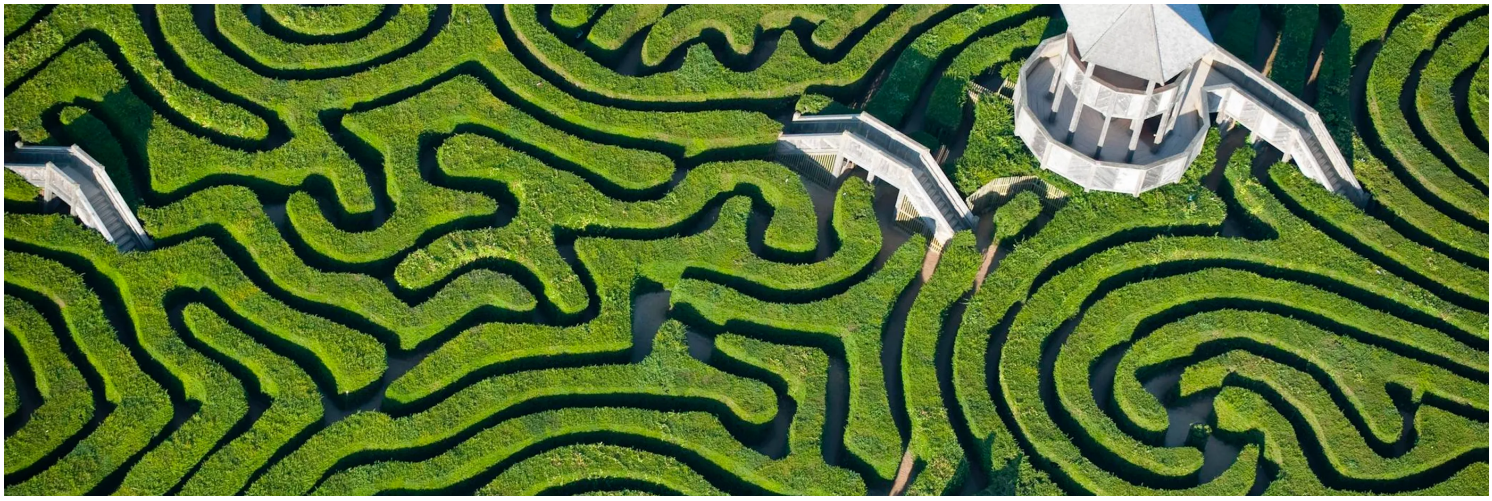
Making Sense of the Bias / Variance Trade-off in (Deep) Reinforcement Learning

What goes into a stable, accurate reinforcement signal?

Arthur Juliani · Follow
Published in ML Review · 12 min read · Jan 31, 2018



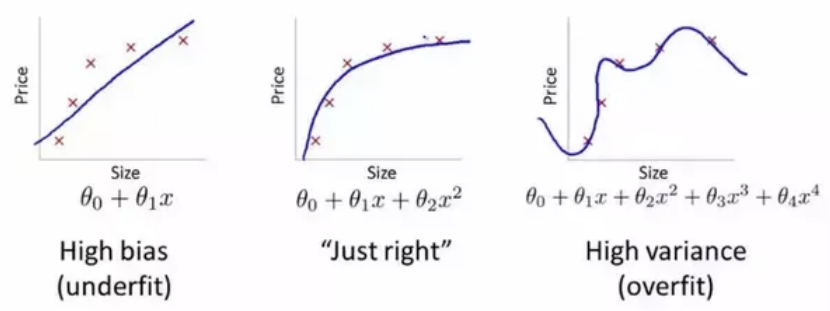
(This post assumes some familiarity with machine learning, and reinforcement learning in particular. If you are new to RL, I'd recommend checking out a series of blog posts I wrote in 2016 on the topic as a primer)

Introduction

Since the launch of the ML-Agents platform a few months ago, I have been surprised and delighted to find that thanks to it and other tools like OpenAI Gym, a new, wider audience of individuals are building Reinforcement Learning (RL) environments, and using them to train state-of-the-art models. The ability to work with these algorithms, previously something reserved for ML PhDs, is opening up to a wider world. As a result, I have had the unique opportunity to not just write about applying RL to existing problems, but also to help developers and researchers debug their models in a more active way. In doing so, I often get questions which come down to a matter of

understanding the unique hyperparameters and learning process around the RL paradigm. In this article, I want to attempt to highlight one of these conceptual pieces: **bias and variance in RL**, and attempt to demystify it to some extent. My hope is that in doing so a greater number of people will be able to debug their agent's learning process with greater confidence.

