurrent neural networks (RNN), with a related setup to our concurrent work (Parmas & Seno, 2022).

The setup is illustrated in Fig. 4. There are two parallel chains of computations performed by a type of sigmoid RNN that was proven to exhibit chaotic behavior by Wang (1991). The variables along the i^{th} paths are denoted with the superscript in closed brackets, $\boldsymbol{x}^{[i]}$; note that this is not the same as indexing the batch dimension. The dynamics of each RNN is $\boldsymbol{x}_{t+1}^{[i]} = \text{Sigmoid}(\beta_i \boldsymbol{W}_i \boldsymbol{x}_t^{[i]}) + \boldsymbol{\varepsilon}$, where $\text{Sigmoid}(y) := \frac{1}{1+\exp(-y)}$, $\boldsymbol{x}^{[i]}$ is 2-dimensional, $\boldsymbol{W} = \begin{bmatrix} -5 & 5 \\ -25 & 25 \end{bmatrix}$, $\boldsymbol{\varepsilon} \sim \mathcal{N}(0, 0.001^2 \boldsymbol{I})$, and the initial state is $\boldsymbol{x}_0^{[i]} = [0.35; 0.55]$ for both $i \in \{1, 2\}$. At the end of the chain, we compute a loss $L := \frac{1}{2}(\boldsymbol{x}_H^{[1]\text{T}} \boldsymbol{x}_H^{[1]} + \boldsymbol{x}_H^{[2]\text{T}} \boldsymbol{x}_H^{[2]})$, and we estimate gradients of L w.r.t. β_1. We compare the gradient estimation variance of TP, TPX, LR and RP.

The inverse temperature, β, controls the dynamics of the system. Around $\beta = 2.5$, there is a phase transition—when $\beta < 2.5$ the system is well-behaved, and when $\beta > 2.5$, the system is chaotic. In the well-behaved region, RP gradients are accurate, but in the chaotic regime, the RP gradient variance explodes. LR follows an opposite pattern: it is robust to chaos, but not as accurate as RP in the well-behaved region. TP automatically chooses an optimal weighting of RP and LR. At the edge of chaos, TP outperforms both LR and RP by orders of magnitude (Parmas & Seno, 2022).

The intention of the experiment is that with two parallel paths with different β_1 and β_2, the dynamics—hence, also the optimal k—would be different for the two paths. As TP chooses a single k to be used in both paths, we hypothesize that it will perform suboptimally in this regime. TPX, on the other hand, chooses a separate k for each dimension and should be unaffected by the other path.

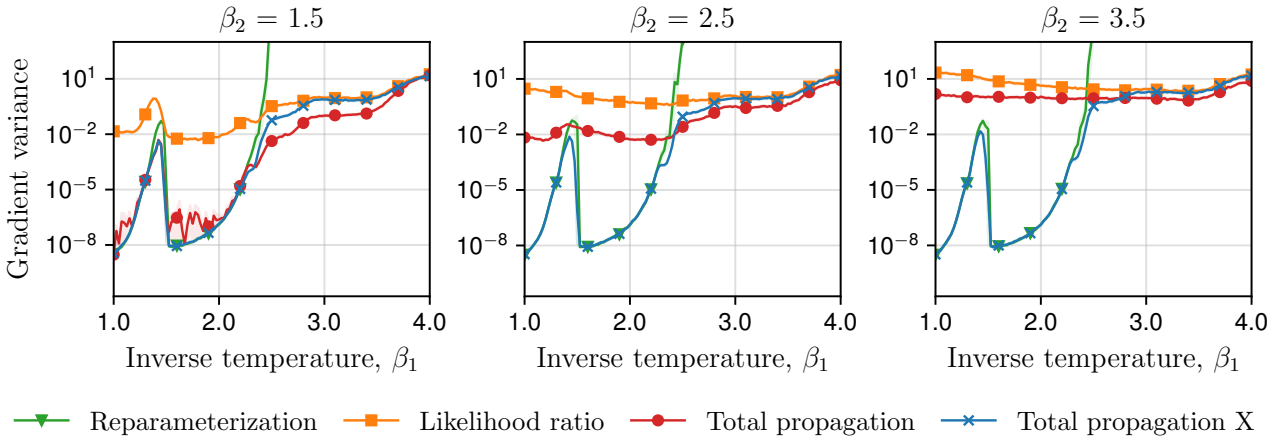


Figure 3: Gradient variance of the double chaotic recurrent neural network in Fig. 4.

The results in Fig. 3 support our hypothesis. Here, we chose a fixed $\beta_2 \in \{1.5, 2.5, 3.5\}$, and varied $\beta_1 \in [1.0, 4.0]$. We see that when $\beta_2 = 1.5$—and hence the dynamics of the parallel path is stable—TP and TPX give a similar performance. However, when $\beta_2 \in \{2.5, 3.5\}$, TP gives poor results when β_1 is in the stable region. When β_1, is small, the RP gradients along the first path are accurate, yet TP is unable to take advantage of these accurate gradients, because the inaccurate gradients along path 2 force TP to place a large weight on the LR gradient. As TPX computes the gradient variance for each of the 4 dimensions separately, the gradients along the two parallel paths do not affect the IVW calculations, and it does not suffer from the same problem as TP.

6. Visual Model-based Reinforcement Learning with Total Propagation

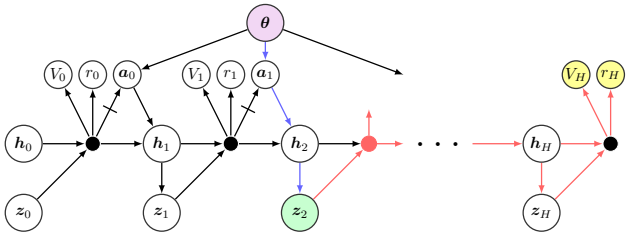


Figure 5: Illustration of imaginary trajectories in the Dreamer algorithm.

We first replicate the MBRL algorithm PIPPS in the original TP paper to show that our implementation is accurate (App. B). The results showed that TP-based methods outperform RP and LR across all learning rates (App. B.2), and were overall much more reliable. Moreover, TPX matched TP, but with a faster computation time.

Next, to show that our methods can scale to modern high-dimensional RL settings, we modify the Dreamer algorithm (Hafner et al., 2020) to use TPS and TPX for computing the policy gradient. Dreamer is an MBRL algorithm that can deal with high-dimensional visual observations (64 by 64 pixel images). It has shown impressive results in continuous control, Atari games (Hafner et al., 2021), recently also in learning to control real robots (Wu et al., 2022), and moreover its latest version is the first algorithm to find diamonds without human data in the game of Minecraft (Hafner et al., 2023). It is one of the algorithms at the forefront of RL research, and demonstrating that TP can be scaled to run together with Dreamer would show for the first time that TP-like composite policy gradient estimators can be applied to state of the art algorithms. We focus our experiments on the original DreamerV1 implementation, and on continuous control environments. We keep the hyperparameters and all settings the same as in the original Dreamer, but only swap the RP gradient estimators in the Dreamer policy gradient estimator to use TPX to test whether our algorithm can scale, and whether it adds any benefit. A full evaluation of how much the performance can be improved with our method would require also tuning the hyperparameters; however, this is expensive, and we leave it to future work.

6.1. Technical Details of Dreamer and DreamerTPX

Architecture. The Dreamer architecture consists of three neural networks: a critic $V(\boldsymbol{x})$, a policy, $\pi_{\boldsymbol{\theta}}(\boldsymbol{x})$, and a latent state-space model (LSSM). The LSSM is similar to a deep Kalman filter (Krishnan et al., 2015), and further decomposes into an encoder, a decoder and a dynamics model. The representation is learned by a variational autoencoder (Kingma & Welling, 2013) that encodes the images into a latent space using the encoder, and reconstructs the image using the decoder. The latent space $\boldsymbol{x} = [\boldsymbol{h}, \boldsymbol{z}]$ decomposes into a deterministic part, $\boldsymbol{h}$ (200 dimensional), and a stochastic part, $\boldsymbol{z}$ (30 dimensional), with a Gaussian distribution and a diagonal covariance. The dynamics model predicts the next latent state $\boldsymbol{x}_{t+1}$ given the current latent state $\boldsymbol{x}_t$ and action $\boldsymbol{a}_t$. The deterministic state $\boldsymbol{h}$ is predicted using a GRU RNN (Cho et al., 2014) while the stochastic state $\boldsymbol{z}$ is predicted from $\boldsymbol{h}$ using a neural network (see Fig. 5).

Training. Learning iterates between applying the policy on the real environment, and optimizing the parameters offline in simulations with the model. The state transitions in the real environment are stored into a dataset. During optimization, the algorithm samples, from the dataset, a batch of 50 sequences of 50 images (2500 data points in total), then encodes the images into the latent space. The LSSM can be directly trained by the reconstruction loss from these images. The part relevant to our paper is the optimization of the policy. The policy is optimized by simulating trajectories in the latent space, computing a return along the trajectory, and maximizing the return via backpropagation (the simulation with the learned model is differentiable). The trajectories start from the 2500 encoded states, and they have length H. The return is computed using λ weighting using the rewards and values (Sutton & Barto, 2018). After one gradient step, we sample a new batch of images from the dataset and repeat the process for 100 iterations. Then, the policy is applied in the real environment again to gather more data.

Combination with TP and Proppo. Our proposed modifications only affect the policy optimization part of the Dreamer algorithm. The computation graph for gradient estimation is illustrated in Fig. 5. The regular Dreamer estimates the gradient through the sampling operation at the $\boldsymbol{z}$ node via RP gradients. In our implementation we enable using LR, TP and TPX gradients instead of RP. To enable TP to scale to this size of model, we compute the variance for IVW using only the parameters at the last layer of the policy (using all of the parameters was impractically slow). On the

other hand, TPX did not require any special modification.

One can observe that if the propagation node is at $\boldsymbol{z}$, the gradients can slip past it through the $\boldsymbol{h}$ node, so the sampling nodes are not blocking the paths of the gradients. This may cause concern when the horizon H is long, as the gradients might explode due to chaos. Moreover, the gradients slipping past the propagation nodes may cause software errors. However, we found that the dynamics of $\boldsymbol{h}$ are governed by a GRU cell, and the gradients were well-behaved. Moreover, we solve the second issue by using Proppo to pause the gradients at $\boldsymbol{h}$ to wait for the gradients from the propagation node to arrive, before they are sent backwards together.

We note that due to implementation peculiarities of Dreamer, its gradient will necessarily be biased. We detail this point in App. C.1 and explain how we match the other gradient estimators to be equivalent to Dreamer.

6.2. Experimental Evaluation

We reproduced Dreamer in PyTorch using the same hyperparameters as in the original article.[5] The performance matched the original, and the computational speed was also similar to the TensorFlow 2 version (our implementation is slightly faster using 32-bit precision, but we have not implemented mixed precision). We compared the performance against our algorithms implemented in Proppo: RP (expected to be the same as Dreamer), TPX, TPS and LR.

Experiments. We evaluated the algorithms on eight continuous control DeepMind Control Suite (Tunyasuvunakool et al., 2020) environments using the MuJoCo simulator: Cartpole Swingup, Cartpole Swingup Sparse, Cheetah Run, Cartpole Balance, Walker Walk, Walker Run, Finger Spin and Reacher Easy. We perform two sets of experiments: (1) *standard* training evaluations as in the Dreamer article (for 1000 episodes), (2) training with a *pretrained* world model (for 400 episodes), but with a randomly initialized policy (in App. C.3). The aim of the second set of experiments is to reduce the influence of the model learning speed on the experiments. If the model learning were the bottleneck in the Dreamer algorithm, then we may not see any change from using different policy gradient estimators, as each method may achieve the optimal policy given the current model (in practice, we did not see a major difference in the experimental results of the two settings). We evaluate each experiment with 4 random seeds. The standard experiments were run on a mix of Nvidia RTX 2080 Ti and V100 GPUs, while the pretrained experiments were run on a uniform setup of V100 GPUs. We perform experiments for simulation horizons $H \in \{15, 60\}$, to test the sensitivity to this hyperparameter for the different algorithms. Finally, we compare the gradient estimation accuracies of the methods, and perform an ablation study on TPX (App. D).

Analysis methods. We evaluate the experimental data from several angles. First, we compute learning curves, and plot the raw data together with confidence intervals (Figs. 16, 17, 18, 19). To provide better statistical significance, we follow the protocol of Agarwal et al. (2021) and create performance profiles using aggregated data (Figs. 6, 15). Moreover, we compute the gradient variances for each environment at different stages of the learning (Apps. C, D). The variances were computed using a checkpointed model from Dreamer, using the same model for each algorithm. As Dreamer includes some implementation tricks that add a bias into the gradient estimator (App. C.1), the LR gradient and RP gradient will not have the same expected value. To deal with this issue, we propose an expected signal-to-noise ratio metric to compare the gradient estimators (see App. C.2 for details). Moreover, we aggregate a normalized version of this metric together over the eight environments to increase the statistical significance (Figs. 7, 28).

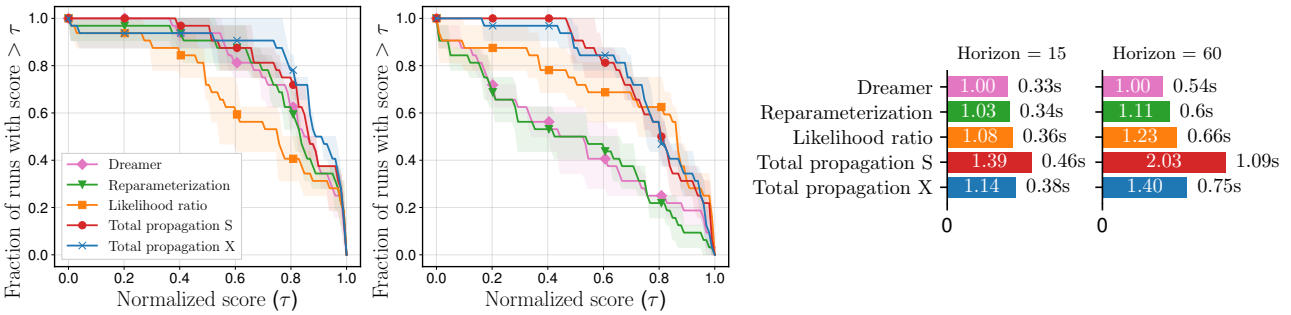
Main results and discussion. The learning curves and performance profiles show that Dreamer performs poorly at the long horizon setting ($H = 60$), LR performs poorly at the short horizon setting ($H = 15$), while TPX performs well in both settings. The poor performance of Dreamer at $H = 60$ is consistent with the results in the original Dreamer paper, and it is caused by the instability in gradient estimation.

The aggregate normalized expected signal-to-noise ratio of the gradient estimators supports that Total Propagation performs well. Total propagation X had the best gradient accuracy metric at all horizons (and it was similar to TP); LR performed poorly at a low H, but well at a high H; RP was the opposite and performed well at a low H, but poorly at a high H; Dreamer followed the pattern of RP, as expected, as it is supposed to be the same.

The ablation study of TPX showed that vectorization of $\boldsymbol{k}$ was important for good performance, while the inclusion of covariance terms did not have a major effect in Dreamer. Nevertheless, as the covariance terms require negligible additional computation, we recommend including them.

The computation times are reasonable: the time per iteration of TPX is only 1.14 times that of Dreamer in the $H = 15$ setting and 1.4 times that of Dreamer in the $H = 60$ setting. We see that TP and TPX perform well and provide better robustness to the hyperparameter selection. These results imply that TP and Proppo are scalable and can be applied to real practical machine learning problems.

[5] Our implementation is a greatly modified version of https://github.com/yusukeurakami/dreamer-pytorch.



(a) $H = 15$ (b) $H = 60$ (c) Normalized computation time per iteration

Figure 6: Performance profiles at episode 1000 in the standard setting (a, b) and computation times (c).

