Investigating Gradient Stability in Policy Based Methods

groupprojectrl.medium.com/test-5d857403e153

October 23, 2020

How does the gradient stability differ between REINFORCE, G(PO)MDP, G(PO)MDP+ whitening during policy learning?

by: Bella Nicholson, Bob Borsboom, Tim van Loenhout, and Jochem Hölscher

Introduction

When it comes to policy based methods, gradient behavior can be very telling. In essence, gradient stability determines: (a) how long it takes a given model to learn and (b) whether it can even learn anything at all. To illustrate what we mean, we will spend this blog post investigating the stability of policy gradients in a few different contexts (i.e., different algorithm-environment combinations). Particularly, we use the REINFORCE, G(PO)MDP and G(PO)MDP+ whitening algorithms, since we can view the latter two algorithms as a progression of the former such that each step of this progression leads to more stable gradients. Meaning, these algorithms are similar enough to each other such that we can make fair comparisons, but also are theoretically guaranteed to exhibit different degrees of gradient stability. We apply these algorithms to two classical RL problems: a self-designed version of GridWorld and OpenAI's CartPole problem. We do so since both environments vary in their rewards distributions and state-action space complexity.

Don't worry if you're not completely up to date on your reinforcement learning knowledge. We'll first briefly introduce all the need-to-know concepts — including a proper explanation of what policy gradient-based methods are, and what our chosen environments look like. So, if you're more confident in your RL knowledge, feel free to skip ahead to our experiments.

Are you ready?

GridWorld

GridWorld is one of the most simple RL problems, since it's so easy to visualize and hence properly understand. As such, the intuition that this problem provides is not only invaluable, but it is also needed for our dive into policy based methods. Therefore it will be the first thing we discuss.