Supervised Machine Learning

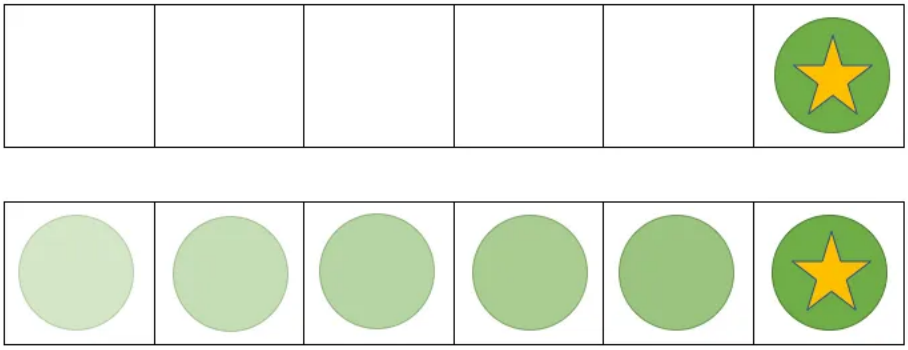


If you've studied ML, these charts may be familiar to you.

Many machine learning practitioners are familiar with the traditional bias-variance trade-off. For those who aren't, it goes as follows: on the one hand, a "biased" model generalizes well, but doesn't fit the data perfectly ("under-fitting"). On the other hand, a high-variance model fits the training data well, too well in-fact, to the detriment of generalization ("overfitting"). In this situation, the problem becomes one of limiting the capacity of a model with some regularization method. In many cases, dropout or L2 regularization with a large enough data set is enough to do the trick. That is the story for typical supervised learning. RL is a little different, as it has its own separate bias-variance trade-off which operates in addition to, and at a higher level than the typical ML one.

Reinforcement Learning

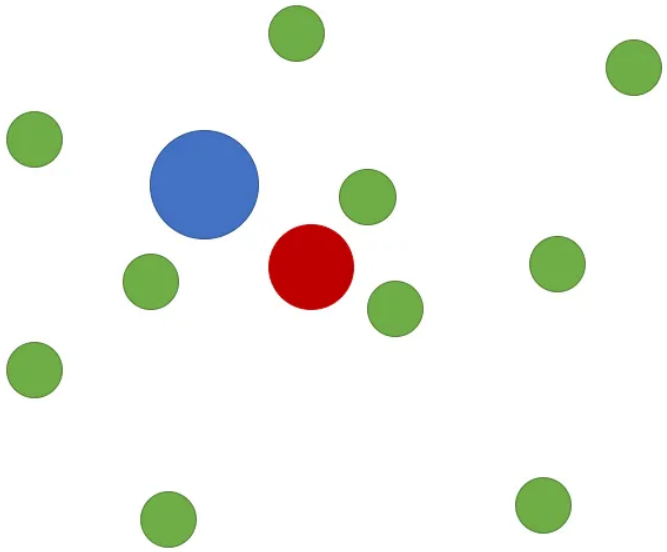
In RL, bias and variance no longer just refer to how well the model fits the training data, as in supervised learning, but also to *how well the reinforcement signal reflects the true reward structure of the environment*. To understand that statement, we have to backup a little. In reinforcement learning, instead of a set of labeled training examples to derive a signal from, an agent receives a reward at every decision-point in an environment. The goal of an agent is to learn a *policy* (method for taking actions) which will lead to obtaining the greatest reward over time. We must do this using only the individual rewards that agent receives, without the help of an outside oracle to designate what count as "good" or "bad" actions.



Rewarding state denoted by yellow star. Value estimates denoted by green spheres. **Above**: Without credit assignment, only rewarding state is seen as being valuable. **Below**: By using discounted sums over future rewards, the trajectory toward star has meaningful value estimates.

A naive approach to an RL learning algorithm would be to encourage actions which were associated with positive rewards, and discourage actions associated with negative rewards. Instead of updating our agent's policy based on immediate rewards though, we often want to account for actions (and the states of the environment when those actions were taken) which lead up to rewards. For example, imagine walking down a corridor to a rewarding object. It isn't just the final step we want to perform again, but all the steps up to that rewarding one. There are a number of approaches for doing this, all of which involving doing a form of *credit assignment*. This means giving some credit to the series of actions which led to a positive reward, not just the most recent action. This credit assignment is often referred to as learning a value estimate: $V(s)$ for state, and $Q(s, a)$ for state-action pair.