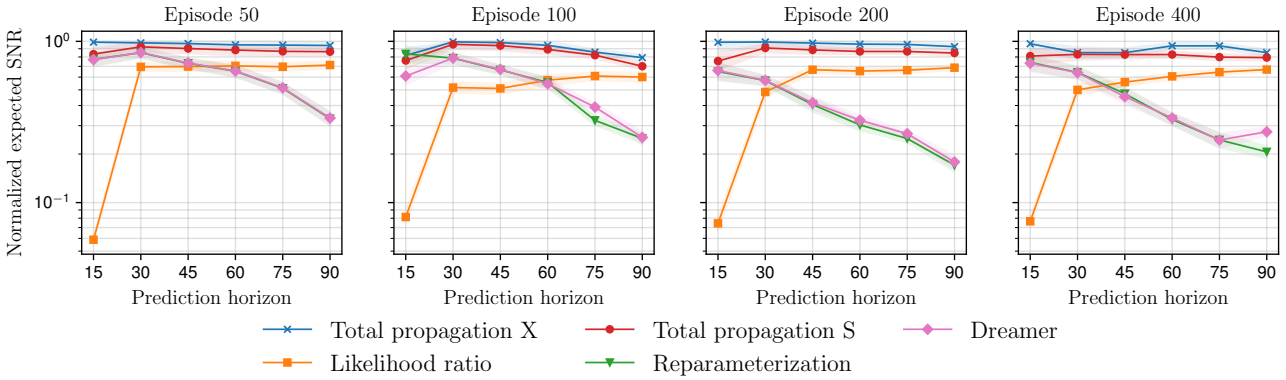


Figure 7: Aggregate normalized expected signal-to-noise ratio of the gradient estimators.

7. Conclusions & Future Work

Neither LR gradients nor RP gradients are satisfactory gradient estimators on their own—LR gradients are inefficient, while RP gradients are unreliable. To achieve reliable and efficient performance across a wide range of tasks and model architectures, it is necessary to examine other gradient estimation techniques. We studied *composite* policy gradient estimators that combine RP and LR, such as TP.

The major bottleneck to the adoption of composite policy gradient estimators was the difficulty of implementing them scalably. Our proposal, Total Propagation X, is easier to implement than previous methods, computes faster, and provides reliable performance. We combined our algorithm with Dreamer, demonstrating that it can be scaled to run at the current frontier of RL research.

There remains future work. For example, other weight estimation schemes that take into account of possible discontinuities in the objective function could be used. Moreover, the LR estimator only used a mean baseline because of Dreamer's implementation peculiarities, but it would be better to use a critic baseline. These future composite policy gradient estimators can also be scaled up by following the recipe provided by TPX.

Author Contributions

Paavo Parmas invented TPX, wrote most of the code and debugged it, ran all of the experiments, did all of the analysis, and wrote the paper. Takuma Seno worked part-time to help write the code for the PIPPS implementation. Yuma Aoki wrote part-time to help write the code for Dreamer.

Acknowledgements

The project was started with funding provided by the Proof of Concept Program at the Okinawa Institute of Science and Technology (OIST). TS and YA were hired to work part-time under the PoC funding by OIST during their contribution to the project. PP finished the project while hired under the Cyborg-AI project supported by NEDO, Japan. PP was supported by JSPS KAKENHI Grant Number JP22H04998. PP would like to thank the Neural Computation Unit at OIST for administrative support while the project was being conducted at OIST.

References

Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A. C., and Bellemare, M. Deep reinforcement learning at the

edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 34:29304–29320, 2021. 6.2

Candela, J. Q., Girard, A., Larsen, J., and Rasmussen, C. E. Propagation of uncertainty in bayesian kernel models-application to multiple-step ahead forecasting. In *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03).*, volume 2, pp. II–701. IEEE, 2003. B.1

Cho, K., van Merriënboer, B., Gulçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014. 6.1

Dangel, F., Kunstner, F., and Hennig, P. Backpack: Packing more into backprop. In *International Conference on Learning Representations*, 2020. 4

Degris, T., White, M., and Sutton, R. Off-policy actor-critic. In *International Conference on Machine Learning*, 2012. C.1

Deisenroth, M. P. and Rasmussen, C. E. PILCO: a model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pp. 465–472, 2011. B.1

Gal, Y., McAllister, R., and Rasmussen, C. Improving PILCO with bayesian neural network dynamics models. In *Workshop on Data-efficient Machine Learning, ICML*, 2016. B.1

Geffner, T. and Domke, J. Using large ensembles of control variates for variational inference. In *Advances in Neural Information Processing Systems*, pp. 9960–9970, 2018. 2

Geffner, T. and Domke, J. A rule for gradient estimator selection, with an application to variational inference. In *International Conference on Artificial Intelligence and Statistics*, pp. 1803–1812. PMLR, 2020. 2

Girard, A., Rasmussen, C., Candela, J. Q., and Murray-Smith, R. Gaussian process priors with uncertain inputs application to multiple-step ahead time series forecasting. *Advances in neural information processing systems*, 15, 2002. B.1

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

Greensmith, E., Bartlett, P. L., and Baxter, J. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov): 1471–1530, 2004. 2

Hafner, D., Lillicrap, T., Ba, J., and Norouzi, M. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2020. 1, 6

Hafner, D., Lillicrap, T. P., Norouzi, M., and Ba, J. Mastering atari with discrete world models. In *International Conference on Learning Representations*, 2021. 6

Hafner, D., Pasukonis, J., Ba, J., and Lillicrap, T. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023. 1, 6

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015. 3

Jordan, M. I. and Rumelhart, D. E. Forward models: Supervised learning with a distal teacher. *Cognitive science*, 16 (3):307–354, 1992. 1

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. B.1

Kingma, D. P. and Welling, M. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013. 6.1

Krishnan, R. G., Shalit, U., and Sontag, D. Deep kalman filters. *arXiv preprint arXiv:1511.05121*, 2015. 6.1

Lee, W., Yu, H., and Yang, H. Reparameterization gradient for non-differentiable models. In *Advances in Neural Information Processing Systems*, pp. 5553–5563, 2018. 2

Madeka, D., Torkkola, K., Eisenach, C., Foster, D., and Luo, A. Deep inventory management. *arXiv preprint arXiv:2210.03137*, 2022. 1

McHutchon, A. *Modelling nonlinear dynamical systems with Gaussian Processes*. PhD thesis, University of Cambridge, 2014. B.1

Metz, L., Maheswaranathan, N., Nixon, J., Freeman, C. D., and Sohl-Dickstein, J. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, 2019. 1, 2

Parmas, P. Total stochastic gradient algorithms and applications in reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 10204–10214, 2018. 3

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

Parmas, P. and Seno, T. Proppo: a message passing framework for customizable and composable learning algorithms. *Advances in Neural Information Processing Systems*, 35:29152–29165, 2022. 1, 2, 3, 4, 5, A, B.1

Parmas, P. and Sugiyama, M. A unified view of likelihood ratio and reparameterization gradients. In *International Conference on Artificial Intelligence and Statistics*, pp. 4078–4086. PMLR, 2021. 2

Parmas, P., Rasmussen, C. E., Peters, J., and Doya, K. PIPPS: Flexible model-based policy search robust to the curse of chaos. In *International Conference on Machine Learning*, pp. 4062–4071, 2018. 1, 2, B.1

Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pp. 1310–1318. PMLR, 2013. 1

Rasmussen, C. E. and Williams, C. K. I. *Gaussian Processes for Machine Learning*. MIT Press, 2006. B.1

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

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

Suh, H. J., Simchowitz, M., Zhang, K., and Tedrake, R. Do differentiable simulators give better policy gradients? In *International Conference on Machine Learning*, pp. 20668–20696. PMLR, 2022. 1, 2

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018. 1, 6.1

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pp. 1057–1063, 2000. 1

Tunyasuvunakool, S., Muldal, A., Doron, Y., Liu, S., Bohez, S., Merel, J., Erez, T., Lillicrap, T., Heess, N., and Tassa, Y. dm_control: Software and tasks for continuous control. *Software Impacts*, 6:100022, 2020. ISSN 2665-9638. doi: https://doi.org/10.1016/j.simpa.2020.100022. URL https://www.sciencedirect.com/science/article/pii/S2665963820300099. 1, 6.2

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

Werbos, P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. 1

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

Wu, P., Escontrela, A., Hafner, D., Goldberg, K., and Abbeel, P. Daydreamer: World models for physical robot learning. *arXiv preprint arXiv:2206.14176*, 2022. 6

Xu, J., Makoviychuk, V., Narang, Y., Ramos, F., Matusik, W., Garg, A., and Macklin, M. Accelerated policy learning with parallel differentiable simulation. In *International Conference on Learning Representations*, 2022. 1

APPENDICES: MODEL-BASED REINFORCEMENT LEARNING WITH SCALABLE COMPOSITE POLICY GRADIENT ESTIMATORS

A. Derivation of Inverse Variance Weighting

Inverse variance weighting (IVW) is a standard technique in statistics and we provide the derivations here for completeness. Given two unbiased random variables X and Y, s.t., $\mathbb{E}[X] = \mathbb{E}[Y]$, IVW constructs a new random variable $Z = kX + (1-k)Y$, and selects k so as to minimize the variance $\mathbb{V}[Z]$. Denote σ_X^2 is the variance, and σ_{XY} is the covariance, then the variance of Z is given by

$$\mathbb{V}[Z] = k^2\sigma_X^2 + (1-k)^2\sigma_Y^2 + 2k(1-k)\sigma_{XY}. \tag{9}$$

This is a quadratic equation in k, and the minimizer is

$$k^* = \frac{\sigma_Y^2 - \sigma_{XY}}{\sigma_X^2 + \sigma_X^2 - 2\sigma_{XY}}. \tag{10}$$

When X and Y are statistically independent, the covariance is $\sigma_{XY} = 0$ and we retrieve the rule used in the original TP Eq. (5). Assuming indepedent random variables ($\sigma_{XY} = 0$), and plugging k^* into Eq. (9) gives the variance

$$\sigma_Z^2 = \frac{\sigma_X^2\sigma_Y^2}{\sigma_X^2 + \sigma_Y^2}, \tag{11}$$

which can also be written using the accuracies (defined as inverse of variance):

$$\frac{1}{\sigma_Z^2} = \frac{1}{\sigma_X^2} + \frac{1}{\sigma_Y^2}, \tag{12}$$

i.e., the accuracies are summed. This allows deriving a simple bound

$$\frac{1}{\sigma_Z^2} \leq 2\max\left(\frac{1}{\sigma_X^2}, \frac{1}{\sigma_Y^2}\right), \tag{13}$$

i.e., the accuracy is less than or equal to twice the higher of the two accuracies. In our previous work (Parmas & Seno, 2022), we showed that TP is able to overcome this bound and can even give 100 times improvement in the accuracy. This was possible thanks to taking advantage of the graph structure of the computations.

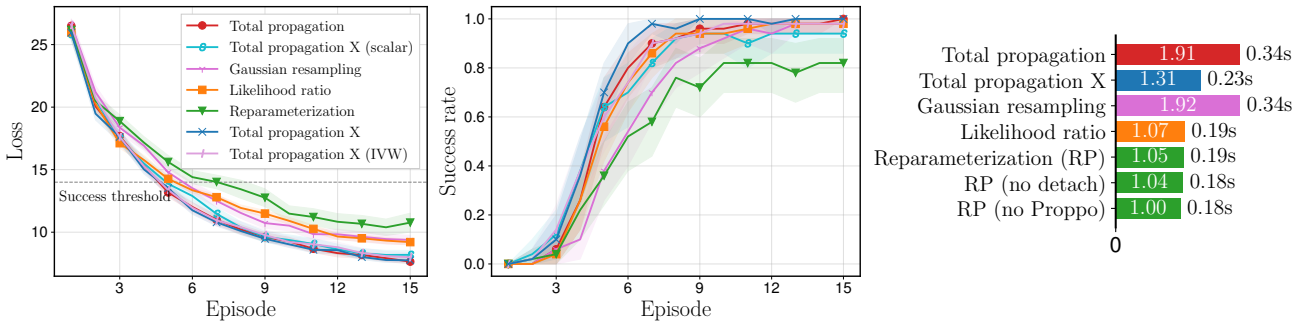
Example with a quadratic function. To give a concrete example of the effect of the covariance terms, we work through an example with a quadratic function $R(x) = \frac{x^2}{2}$, and a Gaussian distribution $x \sim p(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$. This is the same example as the one in Appendix D.2. of (Rezende et al., 2014), except that we consider the covariance and composite gradients. Moreover, we consider $\sigma = 1$ and $\mu = 0$ for simplicity. Then, the gradient estimators for $\frac{\mathrm{d}(\cdot)}{\mathrm{d}\mu}$ are

$$\begin{aligned}\hat{g}_{\text{LR}} &= \frac{x-\mu}{\sigma^2}\left(\frac{x^2}{2} - b\right) = x\left(\frac{x^2}{2} - b\right),\\ \hat{g}_{\text{RP}} &= x.\end{aligned} \tag{14}$$

For LR, we use the mean baseline $b = \mathbb{E}\left[\frac{x^2}{2}\right] = \frac{\sigma^2}{2} = \frac{1}{2}$, and we can compute the variances and covariances:

$$\begin{aligned}\mathbb{V}[\hat{g}_{\text{LR}}] &= \frac{1}{4\sigma^4}\mathbb{E}\left[x^6 - 2x^4\sigma^2 + x^2\sigma^4\right] = \frac{5}{2}\sigma^2,\\ \mathbb{V}[\hat{g}_{\text{RP}}] &= \mathbb{E}\left[x^2\right] = \sigma^2,\\ \operatorname{cov}[\hat{g}_{\text{LR}}, \hat{g}_{\text{RP}}] &= \frac{1}{2\sigma^2}\mathbb{E}\left[x^4 - x^2\sigma^2\right] = \sigma^2.\end{aligned} \tag{15}$$

Now if we use the optimal Eq. (10) taking into account for the covariance, we obtain $k^* = 0$, putting all of the weight on the RP estimator, whereas if we use Eq. (5) ignoring the covariances, we obtain $k_{\text{IVW}} = \frac{1}{3.5}$. Using k^* gives the variance $\mathbb{V}[\hat{g}] = 1$, whereas plugging k_{IVW} into Eq. (9) gives the variance $\mathbb{V}[\hat{g}] \approx 1.12$, which is suboptimal. We have also confirmed these results computationally, and found that TPX finds the optimal solution, whereas IVW, ignoring the covariance terms, is suboptimal. This was a bit of an unfortunate example, as it turns out that for a quadratic, it is optimal to ignore the LR gradient, and use only the RP gradient. Nevertheless, it illustrates that ignoring the covariance terms will in general lead to a suboptimal solution, although empirically we have found that the difference tends to be small.



(a) Learning curves (b) Success rate (c) Normalized computation time

Figure 8: Replication of PIPPS on the cart-pole task, using the Proppo framework. The computation time in (c) was computed per one trajectory simulation and gradient estimator computation on an NVIDIA GTX 1650 Ti GPU.

B. PIPPS: Model-based RL with Gaussian Processes on Cart-pole

B.1. Experimental Details

To demonstrate the correctness of our implementations of TP and TPX, we reproduce the experimental results of PIPPS (Parmas et al., 2018), the first and only article to use the TP algorithm in RL until now. Moreover, we evaluate the different design decisions in TPX, and test the robustness of our algorithms to hyperparameters.