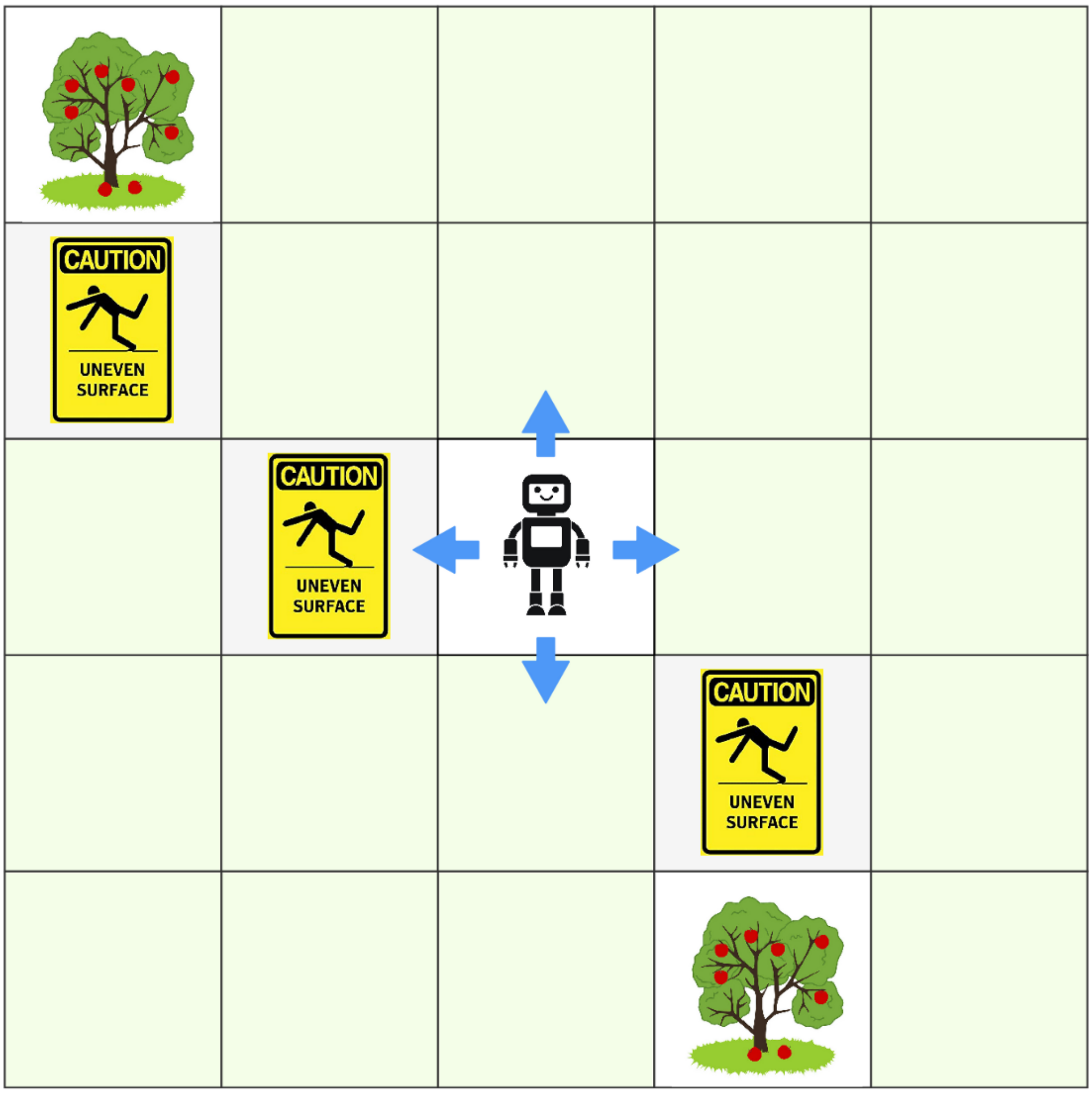


Figure 1: 5x5 Grid world

The premise of GridWorld is as the name suggests: we model our "world" as a finite grid. Generally speaking, our agent knows nothing about this world at first, but it needs to learn the best way to move through it. Thus, our agent relies on trial and failure. Consider the GridWorld problem of teaching a robot to pick an apple from the apple tree. We will model the field our robot is walking through as a 5 x 5 grid, since our robot cannot vary its step size. For simplicity's sake, we'll assume that if the robot reaches the apple tree, then it is always able to grab an apple. When it lands in a square that contains an apple tree, we give it a reward of +2. If the robot is unlucky enough to find a rock to trip over, it gets a penalty of -1 as we want it to walk without hurting itself. To keep things interesting, we don't give the robot any feedback unless it falls or reaches its goal. Whenever the robot gets an apple or runs out of time, we wipe our rewards count clean, pick it up, and place it at the same starting point to try again.

Policy based methods

In Reinforcement Learning, there are different ways to learn a good policy. For example, you can first explore GridWorld by random walk until you either discover an apple tree or a pesky rock. Afterwards, different algorithms have different ways of extracting a policy from the "experience" we've gained. Suppose, we want to learn this policy directly from our past experiences rather than rely on indirect methods. In essence, this is what **policy based methods** do. We convert policy learning into a calculus problem, where we model the cumulative rewards our robot receives as a function with the intent to optimize it.

REINFORCE

With REINFORCE, we play a game (e.g. our apple-picking robot in GridWorld) until we reach some terminal state (e.g., arriving at an apple tree). Afterwards, we start all over again to gather even more information about how to formulate the best strategy. During the course of each run through this game, we sum all the derivatives over the log probabilities of an action given its current state. Once we hit our terminal state, our trajectory gets cut off and we multiply the sum of gradients with that trajectory's reward. We formalize this process as:

$$\nabla_\theta J = G(\tau) \sum_{t=0}^{T} \nabla_\theta \log p_\theta(a_t | s_t)$$

We run the algorithm N times and then take the average over these runs to get a more stable and thus less-noisy update. Thus, the expression above becomes

$$\nabla_\theta J = \frac{1}{N} \sum_{i=0}^{N} G(\tau_i) \sum_{t=0}^{T} \nabla_\theta \log p_\theta(a_t^i | s_t^i),$$

where we not only loop over every *i-th* trajectory but also every state-action pair *t* that we encounter.

Once our gradients are calculated, we use the following update rule to calculate our new parameter values. This rule will improve the policy.

$$\theta_{t+1} = \theta_t + \alpha \cdot \nabla_\theta J$$

where alpha represents our **learning rate**.

If we refer to Figure 2, alpha literally determines the step size we take along our expected rewards surface once we learn something new. For these reasons, it is also interchangeably referred to as the **step size**. While more complex methods anneal the step size with respects to time, we've kept things simple and treated alpha as a constant hyperparameter.

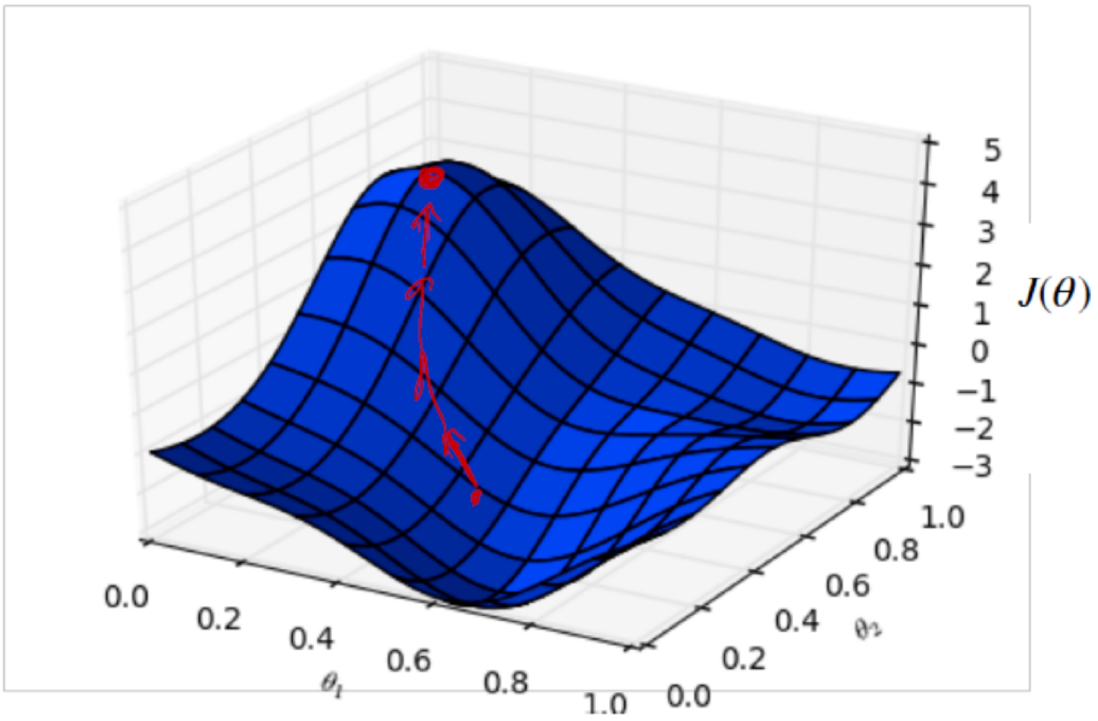


Figure 2: Surface of our expected rewards function J.