We control how rewarding past actions and states are considered to be by using a discount factor (γ, ranging from 0 to 1). Large values of γ lead to assigning credit to states and actions far into the past, while a small value leads to only assigning credit to more recent states and actions. In the case of RL, variance now refers to a noisy, but on average accurate value estimate, whereas bias refers to a stable, but inaccurate value estimate. To make this more concrete, imagine a game of darts. A high-bias player is one who always hits close to the target, but is always consistently off in some direction. A high-variance player, on the other hand, is one who sometimes hits the target, and is sometimes off, but on average near the target.



Red: True value of a given state/action. **Blue**: Low-variance, high-bias estimate. **Green**: low-bias, high-variance estimates.

There is a multitude of ways of assigning credit, given an agent's trajectory through an environment, each with different amounts of variance or bias. *Monte-Carlo sampling* of action trajectories as well as *Temporal-Difference learning* are two classic algorithms used for value estimation, and both are prototypical examples of methods which are variance and bias heavy, respectively.

High-Variance Monte-Carlo Estimate

In Monte-Carlo (MC) sampling, we rely on full trajectories of an agent acting within an episode of the environment to compute the reinforcement signal. Given a trajectory, we produce a value estimate $R(s, a)$ for each step in the path by calculating a discounted sum of future rewards for each step in the trajectory. The problem is that the policies we are learning (and often the environments we are learning in) are stochastic, which means there is a certain level of noise to account for. This stochasticity leads to variance in the rewards received in any given trajectory. Imagine again the example with the reward at the end of the corridor. Given that an agent's policy might be stochastic, it could be the case that in some trajectories the agent is able to walk to the rewarding state at the end, and in other trajectories it fails to do so. These two kinds of trajectories would provide very different value estimates, with the former suggesting the end of the corridor is valuable, and the latter suggesting it isn't. This variance is typically mitigated by using a large number of action trajectories, with the hope that the variance introduced in any one trajectory will be reduced in aggregate, and provide an estimate of the "true" reward structure of the environment.

$$R_t(s,a) = \sum_{t}^{T} \gamma^t r_t$$

Monte-Carlo Estimate of Reward Signal. **t** refers to time-step in the trajectory. **r** refers to reward received at each time-step.

High-Bias Temporal Difference Estimate

On the other end of the spectrum is one-step Temporal Difference (TD) learning. In this approach, the reward signal for each step in a trajectory is composed of the immediate reward plus a learned estimate of the value at the next step. By relying on a value estimate rather than a Monte-Carlo rollout there is much less stochasticity in the reward signal, since our value estimate is relatively stable over time. The problem is that the signal is now biased, due to the fact that our estimate is never completely accurate. In our corridor example, we might have some estimate of the value of the end of the corridor, but it may suggest that the corridor is less valuable than it actually is, since our estimate may not be able to distinguish between it and other similar unrewarding corridors. Furthermore, in the case of Deep Reinforcement Learning, the value estimate is often modeled using a deep neural network, making things worse. In Deep Q-Networks for example, the Q-estimates (value estimates over actions) are computed using an old copy of the network (a "target" network), which will provide "older" Q-estimates, with a very specific kind of bias, relating to the belief of an outdated model.

$$R_t(s,a) = r_t + \gamma V(s_{t+1})$$

Temporal-Difference Estimate of Reward Signal. **r** refers to reward at time t. **V(s)** refers to parameterized value estimate.

Approaches to Balancing Bias and Variance

Now that we understand bias and variance and their causes, how do we address them? There are a number of approaches which attempt to mitigate the negative effect of too much bias or too much variance in the reward signal. I am going to highlight a few of the most commonly used approaches in modern systems such as Proximal Policy Optimization (PPO), Asynchronous Advantage Actor-Critic (A3C), Trust Region Policy Optimization (TRPO), and others.

1. **Advantage Learning**

One of the most common approaches to reducing the variance of an estimate is to employ a baseline which is subtracted from the reward signal to produce a more stable value. Many of the baselines chosen fall into the category of Advantage-based Actor-Critic methods, which utilize both an actor which defines the policy, and a critic (often a parameterized value estimate) which provides a more reduced variance reward signal to update the actor. The thinking goes that variance can simply be subtracted out from a Monte-Carlo sample (R/Q) using a more stable learned value function V(s) in the critic. This value function is typically a neural network, and can be learned using either Monte-Carlo sampling, or Temporal difference (TD) learning. The resulting Advantage A(s, a) is then the difference between the two estimates. This advantage estimate has the other nice property of corresponding to *how much better the agent actually performed than was expected on average,* thus allowing for intuitively interpretable values.

$$A^{\pi}(s_t, a_t) := Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$$

Advantage Estimate Equation. **Pi** refers to the current policy. **Q(s, a)** here refers to Monte-Carlo sampled reward signal analogous to R(s, a), rather than a learned estimate. **V(s)** refers to parameterized value estimate.