PIPPS is a model-based RL algorithm based on Gaussian process (Rasmussen & Williams, 2006) dynamics models similar to PILCO (Deisenroth & Rasmussen, 2011), but instead of the moment matching (MM) Gaussian approximations of the trajectory distribution (Girard et al., 2002; Candela et al., 2003) it uses particle sampling (McHutchon, 2014), and instead of exact differentiation of the trajectory approximation to obtain the policy gradient, it uses sampling-based stochastic gradient estimators, such as TP, RP or LR. PILCO was one of the early successful MBRL methods that took advantage of differentiating the learned dynamics model; however, the algorithm had limited scalability due to the high computational cost and inflexibility of the MM approximations. PIPPS overcame these conceptual limitations, yet the original work did not apply the TP algorithm on high-dimensional tasks. Therefore, here we first replicate the previous results on the cart-pole task, then in Sec. 6 we apply TP and TPX to high-dimensional visual MBRL tasks, and show that these algorithms scale.

Task. The cart-pole swingup and balancing task is a common benchmark in control. The system consists of a cart with an attached hinged pole. The cart can be moved back and forth by applying a horizontal force, and the task is to swing up the pole and balance it. The system state, $\boldsymbol{x}$, is four dimensional—the cart position and pole angle, and their velocities. Moreover, there is observation noise on the state, matching the setup in the PIPPS paper with noise multiplier 1. The control signal is a scalar, u, output by a radial basis function network, $u = \pi_\theta(\boldsymbol{x})$, with ~ 250 parameters.

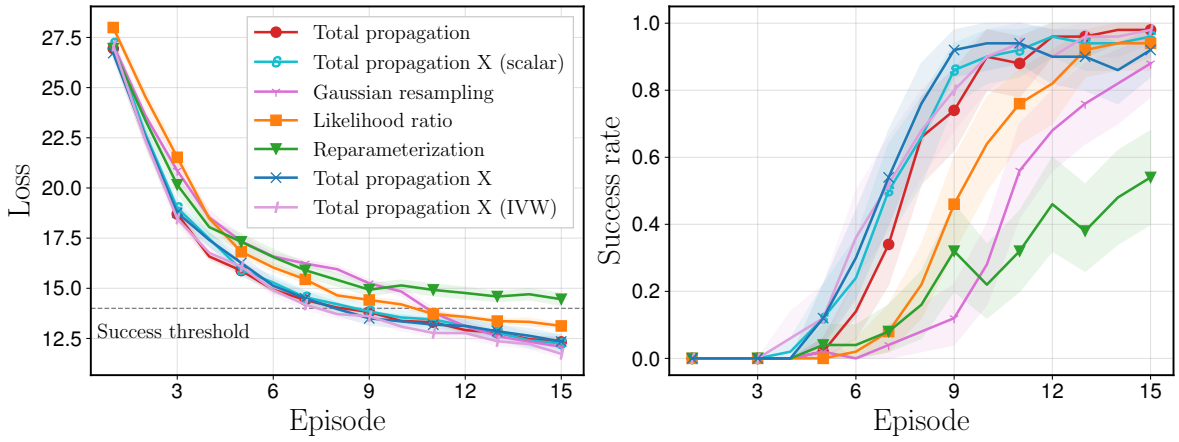
Methods. We compare the MC gradient estimators: RP, LR, TP and TPX. We do not employ the *batch importance weighting* (BIW) for LR introduced in the PIPPS paper, but instead use the standard LR method for simplicity. We also compare with Gaussian resampling (GR) a method that fits a Gaussian distribution on the particles at each time step, and resamples the particles from this distribution. The aim of GR was to replicate MM but with a more flexible sampling based approach. Its use was proposed by McHutchon (2014), and it was later used in a Deep PILCO implementation (Gal et al., 2016). Moreover, we compare with other variants of TPX: the *scalar* version adds a summation into the $\hat{k}$ estimator, and the IVW version omits the covariance terms in Eq. (8).

Setup. The setup mimics the original experiments in the PIPPS article, with slight changes to utilize PyTorch. In the PIPPS algorithm, the system alternates between applying the policy π on the real system for one episode, fitting a Gaussian process model on all of the data observed until that point, training the policy in simulations with the learned GP model, and repeating the process by applying the trained policy on the real system again. The process starts by collecting 1 episode with random actions, then learning to control the system for 15 episode and policy optimization iterations. Total propagation or other MC gradient estimation techniques are used in the policy optimization phase of this process. We use Adam (Kingma & Ba, 2014) as the optimizer, instead of the custom RMSprop-like optimizer in the original PIPPS article. We tested the learning rates in $\eta \in \{0.002, 0.005, 0.01, 0.03, 0.06, 0.12\}$, and plot results for all methods for $\eta = 0.01$, which was the optimal rate for RP. The optimal rates for TP were either $\eta \in \{0.03, 0.06\}$. The number of particles used for trajectory

predictions was $N = 300$, and the number of simulations and policy gradient steps per episode was 250. The prediction horizon was 30 steps. We used the Tip Cost described in the original work, and the loss to be optimized is the sum of the cost over the trajectory. We ran 50 experiments per algorithm (in parallel on a CPU cluster), and show the mean performance (error bar is one standard error) and the success rate (error bars are 95% confidence intervals via bootstrapping).

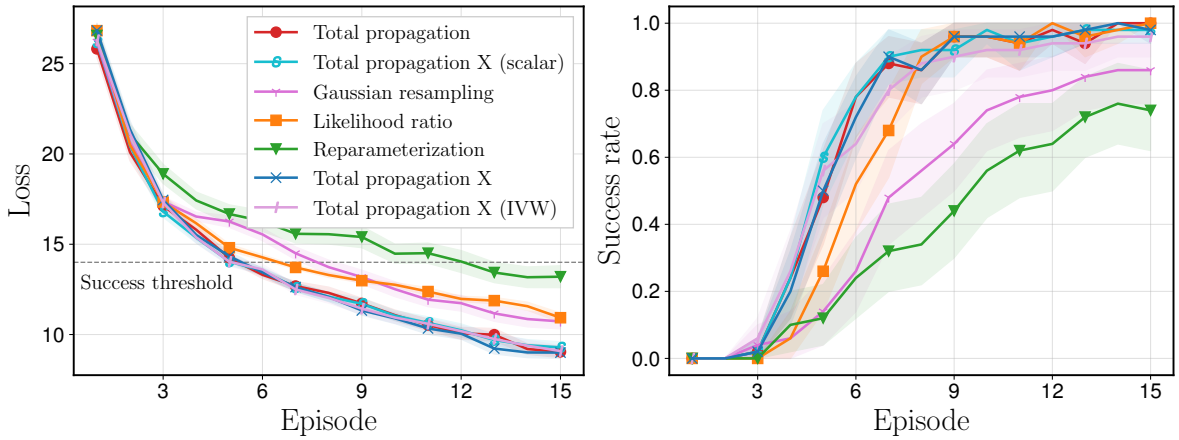
Results. The results in Fig. 8 follow the same pattern as in the original work (Parmas et al., 2018), but with slightly better performance. Results for other learning rates are in App. B.2. The main results are that TP and TPX reliably outperform the other methods at all learning rates. Moreover, TPX matches the performance of TP, but with a simpler implementation and faster computation speed. The IVW and scalar variants of TPX also showed similar performance, but the *scalar* variant had a downward trend, particularly for $\eta = 0.12$. We see that the average performance of RP is poor due to many learning runs being unsuccessful. Parmas et al. (2018) attributed this poor performance to instabilities in gradient estimation. The LR gradient based method is robust to this issue and has a higher success rate. The TP based methods combine the best of RP and LR—they take advantage of the efficiency of RP while having the robustness of LR, ultimately showing a high success rate and faster learning than LR. In Fig. 8c we additionally show the RP *no detach* and *no Proppo* variants. The *no Proppo* variant is an RP implementation without relying on our custom ML framework (Parmas & Seno, 2022), while the *no detach* variant merely uses our framework as an interface, while using the typical RP internal implementation. These computation time results showed that the overhead from using our ML framework is minuscule. We note that unlike the original implementation of TP by (Parmas et al., 2018), our implementation can use PyTorch and run on a GPU leading to much faster computations. Training completes in minutes on a GPU. These results confirm that our implementation is a good reproduction of the original TP.

B.2. Full Experimental Results



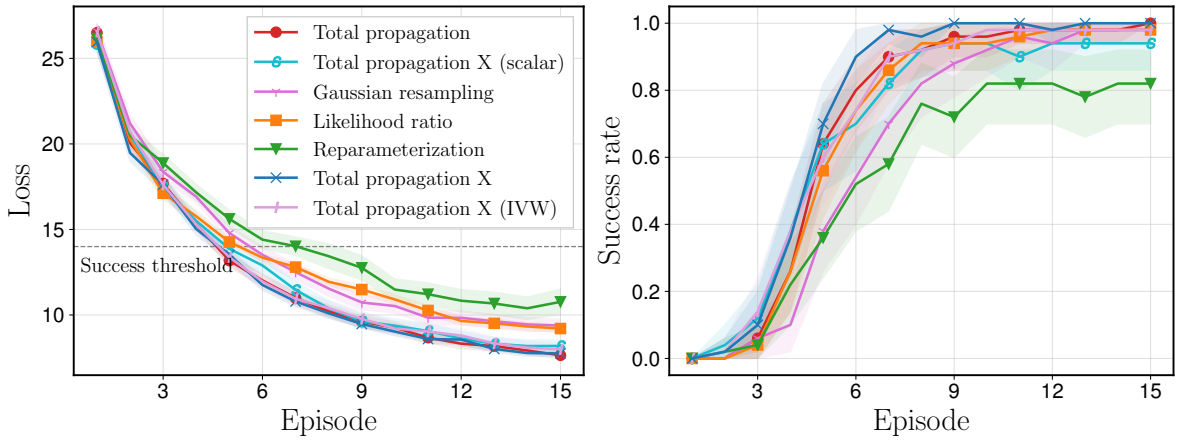
(a) Learning curves (b) Success rate

Figure 9: PIPPS cart-pole results, Learning rate: **0.002.**



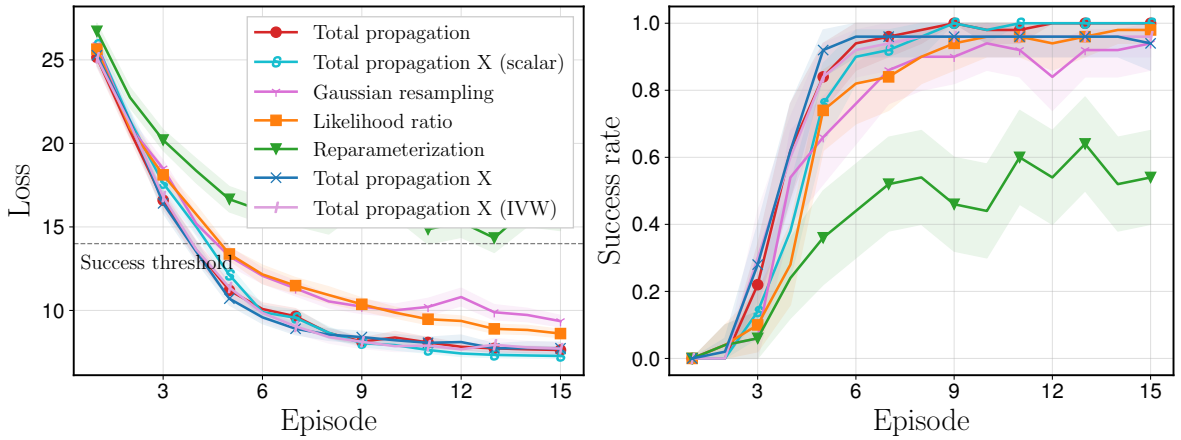
(a) Learning curves (b) Success rate

Figure 10: PIPPS cart-pole results, Learning rate: **0.005.**



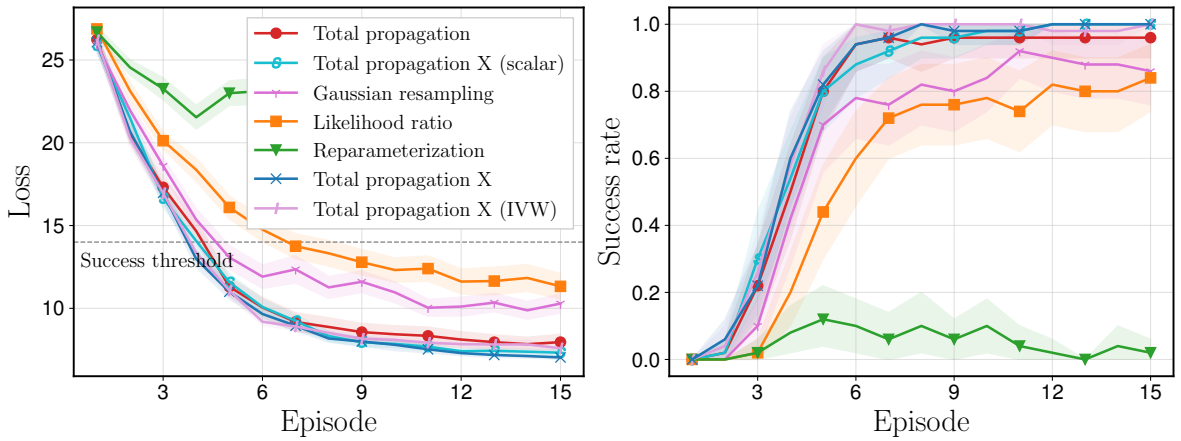
(a) Learning curves (b) Success rate

Figure 11: PIPPS cart-pole results, Learning rate: **0.01.**



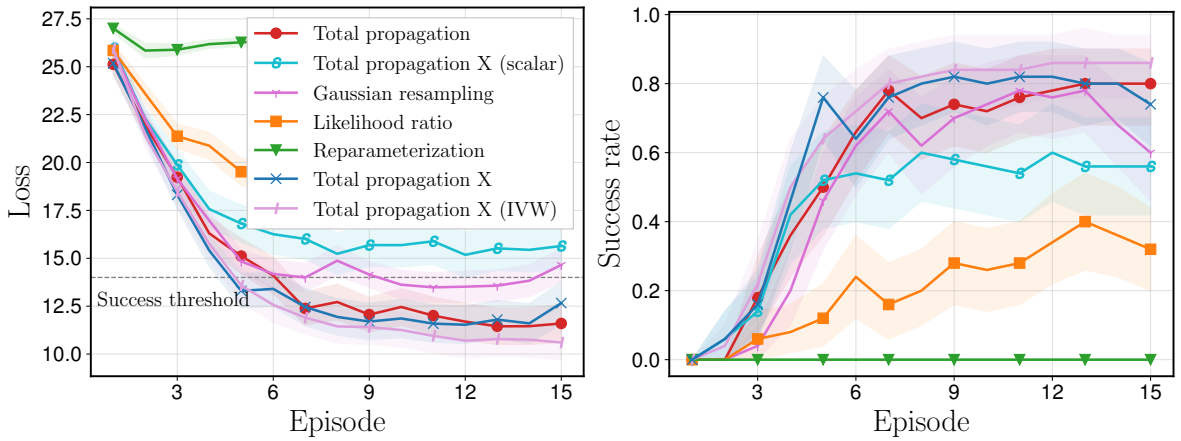
(a) Learning curves (b) Success rate

Figure 12: PIPPS cart-pole results, Learning rate: **0.03.**



(a) Learning curves (b) Success rate

Figure 13: PIPPS cart-pole results, Learning rate: **0.06.**



(a) Learning curves (b) Success rate

Figure 14: PIPPS cart-pole results, Learning rate: **0.12.**

C. Additional Dreamer Details and Experiments

C.1. Implementation Details in Dreamer Leading to a Bias

We will explain two peculiarities in the Dreamer implementation that require more attention when discussing any potential combination with Proppo: the λ-return value weighting, and blocked gradients at the actions (see Fig. 5).