An Intuitive Look at REINFROCE

As we've previously established, we multiply a trajectory's reward by the sum of its gradients. Suppose, we obtain a cumulative negative reward, because our robot walked over a few obstacles before it reached an apple tree. Now, the gradients are multiplied with a negative number. Meaning, our new policy lowers the likelihood of performing the same actions given each respective state. On the other hand, suppose that we get a positive reward, since our robot successfully walked toward an apple tree.
In this case, our policy will make the selection of these actions for these given states more probable in future trajectories.

The Problems with REINFORCE

Thus far, we've painted a rather rosy picture of the REINFORCE algorithm. Now, it's time to take a more critical look at REINFORCE and see what room there is for future improvements. Let's look at two paths that our robot can take in a slightly different version of GridWorld .

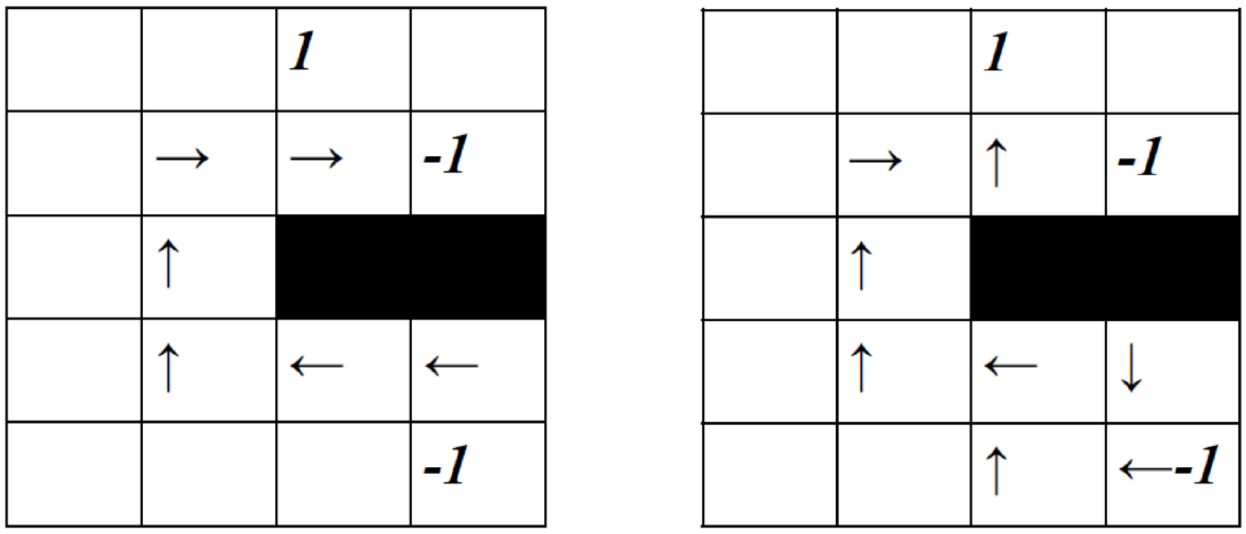


Figure 3: Grid world with two different trajectories

Once again, our robot's objective is to walk towards the positive reward without hurting itself. In the lefthand side of the figure, the agent's performance is perfect until its last step. Now, REINFORCE will make ALL of these actions less probable. Whereas, we only want to "punish" the last action.

A similar mistake happens again on the right hand side of figure 3. Here, the total reward for that trajectory is zero, so the model won't change the action-state probabilities. In other words, we don't learn from the mistakes we made by choosing the wrong first action. Over enough trajectories, REINFORCE may be able to find the optimal policy, but this high degree of variance means that this process takes longer.

G(PO)MDP

G(PO)MDP is an improvement as it fixes the previously mentioned mistakes of REINFORCE. Instead of multiplying the reward of the trajectory by all the actions performed in that trajectory, it handles this situation in a more clever way. G(PO)MDP only multiplies the reward from a trajectory by the steps which lead to this action. We formalize this introduced temporal-reward dependency as,

$$\nabla_\theta J = \sum_{t=1}^{T} r_t \sum_{t'=1}^{t} \nabla_\theta \log p(a_t'|s_t')$$

We simply sum over all the steps in a trajectory, but here comes the interesting part: the right summation sums all the log derivatives of actions taken until we get a reward at time step t'. Afterwards, we multiply this reward by the steps which lead to this reward. Meaning, only the actions that lead to a reward will be "influenced" by that reward. While all the actions were previously influenced by the rewards which came before that set of actions, G(PO)MDP provides a fairer policy update. Simply put, our action probabilities now increase or decrease solely based on the reward a given action leads to.