2. Generalized Advantage Estimate

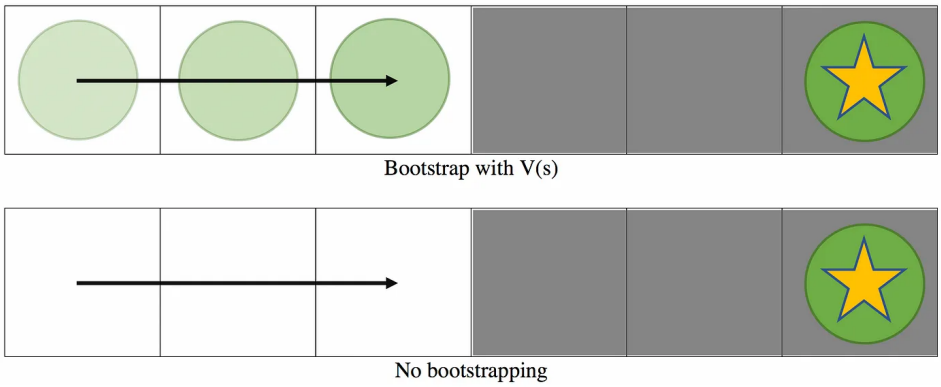
We can also arrive at advantage functions in other ways than employing a simple baseline. For example, the value function can be applied to directly smooth the reinforcement signal obtained from a series of trajectories. The Generalized Advantage Estimate (GAE), introduced by John Schulman in 2016 does just this. The GAE formulation allows for an interpolation between pure TD learning and pure Monte-Carlo sampling using a lambda parameter. By setting lambda to 0, the algorithm reduces to TD learning, while setting it to 1 produces Monte-Carlo sampling. Values in-between (particularly those in the 0.9 to 0.999 range) produce better empirical performance by trading off the bias of *V(s)* with the variance of the trajectory.

$$\begin{aligned}
\mathrm{GAE}(\gamma, 0): \quad & \hat{A}_t := \delta_t && = r_t + \gamma V(s_{t+1}) - V(s_t) \\
\mathrm{GAE}(\gamma, 1): \quad & \hat{A}_t := \sum_{l=0}^{\infty} \gamma^l \delta_{t+l} && = \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V(s_t)
\end{aligned}$$

Generalized Advantage Estimate under two edge cases which reduce to: TD Learning (**Above**), and MC Sampling (**Below**).

3. Value-Function Bootstrapping

Outside of calculating an advantage function, the bias-variance trade-off presents itself when deciding what to do at the end of a trajectory when learning. Instead of waiting for an entire episode to complete before collecting a trajectory of experience, modern RL algorithms often break experience batches down into smaller sub-trajectories, and use a value-estimate to bootstrap the Monte-Carlo signal when that trajectory doesn't end with the termination of the episode. By using a bootstrap signal, that estimate can contain information about *the rewards the agent might have gotten, if it continued going to the end of the episode*. It is essentially a guess about how the episode will turn out from that point onward. Take again our example of the corridor. If we are using a time horizon for our trajectories that ends halfway through the corridor, and if our value estimate reflects the fact that there is a rewarding state at the end, we will be able to assign value to the early part of the corridor, even though the agent didn't experience the reward. As one might expect, the longer the trajectory length we use, the less frequently value estimates are used for bootstrapping, and thus the greater the variance (and lower the bias). In contrast, using short trajectories means relying more on the value estimate, creating a more biased reinforcement signal. By deciding how long the trajectory needs to be before cutting it off and bootstrapping it, we can propagate the reward signal in a more efficient way, but only if we get the balance right.



Arrow corresponds to agent trajectory. **Above**: The value estimate at time step 3 is used to bootstrap trajectory value estimate. **Below**: no bootstrapping is used, and no value is assumed for these states.

Say you have some environment you'd like to have an agent learn to perform a task within (for example, an environment made using Unity ML-Agents). How do you decide how to control the *GAE lambda* and/or *trajectory time horizon*? The outcome of setting these hyperparameters in various ways often depends on the task, and come down to a couple of factors:

- **How well can your value estimator (using function approximation) capture the "true" reward structure of the environment?** If it is

possible for the value estimate to be reflective of the actual future expected reward, then the bias of the reinforcement signal is low, and using it to update the policy will lead to stable reward-increasing improvement. One simple proxy for the quality of the value estimate can be the loss when updating the model responsible for V(s)/Q(s, a). If the loss decreases to (near) zero, then the value estimate is reflective of the rewards the agent is receiving from the environment. More often that not however, this won't be the case. There are a number of reasons a value estimate may be biased. One simple reason comes back to problems of supervised learning, where an under-capacity model being used to learn *V(s)* will likely produce a biased estimate. Increasing the capacity of the model might help here, as well as providing a richer state representation. The other problem is that the "true" reward structure is always a moving target. As an agent improves its policy, then the expected rewards should increase as well. This causes the unintuitive experience of the loss function increasing, even though the agent's policy is more successful.