Dreamer value weighting. Dreamer uses a λ-return both to construct the targets for its value critic, as well as to create a loss for its policy. Given a sequence of rewards r_t and values V_t, the λ-return, V^λ, can be recursively defined by

$$V_H^\lambda = V_H, \quad V_t^\lambda = r_t + (1-\lambda)\gamma V_{t+1} + \lambda\gamma V_{t+1}^\lambda, \tag{16}$$

where γ is the discount factor. This definition is fine for the value targets; however, for the policy loss, all of the returns are simply averaged, $\frac{1}{H}\sum_{t=0}^{H-1} V_t^\lambda$, leading to a peculiar implementation. It would be natural to backpropagate the gradient of V_t^λ from the state $\boldsymbol{x}_t$; however, this is not how the original Dreamer is implemented—the gradients will start backpropagating separately from each V_t and r_t, and the λ-weighting merely serves to provide a weight for each value function. It is unclear whether this is a good design choice—there may be some justification to it based on its similarity with the Off-PAC objective (Degris et al., 2012). Our aim is not to examine the effects of this choice, but if we want to accurately replicate Dreamer using Proppo, we need to compute the weight assigned to V_t and r_t. A simple calculation leads to

$$\begin{aligned} w_V^H = \frac{1-(\lambda\gamma)^H}{H(1-\lambda\gamma)}\gamma, \quad w_V^t = \frac{1-(\lambda\gamma)^t}{H(1-\lambda\gamma)}(1-\lambda)\gamma \\ w_r^t = \frac{1-(\lambda\gamma)^{t+1}}{H(1-\lambda\gamma)}, \end{aligned} \tag{17}$$

where $w_V^H V_H + \sum_{t=0}^{H-1}\left(w_r^t r_t + w_V^t V_t\right) = \frac{1}{H}\sum_{t=0}^{H-1} V_t^\lambda$. We call $R_t = w_V^H V_H + \sum_{h=t}^{H-1}\left(w_r^h r_h + w_V^h V_h\right)$ the Dreamer-return, and this is the return that has to be used for LR for it to estimate the same gradient as is estimated in the standard Dreamer implementation. However, there is yet another implementation detail in Dreamer that adds a bias, and prevents it being possible to estimate the same gradient using LR—there is an operation to block the gradients at the input to the action.

Gradient blocking at the actions. The blocked gradient paths are highlighted in Fig. 5 by the crossed out lines leading from the latent state $\boldsymbol{x} = [\boldsymbol{h}, \boldsymbol{z}]$ to the action $\boldsymbol{a}$. It is easy to see that for an unbiased gradient estimate, the gradients should not be blocked. For example, if $\gamma = 1$ and $\lambda = 1$, so that we are estimating an MC return using only the rewards, to obtain the unbiased gradient, the full trajectory should be differentiated without blocking the gradients anywhere. Therefore, blocking the gradients is not justified from the point of view of estimating unbiased gradients. However, we found that allowing the gradients to pass through increased the gradient variance, and made the learning inefficient. Therefore, we believe this bias was added to improve the performance. The main effect of this bias on our experiments is that it becomes impossible to estimate LR gradients with the same expectation as the Dreamer gradient, because the LR estimator does not use the backpropagated gradients in its estimation, and is, thus, not affected by the gradients being blocked. Nevertheless, we evaluate LR and TP by implementing them as close as possible to an equivalent implementation of the original Dreamer.

C.2. Gradient Variance Estimation Experimental Protocol

Experimental details. To evaluate the gradient variance of the different methods, we load checkpointed models trained by Dreamer, and evaluate the gradients many times to compute their variance.

The experiments consist of 50 iterations of 50 gradient evaluations each. In each iteration, we encode a set of image sequences, and randomly select one of the encoded latent states as the initial state. Given a latent state, we perform 50 gradient evaluations, with all samples in each evaluation starting their trajectory from the same encoded latent state $\boldsymbol{x}_\mu$ (moreover, we match this latent state for all of the algorithms that we are comparing). Each gradient evaluation uses 2500 sample trajectories of length H starting from $\boldsymbol{x}_\mu$. In summary, we sample 50 different $\boldsymbol{x}_\mu$ states, and for each state we obtain a set of 50 different gradient estimates $\hat{\boldsymbol{g}}$ giving a total of 2500 gradient estimates (where each estimate was also performed by averaging gradients from a batch of 2500 trajectories).

The total variance can be decomposed using the law of total variance:

$$\mathbb{V}\left[\hat{\boldsymbol{g}}\right] = \mathbb{V}_{\boldsymbol{x}_\mu}\left[\mathbb{E}\left[\hat{\boldsymbol{g}}|\boldsymbol{x}_\mu\right]\right] + \mathbb{E}_{\boldsymbol{x}_\mu}\left[\mathbb{V}\left[\hat{\boldsymbol{g}}|\boldsymbol{x}_\mu\right]\right]. \tag{18}$$

The first of these terms is the variance of the expected gradient (which is irreducible independent of the gradient estimator), and the second term is the expected gradient variance (this term can be reduced by improving the gradient estimator). The

accuracy is defined as the inverse of the variance $\frac{1}{\mathbb{V}[\cdot]}$. We evaluate our algorithms both based on the expected accuracy as well as based on the total accuracy. We argue that the expected accuracy may be a more important metric, as the total accuracy includes the variance of the expectation, which is irreducible.

Normalized expected signal-to-noise ratio. In our experiments, simply looking at the variance (or accuracy/inverse-variance) of the gradient estimator is not suitable, because the Dreamer algorithm includes a bias in its gradient estimation (see discussion of Dreamer value weighting and gradient blocking). To overcome this issue, a natural way might be to compute the true gradient exactly, and compare the error of the estimator to the true gradient. However, this is computationally infeasible. Moreover, it is not clear that we should be comparing the distance to the expected exact gradient as opposed to the expected biased gradient. To tackle such concerns, we propose an expected signal-to-noise ratio metric for comparing gradient estimators, defined as

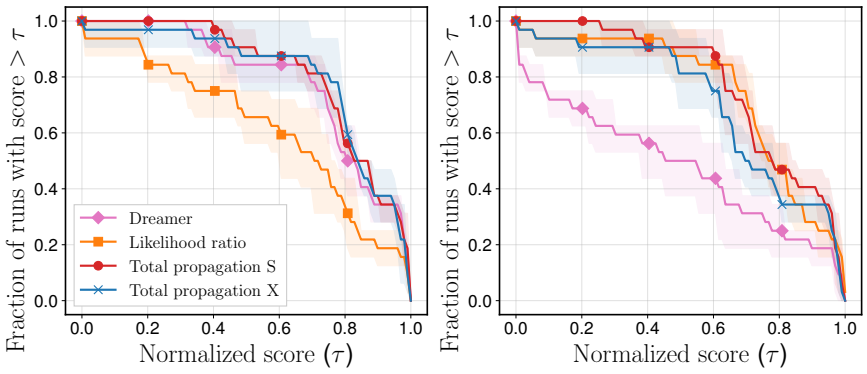
$$ESNR = \mathbb{E}_{\boldsymbol{x}_\mu}\left[\frac{\sum \mathbb{E}\left[\hat{\boldsymbol{g}}|\boldsymbol{x}_\mu\right]^2}{\sum \mathbb{V}\left[\hat{\boldsymbol{g}}|\boldsymbol{x}_\mu\right]}\right]. \tag{19}$$

This metric will compute the expected gradient squared for each $\boldsymbol{x}_\mu$, and take its ratio to the local variance given $\boldsymbol{x}_\mu$. It is thus a signal-to-noise ratio of the gradient estimates starting from $\boldsymbol{x}_\mu$. We take the expectation of this ratio across the different $\boldsymbol{x}_\mu$ to compare the different algorithms.

We note that this metric also has its limitations. Consider that a gradient with an arbitrary bias can have a large ESNR as long as its expected variance is low. Therefore, for biased gradient estimators, it is not possible to conclude that one gradient estimator is better than the other based on this metric. Nevertheless, we believe that for similar gradient estimators, it provides an indication of which gradient estimator is better, and can provide corroborative evidence in support of one method performing better than the other.

Finally, as we are comparing our algorithms across 8 different MuJoCo environments, we propose to aggregate the results to obtain an overall metric. Specifically, we normalize the ESNRs in each environment based on the maximum of the different methods, and average the results across all environments.

C.3. Additional Experimental Results



(a) $H = 15$ (b) $H = 60$

Figure 15: Performance profiles at episode 400 in the pre-trained setting.

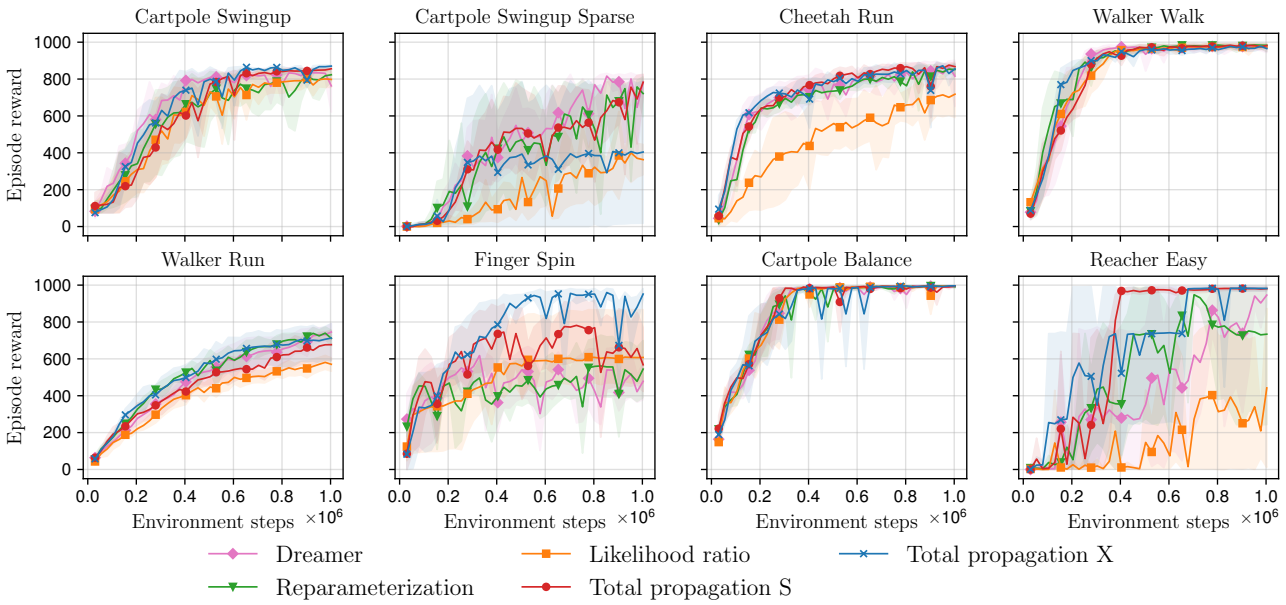


Figure 16: All experimental results in the standard setting with $H = 15$.

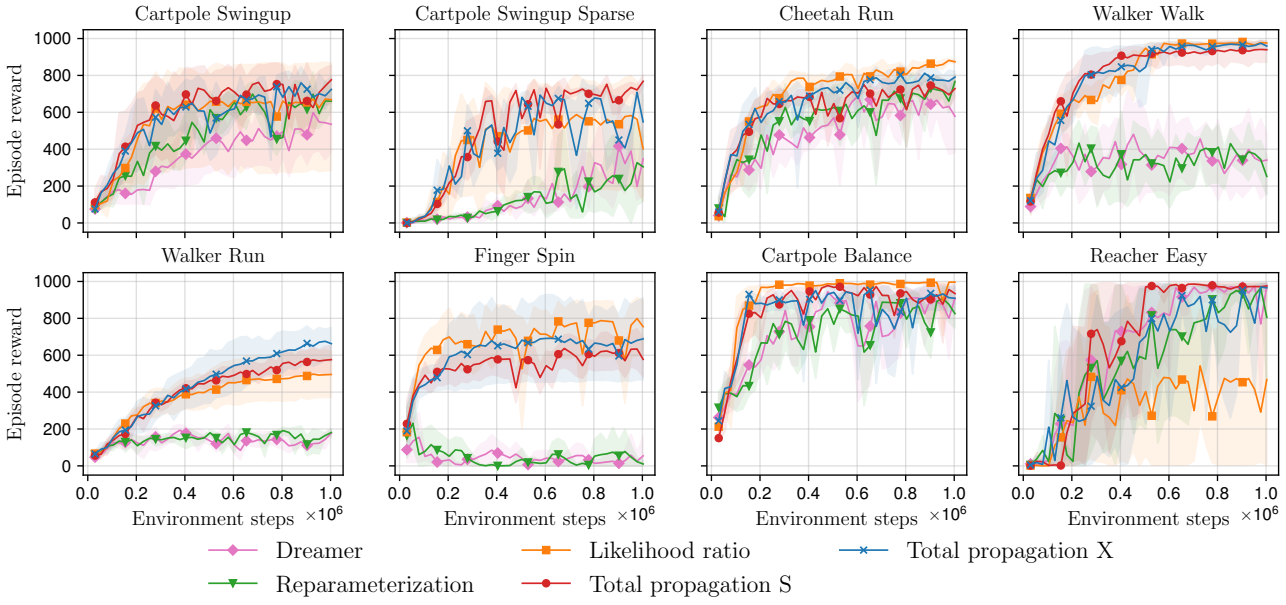


Figure 17: All experimental results in the standard setting with $H = 60$.

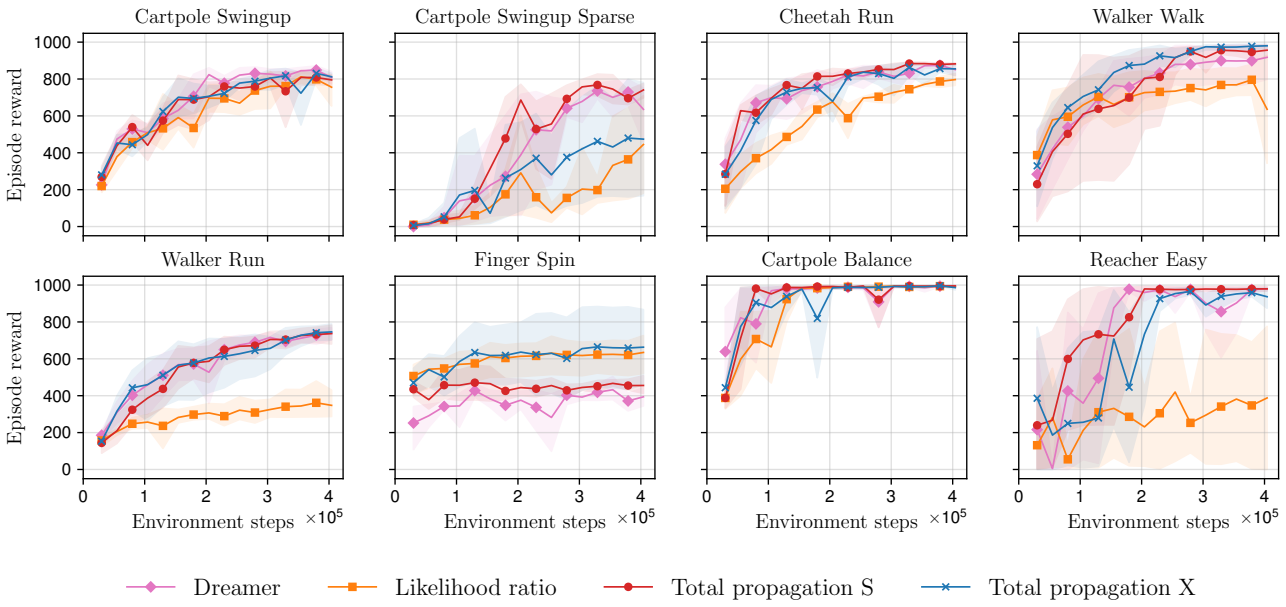


Figure 18: All experimental results in the pre-trained setting with $H = 15$.