This tweak in our objective function makes G(PO)MDP more fair in its assignment of action-state likelihoods and lowers the variance of our policy gradients. However, we're still interested in lowering it even further, since this can decrease the time it takes us to figure out what the optimal strategy is.

Baselines

We'll take a look at baselines now as a means to further lower policy gradient variance. Baselines are a trick researchers found to stabilize policy learning through the reduction of gradient variance. Fundamentally, we subtract a constant estimate of our expected rewards from our current return G_t Meaning, our parameter update becomes

$$\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t))\nabla \log \pi(A_t|S_t, \theta),$$

where we refer to this expected rewards estimate as a **baseline**. Many baselines variants have been proposed and tested, and each comes with its own set of advantages and disadvantages. To keep things simple, we've settled for a baseline technique called **whitening**, where we scale our cumulative returns by their mean and standard deviation values. i.e.,

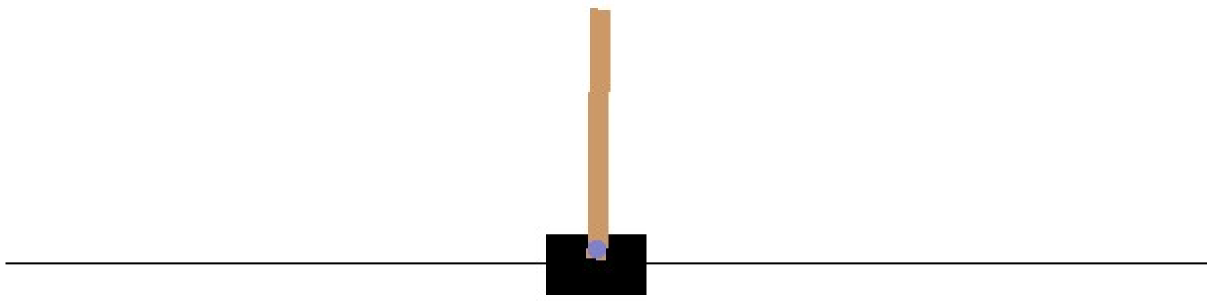
$$b(S_t) = \frac{G_t - \overline{G}}{\sigma_G}$$

While more sophisticated methods do exist, whitening gives us a basic understanding of a baseline alters policy gradient behavior.

Different environments

With the premises of our chosen algorithms established, we also need to discuss the problems we'll be using these algorithms to solve. Since algorithm behavior largely depends on the given environment, our GridWorld problem isn't enough to gain a comprehensive picture of gradient stability. As a result, we also need to consider an environment that is significantly different from GridWorld — both in its state-space structure and in its reward distribution. That is, we want a continuous state-space environment that offers more immediate feedback.

Thus, we settled for OpenAI's CartPole environment, which is described as follows:



CartPole. A pole is attached by an unactuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force left or right to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time step that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. A link to the environment can be found here.

If you want to learn more about the OpenAI Gym, take a look here

Through this choice of environments, we can see two different "forces" that contribute to a problem's difficulty: (1) the size of the state-action space we have to search through, and (2) the level of feedback our agent receives for every good or bad action that it takes. Intuitively speaking, the more feedback you receive as you try to learn some new skill or task, the easier the learning process becomes.

From this perspective, neither problem is indisputably more difficult than the other. The CartPole environment requires our agent to navigate through a more complex state-space; however, GridWorld does not always offer feedback. Our agent can move in such a way where it neither encounters an apple tree or a rock, and thus receives no reward. This level of non-immediate feedback adds complexity to what seemed like a rather simple problem.

Gradient stability

Up to this point we've established the importance of gradient stability but have not yet detailed a means to quantify it. Given that the concept of gradient stability is simultaneously intuitive and nebulous, we need to reframe the way we view it in order to make it measurable. More specifically, we define gradient stability by how similar policy gradients are to one another within each time step. To do so, we use the statistical analysis tools of variance, kurtosis and gradient sign change.

Variance

Also known as the second central moment, variance describes the spread of the gradients, and thus clues us in to our model's update precision. i.e., How consistent or steady are our model updates from to run? High levels of variance indicate that the policy gradients go in many different directions, while low levels of variance mean our gradients are more or less uniform in the direction they point in.

Kurtosis

However, distributions with identical means and variances can still have have very different shapes. In such cases, we also need to look at kurtosis (the fourth standardized central moment), which describes how heavily tailed a distribution is. In other words, kurtosis quantifies the presence of gradient outliers. The higher kurtosis is, the more extreme these gradient values become. This in turn makes makes our policy gradients become more inconsistent — i.e., less stable.