- **How stochastic are the rewards experienced in the environment?** This deals directly with the variance piece of the equation. Environments with more uncertain outcomes (due to multi-agent dynamics, stochasticity in policies and environment, and other factors) will result in less predictable rewards, and hence greater variance in the reward signal. As mentioned above, the traditional fix for this is to simply use larger batches of data during each step of stochastic gradient descent. This has a downside however. It is known that at least for supervised learning problems, large batches of gradient descent are much less efficient than smaller batches. It is also difficult to fit large batches into the memory of a system, especially when images of any considerable size are being used as observations. This approach is also the most data-intensive, as large batches require more interaction with the environment.

Ultimately, correctly balancing the trade-off comes down to a few things: gaining an intuition for the kind of problem under consideration, and knowing what hyperparameters for any given algorithm correspond to what changes in the learning process. In the case of an algorithm like PPO, this corresponds to the *discount factor, GAE lambda, and bootstrapping time horizon*. Below are a few guidelines which may be helpful:

- **Discount Factor:** Ensure that this captures how far ahead agents should be predicting rewards. For environments where agents need to think thousands of steps into the future, something like 0.999 might

be needed. In cases where agents only need to think a few steps into the future, then something closer to 0.9 might be all that is needed.

- **GAE Lambda**: When using the Generalized Advantage Estimate, the lambda parameter will control the trade-off between bias and variance. While it is typically kept within the high 0.95–0.99 range, this depends on the quality of the value estimate V(s) being used, and more accurate V(s) can allow for greater reliance on it when calculating the Advantage.
- **Time-Horizon**: The "best" time horizon is determined primarily by the reward structure of the environment itself. The time-horizon should be long enough to allow the agent to likely receive some meaningful reward within it, but not too long as to allow large amounts of variance into the signal. In cases where the environment has high-variance in the expected rewards, utilizing a smaller time-horizon will allow for a more consistent learning signal in spite of that stochasticity.

With all the tweaking and tuning that often goes into the process, it can sometimes feel overwhelming, and like black magic, but hopefully, the information presented above can help contribute, even in a small way, to ensure that Deep Reinforcement Learning is a little more interpretable to those practicing it.

If you have questions about the bias-variance trade-off in RL, or if you are an RL researcher and have additional insight (or corrections) to share, please feel free to comment below!

Thanks to Marwan 'Moe' Mattar for the helpful feedback when reviewing a draft of this post.

Machine Learning | Reinforcement Learning | Deep Learning | Neural Networks | Robotics

Written by Arthur Juliani

Follow

12.9K Followers · Writer for ML Review

Postdoctoral researcher at Microsoft. Interested in artificial intelligence, neuroscience, philosophy, and meditation.

More from Arthur Juliani and ML Review

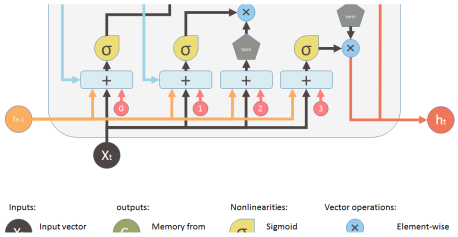
Arthur Juliani

How do psychedelics affect the brain?

A primer on the REBUS/CANAL theory

15 min read · May 2

278 4



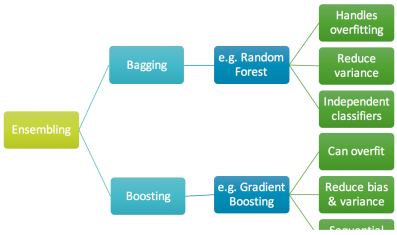
Shi Yan in ML Review

Understanding LSTM and its diagrams

I just want to reiterate what's said here:

7 min read · Mar 13, 2016

9.7K 61



Prince Grover in ML Review

Gradient Boosting from scratch

Simplifying a complex algorithm

8 min read · Dec 8, 2017

9.8K 30

Arthur Juliani in Emergent // Future

Simple Reinforcement Learning with Tensorflow Part 0: Q-Learnin...

For this tutorial in my Reinforcement Learning series, we are going to be exploring a family ...

6 min read · Aug 25, 2016

15.2K 112

See all from Arthur Juliani

See all from ML Review