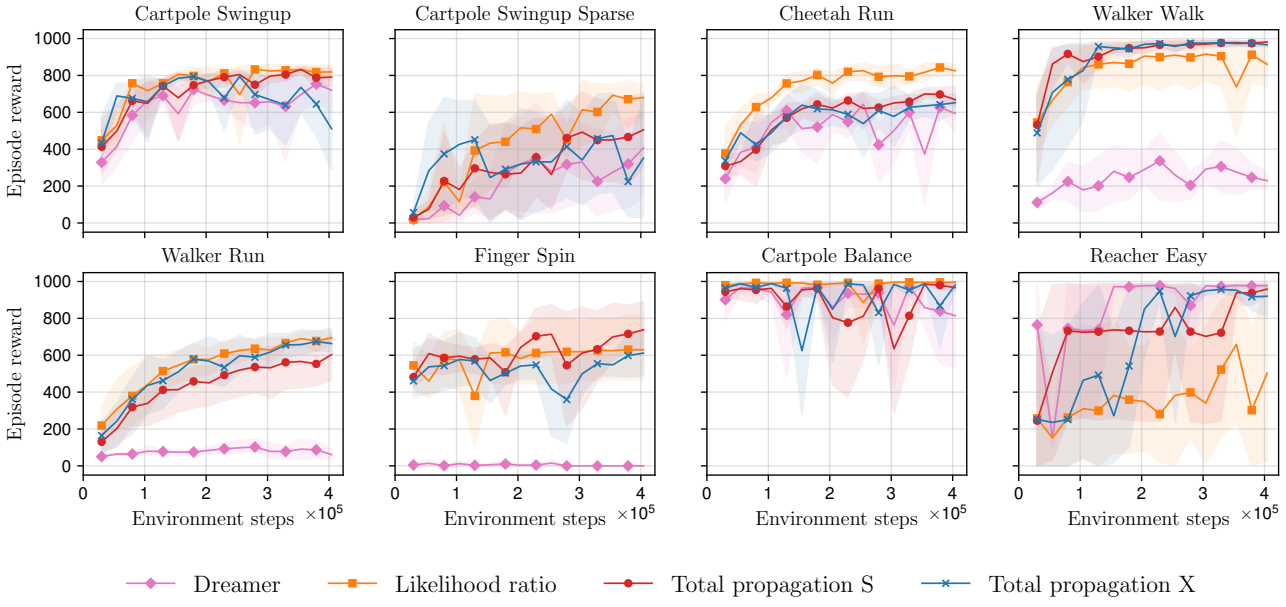


Figure 19: All experimental results in the pre-trained setting with $H = 60$.