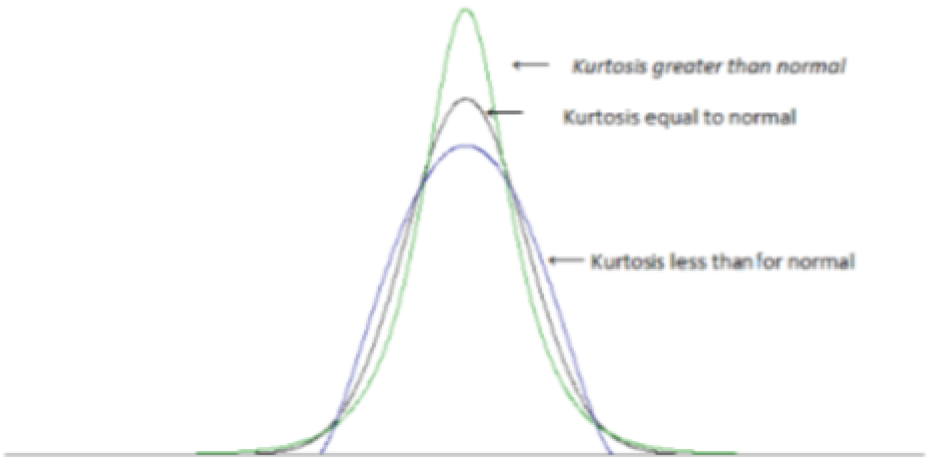


Figure 5: Different kurtosis values for identical variance.

Source can be sound here

Sign changes

Finally, the number of sign changes that occur between time steps also provides clues about gradient stability. If the gradient signs keep changing, this means that we are literally walking back and forth along our loss surface. This implicates one of two possibilities, we either: (1) are consistently overshooting in our updates, or (2) are moving randomly as we do not know where to go. Either way, this indicates less confidence in our parameter updates and hints at model instability.

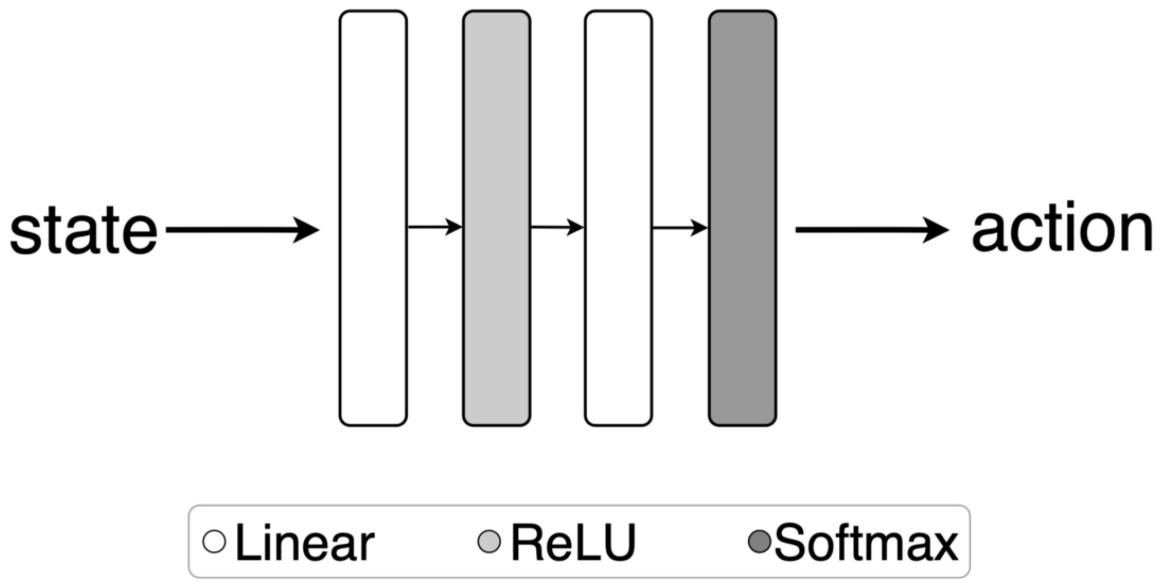
Experimental Setup

We can breakdown our gradient stability analysis into two parts: (a) differences in stability between algorithms in the same environment, and (b) differences in stability between the same algorithm in different environments. Of course, we've already covered part (a) in our background. Over all, G(PO)MDP is not influenced by the noise of rewards from prior actions, and whitening only further reduces the variance of whatever policy based method we apply it to. As such, we expect REINFORCE to have the least stable gradients, followed by G(PO)MDP, and by then G(PO)MDP + whitening.

Meanwhile, the complexity of CartPole's state space could highlight the shortcomings of REINFORCE. However, this overlooks the main distinction between REINFORCE and G(PO)MDP as one handles rewards considerably better than the other. Therefore, the sparsity of GridWorld's rewards distribution may outweigh the problems introduced with high state-action search spaces. As a result, we expect a larger difference in policy behaviour between the algorithms in the GridWorld environment. We used the following experimental setup to verify our hypothesis.

Policy Architecture

Generally, the problem we are trying to solve dictates our policy network architecture. As we've established previously, our state-action space is small and discrete in GridWorld. More specifically, we only have 25 potential states and 4 potential actions to choose from. Given that we define a policy by the action distribution for each state, our policy is nothing more than a look-up table. In practice, this translate to the use of a single linear layer when we apply (deep) policy gradient methods. In contrast, once we delve into continuous state-space problems, we need more complex (i.e., multi-layered) models. Thus, for the CartPole environment, we default to a simplified version of the standard Q-Network architecture, as shown in Figure 6. The original DQN paper can be found here. Note that we use a final softmax layer to get a probability distribution over the possible actions we can take.



If you want to learn more about the basics of neural networks, this is a good starting point.

Parameter Configurations

Initially, we conducted hyperparameter tuning with respects to each algorithm-environment combination, but quickly discovered that the environment more or less determined the optimal learning rate. Here, we define the optimal learning rate as the hyperparameter value which yields the highest cumulative rewards on average. Of course, we cannot apply the same exact process to define our discount factor, since the higher our discount factor is, the higher our cumulative rewards become by definition. However, a good discount factor is still important. So, we "tuned" discount factor by whichever value lead to convergence — which was once again model specific. Table 1 summarizes all of the values that we used.

	Learning Rate	Hidden Dimensions	Discount Factor
GridWorld	0.1	-	0.99
CartPole	0.001	128	0.99

Table 1: Summary of Parameter Configurations Used.

Obtaining policy gradients

We apply each of our three algorithms to the two environments we've previously described, and train our model for N=800 episodes. Every 20 iterations, we pause policy training to test policy performance. In this validation step, we sample a 100 different episodes to gather the gradients of each weight within our network as well as the cumulative rewards observed. The latter is particularly important, since it allows us to contextualize how well our policy is really doing. Meanwhile, the former allows us to understand how gradient behavior varies with respects to time. Due to the sensitivity policy networks exhibit towards their initialized weights, we repeat each model-environment combination using 20 different seeds and report the average statistical measurements that we've observed.

Obtaining variance, kurtosis and sign changes

Since the collected policy gradients values are high dimensional tensors, we cannot directly visualize the policy gradient distribution. Instead, we compute the variance and kurtosis for each individual network weight over the 100 episodes sampled per validation step. Then, we take the average over these statistics to obtain a set of aggregated variance and kurtosis measurements. Together, these values describe the average gradient distribution for a single algorithm-environment combination at a specific time step. We repeat these calculations for all sampled policy gradients and plot our results. Note that the obtained gradients are squeezed between -1 and 1 across the episode dimension. This rescaling does not impact the variance/kurtosis ratio between two algorithms for a single weight gradient, but it does make sure that all weights contribute equally to our aggregate variance and kurtosis values.

Results

Starting with a look at gradient variance, Figure 7 proves that a choice in environments makes a stark difference in policy gradient behavior. As expected, we observe clear differences in gradient stability when in GridWorld, but when not when we're in CartPole. This confirms our hypothesis that the rewards distribution has the capacity to incite more instability than the complexity of our search (state-action) space.

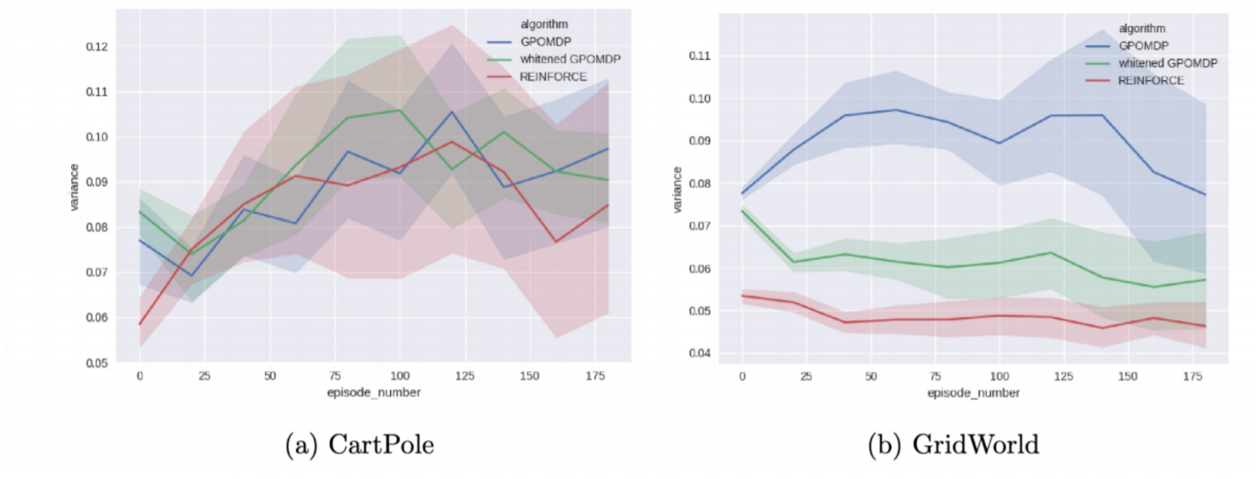


Figure 7: The variance of the policy gradients for all three policies in both the CartPole and GridWorld environments. The variances are reported as the averages observed over all 20 seeds, where the shaded areas represent standard deviation.

Interestingly enough, Figure 7b suggests that REINFORCE somehow exhibits the least amount of variance. Outside of any other context, one could mistakenly presume that this means REINFORCE is the best performing algorithm. However, Figure 8 reveals a very different reality: REINFORCE performs so poorly that our model never learns anything. In the context of our apple-picking robot, our robot literally runs in circles and continuously trips over the same rock. Often times, it only stops when it runs out of time (i.e., the number of steps it can take per episode). With this knowledge, we can re-contextualize the abnormality of REINFORCE's variance as a sign that REINFORCE is consistently random.