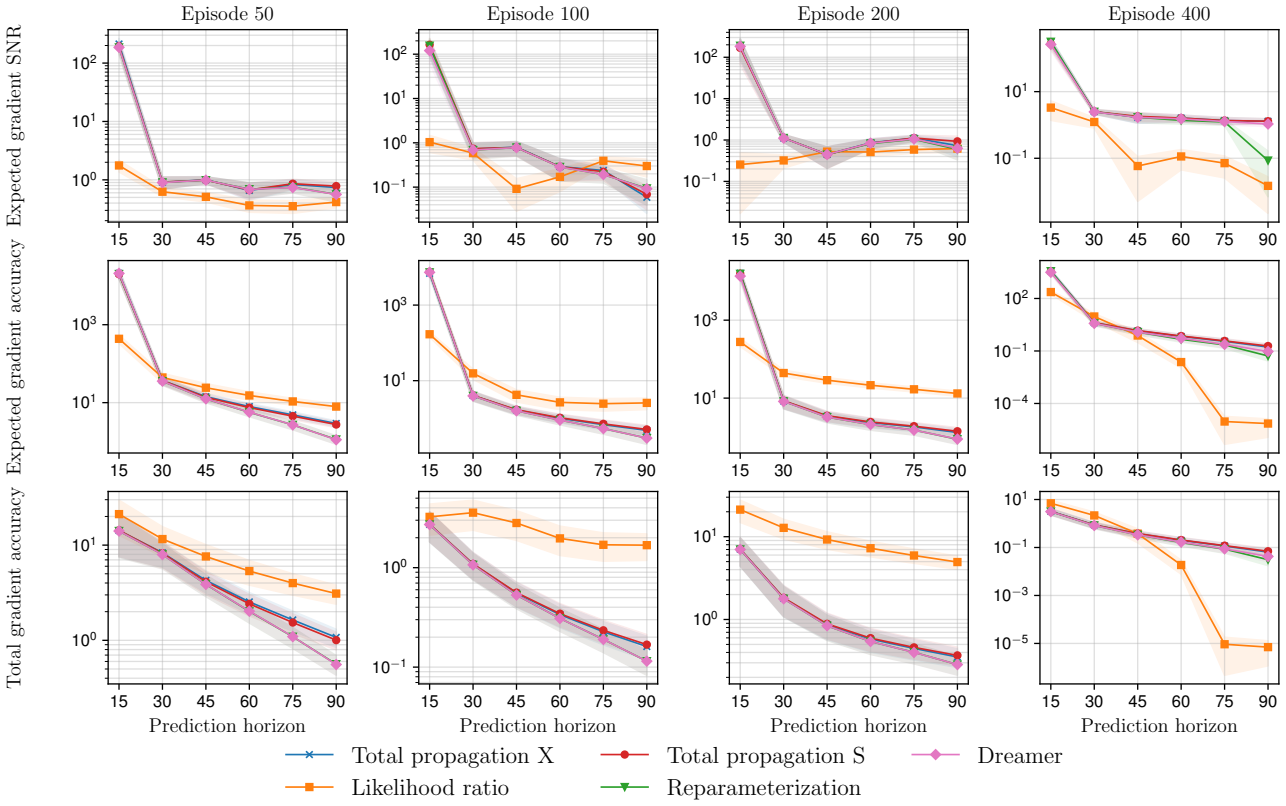


Figure 20: Reacher Easy, raw gradient data

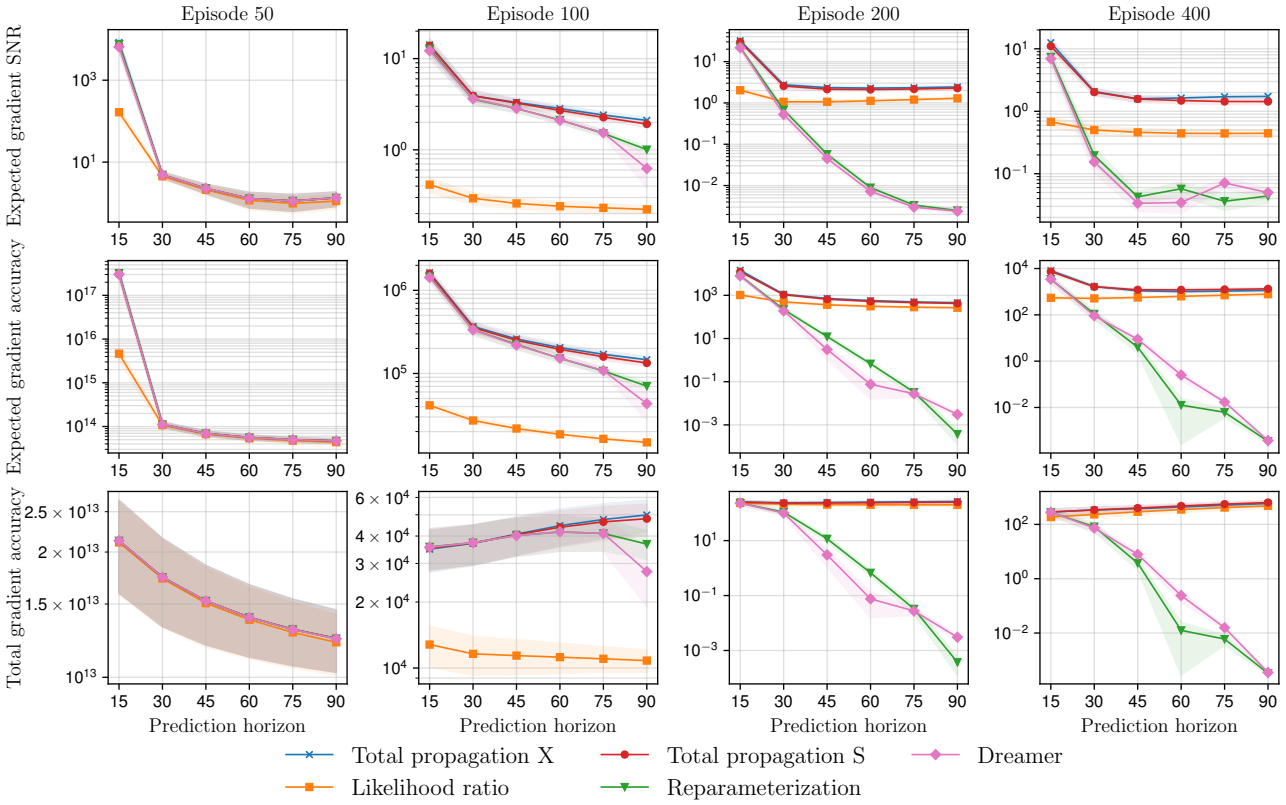


Figure 21: Walker Walk, raw gradient data

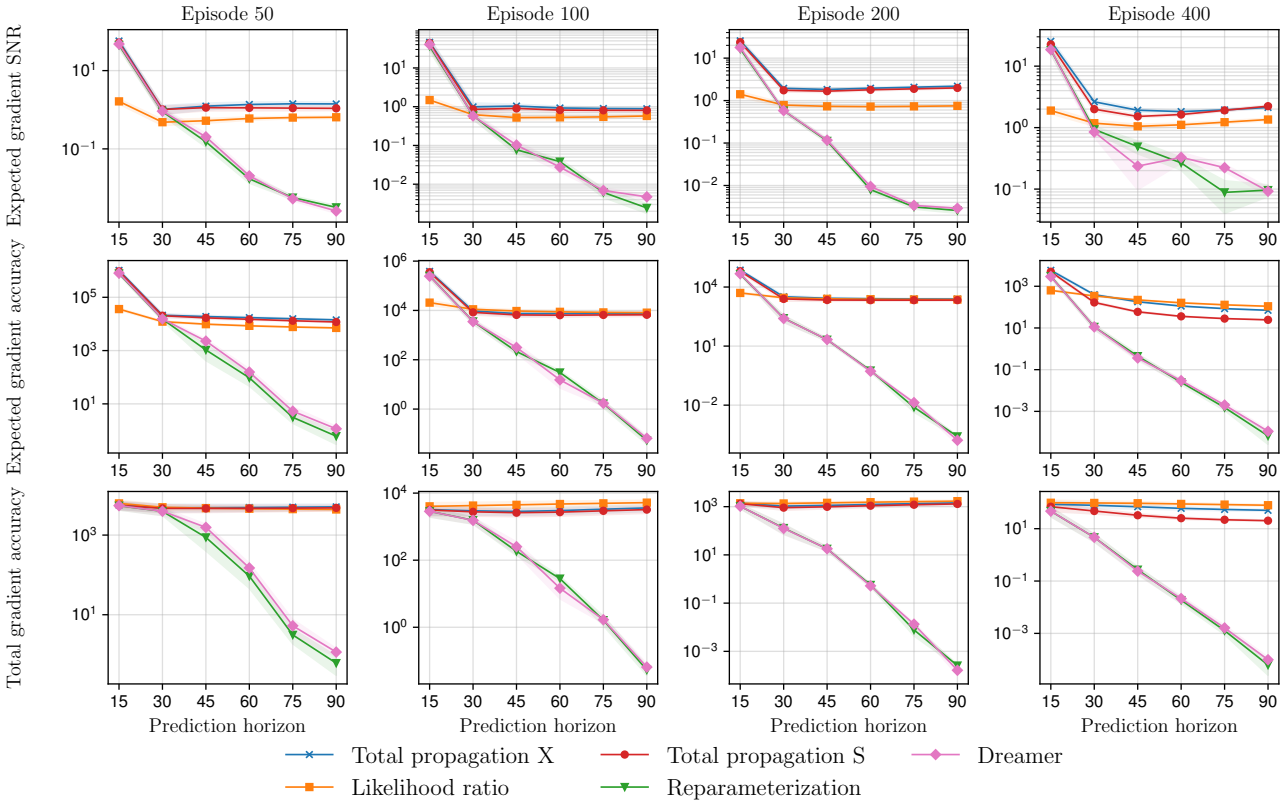


Figure 22: Walker Run, raw gradient data

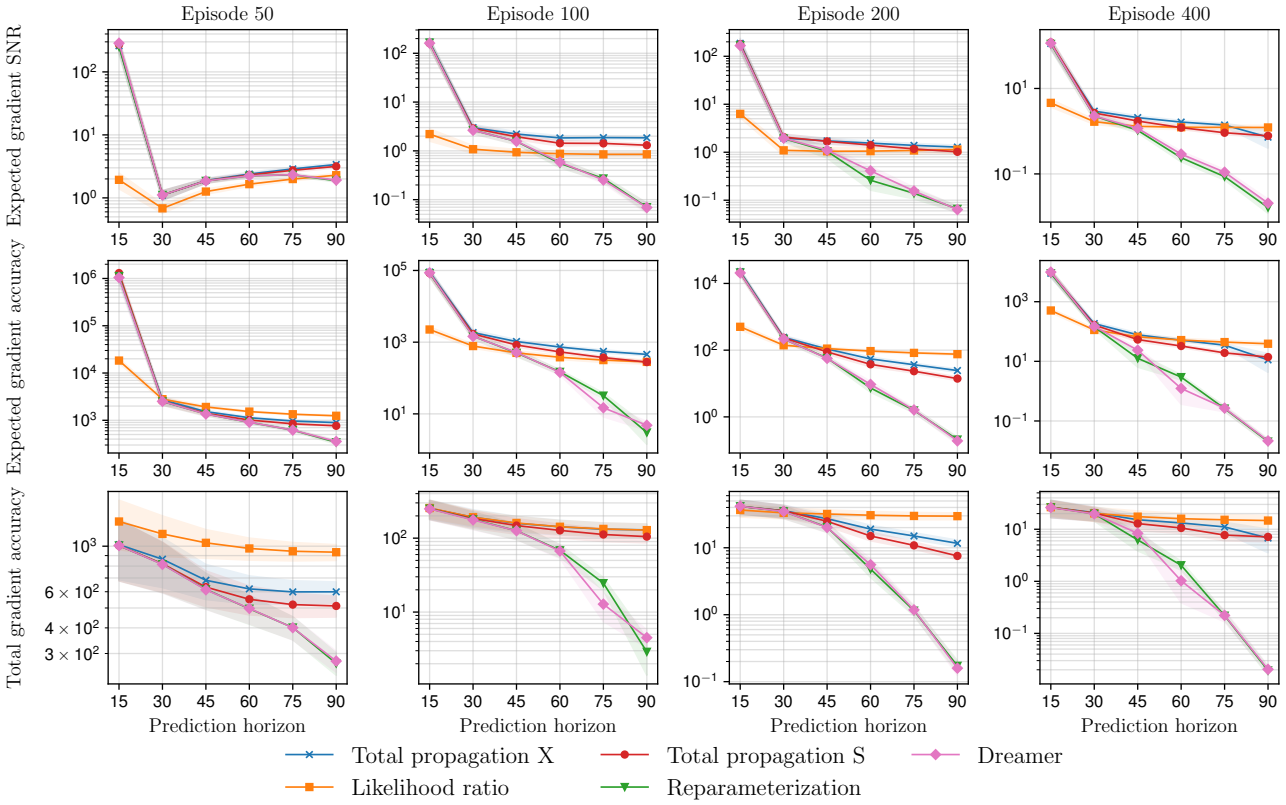


Figure 23: Cheetah Run, raw gradient data

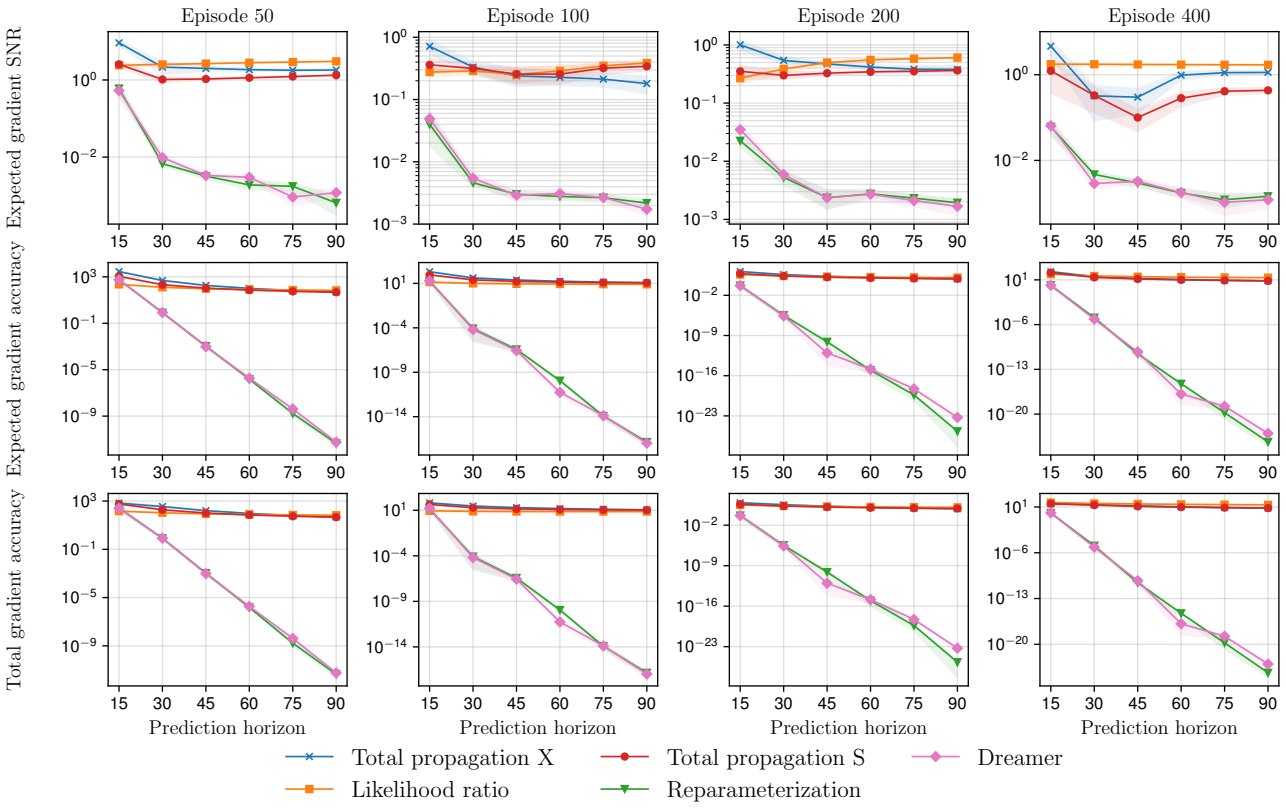


Figure 24: Finger Spin, raw gradient data

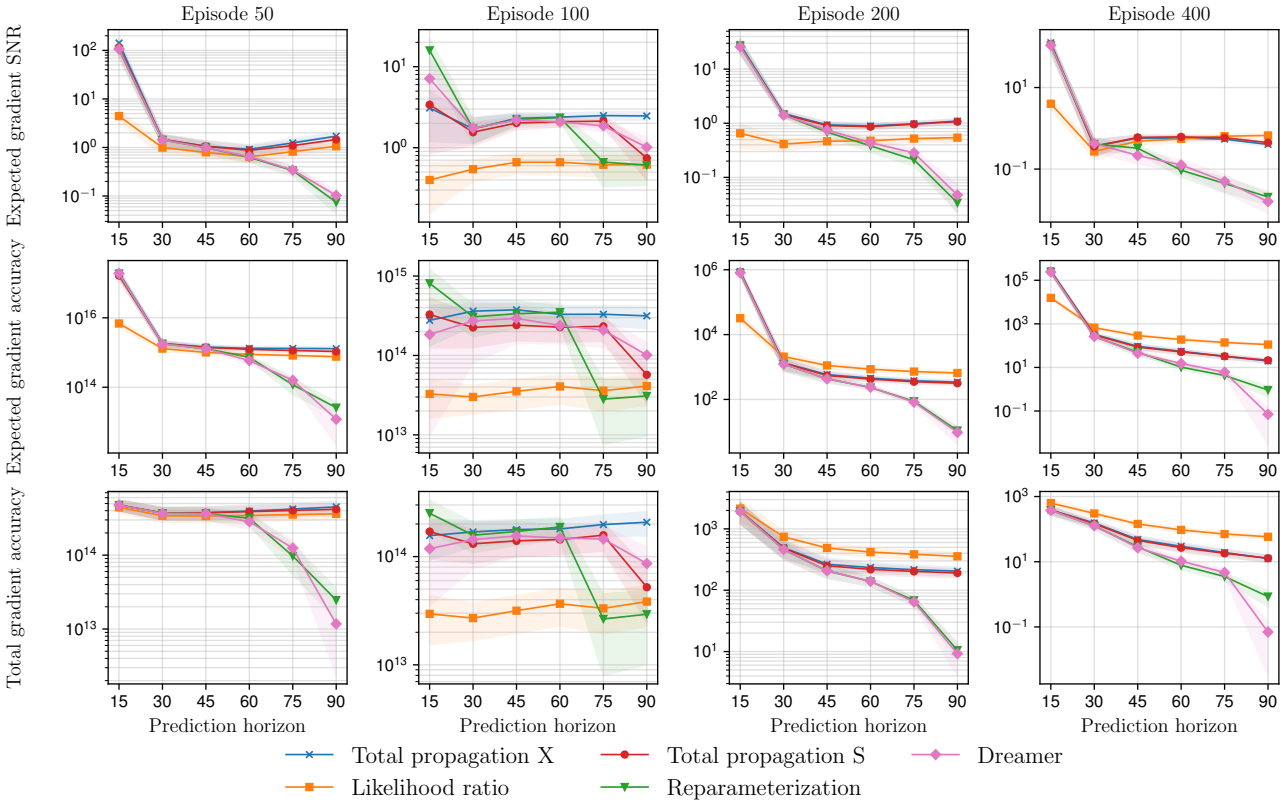


Figure 25: Cartpole Swingup, raw gradient data

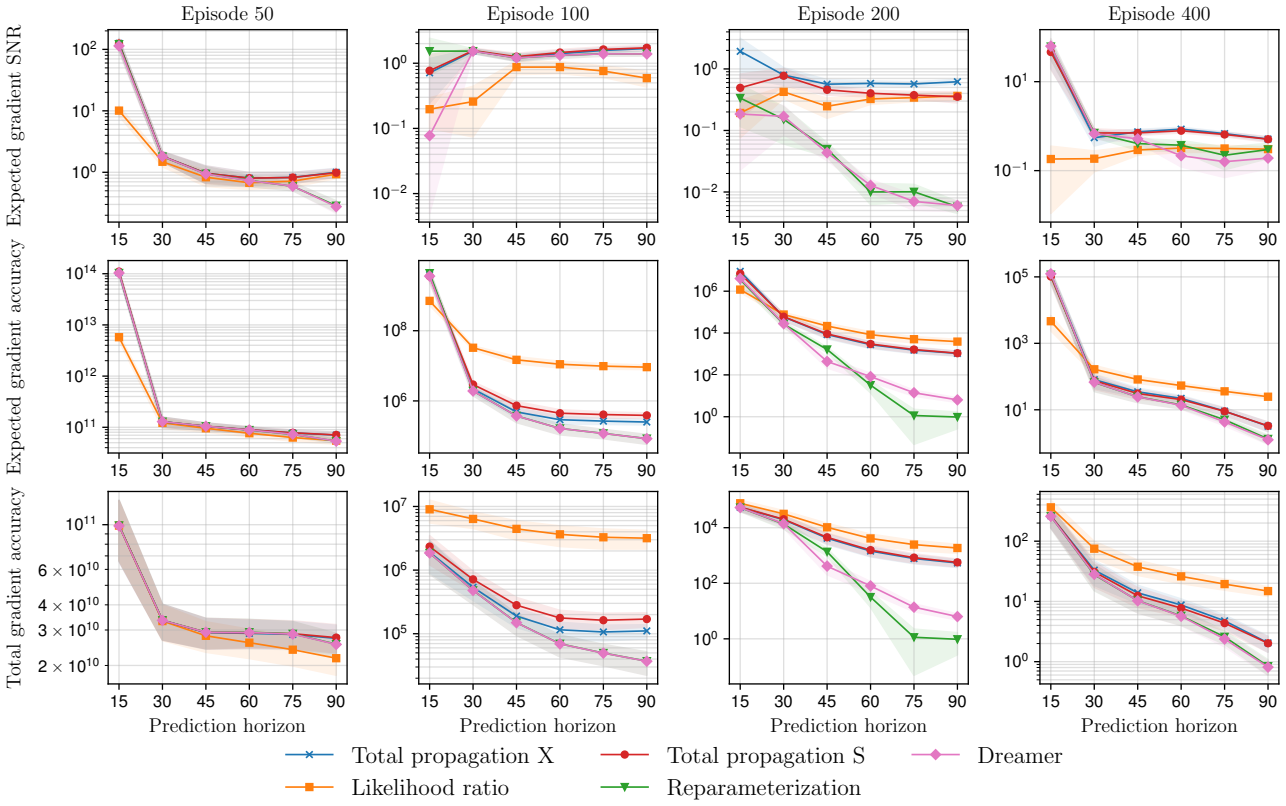


Figure 26: Cartpole Swingup Sparse, raw gradient data

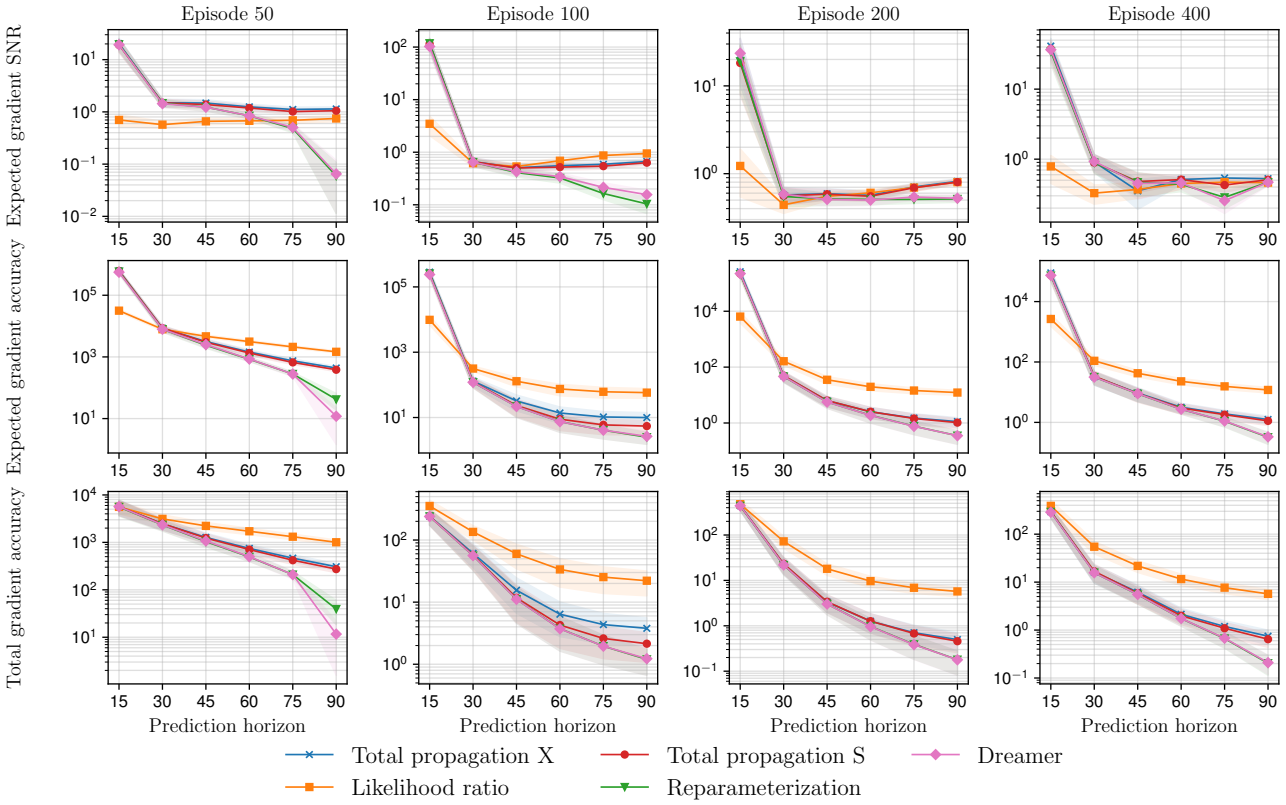


Figure 27: Cartpole Balance, raw gradient data

D. Dreamer Total Propagation X Ablation Experiments

In this section, we include ablation experiments on the different components of TPX. We compare against TPX with a scalar k (TPX scalar), TPX that omits the covariance terms and only includes the IVW terms (TPX IVW), and TPX with a scalar k that omits the covariance terms (TPX IVW scalar). We perform experiments to test the gradient accuracy according to the protocol in App. C.2. The aggregated results are in Fig. 28, and they show that in Dreamer, the covariance terms had little effect, while using a tensor-valued $\boldsymbol{k}$ was important to improve the performance.

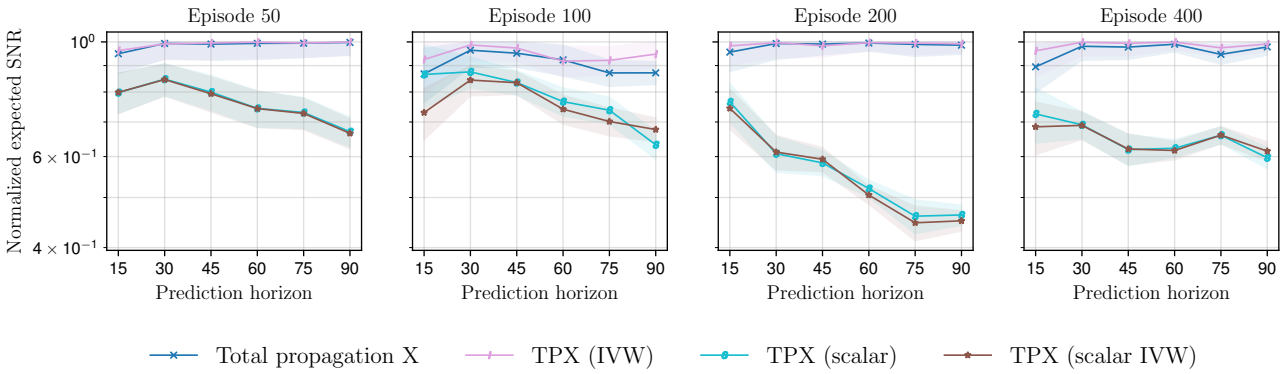


Figure 28: TPX ablation. Aggregate normalized expected SNR of the gradient estimators.