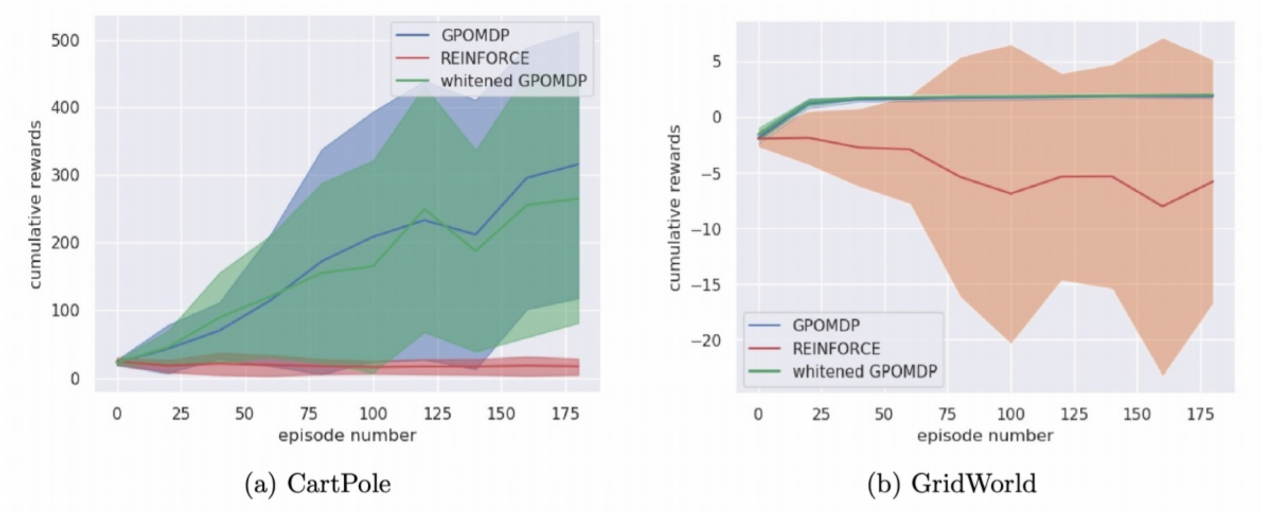


Figure 8: The average cumulative rewards observed over 20 different seeds for all algorithm-environment configurations. The shaded areas represent standard deviation.

Of course, this begs the question: How would REINFORCE perform as an objective function *if* we were able to get a sensible (i.e., non-random) policy? To answer this question, we trained our policy networking using our most stable algorithm, G(PO)MDP + whitening, and only use each algorithm's respective loss function for policy gradients sampling (Figure 9). While the results observed for Figures 7a and 9a are more or less interchangeable, the same cannot be said for Figures 7b and 9b. In Figure 9b, REINFORCE's variance starts low as the policy is close to random, but it soon enough converges to the variance of G(PO)MDP. Note that G(PO)MDP + whitening has the lowest variance of all, but this is only logical since the model has been trained to minimize this specific loss function.

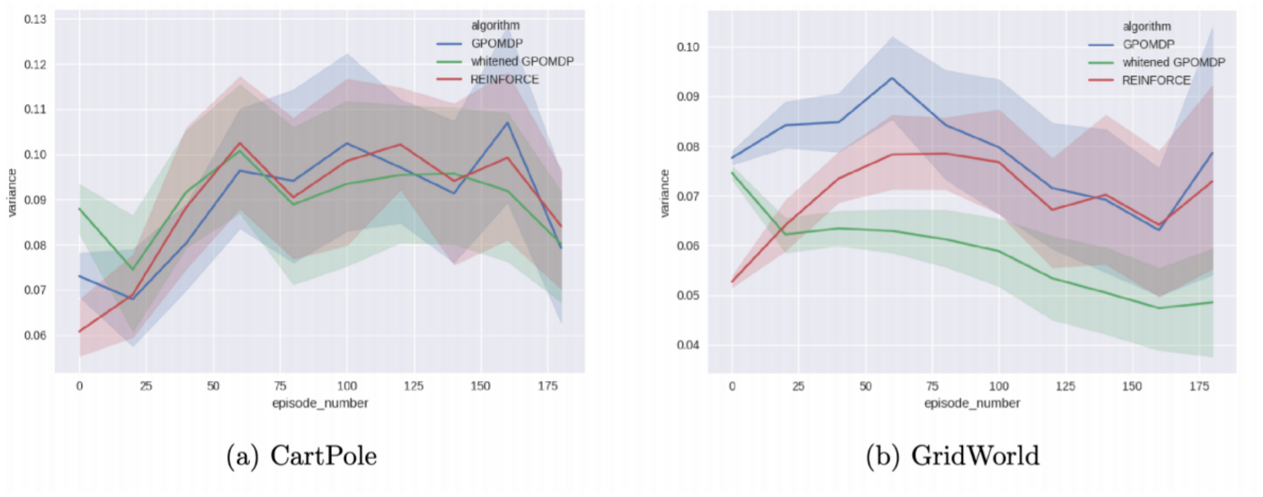


Figure 9: The average variance of the gradients for all algorithm-environment configurations when the parameters are trained using GPOMDP + whitening. The shaded areas indicate standard deviation.

Nonetheless, we have to acknowledge that having the same variances does not implicate different policy gradient distributions to be the same. Meaning, the results we obtain from Figures 7a and 9a are, in a sense, inconclusive. For these reasons, we also consider the kurtosis of our policy gradients (Figure 10). REINFORCE exhibits the highest levels of kurtosis followed by G(PO)MDP and then by G(PO)MDP + whitening. Once again the higher kurtosis is, the more gradient outliers a given algorithm has and the noisier its updates become. Meaning, the stability of each algorithm in CartPole is as we expected it to be.

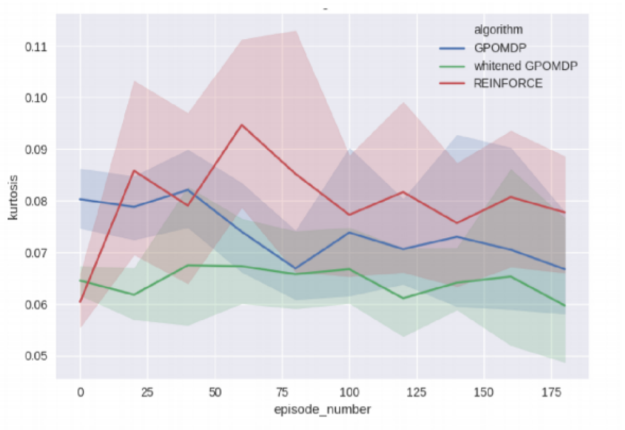


Figure 10: The average kurtosis for all algorithms in the CartPole environment. Results are averaged over the 20 different random seeds. The shaded areas indicate standard deviation within kurtosis.

With the stability established with respects to each algorithm and each problem, we also want to look more closely at how gradient stability varies over time. To keep our figures legible, we only visualize the percentage of positive gradient sign changes. As seen in Figure 11, our policy gradients generally start off as relatively balanced as we begin to make our away along our loss surface. However, during the progression of policy learning, this balance tips in favor of negative gradient sign changes. When this decline in percentage of positive gradient changes is monotonic, we deem overall model learning to be stable. Given that our model relies on these policies to learn, we can view the stability of policy gradients and model learning as one and the same.

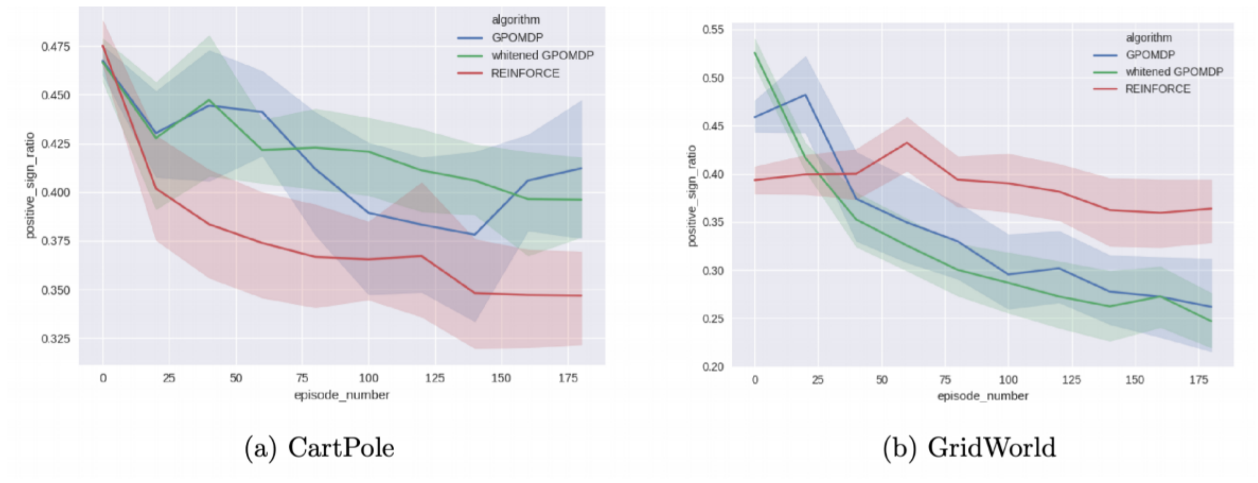


Figure 11: Ratio of positive gradient sign of the three policies in the CartPole environment and the GridWorld environment. The shaded areas indicate standard deviation within sign changes.

Surprisingly, there is a very slight decline in the percentage of positive gradient sign changes observed in REINFORCE's gradients (Figure 11b). However, we don't believe that this model slightly improves on a consistent basis, since GridWorld is too simple to provide opportunities for nuanced policy learning. Rather, we attribute this slight decline to the sensitivity policy networks have towards their initialized weights. Likely, there might be one or two policies out of the 20 seeds that we averaged over where our agent was actually able to learn something using reinforce — even if this something was to avoid states where it would trip. Similarly, we can also use this to explain the erratic behavior of REINFORCE with respects to its policy gradient kurtosis in CartPole.

Conclusion

As seen throughout our results, context is critical when describing policy learning stability. Whenever we rely on a single figure, we risk critically misinterpreting our results and landing on a faulty assumption — such as mistaking our worst performing algorithm for our best performing one. This need for context extends well beyond the consideration of many measurements and goes into explicitly stating the underlying assumptions we make when formulating a hypothesis. After all, in all of our predictions, we assumed the models would eventually learn something about how to play each game, even this something was far from perfect. By never formalizing this assumption, we never considered the alternative possibilities: models are never guaranteed to learn. Especially, models as simple as the ones we implemented here.

GitHub

If you want to take a closer look to our implementation, here you can find a link to our github. Have fun!