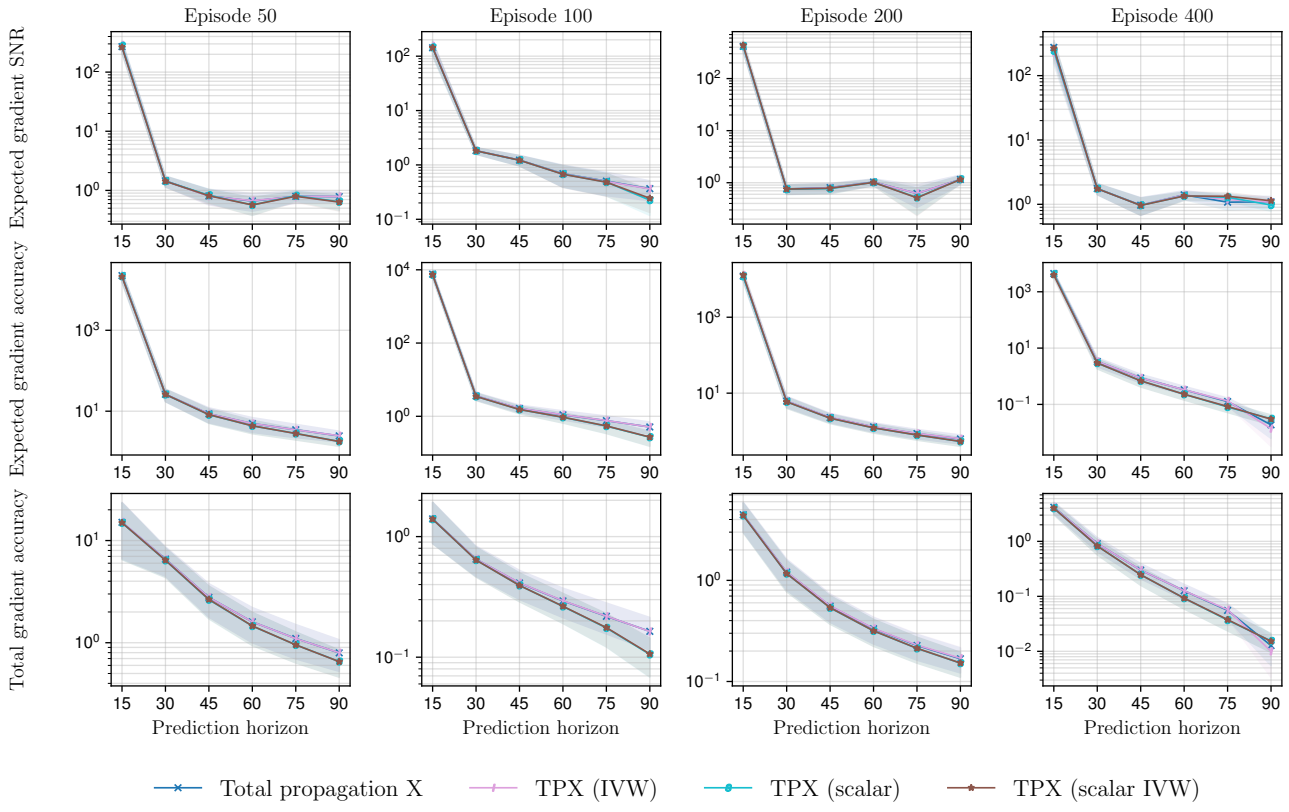


Figure 29: Reacher Easy, TPX ablation raw gradient data

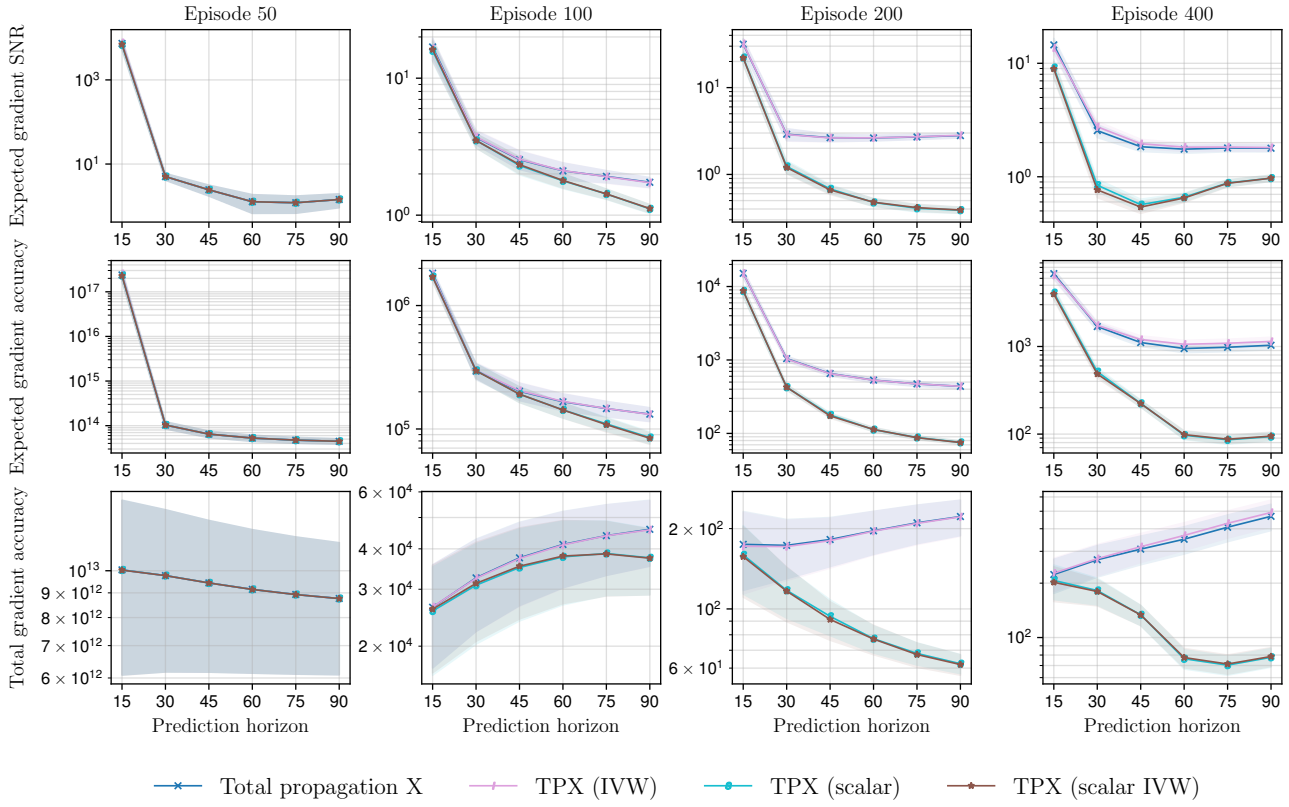


Figure 30: Walker Walk, TPX ablation raw gradient data

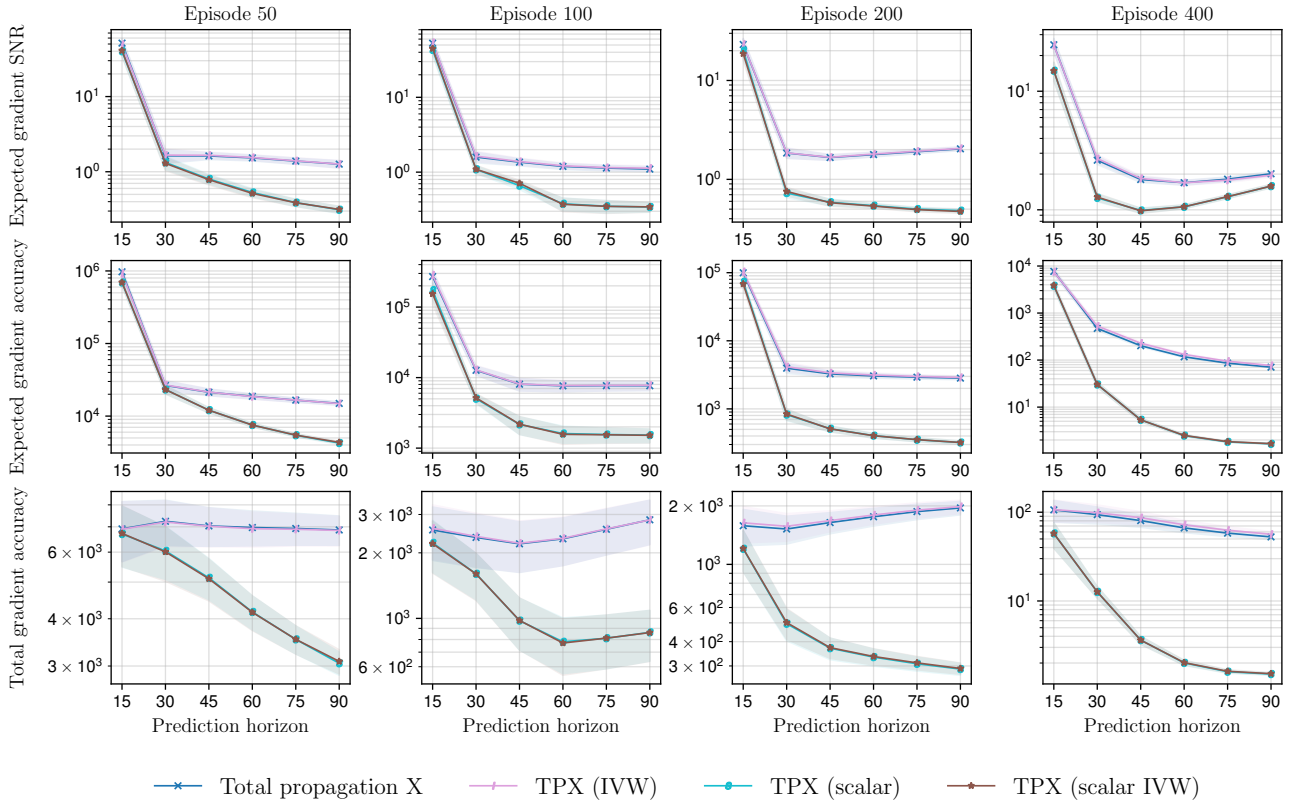


Figure 31: Walker Run, TPX ablation raw gradient data

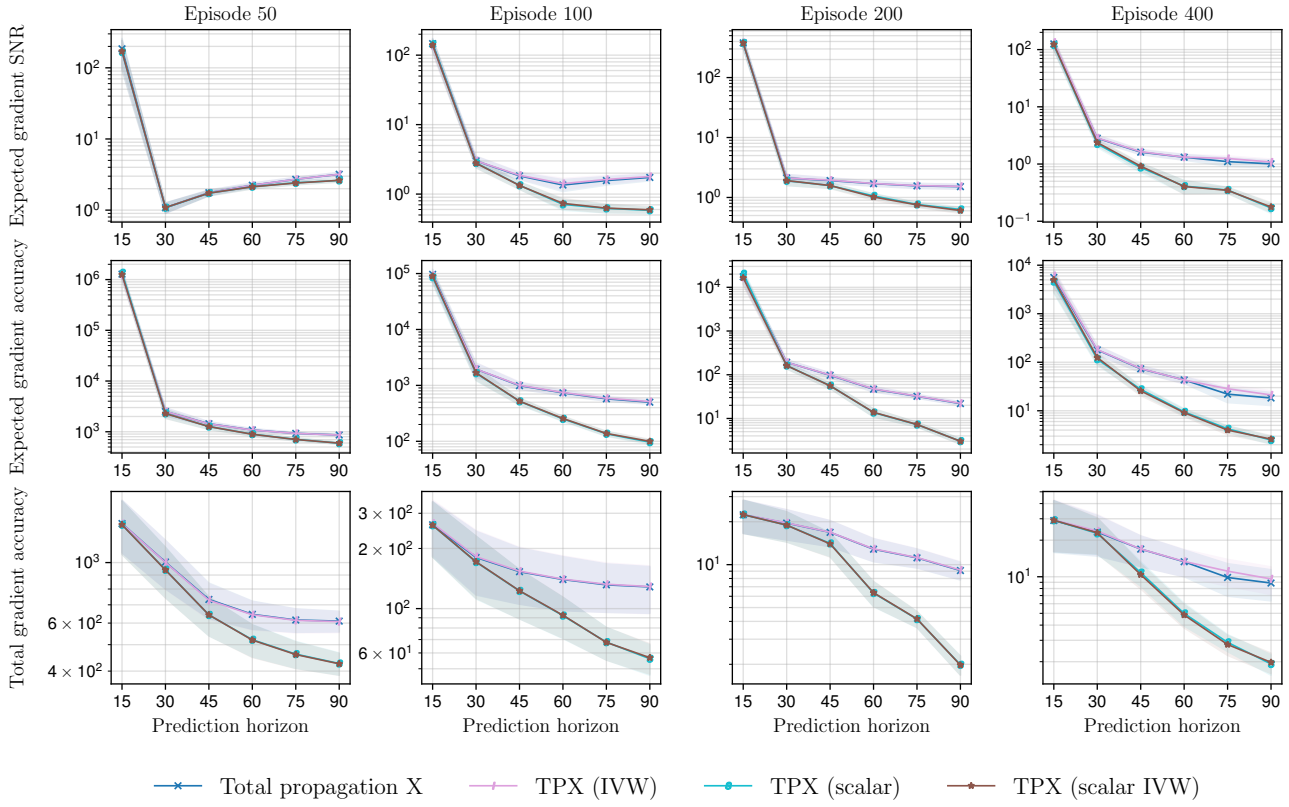


Figure 32: Cheetah Run, TPX ablation raw gradient data

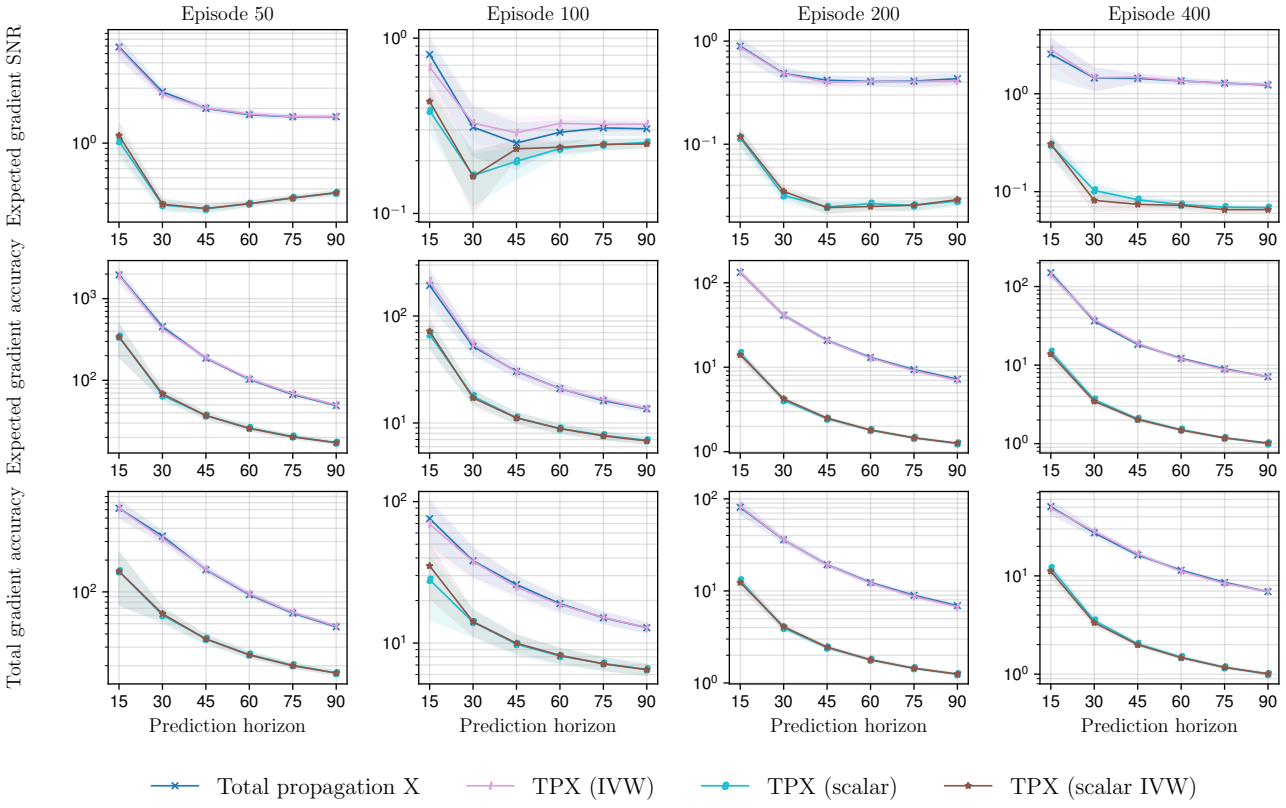


Figure 33: Finger Spin, TPX ablation raw gradient data

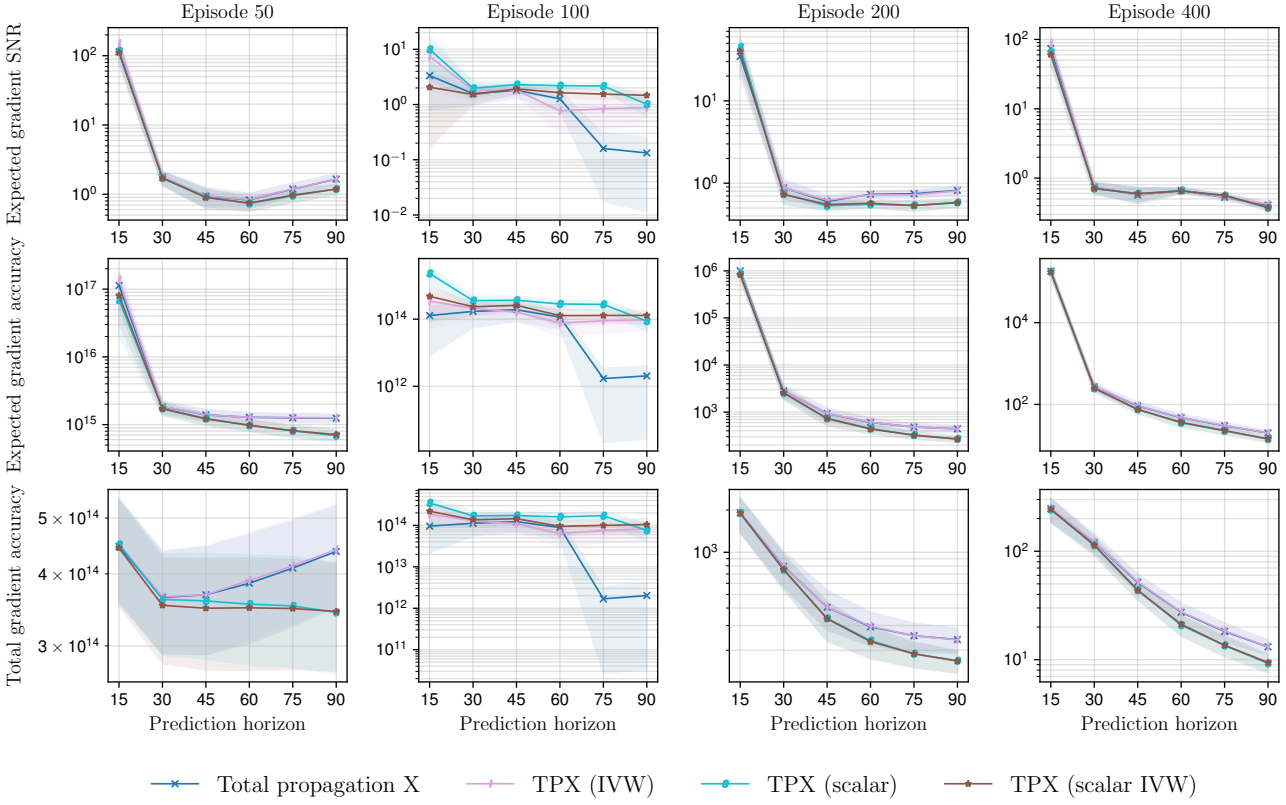


Figure 34: Cartpole Swingup, TPX ablation raw gradient data

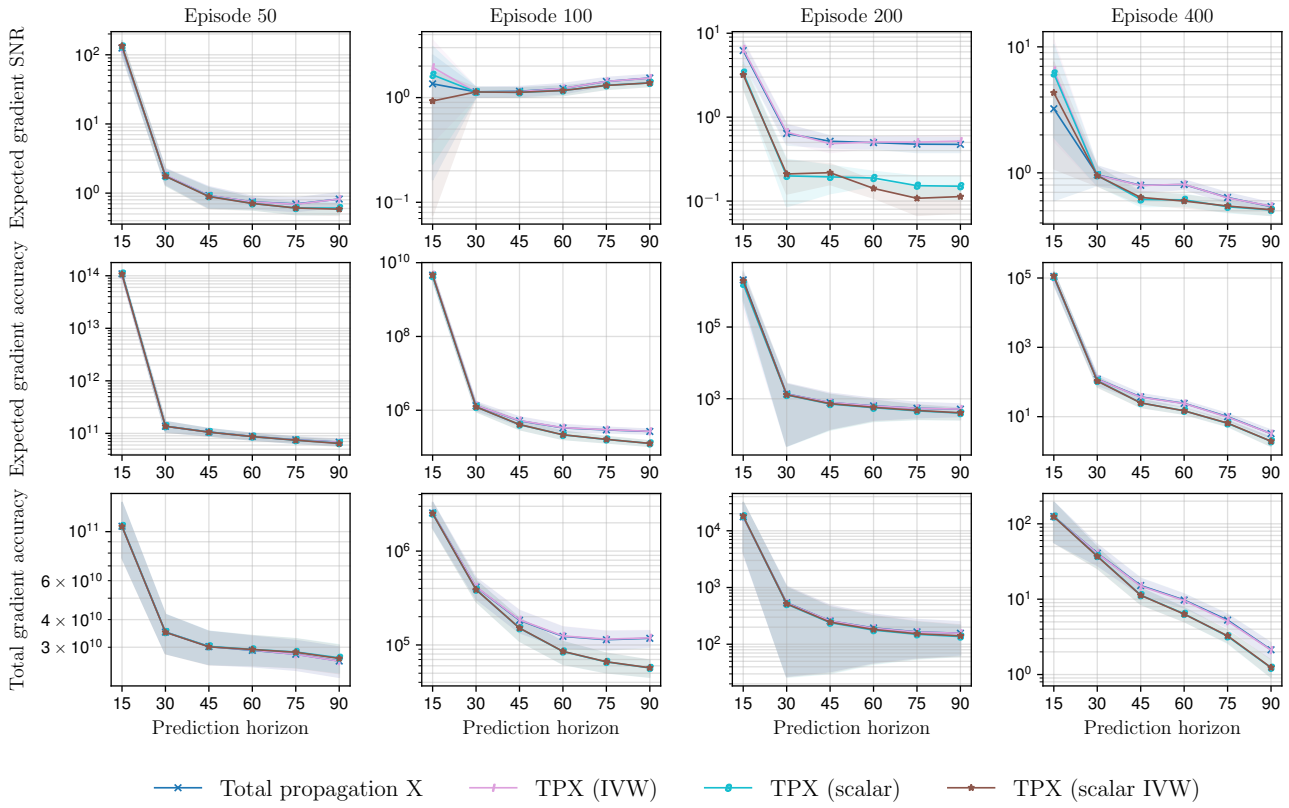


Figure 35: Cartpole Swingup Sparse, TPX ablation raw gradient data

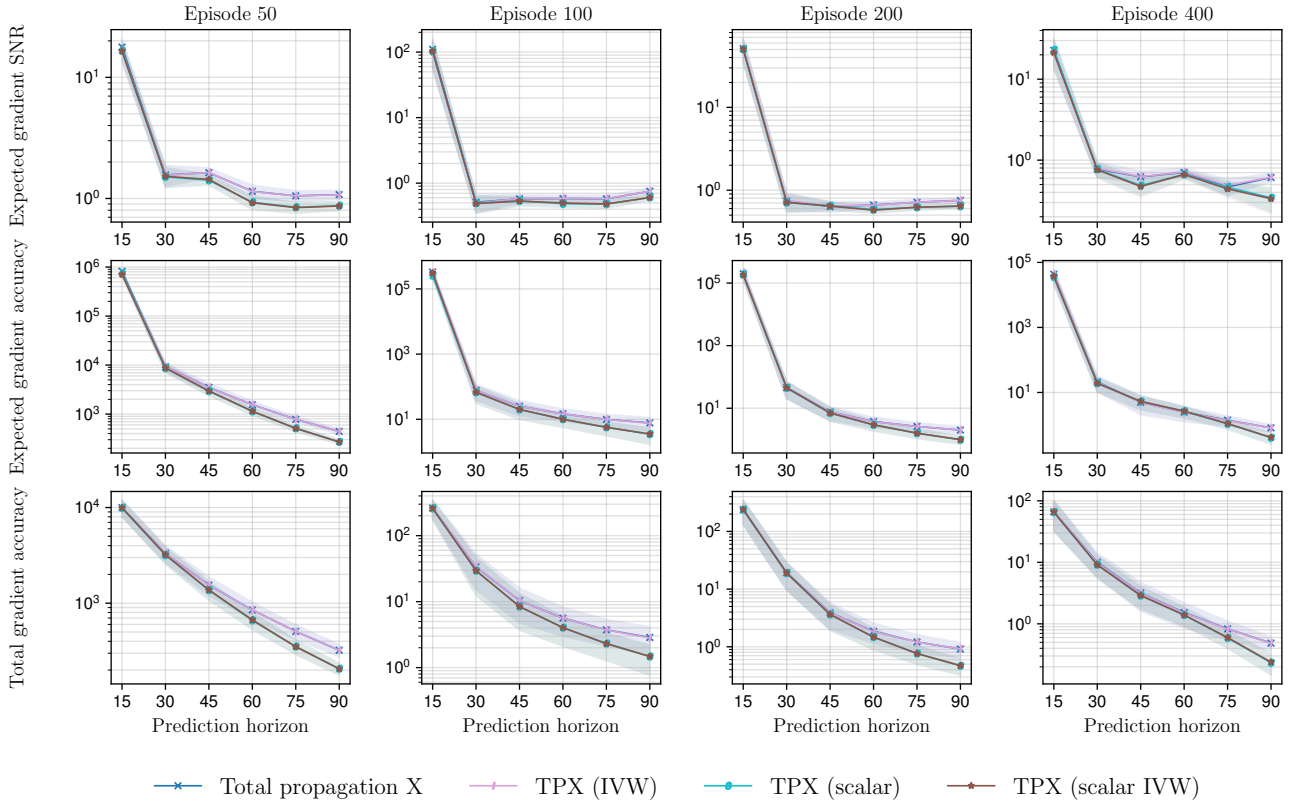


Figure 36: Cartpole Balance, TPX ablation raw gradient data

References

Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A. C., and Bellemare, M. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 34:29304–29320, 2021. 6.2

Candela, J. Q., Girard, A., Larsen, J., and Rasmussen, C. E. Propagation of uncertainty in bayesian kernel models-application to multiple-step ahead forecasting. In *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03).*, volume 2, pp. II–701. IEEE, 2003. B.1

Cho, K., van Merriënboer, B., Gulçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014. 6.1

Dangel, F., Kunstner, F., and Hennig, P. Backpack: Packing more into backprop. In *International Conference on Learning Representations*, 2020. 4

Degris, T., White, M., and Sutton, R. Off-policy actor-critic. In *International Conference on Machine Learning*, 2012. C.1

Deisenroth, M. P. and Rasmussen, C. E. PILCO: a model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pp. 465–472, 2011. B.1

Gal, Y., McAllister, R., and Rasmussen, C. Improving PILCO with bayesian neural network dynamics models. In *Workshop on Data-efficient Machine Learning, ICML*, 2016. B.1

Geffner, T. and Domke, J. Using large ensembles of control variates for variational inference. In *Advances in Neural Information Processing Systems*, pp. 9960–9970, 2018. 2

Geffner, T. and Domke, J. A rule for gradient estimator selection, with an application to variational inference. In *International Conference on Artificial Intelligence and Statistics*, pp. 1803–1812. PMLR, 2020. 2

Girard, A., Rasmussen, C., Candela, J. Q., and Murray-Smith, R. Gaussian process priors with uncertain inputs application to multiple-step ahead time series forecasting. *Advances in neural information processing systems*, 15, 2002. B.1

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

Greensmith, E., Bartlett, P. L., and Baxter, J. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530, 2004. 2

Hafner, D., Lillicrap, T., Ba, J., and Norouzi, M. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2020. 1, 6

Hafner, D., Lillicrap, T. P., Norouzi, M., and Ba, J. Mastering atari with discrete world models. In *International Conference on Learning Representations*, 2021. 6

Hafner, D., Pasukonis, J., Ba, J., and Lillicrap, T. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023. 1, 6

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015. 3

Jordan, M. I. and Rumelhart, D. E. Forward models: Supervised learning with a distal teacher. *Cognitive science*, 16(3): 307–354, 1992. 1

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. B.1

Kingma, D. P. and Welling, M. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013. 6.1

Krishnan, R. G., Shalit, U., and Sontag, D. Deep kalman filters. *arXiv preprint arXiv:1511.05121*, 2015. 6.1

Lee, W., Yu, H., and Yang, H. Reparameterization gradient for non-differentiable models. In *Advances in Neural Information Processing Systems*, pp. 5553–5563, 2018. 2

Madeka, D., Torkkola, K., Eisenach, C., Foster, D., and Luo, A. Deep inventory management. *arXiv preprint arXiv:2210.03137*, 2022. 1

McHutchon, A. *Modelling nonlinear dynamical systems with Gaussian Processes*. PhD thesis, University of Cambridge, 2014. B.1

Metz, L., Maheswaranathan, N., Nixon, J., Freeman, C. D., and Sohl-Dickstein, J. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, 2019. 1, 2

Parmas, P. Total stochastic gradient algorithms and applications in reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 10204–10214, 2018. 3

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

Parmas, P. and Seno, T. Proppo: a message passing framework for customizable and composable learning algorithms. *Advances in Neural Information Processing Systems*, 35:29152–29165, 2022. 1, 2, 3, 4, 5, A, B.1

Parmas, P. and Sugiyama, M. A unified view of likelihood ratio and reparameterization gradients. In *International Conference on Artificial Intelligence and Statistics*, pp. 4078–4086. PMLR, 2021. 2

Parmas, P., Rasmussen, C. E., Peters, J., and Doya, K. PIPPS: Flexible model-based policy search robust to the curse of chaos. In *International Conference on Machine Learning*, pp. 4062–4071, 2018. 1, 2, B.1

Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pp. 1310–1318. PMLR, 2013. 1

Rasmussen, C. E. and Williams, C. K. I. *Gaussian Processes for Machine Learning*. MIT Press, 2006. B.1

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

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

Suh, H. J., Simchowitz, M., Zhang, K., and Tedrake, R. Do differentiable simulators give better policy gradients? In *International Conference on Machine Learning*, pp. 20668–20696. PMLR, 2022. 1, 2

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018. 1, 6.1

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pp. 1057–1063, 2000. 1

Tunyasuvunakool, S., Muldal, A., Doron, Y., Liu, S., Bohez, S., Merel, J., Erez, T., Lillicrap, T., Heess, N., and Tassa, Y. dm_control: Software and tasks for continuous control. *Software Impacts*, 6:100022, 2020. ISSN 2665-9638. doi: https://doi.org/10.1016/j.simpa.2020.100022. URL https://www.sciencedirect.com/science/article/pii/S2665963820300099. 1, 6.2

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

Werbos, P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. 1

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

Wu, P., Escontrela, A., Hafner, D., Goldberg, K., and Abbeel, P. Daydreamer: World models for physical robot learning. *arXiv preprint arXiv:2206.14176*, 2022. 6

Xu, J., Makoviychuk, V., Narang, Y., Ramos, F., Matusik, W., Garg, A., and Macklin, M. Accelerated policy learning with parallel differentiable simulation. In *International Conference on Learning Representations*, 2022. 1